Towards an embedded real-time Java virtual machine

Anders Ive

Licentiate thesis, 2003

Department of Computer Science
Lund Institute of Technology
Lund University

ISSN 1404-1219
Dissertation 20, 2003
LU-CS_LIC:2003-4

Thesis submitted for partial fulfillment of
the degree of licentiate.

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

E-mail: Anders.Ive@cs.lth.se
WWW: http://www.cs.lth.se/~ive

Abstract

Most computers today are embedded, i.e. they are built into some products or system that is not perceived as a computer. It is highly desirable to use modern safe object-oriented software techniques for a rapid development of reliable systems. However, languages and run-time platforms for embedded systems have not kept up with the front line of language development. Reasons include complex and, in some cases, contradictory requirements on timing, concurrency, predictability, safety, and flexibility.

A carefully tailored Java virtual machine (called IVM) is proposed as an approach to overcome these difficulties. In particular, real-time garbage collection has been considered an essential part. The set of bytecodes has been revised to require less memory and to facilitate predictable execution. To further reduce the memory footprint, the class loader can be located outside the embedded processor. Since the accomplished concurrency is crucial for the function of many embedded applications, the scheduling can be defined on the application level in Java. Finally considering future needs for flexibility and on-line configuration of embedded system, the IVM has a unique structure with which, for instance, methods being objects that can be replaced and GCed.

The approach has been experimentally verified by a full prototype implementation of such a virtual machine. By making the prototype available for development of real products, this in turn has confronted the solutions with real industrial demands. It was found that the IVM can be easily integrated in typical systems today and the mentioned requirements are fulfilled. Based on experiences from more than 10 projects utilising the novel Java-oriented techniques, there are reasons to believe that the proposed approach is very promising for future flexible embedded systems.

Acknowledgements

This thesis would not be without the support and help from my supervisors, Boris Magnusson and Roger Henriksson that tirelessly followed my progress. Their valuable ideas have made drastic improvements to the disposition of the thesis. The support and comments of Klas Nilsson have been especially valuable for me as they returned my focus to the original problems and objectives of this work.

This thesis is also the product of many projects in cooperation with other companies and researchers. I am especially grateful for the project with BlueCell that resulted in a product where the IVM was an integral part, but above all I am glad for the acquaintance with the managers of BlueCell, Mats Iderup and Björn Strandmark, whose practical knowledge excel in the hardware and software field. The cooperation with Ericsson and ABB has been invaluable during the development of the machine. At Ericsson I thank Magnus Larsson, Elizabeth Bjarnasson, Christer Sandahl, and Sten Minör, for their supportive and positive attitude towards reaching a "Java-in-the-ear" solution. The constructive collaboration with Magnus Larsson during a couple of hectic weeks at Ericsson resulted in major improvements to the IVM code. The master thesis of Thomas Fänge and Daniel Linåker at Ericsson inspired me to improve the IVM with some optimisation. The projects with Anders Roswall at ABB Corporate Research in Västerås and Michael Meyer at ABB Automation Technology Products in Malmö have resulted in valuable contributions to the machine. I thank Anders Lindwall, Andreas Rebert, Johan Gren, and Jens Öhlund for their cooperation in their excellent student project in which they utilised the IVM in a real-time application. Their results have been most valuable in this licentiate thesis. Their summer project at ABB in Västerås, where the IVM was integrated into an embedded platform produced many new ideas concerning the IVM. The master theses of Johan Gren and Jens Öhlund, and Tor Andrœ and Johan Gustavsson at ABB Automation Control in Malmö resulted in further developments of the IVM code.

The IVM has been integrated in many research projects. First I thank Patrik Persson for his support and his friendship. His ideas from "Skånerost" are a valuable part of the WCET analysis of the bytecodes. I thank Anders Nilsson for his work on the Java2C converter that resulted in a unified object model of the IVM. Torbjörn Ekman also contributed with his master thesis concerning a hard real-time kernel on an AVR processor. The unsurpassed knowledge of embedded real-time behaviour of Anders Blomdell resulted, together with the rest of the group, in the garbage collector interface that has been successfully utilised in the IVM and in the Java2C converter. I also thank all the other members of our group Görel Hedin, Sven Gestegård Robertz, Ulf Asklund, and the new members Torbjörn Eklund and David Svensson, for valuable discussion and project ideas. I thank Göran Fries, Lennart Andersson, and all the other colleagues at the Department of Computer Science in Lund for pushing me forward. I also would thank Anders Robertsson and Johan Eker for their support during my early days as a "robot" researcher.

I especially thank Christian Andersson as a minute proofreader and for our "after-work" discussions providing me with determination. In this context, I would also thank Fredrik Jönsson for his understanding and support.

I thank Daniel Einarsson and Flavius Gruian for their friendship and their tireless determination to include the IVM in their projects.

I thank Mads Bondo Dydensborg for his cooperation in the Koala project. His invaluable knowledge of the open-source community and his practical knowledge of all the cool tools have increased quality of the IVM source code considerably and increased my interest in the open source community.

I thank Magnus Landquist for his master thesis work together with the IVM and the PalmOS. His work pinpointed crucial requirements of the IVM that had to be implemented, but above all, he made me laugh so much that my muscles in my stomach cramped.

Finally, I thank Madeleine Emmerfors for her support and love, but also for her determination to proofread the thesis. She forced me to confront the darkest sections of the thesis, which improved the text considerably. Above everything, she made me laugh at myself in moments of despair.

Contents

Chapter 1

Introduction

The purpose of this thesis is to provide a foundation for the integration of high-level object-oriented language features in real-time embedded systems. This is achieved by an implementation of a specially designed virtual machine for Java.

In the field of embedded systems, the state-of-the-art high-level programming languages have not made any major impact, because the imposed restrictions are difficult to cope with in a high-level context. Limited computational power and limited memory resources restrict the incorporation of desirable high-level language features. High-level programming languages have been developed on and adapted to general systems with relatively powerful processors and vast memory resources. Hard real-time requirements, such as predictability, impose further issues that are not even resolved by powerful computers.

Throughout computer history, programming languages have become more expressive and more secure. They have developed from low-level instructions into more abstract constructs that relate to the algorithms. Program complexity decreases with high-level programming languages. The vision is to unambiguously describe the execution of computer programs with few building blocks, sufficiently few for the human mind to grasp (see [Nør99]). Introduction of modern high-level programming languages into the development of embedded systems is desirable and in great demand from the industry.

Common programming issues, such as the problem of encapsulation, or issues regarding re-usability, scalability, and portability are elegantly handled in modern high-level programming languages. High-level languages often provide program organisation and structure. The time to develop software has decreased and the code quality has increased with the utilisation of high-level programming languages.

High-level languages also focus programmers to essential programming tasks. Purely administrative tasks, such as memory management, are handled by the language itself. Programming errors can thereby be avoided. In many high-level languages, a *garbage collector* (GC) automati-

cally performs memory management. Manual memory management, where the programmer allocates and deallocates the memory, has been a major source of severe programming errors.

The principal real-time requirements are worst-case execution time, WCET, predictability, and worst-case live memory, WCLM, predictability. High-level programming languages have not addressed these requirements.

The modern high-level programming language, used as a platform for this thesis, is the secure and platform independent Java programming language (see [JLS00]). As the name indirectly implies, Java was originally designed to be a platform for embedded systems, for instance, coffee machines. However, this original vision has not been implemented during the development of Java. It now requires vast memory resources and high performance computers to execute adequately. This work is an attempt to return to the original vision of Java by implementing a tiny Java Virtual Machine, JVM that executes the platform independent Java bytecode. The *Infinitesimal Virtual Machine*, IVM, is our implementation of a memory efficient real-time adapted JVM. There has been made many attempts to implement this original vision of Java. However, the resulting contributions have often suffered from severe restrictions or overhead.

The work presented in this thesis furthermore takes an important step towards integration of the object-oriented paradigm and real-time embedded systems. As a foundation for further development and research, it thoroughly examines the implications of the requirements introduced in an object-oriented context.

The thesis is structured as follows:

- Chapter 1 includes background and requirements for the work presented in this thesis. Focus is on embedded systems, real-time, and Java.
- Chapter 2 describes the design of the Infinitesimal Virtual Machine considering the requirements mentioned in Chapter 1.
- Chapter 3 discusses the start-up procedure of the IVM that includes class loading, linking, and initialisation. A discussion about bytecode conversion completes the chapter.
- Chapter 4 deals with the runtime description of the IVM with a subsection about real-time considerations.
- Chapter 5 contains experiences with different platform ports of the IVM.
- Chapter 6 discusses related work with embedded systems limitations and real-time requirements.
- Chapter 7 discusses the conclusions of this thesis, together with an elaboration of future work.

1.1 Embedded systems

An embedded system is characterised by a specific application domain — typically something else than the system itself — for example, sensors and controllers. The concepts of embedded computer systems are, how-

ever, difficult to clearly separate from those of general-purpose computer systems. The flexible general-purpose systems are prepared to execute a vast range of applications, and the embedded systems are inexpensive and power efficient.

Figure 1.1 shows these characteristics of different kinds of computer systems.

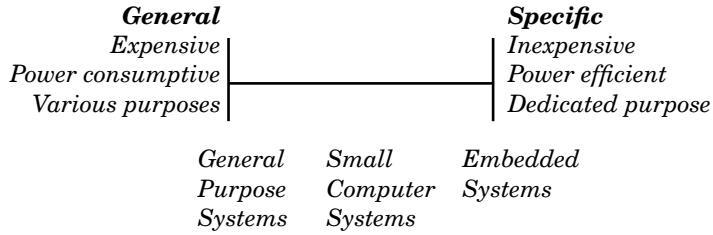


Figure 1.1 *Even though the embedded systems are much simpler than the general-purpose systems, they have other attractive chraracteristics.*

An example of a small computer system is the personal digital assistant, PDA. Embedded systems are, for instance, cellular phones, sensors, and controllers. General-purpose systems are typically found as desktop computers with applications ranging from mathematical calculation and simulation to word processing and entertainment.

The popularity of embedded systems is reflected in their large production quantities. A complete system often combines many embedded systems together with general-purpose systems in a network, in order to benefit from both. Examples of such networks are Supervisory Control & Data Acquisition, SCADA (see [SCADA]), and Controller Area Network, CAN (see [CAN91]).

Even though there are differences between the embedded systems and general-purpose systems, the software languages do not have to differ. The flexibility and the greater power of the general-purpose system have, however, lead to improved language features for general-purpose systems. Computer language development for embedded systems has been lagging behind the state-of-the-art language development due to the restrictions and limitations of the embedded systems. Software in embedded systems is normally developed in a low-level language, typically in C. General-purpose systems are often developed using object-oriented languages like C++ or Java.

1.1.1 Embedded systems overview and restrictions

The restrictions imposed by embedded systems are limited computational power and restricted memory. Depending on the level of the restrictions, the JVM may be utilised in various ways. Preferably, the embedded system has both RAM for the dynamic heap and ROM for the JVM and basic Java programs.

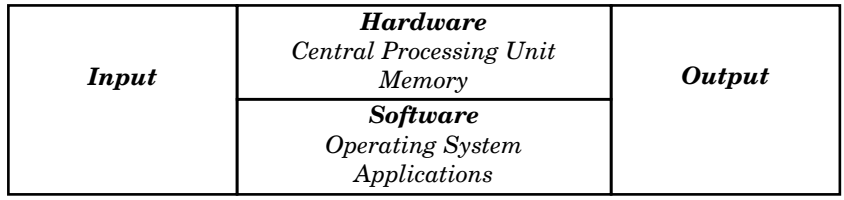


Figure 1.2 *The size of the blocks in a computer system varies depending on the type of the system.*

Figure 1.2 shows the main parts of a computer system in general. The memory area in embedded systems is often divided into several kinds of memory, e.g. RAM, ROM, EEPROM, flash memory, hard drives etc. Our work has targeted embedded systems with a small flash memory and a small RAM area.

Embedded systems may also work together with other systems. An interesting situation where a network contains both general-purpose systems and embedded systems offers a *split machine* approach for the JVM. In those networks, the IVM may be split into an interpreter that resides in the embedded system, and the class loader that resides in the general-purpose system.

The limited memory of embedded systems imposes restrictions, such as a small heap. It is essential to keep a low memory overhead. Fortunately, small memory sizes also lead to shorter pointers. The GC may be designed to accommodate to a small memory area, which can lead to memory efficient and fast garbage collecting algorithm implementations. Real-time behaviour is not affected by the limited memory requirement.

Evidently, the limited computational power imposes requirements on a small overhead for managing the programs. Small and embedded computers often tend to be simple and predictable, which is advantageous when performing hard real-time analysis.

1.1.2 Embedded operating systems

The software organisation in an embedded system is typically divided into an operating system and the application programs. The operating system controls all the computer's resources and provides the basis upon which the application programs can be written.

A *scheduler* manages the threads in a real-time application. The scheduler may reside in the operating system, i.e. *tightly coupled*, or in the application itself, *loosely coupled*. Loosely coupled applications share the processor resource with other applications, or utilise the processor exclusively as a single application or as a high-priority application.

Figure 1.3 shows the different types of software organisation relevant in embedded systems. There are systems that combine both the tightly coupled and the loosely coupled thread management strategies. Those systems are called *combined* in this thesis. For example, a time critical application may execute together with other applications. To ensure dead-

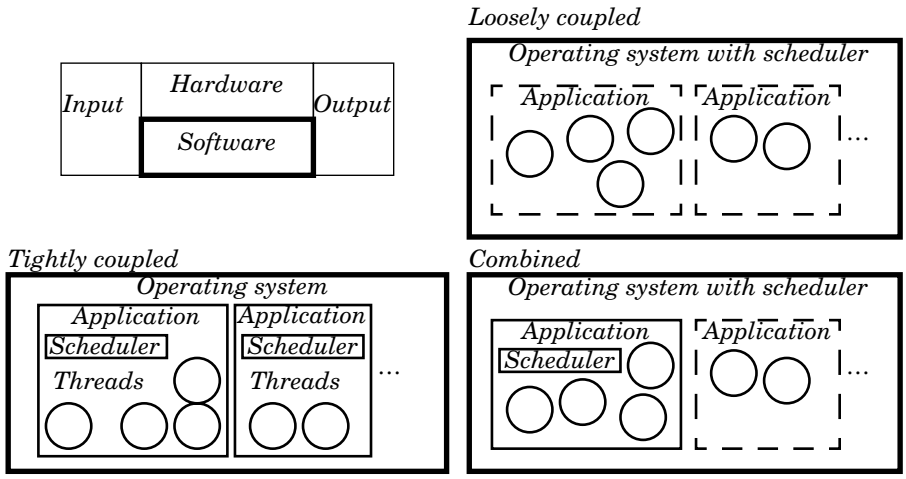


Figure 1.3 *A program is tightly coupled with the operating system if it is able to utilise the operating system threads as its own. Otherwise, it is loosely coupled; it has to manage its own threads.*

lines, the operating system has to guarantee the processor allocation for the hard real-time application.

Migration of new software disciplines into existing embedded systems may take the combined approach to maintain the original code and, at the same time, benefit from the advantages of modern programming languages.

Other embedded systems do not utilise an operating system. Applications for those systems implement their own scheduler, as in the loosely coupled case. In those cases, the processor is exclusively utilised by the application itself.

1.2 Real-time programming

Real-time programming handles applications with time and timing requirements. A real-time program is considered correct only if it executes correctly within a specified period. The *deadline* is the latest time instance before which a calculation has to be completed. Embedded systems often execute real-time programs. Sensors and controllers must calculate and deliver values within a specified time frame. Aeroplanes and their passengers would suffer from unexpected and possibly fatal consequences if the calculations performed by the controllers were based on old or late data from the sensors, or if the controllers spent too much time calculating control signals. Other time critical application domains are for example space probes, robots, and alarm systems for nuclear power plants.

The real-time systems focused on in this thesis utilise one computer and one private memory area. A single computer, however, often has many different tasks to perform simultaneously. *Parallel programming* allows the tasks to be expressed as separate programs. The idea of parallel programming is to give the impression of concurrently executing pro-

grams, *threads*[1]. The idea of real-time programming is to schedule the execution order of the threads in such a way that every deadline is met. The threads typically execute a single loop indefinitely and periodically.

The single-processor approach is called *multiprogramming*. If more computers share the same memory, it is called *multiprocessing*. If the computers are connected in a network with private memory areas it is called *distributed programming*. The approach in this thesis is to study the real-time issues for multiprogramming. The other domains are briefly discussed.

If a number of threads simultaneously read from and write to the same memory area, the program can enter an unpredictable state. Code sections that must be handled *atomically* are called *critical regions*. Mutually excluding threads from concurrently executing the same code region is often realised with *semaphores*, *monitors,* or *events*.

Threads are often given *priorities* to support the *scheduling algorithm*. The scheduler switches threads according to a scheduling algorithm. The basis of the scheduling algorithms is that real-time programs are predictable and schedulable. These concepts are described in the following subsections. This section is concluded with a detailed study of the preemption mechanism, a description of a real-time garbage collector (RTGC) and a summary.

1.2.1 Predictability

The fundamental prerequisite of real-time programming is timing predictability of program behaviour during runtime. Deadlines cannot be guaranteed to be met unless the execution time of the thread loop is known. It is also necessary to be able to predict memory consumption to ensure the availability of sufficient memory during the execution of a real-time program.

Calculation of execution time is mainly based on summation of executed instructions. Control-flow analysis determines the most time-consuming execution path, if there is one. Indefinite loops (cf. the *halting problem* in [AT36]) increase the complexity of the control-flow analysis.

The execution time of instructions is often specified in old Complex Instruction Set Computers, CISC. However, more modern and complex Reduced Instruction Set Computers, RISC, utilise optimisation techniques to increase average execution time, which complicates instruction execution time predictions. Caches, pipelines, instruction level parallelism, and speculative control flow estimation are some performance enhancing techniques that complicates the prediction of instruction execution time. A thorough description of the techniques can be found in [HePa96]. Common, but inexact, solutions to overcome the analysis complexity are program simulation and benchmark measurements (see [CE00]).

Some real-time systems tolerate a percentage of deadline misses. Those systems have *soft deadlines* as opposed to systems with *hard dead-*

1. Threads will also be referred to *processes* in this thesis.

lines where every deadline has to be met. *Dynamic scheduling* can be utilised by soft real-time systems. The scheduler is supported by execution time measurements during runtime to increase the real-time performance.

Prediction of execution time and memory utilisation is focused on the worst possible outcomes. The *Worst-Case Execution Time*, WCET, is the longest possible effective execution time needed to execute a code sequence if the code is executed on a single processor. The overhead of the scheduler is not included in the WCET calculation. Typically, the relevant WCETs are located in the task loops of the threads.

The *Worst-Case Live Memory*, WCLM, value describes the maximum amount of utilised (live) memory during the life of a program. Of the three program phases, start-up, working, and termination, the working phase is the most important. It is desirable to locate WCLM during that phase. There are three different techniques used in the analysis of WCLM:

- **Manual memory analysis** is the sum of statically allocated activation records, variables, and objects. Memory allocation during runtime, *dynamic memory allocation*, is not permitted in these real-time systems. Since all memory that is needed by an application is allocated before runtime (statically), it tend to be much larger than the actual utilised memory, thus WCLM tend to be lower than, and not equal to, the statically determined memory.
- **Automatic memory analysis** examines the code to determine the maximum amount of utilised memory. Generally, the automatic analyser cannot determine the maximum sizes of data structures or the maximum recursion depths.
- **Annotated automatic memory analysis** is supported by annotations in the code set by the programmer. The annotations describe the maximum sizes of data structures and the maximum recursion depths. The annotations enable the programmer to utilise more advanced programming language concepts, for example, recursion, and lists, in real-time programs. A detailed study of such code annotation techniques can be found in [Per00].

1.2.2 Context switch

The procedure where an executing thread is stopped and another thread is started is called a *context switch*. The context, i.e. all the processor register values, for the stopped thread is written to memory and the context of the starting thread is read into the processor. When a thread is restarted, it continues executing from where it was previously stopped, just as if no interruption would have occurred.

Scheduling algorithms for real-time systems rely on involuntary changes of active threads, *preemption*. The scheduler decides when a context switch is to occur. If context switches are only initiated by the application itself, the context switches are called *voluntary*, or non-preemptive. Voluntary context switches result in unpredictable execution times, and they burden the programmer with extra programming tasks.

Preemptive context switches are typically triggered by a clock, or at certain pre-defined *preemption points*. Table 2.2 shows commonly utilised preemption point insertion techniques in a Java perspective. Not all the solutions are deterministic. Non-deterministic preemption points are disqualified in hard real-time systems. The estimated times presented in the table show the average preemption point interval and the maximum time between preemption points. Time is measured by the duration of the execution of a number of Java bytecodes.

The IVM utilises a combination of clock triggered preemption and preemption points. The different times related to a context switch in the IVM are described in Figure 1.4. Implementation of clock triggered context switch is hindered by the problem to determine what registers contain references. References to live objects are important requirements during garbage collection. With preemption points, it is possible to separate references from values.

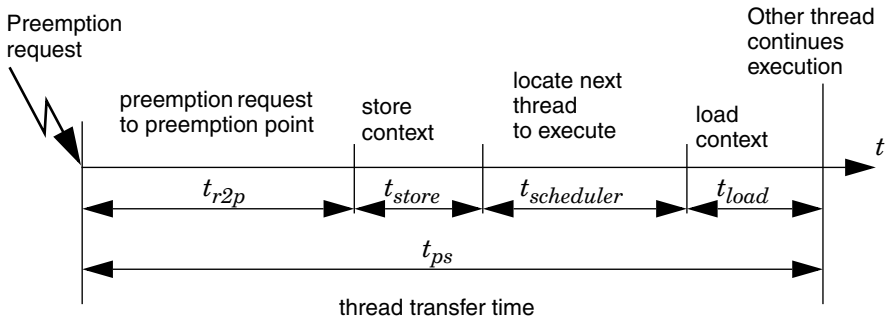


Figure 1.4 *The context switch is not immediately performed as it is requested. The figure shows delays that occur when the context switch is requested until it is performed.*

1.2.3 Schedulability

The schedulability analysis determines if a real-time application will execute correctly. Even though the system is predictable, it is not certain that a real-time program will meet all its deadlines. One approach to ensure schedulability is to measure the behaviour of the system. Such an empirical study, however, will not guarantee correctness, but can give an estimation of the real-time characteristics. Analytical a priori examination of a real-time system, on the other hand, could prove correctness. A third technique for schedulability analysis is to combine the two approaches. This *feedback schedulability* is thoroughly covered in [EHÅ00].

During runtime, the scheduler performs context switches, by executing a *scheduling algorithm*. In hard real-time applications, these scheduling algorithms are based on the predictability of time mentioned in the previous subsection.

This thesis focus is on hard real-time systems. However, all the scheduling techniques could be implemented in the IVM. A short description of different scheduling algorithms is given below.

Static cyclic scheduling
The processor resource is divided into time slots. Every thread is given a specific time slot at a given interval, in which its execution has to finish. The time between the end of a thread's execution and its time slot expiration is not utilised. This approach is simple and straightforward, but burdens the system analyser. Every application, and every software modification, may result in a new thread execution order. That execution schema must be created manually.

Fixed priority scheduling
Every thread is given a priority and scheduled in accordance to these thread priorities. Two popular methods for assigning priority are *rate monotonic scheduling*, RMS, and *deadline monotonic scheduling*, DMS.

In RMS, threads are ordered according to their period, which have to be fixed. Threads with shorter periods receive a higher priority. The threads are not allowed to block each other. RMS lets the thread with the highest priority execute at all times. This solution has been proven optimal by Lui and Layland in [LL73]. Elaboration of the RMS algorithm is expressed in [SLR90] by Sha, Rajkumar, and Lehoczky where thread blocking, scheduling overhead, etc. are covered.

DMS is interesting in systems where threads have deadlines smaller then their period. To achieve the optimal scheduling solution for these systems, the priority should equal the deadline — the shorter the deadline, the higher priority.

Earliest deadline first scheduling
This dynamic scheduling algorithm delays the scheduling decisions until runtime. The thread with the shortest time to its deadline is given the processor resource. This scheduling algorithm was proved optimal by Dertouzous in [Der74].

Feedback scheduling
The scheduler utilises measurements during runtime to schedule the threads in the system. The resource allocation varies during runtime (see [CE00]). This approach cannot sustain hard real-time requirements.

1.2.4 Real-Time Garbage Collection

Automatic memory management, *garbage collection* (GC), is desirable since it relieves the programmer from the burden of doing error-prone manual memory management. Safe modern high-level object-oriented languages include garbage collection. The problems resolved by GC are dangling pointers, memory leaks, and memory fragmentation.

To handle real-time requirements of predictable execution times and predictable free memory, a typical garbage collector must be *incremental*, *exact*, and *non-fragmenting*. The scheduler must schedule the GC in accordance with real-time requirements, (see [Hen98]).

Incremental GC algorithms distribute their execution throughout the execution of the program, as opposed to perform a complete garbage col-

lection when needed. WCET for stop-the-world algorithms is very high, making them unsuitable in real-time systems.

Exact algorithms maintain information to locate references, in contrast to *conservative* GC algorithms that guess if the type of an element is a reference or a numerical value. All elements that resemble to a reference are treated as such. Conservative GC algorithms violate the predictability requirement since the amount of free memory is indefinable. In conservative GCs, values could be treated as references to allocated memory.

Non-fragmenting GC techniques are characterised by the ability to collect live objects into one continuous sequence. This can either be performed by *compaction*, where live objects are pushed together in the same memory area, or by *copy*, where the objects are moved to another memory area. The copying technique splits the memory area into two sections. Typically, as soon as one section is full of objects, the live objects are moved to the empty section.

As an object is moved, it is essential to update all the direct pointers to it. All the direct object pointers are encapsulated in handles, which are presented as references to the programmer. These handles introduce memory and execution overhead.

The real-time garbage collection scheduling algorithm presented in [Hen98] operates as a middle-priority thread, separating the high-priority (HP) time critical hard real-time threads, from the low-priority (LP) soft real-time threads. To increase the real-time performance for the HP threads, their GC work is collected and delayed until the GC thread is allowed to execute, after the HP threads. Scheduling analysis is utilised to prove the schedulability of the HP threads and the GC thread. LP threads perform their GC work as it is generated, i.e. when allocating new objects and assigning references. Figure 1.5 shows a picture of a logic analyser that displays the different types of threads at work. More details about a study of garbage collection and real-time can be found in [Ive98.2]. An important parameter to schedule the garbage collector in

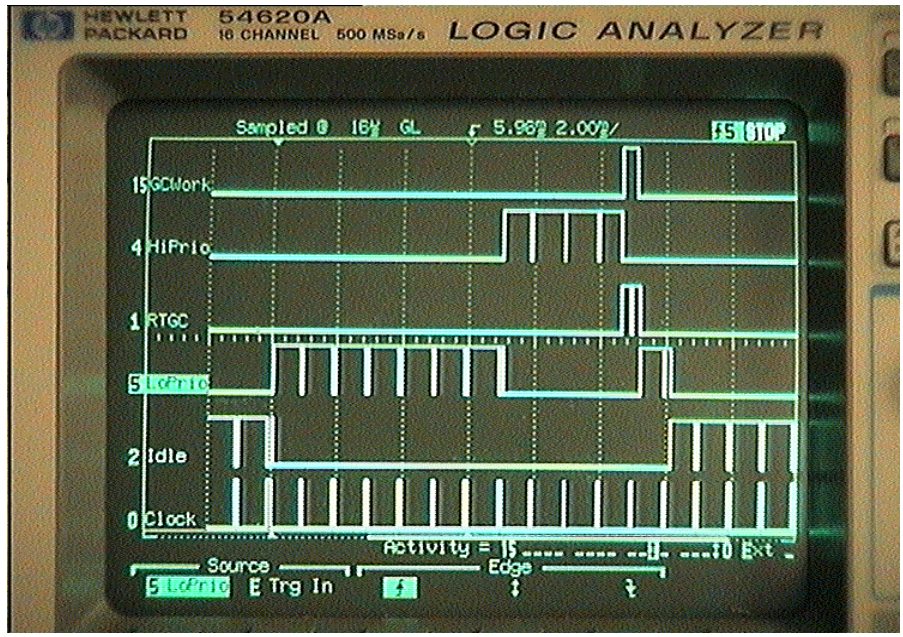


Figure 1.5 *The snapshot of the logic analyser shows how the GC thread co-operates with the high-priority threads and low-priority threads. The six lines show, from above:* ***GCWork*** *– the total GC thread execution time,* ***HiPrio***, ***RTGC***, ***LoPrio*** *– the execution of HP threads, the GC thread, and LP threads respectively,* ***Idle*** *– idle time, and* ***Clock*** *– the context switch handling.*

real-time systems is the memory allocation rate of the high-priority threads.

Before a context switch can be performed in a system with an exact GC, the system must reach a state where the locations of all references are known. References can be stored in memory and in processor registers. In the memory, references are stored in activation frames as local variables, on the stack as global variables, or in objects. The handling of these references is addressed in Section 3.2.2.

1.2.5 Summary

Hard real-time programming addresses problems where time is as crucial as correct calculations. To guarantee correct behaviour, the programs must be predictable with respect to time and memory consumption. The system could then be analysed with a scheduling analysis technique to determine if it is schedulable. Schedulable programs can always be guaranteed to perform all calculations within its deadline limits.

The introduction of automatic memory management in real-time systems increases the complexity of the scheduler. One solution divides the threads in the system into high-priority and time crucial threads, and other threads that are not time critical. Threads that are not time critical are given a lower priority. The GC thread itself cleans the memory after the execution of the high-priority threads and before the low-priority threads are allowed to execute.

The exact RTGC also imposes context switch latency. The system must reach a state where the references to live objects are under control. These problem domains are addressed in this thesis.

1.3 High-level Programming Languages

The development of programming languages is motivated by the vision of attaining higher code quality, e.g. through improvement of the language comprehensibility. This is achieved by abstract *high-level* language concepts, suited for human notions. *Low-level* languages, on the other hand, primarily reflect the hardware functionality. Tor Nørretrander writes in his book "Märk Världen" that the human being is able to keep about seven different pieces of information in mind at the same time (see [Nør99]). These pieces must be carefully selected to increase the comprehensibility of a programming language.

A high-level programming language is characterised, among other things, by the following:

Comprehensibility – the complexity of the language is determined by the syntax and the amount of features covered in the language.

Productivity – the ability to create software products is determined by the programmer's knowledge in programming and by the support from programming tools, e.g. the programming language.

Robustness – a robust programming language is characterised by well-limited concepts, error recovery mechanisms, and the ability to handle heavy program utilisation.

Extendibility – the code size should reflect the program functionality and not increase dramatically as new features are added to a large program.

Portability – the software does not depend on a particular type of hardware. It has the ability to run on a variety of computers.

Hardware specific details are often written in a low-level language and integrated into the high-level domain through a low-level, or *native*, interface. Typical low-level language concepts are memory addresses, pointers, and pointer arithmetic.

The real-time embedded system community primarily utilises low-level programming languages. Modern state-of-the-art high-level programming languages often require vast memory spaces and utilise the processor extensively to manage the language overhead. Average case performance has been optimised, but worst-case execution time analysis has been omitted. These prerequisites conflict with time critical real-time programming and the restricted embedded systems.

This section discusses the advantages of high-level programming languages from the view of the object-oriented programming paradigm. The Java programming language is studied in detail in conjunction with real-time embedded systems. Finally, existing real-time Java solutions are presented before the summary.

1.3.1 Modern object-oriented programming languages

Object-oriented programming (OOP) languages are based on the philosophical fundament of Plato's idea of a perfect entity of which all other instances are implementations. A class describes the ideal entity, and it may be instantiated into objects. The classes can be ordered in a hierarchy to reflect the natural connections of the classes. For example, Linné categorised flowers in a hierarchical order that can be found in The Flora (see [Lin51]). OOP languages support division of code into classes. The programmer has the possibility to organise the software naturally into classes and hierarchies, e.g. according to the functionality of the classes. The intention is to improve the comprehensibility of the code with abstract concepts. A collection of recurrent class diagram designs has been put together in Design Patterns and A System of Patterns (see [GHJV95] and [BMRSS96]).

Subclasses inherit and reuse code from their superclasses. The main idea for code reuse is to increase code quality through "code once, use everywhere". The reused code increases the software robustness through its extensive usage. It is better to test one implementation many times than to test many similar algorithms one time each. However, code reuse may result in a small loss of performance.

A more general way of reusing code is to describe how classes should be created. These descriptions of classes are called *generic types* or *templates*. Algorithms could be made independent of types with generic types. Stepanov and Lee in The Standard Template Library (see [STL95]) describe an excellent example of a general generic type programmer's interface.

OOP languages are suitable to implement automatic memory management. The required information of objects by the garbage collector is defined by classes. Automatic memory management decreases the programming overhead for the programmer, and increases the code comprehensibility and robustness. The memory related pointers are replaced by object related references that either refer to objects or are set to null. Examples of high-level OOP languages are Java, Simula, Beta, and Smalltalk (see [JLS00], [DNM68], [KMMN91], and [GR83] respectively).

In *strong typed* programming languages, the compiler and the runtime system perform controls to assure the correct type before the type entity is utilised. If a situation arises where the program cannot handle the type, the program halts in a controlled manner, e.g. by raising an exception or an error. The idea is to avoid unintentional and undesirable program execution. *Weakly typed* languages, such as C and C++, often provide type controls, but they can be circumvented. Programs could enter a state where the execution is unpredictable.

Low-level languages often include features that extend the language functionality and increase its complexity, for instance, pre-processor directives, and macro expansions. An example of a low-level language is C.

Many OOP languages have both low-level and high-level features. These composite languages must regulate the utilisation of the language by coding conventions, to ensure high-level code standards. The language

itself cannot guarantee the desired robustness of high-level languages. An example of a language with both high- and low-level features is C++ (see [C++91]).

1.3.2 Java

Java is a modern object-oriented programming language primarily designed with the intention to be utilised in embedded systems, for example, coffee machines, remote controls, and portable digital assistants. However, during its development, the language was developed and adapted to general-purpose computers with large amount of memory and powerful processors. A goal of this thesis is to attempt to return to the original vision of Java.

The Java compiler produces an intermediate and symbolic low-level machine code, *bytecodes*, in *classfiles*. A classfile is read by a Java Virtual Machine and converted into an internal representation before execution starts. The functionality of the JVM, especially the functionality of the bytecodes, is specified in [JVM99], The Java Virtual Machine Specification. Some implementations of the JVM compile the code dynamically (see [HS02]). Other JVM implementations interpret the internal code instead.

The main advantage with the classfiles is that they are portable. If a JVM exist for a platform, programs may be written in Java on those platforms. The language specific features of Java are automatic memory management, strong typing, and native code encapsulation.

1.3.3 Real-time aspects of Java

The real-time behaviour of Java is integrated into the language itself and in every object. Processes are termed *threads*.

The two synchronisation mechanisms implemented in Java are *locks* and *events*. Locks are specialised monitors. They are only specified for concurrent systems and not hard real-time systems. According the Java specification, [JLS00] p. 235, “Every object has a lock associated with it, …”. The monitor functionality resides in the Object-class, which every other class inherit from. Another feature of the locks is that they only have one condition variable. A thread can only wait for one single condition to be fulfilled before it is woken.

The implementation of monitors into the virtual machine requires that the machine utilises the monitors every time the `synchronized`-keyword is encountered. The keyword could be a statement or a method modifier

```
synchronized (aLock) {          // synchronised statement
    …                           // The object 'aLock' is locked.
}

synchronized void aMethod() {// synchronised method modifier
    …                           // The object receiving the method call is locked.
}

synchronized static void aMethod() {// synchronised and static method modifier
    …                           // The class-object receiving the method call is locked.
}
```

Figure 1.6 *Locks are located inside objects that are locked through the* `synchronized` *statement and method calls to* `synchronized` *methods.*

(see Figure 1.6). As a statement, the compiler generates lock-related bytecodes to indicate when the thread enters a lock and when it exits the lock. The JVM executes the monitor operations as these bytecodes are encountered. A counter has to be added in every lock since the thread that owns the lock can lock them repeatedly.

In the method modifier case, the lock related bytecodes are not generated. The specification requires that every time a `synchronized` method is invoked, the lock must be acquired before execution continues. The JVM must check the method modifier, and in the synchronised case, try to attain the lock, before the method is executed.

Real-time conflicts in Java

Even though Java is a thoroughly designed modern high-level programming language, there are language constructs that conflict with the requirements of real-time embedded systems. The following subsections relate the quirks in Java with these requirements.

Concurrent monitor specification

The JVM specification states that every object has a lock associated with it. A direct implementation of this statement would consume a lot of memory that will never be used. The overhead of the processor increases, as these locks have to be managed. Solutions to give the impression that every object has a lock are required in memory limited embedded systems. A priori program analysis could determine which classes contain the synchronised method modifier. As objects of those classes are created, an extra lock could also be created. However, the synchronised statement invalidates this procedure since every object could be utilised as a lock in the statement. That removes the possibility for the a priori analysis since anyone may write a program that locks every other accessible object. Objects could, however, be hidden from other programmers.

Unpredictable dynamic class loading

The JVM is specified for dynamic and lazy evaluation techniques. Classes could be loaded as they are needed, and code is analysed and transformed, as it is necessary. In a real-time system, the WCET would be pessimistic if the lazy and dynamic approach would be considered. The static approach is more desirable in real-time systems, where all necessary classes are

loaded before the execution starts. Loading and conversion times should not burden the WCET analysis.

Unpredictable garbage collector behaviour
The garbage collector algorithm is influenced by the JVM at two points. The complexity of the GC algorithm is thereby increased. First, the method `finalize` is inherited into every object from the class `Object`. The method description states that (see [JLS00], Section 12.6):

> *Before the storage for an object is reclaimed by the garbage collector, the Java virtual machine will invoke the finalizer of that object.*

Some garbage collecting algorithms only determine the live object set. The added functionality of dead object determination and `finalize`-method call extends those GC algorithms.

In real-time applications, the WCET analysis would be pessimistic if the `finalize`-methods are incorporated into the scheduling analysis, because the execution time of the finaliser must be included in the WCET analysis.

Native manual memory management
In the Java Native Interface (see [JNI99]), there are methods that lock an object. It may not be moved by the GC until the programmer releases the pointer. This manual memory management conflicts with the operations of the GC. It also introduces low-level pointers and extra overhead for the programmer.

1.3.4 Related work

There are many attempts to implement real-time embedded systems. None of the projects can, however, determine the real-time behaviour of Java programs together with automatic memory management.

Two approaches to the handling of real-time issue in Java can be recognised. First, the API could be extended with a specific real-time module, and the interpreter could be modified. Second, a Java compiler could generate real-time code. This section lists some interesting Java real-time solutions. The projects are examined in Chapter 6.

Real-time Java specifications
The Real-Time Specification for Java is a document describing how the Java Language Specification should be specialised to ensure hard real-time behaviour (see [RTSJ00]). Some manual memory managements have been introduced and a detailed real-time API has been specified.

Real-time Java compiler
A Java compiler could perform the conversion of Java to predictable native code. Either the bytecode or the Java source code is transformed. The compilation could be performed ahead-of-time or by a JIT.

Interesting works in this area are the Java-to-C converter by Anders Nilsson in [NE01], the commercial RTOS and bytecode to native compiler (see [JBed], and [PERC02]).

1.3.5 Summary

The incorporation of high-level languages in real-time embedded systems is complex since the restricted memory and limited computational power requirements often interfere with high-level functionality. It is, however, desirable to benefit from the advantages of high-level languages in embedded systems; the code quality increases. The major benefits are relief of programming memory management, better language support for software organisation, and clear languages specified for high-level programming.

The programming language studied in this thesis is the object-oriented Java programming language. It covers the crucial high-level functionality and hides the low-level details behind a native interface. Java serves well as a high-level language to prove the concept.

Chapter 2

The Infinitesimal Virtual Machine

The Infinitesimal Virtual Machine, IVM, for Java is a research prototype intended to execute Java programs in embedded systems with real-time demands. Besides proving that object oriented programs can run in real-time environments, the IVM was developed as a research platform intended for a study of code replacement during runtime with real-time requirements. The IVM is also suited to support other research in connection with Java and real-time.

The IVM is designed as an interpreter. Interpreted code is slower than compiled code. However, the goal of this thesis is to prove that it is possible to utilise high-level object oriented languages in real-time embedded systems. Compared to real-time programs that are not optimised, the execution of Java programs by the IVM may perform well. Hard real-time applications often are not optimised to ensure stability, and remain readable and traceable. In this aspect, the interpreted bytecodes may be competitive. Besides, the interpreted bytecode is platform independent, simple, more expressive than binary code, and thus suitable as an interface for real-time analyses.

This section describes the design of the IVM and the design considerations. First, the overall static data structures of the IVM are described as modules and interfaces between the modules. Then the dynamic runtime data structures are described, for example, classes, objects, and method calls. A split variant of the IVM is introduced. It imposes further design issues. The runtime behaviour is discussed and the implications of *preloaded* classes are discussed. The section is concluded with a general design discussion and a section summary.

2.1 Java Virtual Machine Overview

The overall structure of a Java Virtual Machine, as it is described in the JVM specification, is depicted in Figure 2.1, (see [JVM99] pp. 67-70).

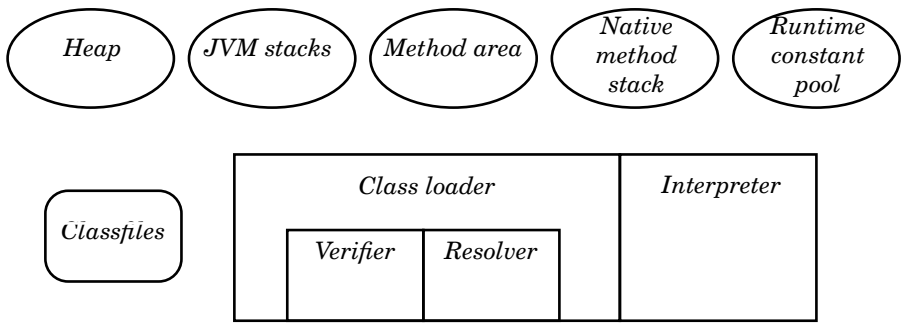


Figure 2.1 *This overall structure of the Java Virtual Machine shows the main parts, i.e. the modules and memory areas, according to the JVM specification.*

Classfiles are loaded by the *class loader*. It verifies that the code is secure. The code is then resolved by the *resolver*. During resolution, the symbolical references in the classfile are substituted into internal references to increase the overall runtime performance during the execution of the methods in the class. The interpreter utilises the internal references to execute the program.

The memory of the JVM is organised in five areas:

The **Java Virtual Machine stacks** contain one stack per thread. The stack stores local variables, temporary results, and manages the method calls by the JVM stack.

The **heap** is the runtime data area. Objects and arrays are located on the heap, which is managed by the garbage collector.

The **method area** is shared among all threads. It contains constants, class descriptions, method data, and code.

The **runtime constant pool** contains the symbols and constants of classes. The information is relevant to transform the class into an internal representation or to examine the class *retrospectively*.

The **native method stacks** are typically allocated one per thread. Native machine dependent methods utilise the native stack to perform its execution.

2.2 Modules and interfaces

The IVM is divided into modules to comply with various demands that originate from its usage. The rationale is the embedded system limitations and the real-time requirements, which necessitate modifications to the original JVM design. Another design goal for the IVM is to facilitate the port process to other platforms. It is achieved by division of platform specific code and platform independent code. Platform specific code is encapsulated in modules and accessed via a *port* interface. The intention

has been to create a simple design intended to be extendable and flexible. Other JVM research projects could utilise the IVM as a platform for research on Java or JVM related ideas.

The overall structure of the IVM is depicted in Figure 2.2. Two new modules, the optimiser and the real-time analyser, are added to meet the requirements of embedded systems and of real-time systems. The scheduler and the garbage collector are explicitly shown because they have different behaviour in real-time systems and concurrent systems. The real-time requirements inflict special solutions to those parts that are superfluous in concurrent systems.

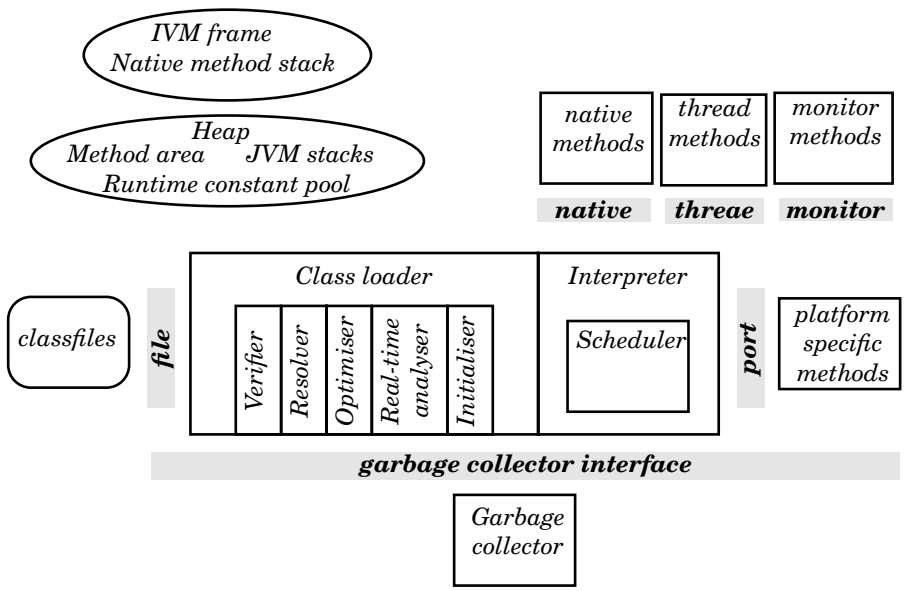


Figure 2.2 *The overall structure of the Infinitesimal Virtual Machine shows its modules, interfaces, and memory areas. The difference from the original JVM specification is the real-time analyser and optimiser in the class loader.*

The heap is utilised for the JVM stacks, the method area, and the runtime constant pool. This solution simplifies the overall structure of the IVM and reduces the amount of design decisions. Native methods execute on the same frame as the IVM itself.

The modules are:

The **class loader** locates and loads classes into internal data structures.

The **verifier** checks if the classfiles are well formed and secure to execute.

The **resolver** converts bytecodes into an internal form.

The **real-time analyser** creates real-time information about the code for the scheduler.

The **initialiser** initialises the loaded classes.

The **interpreter** executes the bytecodes.

The **scheduler** schedules threads.

The **garbage collector** works together with the scheduler to uphold real-time characteristics.

The **platform specific methods** — the IVM support methods that are platform dependent.
The **thread and monitor methods —** support for different thread and monitor implementations are implemented in this module.
The **native methods** store all native methods.

The original class loader has been split into a verifier, a bytecode resolver, and an initialiser. The real-time analyser prepares the internal class representation with real-time information that is relevant to the scheduler. The information concerns WCET, and WCLM. The optimiser is mainly focused on memory saving optimisations, but it is possible to extend it with other performance-increasing optimisation techniques. The garbage collector interface enables various garbage collector modules. For real-time embedded systems, a scheduling of a GC is available in [Hen98].

The interfaces are:

File: the classfile access protocol
Native: support of and access to native methods
Port: support methods for the IVM
GCI: garbage collector interface
Thread: interface to context switch and thread handling
Monitor: access to lock handling
Bytecode conversion: description of the internal bytecodes

The file interface describes how to access classfiles. It is utilised by the class loader. This interface gathers hardware specific file formats for different platforms, in modules. It consists of simple file accessing methods, for example, open and close files, and read bytes.

The GCI is platform independent; the various garbage collectors that comply with the interface can be interchanged. The GCI also supports thread safe GC utilisation and a debug layer to support IVM and GC development. The debug layer can also be utilised when different garbage collectors are tested and evaluated. Real-time requirements necessitate GC algorithms that are unnecessary complex for concurrent systems. GCI enables the ability to change GC implementations in accordance with the purpose of the application. The GCI is utilised throughout the code of the IVM.

Some methods are inherently platform dependent. For instance textual output could be presented on a monitor or a LCD display. Such platform dependent methods are collected in the port interface.

The native interface differs from the other interfaces. It has two parts, one with access to native methods from the IVM, and another with access of Java objects and Java methods from native code. The latter is similar to the JNI specification [JNI99]. In the IVM design, the native methods are statically linked during compilation. New native methods cannot be added during runtime. They are statically linked with the interpreter. Native methods are generated from native method descriptions. Many native method descriptions stem from the Java API, but platform specific implementations could override the native methods. The programmer could also add native method descriptions. The generated native method file contains all the accessible native methods during runtime.

The monitor and the thread interface describe the methods that are relevant for the IVM to be able to reschedule threads and perform synchronisation of threads.

The following subsections contain detailed descriptions of the interfaces. Another interface, the bytecode conversion interface, offers alternative bytecode implementations suitable for specific platforms. The concluding discussion covers an interface to threads in the IVM.

2.2.1 File interface

The file interface is a universal and platform independent interface to access classfiles. The underlying file system may for example store classfiles on a hard drive, via a network, or on a flash memory module. Only the fundamental file methods are implemented in the interface. The interface concerns:

- Open and close classfiles.
- Read information (byte, short, or int).
- Check if a classfile exists.

The interface should implement a temporary buffer to enhance file accesses. Then chunks of information could be read from the file instead of single bytes.

2.2.2 Native interface

The native interface describes how the JVM and Java objects can be accessed from native code, and how native methods are invoked and added.

The native methods in the IVM are implemented in C. To support the programmer, a tool, *Java native extractor*, has been developed to extract declarations of native methods from Java files and provide a default native method implementation, i.e. an implementation that displays a message that the native method is not finished. Arguments are popped from the stack and a default return value, if any, is pushed. The Java native extractor also forces the native programmer to encounter the coding standards of the IVM. It is imperative to utilise the heap correctly. Native code has to follow the GCI correctly. The programmer is supported by the default implementation generated by the extractor, and by the debug layer of the GCI that examines if the memory is handled correctly.

The native implementations are collected by the *native code generator* and put into a single file that is compiled and statically linked into the IVM. The Java Language Specification states that native methods should be loaded dynamically, i.e. the native methods should be located in shared objects, or dynamic link libraries. At this point, the IVM breaches the specification to the benefit of decrease of the complexity in the IVM. Hard real-time analysis is simplified if loading times of native methods are excluded from the analysis.

Native method implementations are supported for different platforms and different thread models. The native code generator selects the native

implementations due to the given characteristics of the current IVM compilation. Figure 2.3 describes the process of native code integration into the IVM.

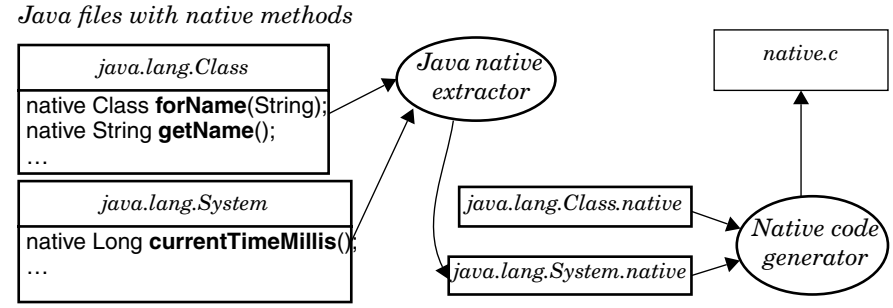


Figure 2.3 *One part of the native interface describes how native code should be added into the IVM. The Java native extractor supports the programmer with a default native method implementation that fulfils the native method interface.*

Inside the IVM, each native method is represented by a unique index number. The number is used to locate the native method during runtime. The native code generator generates a switch statement where all the native methods are case alternatives. Figure 2.4 shows this generation and the resulting switch statement.

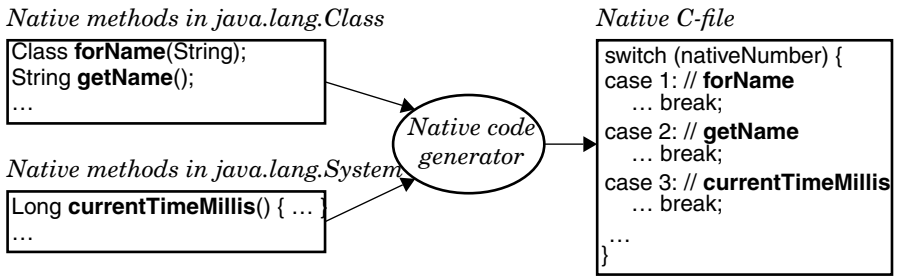


Figure 2.4 *In the IVM, the native methods are identified as numbers that are utilised in a switch statement to locate the method, when the native method is to be executed. The switch statement is generated by the native code generator from the native method implementations. The resulting native file is statically linked into the IVM.*

Native methods in the IVM execute on the same stack as the interpreter. This simplifies the design of the C stacks. One stack is needed for the IVM itself and for the native methods. However, this influences the real-time behaviour, since only one native method is allowed to execute. The interpreter is blocked from further context switches until the native method is finished. This restriction complicates the analyses of WCLM and WCET for native methods. WCET analysis for native methods is omitted in the IVM. Only the bytecodes are studied in the WCET analysis. WCLM analysis is relevant to design the size of the C stack for the IVM. The IVM native interface can be utilised to analyse the memory consumption for native methods. However, if the methods are non-deterministic in size, the WCLM is only an approximation.

Inside the native methods, it is possible to invoke Java methods and access Java objects. These procedures are described in the Java Native Interface Specification (see [JNI99]). The main functionality of the JNI consists of the following tasks:

- Call virtual or static methods.
- Pass arguments to and from the virtual machine.
- Get and set object-, static- and array fields.
- Handle strings and arrays, e.g. get number of elements, or get a subsection.
- Handle exceptions, i.e. generate, and throw exceptions.
- Check types.
- Get internal identification for methods and fields.
- Get class, superclass, or virtual machine.
- Native method registration.
- Synchronise threads.

The native interface implemented in IVM includes subsets of the categories above. The reasons for this are to decrease the overall code size of the IVM and that the full JNI implementation has low priority in the project. It is possible to implement almost every JNI method without difficulty. However, some of the methods in the JNI are related to memory management. For instance, it is possible to lock the position of an object in memory. This procedure intrudes on the workings of the GC and conflicts with the hard real-time requirement of a predictable memory area without fragmentation, to ensure the size of allocated memory. These methods are introducing manual memory management and thereby introducing low-level concepts into Java.

The static loading of native methods interferes with the JVM specification. It states that dynamic loading of native methods is necessary. The specification breach is not considered critical. Dynamic loading of native methods would increase the complexity of the real-time analysis.

2.2.3 Port specific interface

Methods not covered in another interface and necessary for the IVM are collected in the port interface. Primarily it serves the supportive purpose of printing and context switching. Types utilised throughout the IVM code are also defined as compiler dependent types in the port interface.

2.2.4 Garbage Collector Interface

This interface allows different garbage collecting algorithms to be implemented and utilised in the IVM. However, it also burdens the IVM implementer and the native method implementer, with code regulations. Every reference assignment and reference utilisation has to be capsulated and accessible from the garbage collector. The interface also defines a description of object layouts, and garbage collecting object overhead. The following coding regulations are added:

- Declare (allocate) a reference under the supervision of the garbage collector.
- Use (access) GC reference.
- Compare GC references.
- Check GC reference assignment.
- Add/remove GC reference to/from live reference set.
- Initialise heap.
- Allocate object on heap.

The GC fields in every object ordinarily consist of a handle location (forwarding pointer). However, different garbage collecting algorithms require different GC fields. For instance, mark algorithms require a mark bit.

The object layout description shows the location of GC references inside an object. Figure 2.5 explains the grammar and Figure 2.6 shows an example of object descriptions. In arrays, the number of pointers is noted as variable in the object description. The actual number of references is contained in the array object itself. The variable marker also indicates the position.

<pointer locations>	→	*<pointers>* | *<stop>*
<pointers>	→	*(<number of pointers> <skip bytes>)* | *<stop>*
<skip bytes>	→	*(<number to skip> <pointers>)* | *<stop>*
<number of pointers>	→	*<number>*
<number to skip>	→	*<number>*
<number>	→	*<byte number>* | *<word number>* | *<variable size>*
<byte number>	→	0 – 253
<word number>	→	255 *<high byte> <low byte>*
<high byte>	→	*<byte>*
<low byte>	→	*<byte>*
<byte>	→	0 – 255
<variable size>	→	254
<stop>	→	*<end of array>*

Figure 2.5 *The garbage collector utilises strings formed from the grammar to locate pointers in objects. Variable size entries indicate that the number of pointers or bytes is found in the object itself at the location.*

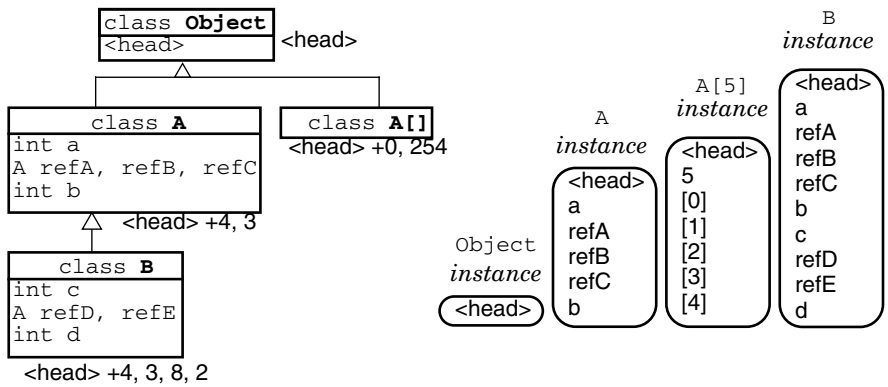


Figure 2.6 *The object descriptions show where the GC references are located in objects. The references are counted and the space between references is given in bytes. Variable length is given as the number 254. It shows that the corresponding location in the array-object contains the number of elements. It is five in the array-object.*

2.2.5 Thread interface

Real-time threads are not specified in the JVM specification. To achieve real-time characteristics a specific implementation of the threads in Java must be implemented. The implementation of threads is modularised to allow different thread implementations. Real-time threads may be too complex for applications without real-time demands. Those applications could choose a lighter thread implementation.

The thread interface involves context switching and monitor operations. The IVM is designed to check if a context switch is about to occur, after the execution of every bytecode.

2.2.6 Bytecode conversion interface

The IVM supports alternative implementations of the internal instruction set. Many of the bytecodes specified in the JVM specification (see [JVM99]) are utilising symbolic references as operands. These references are time-consuming to follow and they should be replaced with direct references to increase runtime performance. Other reasons for alternative bytecode implementations are introduction of performance-increasing bytecodes, and removal of unnecessary bytecodes. An example on how to increase the performance is to map platform specific abilities to bytecodes.

In some embedded systems, the restricted memory requirement is especially prominent. Reduction of the internal bytecode instruction set decreases the size of the interpreter. However, some performance-increasing bytecodes are also removed.

In some embedded systems, there are bytecodes that are never utilised. These unnecessary bytecodes are, for example, floating-point operations or monitor related bytecodes in a single threaded application.

Conversion of symbolic bytecodes
Symbolic references are represented in textual form and should be converted into direct references to gain performance. Textual comparisons during runtime are time-consuming. In applications where symbols are only utilised to resolve symbolic references, the symbols may be removed to save memory space after the conversion.

Direct references may be pointers or indirect references, i.e. offsets. There are three groups of references from the bytecode:

1. Class references
2. Virtual methods references and object field references
3. Static method and field references, and constants

Some bytecodes utilise two of the references, but most of them utilise only one. Symbolic class references are replaced by direct references to the class. Virtual methods are often replaced by an offset in the virtual method table. Fields are replaced by offset into the object. A straightforward resolution of static methods is by an offset into the static method table of the object. Constants and fields are also located by an offset into the static field array of the class.

There are other solutions to the constant bytecode resolution. For example, inline methods, or propagate constants into the bytecode.

Minimal bytecode instruction set
The JVM instruction set operates on four different components: the objects, the static objects and static constants, the stack, and the local variable area. Operations exist to transfer information between the components, and to directly operate on them. Since the machine is stack-oriented, operations on the stack attract most bytecodes. Figure 2.7 shows the overall bytecode operations of a JVM.

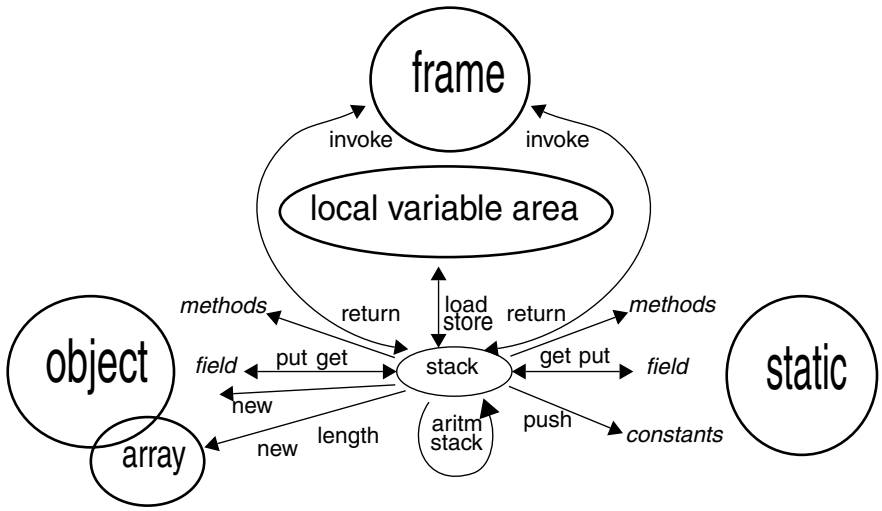


Figure 2.7 *The Java bytecode instruction set contains mainly operations on the stack and data transfer to and from the stack.*

The main component is the stack. The JVM is a stack-based machine without registers. All operations are performed on the stack. Data is loaded and stored in three different places, besides the stack. Those are the other components, i.e. the objects, the local variable area, and the static variables. The objects are created from the application program. The static objects are created by the JVM itself. The local variable area is a part of the frame. In the IVM, the stack is also a part of the frame, and frames are stored as objects on the heap.

Data transfer between the different memory areas (objects, static variables, constants, local variable area, and stack) are described as put and get, load and store, push, and stack manipulating bytecodes. Other essential bytecodes handle method invocations. They occur in objects, either static or ordinary, and during creation of objects (`new`). These bytecodes, together with the arithmetic bytecodes, span the fundamental operations in the machine. Arrays are treated specially in the JVM. They have special creation bytecodes and a specific bytecode that delivers the size of an array to the stack. Bytecodes that address the frame directly are related to control flow and exception handling. Synchronisation bytecodes are related to thread handling.

The static objects and the frames are also located as objects on the heap. It would be possible to substitute those bytecodes with object related bytecodes to decrease the size of the interpreter even further. The description of internal data structures as classes is dealt with in Section 7.6, "Meta virtual machine".

Unnecessary bytecodes in embedded systems

The restrictions of the embedded system hardware may be utilised to decrease the size of the IVM code and the bytecodes of the method. The bytecode converter could also be made smaller in size. The following groups of bytecodes may be superfluous in some platforms:

- Type reduction and unsupported types: Not all the Java types may be supported by the platform. Those bytecodes may be removed for those systems. For example, if the platform does not support floating-point arithmetic, the types `float` and `double` may be removed. Other types may not be relevant for the underlying platform, e.g. `int`, `short`, `char`, or `byte`.
- Single threaded application: If the application is single threaded, the overhead for multi-threading could be removed. Synchronisation bytecodes and synchronisation code in the IVM could be removed.
- Limited memory area: Since the RAM size in embedded systems are limited, the heap has a definitive maximum size. References and addresses could be made smaller within the IVM code. The number of classes could be limited by the memory size. Indirection to a smaller range of classes could decrease the size of offsets.
- Some runtime checks may be removed after an analysis. For example, situations where array accesses cannot exceed the array limits may be removed. See [ACL98] for more details on such optimisations.

2.2.7 Discussion

The IVM is split into modules connected with interfaces, to support many different implementations. The modularisation serves well as a research platform. Different implementations of the modules could support examination of the behaviour and functionality of the virtual machine.

The focus during the design of the IVM has been to accommodate the requirements of real-time embedded systems in the machine. The memory consumption is important to minimise and real-time threads impose requirements that are necessary to deal with in the JVM design.

2.3 Internal data structures

The internal data structures lay the foundation of the work of the IVM. Internal representation of classes, objects, and methods, are described in this section. During runtime, *template* structures support, for example, the automatic memory management, the dynamic type checks, and the localisation of virtual methods. *Symbol tables* support the class loader to transfer classfiles into an internal representation of the class.

In general, the template structure collects information common to underlying templates and objects. However, these structures do not correspond to the Java class inheritance structure. The difference is analysed in a section after the description of templates, object layouts, and inheritance structure.

2.3.1 Object design

The primary design goal for the objects, i.e. instances, was to make them simple and to design them for real-time and dynamic code replacement purposes. Performance was considered a secondary goal.

An object consists of an object head and the attributes designed by the programmer. The object overhead consists of garbage collecting part, the template reference, and information concerning the lock of the object. The size of the garbage collecting part is dependent on the algorithm of the GC. The template reference refers to the template describing this object and containing all common information for all objects of that type. The lock is due to the Java specification. See the Java Virtual Machine ([JVM99]) for more information about the lock mechanism.

Because methods are common to all objects of the same type, they are collected in the template. The attributes, as described in the class, reside in the object, since they are unique for every object. The object structure implemented in IVM is described in Figure 2.8. Other information that is common to objects of the same type is a description of the object, for example, the object size. The template of objects is actually class descriptions. They contain methods, static variables, symbolic information about the class for further class loading, and an interface array to keep track of the implemented interfaces.

Some garbage collecting algorithms use handles. The mark-and-sweep algorithms or mark-and-compact algorithms also utilise a mark pointer

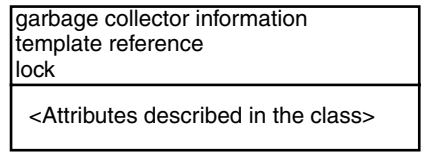


Figure 2.8 *The object structure layout consists of an internal overhead for managing the object and the attributes described in the object's class.*

field. The IVM is designed with the intention of different object layout techniques.

2.3.2 Templates

The internal hierarchical template structure contains runtime information common to children of the template. All IVM objects are referring to a template that describes their layout and design. Other information gathered in templates is garbage collecting information, i.e. object size and the location of references inside the object. The reason to collect the common information in a template instead of inside the objects is to save memory space. It is also a principal decision to strive to gather information affecting many objects in one place. The major drawback is performance loss. It is quicker to access the information immediately, in the objects, than through indirection via a template reference.

In one specific case, the common information is contained in the objects themselves. Garbage collector information for array-objects is also described in the objects and not solely by their templates. Instead of having a separate template for every array, all arrays may share one single template at the expense of slightly increased array sizes.

The template hierarchy inside the IVM is shown in Figure 2.9. The internal data structures in the figure are created prior to execution and class loading. Classes are created during class loading. Dynamic data structures are created during runtime as described by the executing Java program. At the top of the hierarchy the meta meta template is located. It describes itself as well as its children. They are the meta method template and the meta meta class template.

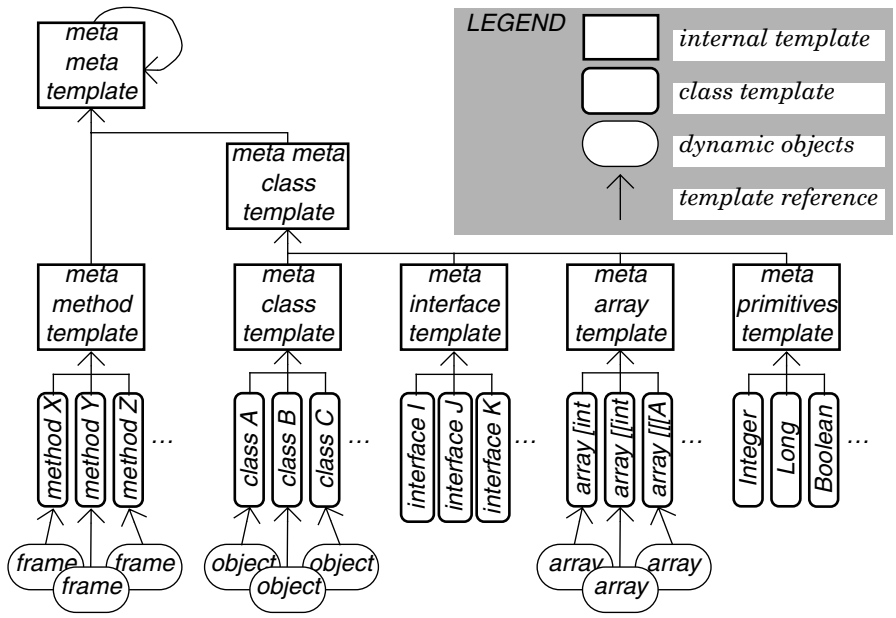


Figure 2.9 *The template structure in the IVM shows how the objects and templates relate to each other in the IVM system. Methods are marked as classes since they are created during class loading.*

The separation of classes, interfaces, primitive classes, and array classes enables the runtime system to determine the type of an object during runtime. Some Java methods require this distinction. For example, in the class `Class` there are methods, `isPrimitive` and `isInterface`, that examine the type of the object.

The template structure is utilised by the runtime system to support the GC with the layout and sizes of objects and the other data structures in the runtime system. The interpreter compares types with template reference comparisons. Virtual methods are found by following the objects template reference. Similar data structures can be found in [KM93].

The design of a template head

Every template is an object and thus located on the heap. All objects have information concerning the GC state of the object. The templates also describe instances with a reference location description and an object size.

Figure 2.10 describes graphically the outlook of the template head in the IVM system. The reference location description is explained in detail in Section 2.2.4.

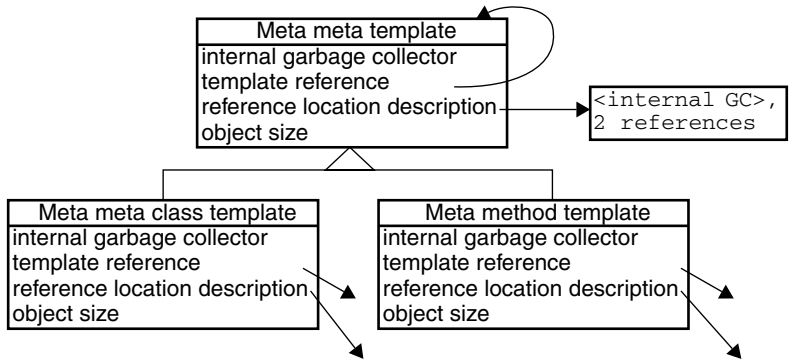


Figure 2.10 *The templates describe its children. The meta meta template is also its own subtemplate.*

Templates may be extended to hold more information common to their children. The meta method activation template and the meta meta class template have the same outlook, but they describe different children. Children to the meta meta class template have an extra virtual method table. In Java, it is possible to call methods in classes. The virtual method table contains the methods that are accessible from every Java class object. Those methods are described in the Java class named Class.

Templates for classes, arrays, interfaces, and primitive types
The objects of templates are the instances of the classes that the templates represent. Information common to all objects of a class is collected in the corresponding class template. The information in a template for a Java object is described by the following fields (the template head is excluded):

- **Access flags** — the flags describe the access modifiers and property modifiers of the class (see [JVM99], Table 4.1, p. 96).
- **Superclass**: The reference refers to the Java superclass of this object.
- **Virtual method table** — the table contains all the virtual methods in the class. The methods are represented as activation templates.
- **Static method table** — the table contains the static methods declared in the class. The methods are represented as activation templates.
- **Constant value table** — the table contains the value constants declared in the class.
- **Constant reference table** — the table contains the references constants declared in the class.
- **Interface table** — the table contains the interfaces and the corresponding virtual method array, implemented by this class.

- **Fields** — the content describes fields declared by the class. The following information is stored: name of the field, the descriptor of the field, the offset to the field, the access flags of the field, and the type of the field. The name and the descriptor are stored as indices to the internal symbol table that is explained in Section 2.3.5.
- **Class references** — the references utilised in the method are stored in this array. A reference entry contains indices to its class, name, and descriptor. Class indices are offsets in the class template table and the class symbol table. The name and descriptor indices are offsets in the symbol table. See Section 2.2.5 for more information about the internal tables.
- **Class index** — the index shows which location in the class symbol table that contains the class template table representing this class.
- **Debug information** — the extra information about the class is stored into the debug information table.

Activation templates
The method templates describe the methods in the IVM. As methods are called, their invocations are stored as objects on the heap with a reference to their template. The activation template is depicted in Figure 2.11 and it contains the following information:

- **Access flags** — the flags describes the method access modifiers and properties, see [JVM99], table 4.5, p. 115 for more information.
- **Class template reference** — the reference refers to the class implementing this method.
- **Name and descriptor indices** — the indices describes the location of the symbol of the name and descriptor of this class in the class symbol table.
- **Number of reference and value arguments** — the number of arguments shows how many arguments are transferred to the new *activation* or *frame*.
- **Start of local variable area and stack** — the indices show where the local variable area and the stack start in the frame. Since the local variables and the stack are split into reference and value parts, there are four indices to locate the internals of the frame.
- **Exception table** — the exception table contains indices to the exceptions and their ranges in which the exception can be caught. A handler index indicates where in the bytecode to proceed if the exception is caught.
- **Code reference** — the code reference refers to the bytecode array.

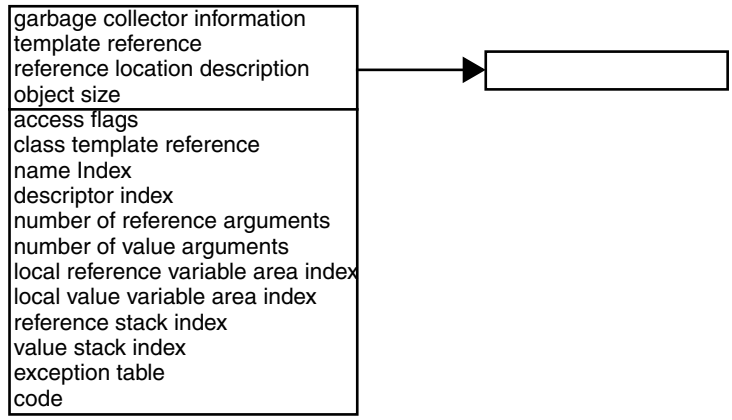


Figure 2.11 *The activation template contains information that is common to all method calls of the method.*

2.3.3 Inheritance structure

The hierarchical inheritance structure represents the type of the objects of the class, and the contents of objects as designed by the programmer. Attributes and methods in an object consist of the collection of inherited attributes and methods plus those implemented by the class. Figure 2.12 depicts an inheritance situation.

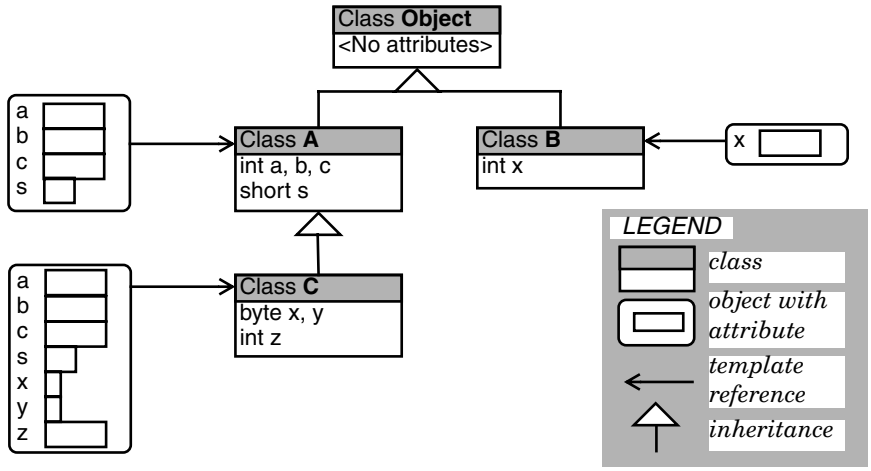


Figure 2.12 *The inheritance structure describes the type and content of objects.*

The template structures are utilised to locate the direct superclass of an object. However, the superclass of the object's direct superclass is not found via the template reference of that class. Instead, the class reference in the class template is utilised to find the superclass of the object. The template reference in the class template leads to the meta template of the template. The distinction between the class and its template is due to the

JVM specification. It says that all classes are instances of the class `Class`.

All the templates that are possible to inherit have to implement a superclass reference to support the class inheritance structure.

2.3.4 Java class structure

The Java class structure shows a template structure as described to the programmer by the API. The Java description of classes and objects is found in the class `Class` defined in the Java API, e.g. Java 2 Standard Edition API [J2SE] and Java 2 Micro Edition API [J2ME].

The Java class structure differs from the IVM template structure and from the class inheritance structure. It does not describe any garbage collecting information and other implementation specific details. Neither does it collect all common information in children. It only enables everything written by the programmer for the Java program. The class `Class` supports symbolic field and method access. Figure 2.13 shows how the Java classes are related to the class `Class`. This class is important in a JVM implementation even though it belongs to the Java API. Inside the class `Object`, there is a method returning the class of the object. The method is inherited into every Java object in the system. The objects and classes in Figure 2.13 are related to each other according to the Java API. The instances of classes refer to their classes. The internal template structure and the Java class structure utilise the same template reference.

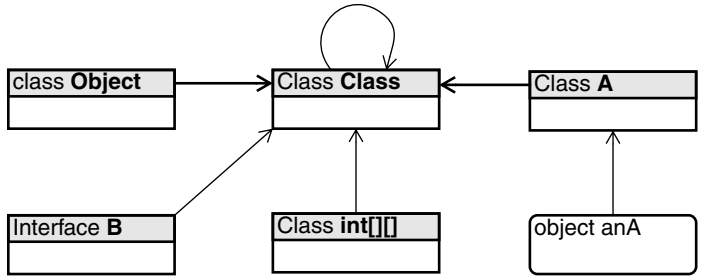


Figure 2.13 *The Java class structure describes relations of objects and classes.*

2.3.5 Internal memory and data structures

This section describes the memory organisation and internal data structures that support the IVM. Memory organisation is primarily a garbage collection design issue. The GC algorithms decide the outlook of the references, objects, and heap. The internal data structures support the IVM during runtime.

To decrease memory utilisation, symbols in classfiles are reused. They are collected in a global symbol table. Every reference to a symbol is represented by an index in the symbol table. Class symbols and class tem-

plates are referred from a global class symbol table and a global class table, respectively. The symbol tables are utilised during class loading and by introspection. If introspection is removed from the API, the symbol tables do not need to reside in the runtime environment of the interpreter. The converter, on the other hand, requires the symbols to resolve symbolic links in classfiles during class loading.

The memory in the IVM is concentrated to the heap. Other memory areas have been transferred to the heap in order to simplify the memory organisation of the IVM. The cost of simplicity is performance loss. The only memory area outside the heap is the C stack that the IVM utilises during execution.

The organisation of the heap is dependent on the garbage collector algorithm. The GC algorithms implemented in the IVM are batch-copy, compacting incremental mark-and-sweep, and a compacting generational incremental mark-and-sweep. The algorithms will be touched briefly upon here, but more information about them can be found in "Garbage Collection", a book written by Richard Jones and Rafael Lins ([JL96]). The memory structure and the outlook of references of the algorithms are described next. All GCs utilise a root set containing references to live objects.

A batch-copy algorithm divides the memory into two areas of the same size. Allocation is performed in one area until it is filled. Then the program execution is abruptly halted while all live objects are selected and transferred to the other area. Dead objects are left in the old area. References are direct pointers to objects. During the *flip*, all the live references are updated to point to the new location of the object. Even though this algorithm induces little overhead, the unused memory area conflicts with the restricted memory of embedded systems.

The compact and incremental mark-and-sweep algorithm utilises one single memory area. The GC compacts objects inside the heap to avoid internal fragmentation. Every reference points at an internal object table where all objects in the heap are referred. When an object is moved onto the heap, it is only necessary to update the object table since all references to that object go through the corresponding object table entry. Every object is fitted with a handle that locates its entry in the object table. The memory state of the object is also noted inside every object.

The compact incremental generational mark-and-sweep algorithm combines the two algorithms mentioned above in an attempt to gain from the advantages. It has a small and fast batch-copy area, and objects surviving one flip are placed in a compacted heap that is updated with long intervals.

The disadvantage of placing the JVM stacks on the heap is performance loss. Compared to the stack solution due, extra indirection is introduced and extra overhead decreases performance. Another disadvantage is introduction of memory overhead in the frames. However, the stack solution requires beforehand determination of the stack sizes that could result in reserved memory that isn't utilised. Even if the maximum stack size could be determined, it is not probable that all stacks are utilised to the fullest at all times. The heap solutions do not suffer from these memory problems.

2.4 Split machine

The split machine executes the class loader and the interpreter on different machines connected via a network. Classes and objects are sent over the network to the interpreter, as they are needed. The size of the interpreter and the heap of the interpreter are reduced significantly without the class loader. Since the memory is not shared between the interpreter and the class loader, the classes loaded to the interpreter have to be linked into the runtime system of the interpreter. The class-loading module in the interpreter is replaced with a linker.

The split machine could be utilised in distributed systems consisting of small embedded nodes and a more powerful server, e.g. SCADA [SCADA]. An example of a system suitable for the split machine is depicted in Figure 2.14. The nodes contain embedded computers with an interpreter each. A supervisor computer keeps track on the state of the different nodes.

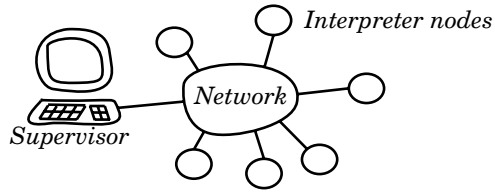


Figure 2.14 *The split machine is suited for systems with a more powerful computer, the supervisor, connected to interpreter nodes. The supervisor contains the class loader and prepares classes to be sent to the nodes.*

Another benefit of the split solution is that only parts of a class have to be within the memory of the interpreter. If a method is called without being loaded, it is requested from the supervisor. Other threads could continue to execute during the loading of the method, reducing the memory requirement even more. This approach supports real-time systems as well. A typical real-time program often executes a loop. The necessary methods and data structures to execute the loop — the *working set* — could be loaded into the interpreter's memory, while the rest could be thrown away. A simple way to find the main parts of the working set is to clean the memory as the control loop is reached. All the necessary methods and classes would then be requested from the supervisor instead. Figure 2.15 depicts the extensions of the original IVM structure.

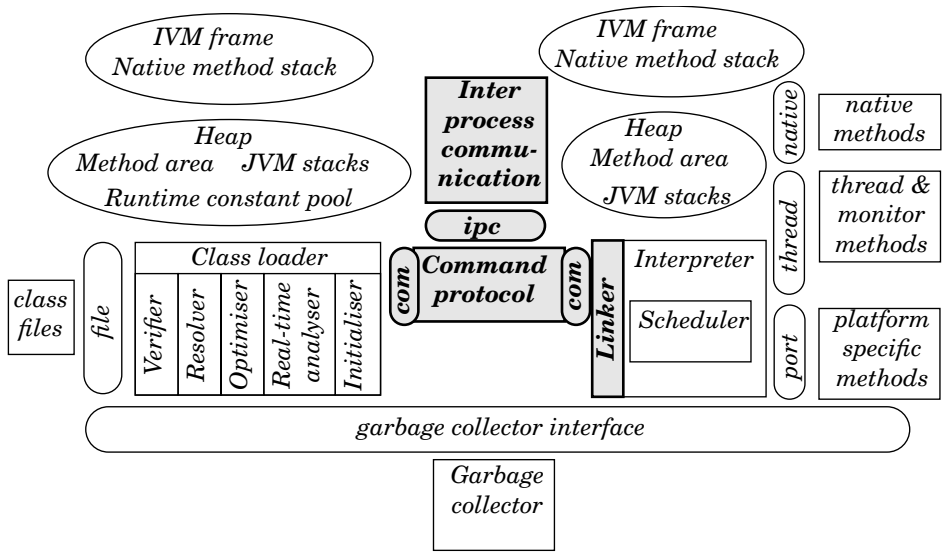


Figure 2.15 *The overall structure of the Java Virtual Machine shows the main modules and interfaces. The greyed interfaces and modules are related to the split machine. Two separate memory areas are needed for the class loader and the interpreter respectively. Objects and commands are sent via the com interface to the IVM. The command module utilises basic methods in an underlying physical interface, ipc, to send and receives bytes.*

The physical layer is encapsulated in the inter process communication, *ipc*, interface. It consists of methods to open and close a channel between the converter and the interpreter. It also defines how to read and write bytes to an opened channel. On top of the physical interface, the converter utilises a command interface to send instructions to the interpreter, for example, load a class, and start executing a program. This command interface is called *com* in the figure.

2.4.1 Interfaces and modules of the split machine

The interfaces and modules introduced by the split machine concern communication between the interpreter and the class loader. The interpreter is extended with a linker to incorporate the classes sent from the class loader into the runtime data structures of the interpreter. At the same time, the class loader is extended with an IVM control module that sends commands and objects to the IVM node. Figure 2.16 shows the communication layers:

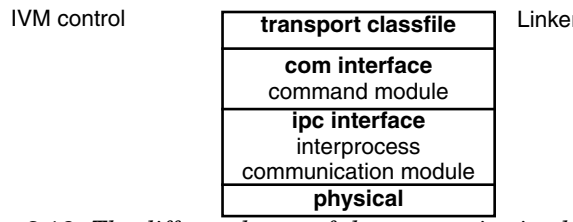


Figure 2.16 *The different layers of the communication between the IVM and the class loader are designed to support different implementations.*

The extra modules in the split IVM are as follows:

Inter process communication — a transport layer of bytes between communicates between the interpreter and the class loader.
Command protocol — defines commands and the representation of the *transport classfile*.
Ivm control — sends commands and objects to the IVM node.
Linker — implements the commands defined in command protocol.

The extra interfaces in the split IVM are:

Com — the command protocol transfers commands and objects to and from the IVM.
Ipc —the inter process communication encapsulates the physical transport layer. It contains methods to open and close channels, as well as methods to send and receive bytes.

The com interface describes object serialisation between the converter and the interpreter. It also describes control signals. The command module implements the com interface and the underlying physical transport layer is encapsulated in the ipc interface.

The linker module has to be added to the IVM. It converts the loaded objects into runtime objects, by resolving the pointer references to real references. The class loader sends and receives commands to the IVM via the IVM control module.

The transport classfile layer consists of the internal runtime data structures. However, the references have to be recalculated in the interpreter if a separate memory area is utilised.

Initialisation of the IVM node's internal data structures
Classes are sent in a transport representation. All references have to be recalculated in the IVM. It is essential to install the referred objects before reference recalculation is performed. The internal hierarchical template structure can also be sent to reduce the size of the interpreter.

2.4.2 Memory model

There are two alternative solutions to the split machine memory design. The class loader memory area should be shared amongst the nodes to save space on the supervisor computer. The other alternative is more memory consumptive where a separate heap is allocated for every node. The advantage of such a solution is to avoid irrelevant classes to reserve memory in the interpreter memory area. A combination of both ideas is also an alternative. The common classes defined in the API are the same for every interpreter node.

2.4.3 IVM references

There are two different ways to refer to an object and there are two different ways to refer to elements in an object. The object may be accessed directly by a pointer, or via a reference. The pointer is only utilised by the garbage collector. Java objects and internal data structures are referred to

via references. The reference implementation is dependent on the GC algorithm. It may be implemented as a pointer or as an indirect pointer, i.e. a pointer pointer. Elements inside an object are accessed as offsets from the object pointer. Array elements are to be accessed via indices, or as offsets.

For instance, an object accesses its class by directly referring to its template. A class is referred from the bytecode by an index to it. The index is utilised in the global class template table to access the class. Index references are also utilised by Java arrays. The global class template table could be accessed from Java, if a reference to it is provided to the program. Figure 2.17 describes the different reference types.

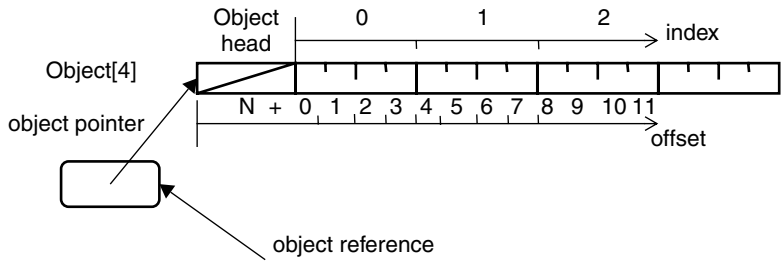


Figure 2.17 *In the IVM, the array would be referred to by an object reference. Only the garbage collector is aware of the direct pointers to the array. Elements in the array are accessed by an index. Internally, the elements of a general object are reached by offsets.*

The utilisation of indices is more secure than that of offsets. If the index reaches outside the array, an exception is thrown. Offsets do not have this feature.

2.5 Runtime

The runtime design of the IVM concerns the execution of bytecodes, method calls, threads, and context switches. The design is a specialisation of the JVM specification, which is too general for real-time embedded systems.

In Java, the runtime system is tightly coupled with the JVM, and the Java API. Programmers are allowed to work with threads and *locks*, i.e. monitors. Every object in Java has a monitor, and thread handling methods are implemented in the superclass of all classes, `Object`. All other classes inherit the class. Inside the JVM, there are monitor-handling bytecodes associated with the runtime system.

2.5.1 Method calls

The method call design of the IVM utilises the heap instead of stacks. The incurred performance loss is motivated by a simple and more predictable behaviour of the IVM. The advantages are avoidance of stack size calculation and better memory utilisation. The method call procedure is depicted

in Figure 2.18. As one method is called, a new method activation record,

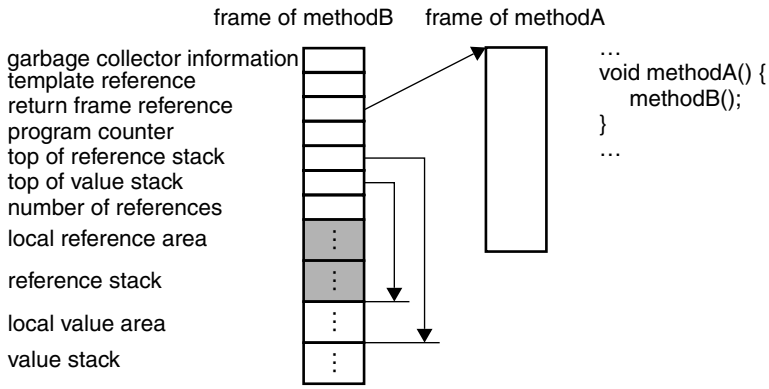


Figure 2.18 *The design of a method call in IVM describes the utilisation of the heap for all Java frames instead of one stack for every thread. The greyed area in the frame shows the references in the local variable area and the stack.*

i.e. *frame*, is created for the new method. This frame contains the local variable area and the stack for the called method, according to the information in the method template. It is linked with the frame of the caller in order to store the returning frame after the method is completed. The other contents of a frame are the program counter and the location of the top of the stacks. Both the stack and the local variable area are divided into a reference part and a value part that supports the garbage collector with the locations of the references in a simple way. The number of total references in the frame is also noted to support the GC.

The procedure of a method call is as follows:

1. Allocate the new frame.
2. Initialise internals of frame:
   - Set all references to null.
   - Set method template reference; return frame reference, program counter, and top of stacks.
   - Transfer arguments.
3. Transfer the execution point to the new frame.

The arguments are moved from the stack of the caller to the local variable area of the new frame. If the method is declared virtual, a reference of the object that receives the method call is also transferred as an "invisible" argument.

As the IVM is started, the specified main-class must contain a method that is named `main` and declared public and static. The IVM automatically creates the frame of the main-method, and starts execution of the main-method.

2.5.2 Java runtime

Java enables access to the runtime system via the language itself as wall as via the standard API. The main-method describes the first method to execute in a thread. It does not differ from other threads in other ways.

New threads can be spawned from classes inheriting the `Thread` class or implementing the `Runnable` interface, which is then given to a Thread constructor. The class `Thread` specifies operations for voluntary rescheduling, i.e. sleeping a period of time, and yielding the processor resource to other threads. Preemption occurs when the time slice of the currently active thread has expired, or when higher priority threads are activated. Every thread has a priority to indicate its importance. See [JLS00] for more details on the behaviour of threads.

The protection of critical regions and synchronisation is described as follows: every object has a lock associated with it and that lock can be acquired and released through the use of methods and statements declared `synchronized`. Since the lock is implemented as a monitor there are standard monitor operations associated with it. They are located in the `Object` class. It is possible for a thread inside a monitor to release it and wait for a condition to occur. As the monitor is released, other threads can enter it, change conditions, and notify one or all of the waiting threads. The JVM specifies the monitor behaviour in [JVM99].

Differences from general monitors are that the Java monitors only have one condition variable, and that the Java monitors are incorporated into the JVM. More condition variables allow threads to wait for different conditions to occur in the monitor. It is also common to treat the monitors as objects in an object oriented system, e.g. in BETA ([KMMN91], Section 12). The behaviour of monitors in Java is specified in the Java Virtual Machine Specification ([JVM99] Section 8, “Threads and Locks”). The specification does not describe the monitor or the scheduler behaviour exactly. The following citation is from [JLS00], 10.6 Thread Scheduling, pp. 248-249:

> *Exactly when a preemption can occur depends on the virtual machine you have. There are no guarantees, only a general expectation that preference is typically given to running higher priority threads. … You can make no assumptions about the order in which locks are granted to threads, nor the order in which waiting threads will received notifications – these are all system dependent.*

In short, the Java Specification specifies the contents of the JVM runtime handling while leaving the details to the JVM implementation.

Other common synchronisation mechanisms are semaphores and event handling. They are omitted in the Java specification.

2.5.3 JVM runtime

Even though the specification does not state the exact behaviour of the JVM runtime system, many tasks are thoroughly covered. They can be collected into the following list:

- Threads
- Locks
- Preemption
- Priorities

- Runtime API, e.g. `Thread` and `Object`

The runtime API is the interface for the programmer to the runtime system, and the scheduler. Exactly how the scheduler is implemented varies greatly. In some systems, the scheduler is implemented by a thread; in other systems, the scheduling work is distributed throughout the program. An example of the active thread organisation within the JVM is shown in Figure 2.19. Active threads are placed in different priority ready queues. When preemption occurs, the running thread is placed last in its priority list in a round-robin manner. Threads that are not in a ready queue are inactive, or blocked. Blocking can occur, for example, when a thread waits in a condition queue of a monitor. Sleeping threads are placed in a separate sleeping queue. An example of threads during runtime is described in Figure 2.19.

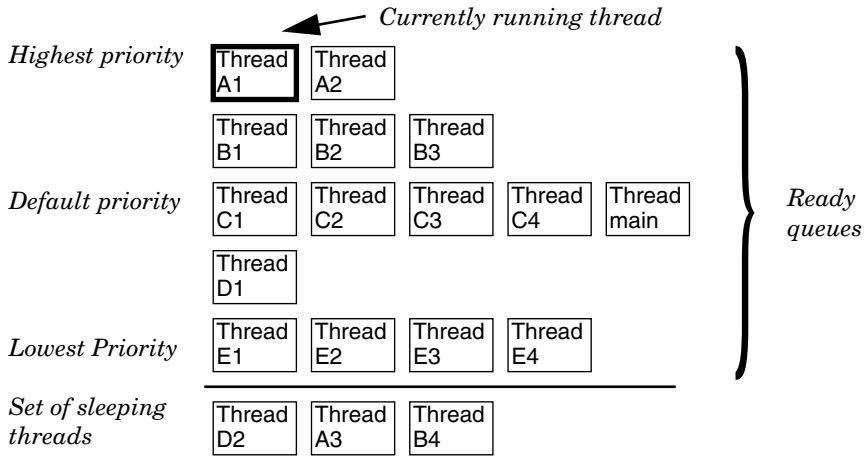
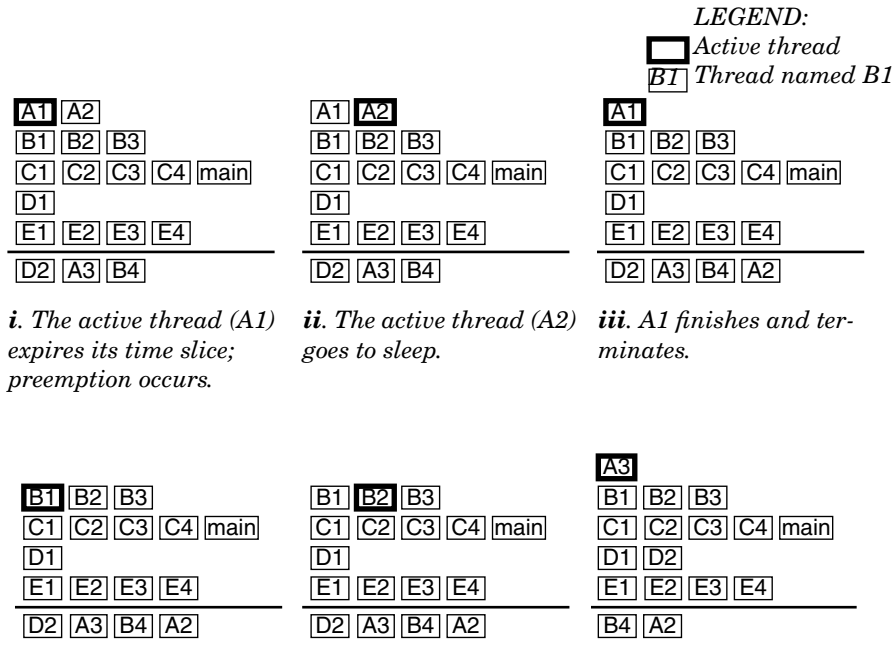


Figure 2.19 *Ready queues for different priorities help the scheduler to keep track on which thread it has to execute. Sleeping threads are woken as their sleeping time expires and re-inserted into their priority queue. This system contains 15 threads that are ready to run and 3 sleeping threads.*

The pictures in Figure 2.20 show the workings of the scheduler as preemption occurs, i.e. when the time slice of the active thread expires, when the active sleeps, and when a sleeping thread is woken.



i. *The active thread (A1) expires its time slice; preemption occurs.*

ii. *The active thread (A2) goes to sleep.*

iii. *A1 finishes and terminates.*

iv. *B1 yields and transfers the processor resource to the next thread in line.*

v. *During the execution of thread B2, two threads D2 and then A3 awake. A3 resumes execution.*

vi. *The time slice for A3 expires. The scheduler transfers the execution to A3 due to priority.*

Figure 2.20 *The ordinary workings of the scheduler consist of preemption due to expired time slice, and voluntary rescheduling, i.e. yielding and sleeping. When a thread terminates, rescheduling occurs to the next thread in line.*

Java locks

A typical layout of a Java monitor, called *lock*, is shown in Figure 2.21. It consists of a waiting queue where threads are lined up if the monitor is occupied. The event queue of the monitor contains threads that wait to be notified, probably after some change of a condition inside the monitor.

Threads that are located in the monitor queues or in the sleeping queue are blocked. Only threads in the ready queue are allowed to execute. A thread cannot reside in different queues at the same time.

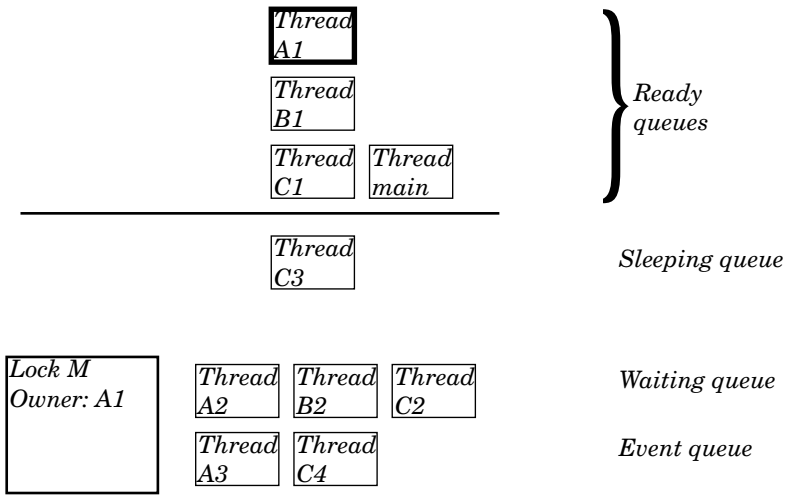
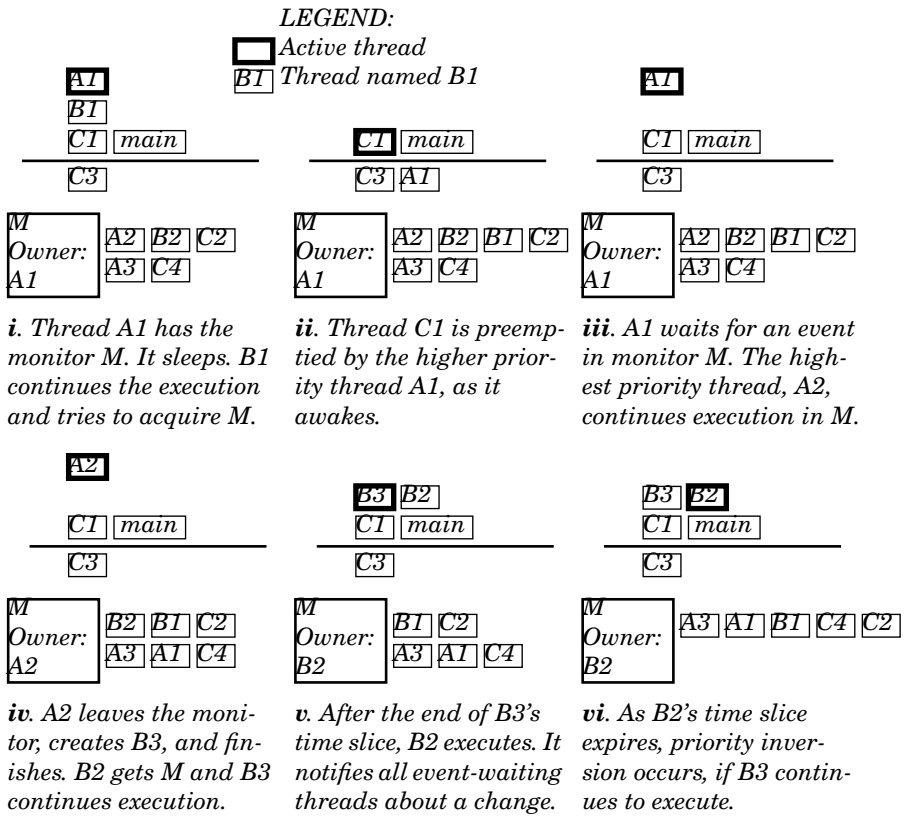


Figure 2.21 *The workings of a lock can be described with two queues associated with it. They are the waiting queue, where the active thread is placed when trying to acquire an occupied lock. The other queue is the event queue. As the thread that holds the lock decides to wait for a condition to change, it can wait in the event queue for this to happen. Other threads may then change the condition and notify the threads in the waiting queue. The name of the threads indicates their priority: A is highest priority and D is lowest priority. The thread that executes* `main` *has priority C.*

In Java, it is specified that there is a lock associated with every object. The workings of the monitor methods are described in Figure 2.22. Inside the monitor, a thread can wait for a condition to occur, or notify waiting threads of condition changes.



i. *Thread A1 has the monitor M. It sleeps. B1 continues the execution and tries to acquire M.*

ii. *Thread C1 is preempted by the higher priority thread A1, as it awakes.*

iii. *A1 waits for an event in monitor M. The highest priority thread, A2, continues execution in M.*

iv. *A2 leaves the monitor, creates B3, and finishes. B2 gets M and B3 continues execution.*

v. *After the end of B3's time slice, B2 executes. It notifies all event-waiting threads about a change.*

vi. *As B2's time slice expires, priority inversion occurs, if B3 continues to execute.*

Figure 2.22 *The workings of the monitor M are shown graphically. However, the exact implementation may vary from system to system. In pictures* ***v-vi*** *the notified threads are sorted into the waiting queue. According to the Java specification, this sorting procedure may not be taken for granted. It is implementation dependent. The handling of the* ***vi-****priority inversion situation is also implementation dependent. It is not reasonable to let lower priority thread block a higher priority thread's execution.*

Java does not explicitly specify the exact behaviour of the monitor operations. For example, the priority levels may not be completely valid because the behaviour of the monitor is implementation specific. In some systems it is feasible to let every thread, even those with lower priority, execute occasionally, in order to prevent starvation. In hard real-time systems, however, the priorities are typically followed stricter.

Traditionally, an implementation of a monitor works as follows with priorities:

- The waiting queue of a monitor is sorted due to priority. It is feasible that threads with higher priority acquire the monitor before lower priority threads do, even though the threads with lower priority have to wait longer. An example of this procedure is described in Figure 2.22.i.
- The monitor event queue is sorted due to priority. If one waiting thread is notified it is feasible to wake the thread with the highest

priority. If more thread share the same priority, the thread that has waited for the longest time will be awaken.

- As all threads in the event queue are notified simultaneously, they are sorted into the waiting queue according to their priorities. Threads from the event queue are placed before other threads with the same priority in the waiting queue. They had to hold the lock before they were able to wait for an event. See this transition in Figure 2.22.v.
- It is necessary to implement a priority inheritance protocol, for example, in hard real-time applications, in order to avoid priority inversion. An example of possible priority inversion is found in Figure 2.22.vi, where the B1 thread blocks higher priority threads. B1 does not have the lock M.

The Java runtime access

The Java J2SE API [J2SE] covers the workings of the scheduler in the classes Object and Thread. The Java language supports handling of monitors through the synchronised statement and the synchronised method declaration. The classes contain the following methods related to workings described above:

public class ***Object***	
`public final void` **`notify`**`()`	Wake up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. A thread waits on an object's monitor by calling one of the `wait` methods.
`public final void` **`notifyAll`**`()`	Wake up all threads that are waiting on this object's monitor. A thread waits on an object's monitor by calling one of the `wait` methods.
`public final void` **`wait`**`()` **`wait`**`(long timeout)` **`wait`**`(long to, int ns)` `throws InterruptedException`	Cause current thread to wait until either another thread invokes the `notify` method or the `notifyAll` method for this object, or the specified amount of time has elapsed, or the thread is interrupted by another thread.
*`public class` **`Thread`** `extends Object implements Runnable`:*	
`public` **`Thread`**`()`	Allocate a new Thread object. Other variants of the Thread constructor take arguments such as the name, `Runnable` interface, or the `ThreadGroup` the thread belongs to.
`int` **`getPriority`**`()`	Return this thread's priority.
`void` **`setPriority`**`(int prio)`	Change the priority of this thread.
`void` **`interrupt`**`()`	Interrupt this thread.
`void` **`run`**`()`	If this thread was constructed using a separate `Runnable` run object, then that `Runnable` object's run method is called; otherwise, this method does nothing and returns.

Table 2.1 *The classes in the J2SE API that relates to thread handling are* `Object` *and* `Thread`.

public class ***Object***	
`static void` **`sleep`**`(long ms)` **`sleep`**`(long ms,int ns)`	Cause the active thread to sleep (temporarily cease execution) for the specified number of milliseconds.
`void` **`start`**`()`	Cause this thread to begin execution; the Java Virtual Machine calls the run method of this thread.
`static void` **`yield`**`()`	Cause the active thread object to temporarily pause and allow other threads to execute.

Table 2.1 *The classes in the J2SE API that relates to thread handling are* `Object` *and* `Thread`.

Voluntary rescheduling may occur in the following methods in the Java J2SE API: `wait`, `setPriority`, `sleep`, `start`, and `yield`. The `setPriority`-method may lower the priority of the active thread below other ready threads. Execution continues among the other threads with higher priority. The `start`-method may initiate a higher priority thread, which should resume the execution.

Java Specification [JLS00] states nothing about the implementation issues like time slicing, priority inheritance protocols, periodic threads, and semaphores. Preemption, i.e. time slicing, and periodic threads are especially important for hard real-time scheduling. It may be implemented in different ways. See Section 2.5.4 for alternative implementations. A real-time adapted API, similar to the Java J2SE API, can be found in [Big98], where a semaphore API is also specified.

2.5.4 Preemption models

The procedure of preemption is crucial to real-time analysis techniques. The real-time systems aimed at in the IVM contain many threads executing their code repeatedly within specified time limits. Deadlines may vary between the threads. It is the scheduler that decides which thread that will execute after a context switch, in contrast to voluntary context switches where the context switch decisions are transferred from the scheduler into the application program, i.e. programmer. The scheduler cannot guarantee that deadlines are met.

There are many ways to implement preemption. Table 2.2 lists a few implementations and some systems utilising the techniques, along with an estimated minimum interval of continuous execution without preemption (in number of bytecodes). The order of the WCET is also presented.

Preemption	*Preemption interval estimation*	*WCET*	*Comments*
Insertion of extra preemption bytecodes.	< maximum interval	Maximum time of most time-consuming control flow path between preemption points.	An analysing tool could suggest insertion of preemption points.
Between source code lines.	~2–10 bytecodes	Longest “one-liner”.	Implemented in Lund Simula [SIM89].

Table 2.2 *Preemption is crucial in real-time systems. The table lists some alternatives of where preemption points can be inserted into the code.*

Preemption	*Preemption interval estimation*	*WCET*	*Comments*
Before every memory allocation (objects or activation frames)	~1–100 bytecodes	Most time-consuming control flow path without memory allocation.	Implemented in Beta [KMMN91].
Before method entrances and backward jumps	~10–50 bytecodes	Most time-consuming control flow path without method calls or backward jumps	Implemented in 1131-1.
After the execution of a number of bytecodes.	= maximum interval	Execution time of maximum bytecode count.	Instruction counting introduces noticeable runtime overhead.
Preemption (interruptions from surrounding system)	time interval between interruptions	Time interval + context switch.	This procedure does not require prior code analysis.

Table 2.2 *Preemption is crucial in real-time systems. The table lists some alternatives of where preemption points can be inserted into the code.*

Preemption from the surrounding system is an attractive option from the programmer's view. Little extra analysis is necessary to calculate the time of a context switch. The complexity of the system, however, increases; preemption can only occur at safe positions in the code where the GC can supervise all references in the system. The GC must be informed of references stored in processor registers. With preemption from the surrounding system, the context switch may be time consuming compared to the other variants. All the registers in the processor must be stored in the context of the thread. Other preemption implementations, e.g. preemption points, restricts the context switches to well-defined positions in the code, where only the necessary registers have to be stored in the context of the thread.

The IVM is restricted to check for a pending preemption after every bytecode. The longest interval between preemptions is set by the most time-consuming bytecode. Native methods are executed as one bytecode. It is possible for the programmer to insert rescheduling checkpoints in the native code in order to decrease the interval between preemptions.

2.5.5 Alternative runtime design

There are many different flavours of runtime design. Some applications require specialised treatment, while others are more machine independent. This section deals with basic design issues that are relevant in some systems and applications.

In an attempt to simplify the implementation of threads and to decrease the memory overhead for the scheduler, it is possible to implement the scheduler in Java. Everything except crucial native methods could be written in Java. The minimal native functionality is to disable and enable interruptions, to avoid preemption during critical regions. It is possible to build a complete runtime system on top of coroutine primitives (call, detach and thread initialisation). Another runtime implementation could describe all the thread handling procedures in native code. Native code inflexible but may increase performance.

The idea of thread handling written in Java is to simplify access and increase flexibility. The IVM is intended as a research project with unforeseen requirements. Flexible code should increase the availability of the code for future research projects. However, the cost is performance loss.

The implementation of a thread API could be influenced by the notion of coroutines. They are utilised in the Simula programming language. More information about coroutines may be found in [KM93], Section 25, “Simula runtime system overview”.

2.6 Preloaded classfiles

It is desirable to start a Java application fast. Many classes could be converted into the internal intermediate format prior to the execution. When an application is started, the preloaded classes have to be linked into the runtime system; conversion has already been performed. Typically, the API should be appropriate to convert before execution, and application classes should undergo conversion as usual. The preloading of classfiles steps outside the scope of real-time systems, but embedded systems may often require fast start-up behaviour. There are three levels of preloading:

1. No classes are preloaded. Everything is loaded and converted during runtime. This approach is time-consuming but flexible.
2. Some classes are preloaded, typically the API. The application itself is not preloaded. Flexibility is maintained reasonably well together with shorter start-up times.
3. Every class is preloaded. This inflexible approach speeds up the activation of the application.

As the classfiles are preloaded to improve start-up time, it is necessary to specify with which classes the preloaded classes are loaded. The same symbols and symbol indices must be utilised by all the preloaded classes in the working set.

The preloaded classes are incorporated into the runtime system by a linker. The purposes of the linker is to load the preloaded classes, allocate memory for them, and set the absolute references in them to their correct values. The linker is similar to the interpreter linker in the two-process variant.

The problems associated to the linker are circular graphs and how to represent the references in a manner that is independent of absolute positions. The circular graph problem could be circumvented with a two-pass procedure. All preloaded objects are allocated and loaded in the first pass. The references, on the other hand, are resolved during the second pass. The exact details of the solutions to these problems are directed in section 4.3, "Loading converted classfiles".

A simple and effective fashion to preload classes is to load all classes that should be preloaded and store the heap as a file. The heap image could then be copied into the heap before the application is loaded. The heap and the loaded heap image must be located on the same physical

memory addresses so that references refer to the same addresses as during the creation of the heap image.

Chapter 3

IVM runtime

The runtime behaviour of the IVM is focused on hard real-time applications. The complex analyses to guarantee timing criteria are achieved through a simple fundamental design of the runtime system that insists of program execution, thread handling, exception handling, and termination. Furthermore, the garbage collector is also an essential part of the execution system.

The subsections describe the runtime data structures of the IVM with special regard to the prerequisites of limited memory and hard real-time, and runtime procedures. The main contributions are the solutions to the prerequisites, i.e. the split machine and the split stack. Functionality and simplicity have dominated over performance during the design of the runtime system. Our aim is not to optimise an already existing program, but to prove that object-oriented programs may function in embedded hard real-time systems.

3.1 Fundamental runtime data structures

The fundamental runtime data structures in the IVM consist of a *template hierarchy* and a *frame stack* for every thread, i.e. a stack of method activations. The purpose of the templates is to save memory by gathering information in one location, instead of copying the data into the children of the template. Maintenance is also simplified if information is expressed in one location. The template hierarchy consists of internal nodes, *templates*. A child of a template can be a template or an instance. For example, all class templates are internal nodes that describe their instances, and the class template is described by the meta class template, etc. The frame stack holds all of the frames of the method invocations that are active at the current execution point.

The descriptions of the runtime data structures in the IVM are presented gradually, beginning with a simple Java example. The template hierarchy of the following program example is shown in Figure 3.1.

```
class SuperHero {
   private int strength;
   public SuperHero(int strength) { this.strength = strength; }
   public void useSuperPower() {…}
   public static void main(String[] argv) {
      SuperHero virtualMan = new SuperHero(10);
      virtualMan.useSuperPower();
   }
}
```

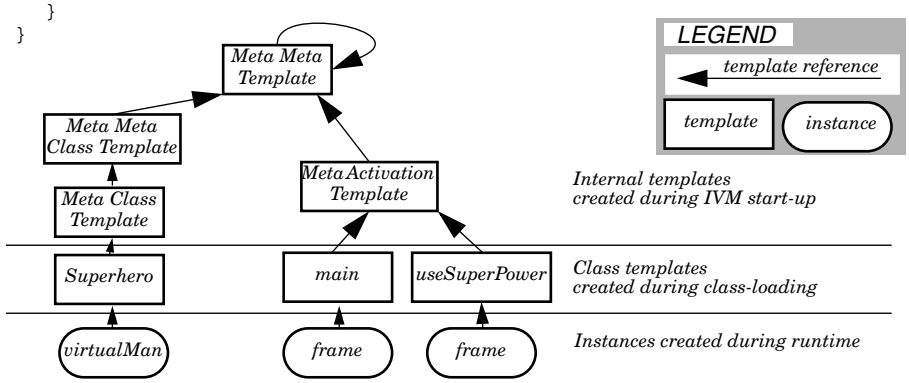


Figure 3.1 *The template hierarchy of the program example shows how the instances and the templates are connected.*

All objects, i.e. templates, instances, and frames, are located on the heap and they are connected to the template hierarchy. The top node, the meta meta template, describes itself as well as its children. To support the template hierarchy, all objects on the heap must have a template reference. Templates also contain the size of their children and garbage collector information describing the layout of their children.

The instance created in the program example of Figure 3.1 is shown in Figure 3.2 together with a template, the `SuperHero`-class. The `virtualMan`-object contains a template reference and an integer attribute, `strength`, which is set to the value 10. The `SuperHero`-class in the picture also contains the size of its instances. The size can vary from one platform to another. The `GCInfo`-field contains information for the GC about the layout of the instances of the class `SuperHero`.

This simplified description of objects omits the object head. However, that is described in Section 3.2.8 together with the `GCInfo`-field.

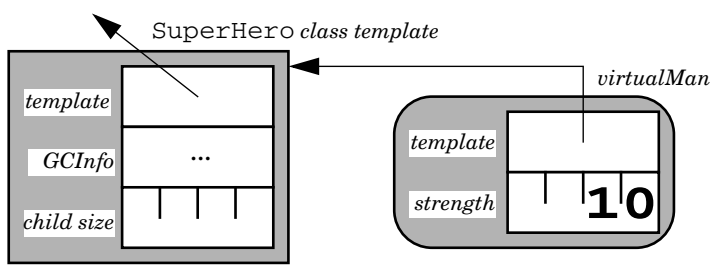


Figure 3.2 *The contents of the newly created virtualMan-instance are a template reference and a strength-attribute as declared in the class.*

Every method invocation results in an allocation of a frame where runtime information about the method invocation is stored — such as the values of local variables. When the method is completed, the previous method resumes execution. A reference to the *dynamic father*, i.e. the caller of the method, is stored in the frame.

At one location in the program example, the `main`-method calls the virtual method, `useSuperPower`, on the `virtualMan`-object. The frames resulting from the method call are depicted in Figure 3.3. In this simplified example, the frame consists only of a template reference, and a dynamic father. The dynamic father of the `main`-method is null to indicate that the execution (of that thread) is finished.

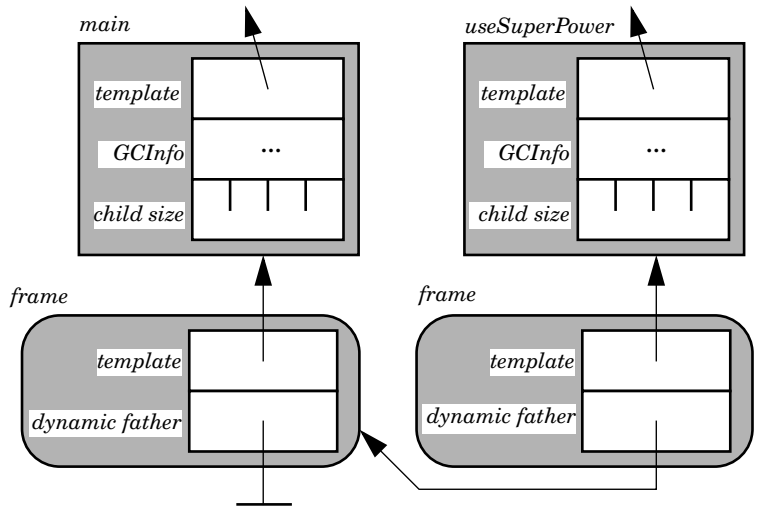


Figure 3.3 *The runtime situation in the figure shows two frames when the* `main`*-method has called the useSuperPower-method. The* `main`*-method does not have a dynamic father (null).*

3.2 IVM runtime system in detail

These subsections focus on the runtime mechanisms of the IVM, and especially on the techniques relevant for hard real-time embedded systems.

3.2.1 Interpreter

The main task of the interpreter is to execute bytecodes. Other essential functionality of the interpreter is managing signals, preferably context switches. The following pseudo code shows this procedure.

```
do {
   fetch an bytecode;
   if (operands) fetch operands;
   execute the action for the bytecode;
   if (signals) handle signals;
} while(there is more to do);
```

During signal handling, the machine may switch the executing thread. However, the interpreter will not notice the context switch. Execution will continue in the active thread as if no interruption had occurred.

Context switches are only performed after the execution of a bytecode, regardless of whether the program voluntarily requests the switch or the switch is triggered by an external signal (preemption). Voluntary context switches are instigated by blocking the thread, e.g. a call to `sleep`, `yield`, or `wait`. There are two different ways to signal for a preemptive context switch. The first way is to let the machine itself signal for a context switch. This may be instigated by, for example, a number of bytecodes that have been executed, or when specific bytecodes are reached. The second alternative is to let a hardware timer signal for the context switch. The latter alternative is often utilised in hard real-time applications. Different solutions to preemption are presented in Table 2.2.

Other signals than those just mentioned are also possible to send to the machine, e.g. command the machine to pause, or to send signals to the application. Signal handling is performed together with the context switch handling. The reason for delaying preemptive context switches until after the execution of a bytecode is to reach a well-defined location in the code where all the references reside in well-defined locations, i.e. in memory, and not within unmanaged processor registers or caches. The garbage collector requires that it must be able to reach all references when it is working.

3.2.2 Method calls

There are four different types of method calls: virtual, static, interface, and native. Synchronised methods extend the execution of the calls with monitor-handling procedures. They are described in the later part of this section.

All method calls, except the native method call, are performed by locating the method template, and creating a frame from that template. The method types differ in how the method template is located. Since the frames are allocated on the heap, no extra memory areas, stacks, are required for each thread. Performance decreases, however, in comparison to stack allocated frames. The incurred performance loss is motivated by a simple and more efficient memory utilisation. No stack sizes have to be calculated and allocated. A detailed study of the performance loss is described in Appendix C.

Frame layout

The contents of a frame are data utilised by the method during runtime, i.e. a program counter, a dynamic father, local variables, and a stack. The local variables and the stack have been split into a value part and a reference part to simplify the description of the reference locations for the gar-

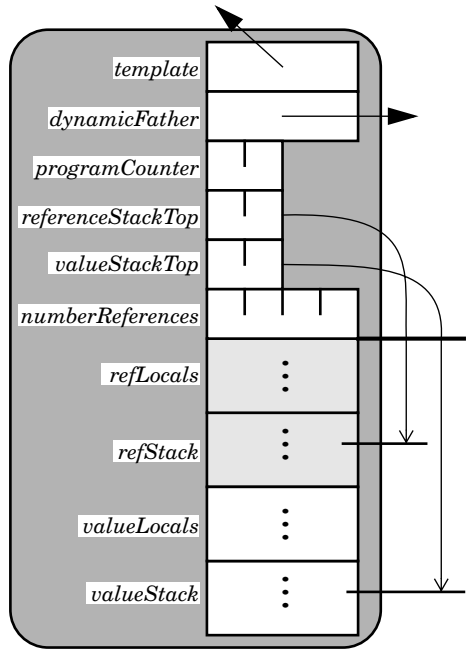


Figure 3.4 *The frame contains the runtime state of a method. The greyed area of the frame contains the references in the local variables and the stack.*

bage collector. The frame is described in Figure 3.4 and the elements in the frame are:

- **Program counter** – the currently executing bytecode offset into the method.
- **Dynamic father** – the caller of this method.
- **Stack tops** (value and reference) – offsets into the method indicating where the top of the stacks are located in the frame.
- **Number of references** – the number of references that follows.
- **Local variables** (value and reference) – variables that are declared in the method.
- **Stacks** (value and reference) – holds temporary values and references.

Method template layout

The method template describes the layout of its frames. Most values are related to the construction of the frame: offsets to the stacks and local variables, and the number of reference and value arguments declared by the method. The bytecodes of the method is stored in a `code`-array. Exceptions that this method can catch are noted in the `exceptions`-array. The name of the method is described as a name index into the symbol table. The descriptor of a method is a textual representation of the argument and return value types. The descriptor index is also an index into the symbol table. The semantics of the descriptor is given in [JVM99], Section 4.3.3. Access flags contain information about the declared method modifiers. See Appendix B for more information.

Figure 3.5 shows the contents of the frame template in relation to a frame.

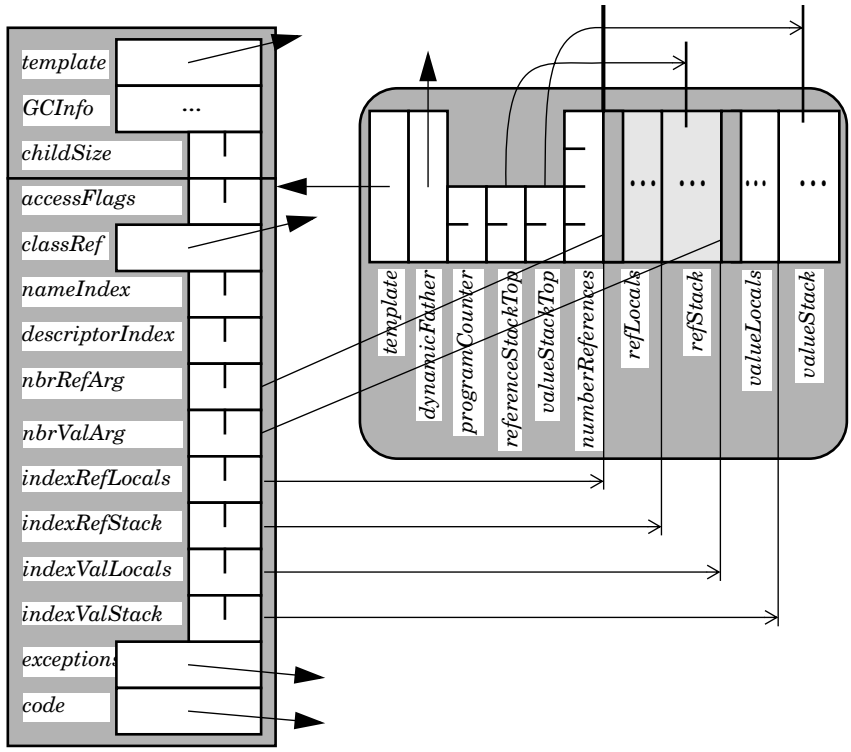


Figure 3.5 *The picture shows the frame template and a frame of the method.*

Method call details

As a method is called, a new frame is created for the new method. The stack sizes and number of local variables of the frame are described in its method template. The new frame is linked with the caller's frame by the dynamic father reference. The other contents of a frame are initialised into default values. The program counter is initialised to the start of the bytecodes, and locations of the stack tops are initialised to indices representing the bottom of the stacks. The number of total references in the frame is also noted to support the GC.

The procedure of a method call consists of the following steps:

1. Locate the method template.
2. Allocate the new frame.
3. Initialise the frame:
   - Set all references to null (performed during allocation).
   - Set template, dynamic father, program counter, and stack tops.
   - Transfer arguments.
4. Transfer the execution point to the new frame.

The arguments are moved from the stack of the caller to the local variable area of the new frame. If the method is declared virtual, a reference

of the object that receives the method call is also transferred as an "invisible" argument. The execution point is then transferred to the new frame. All of the method calls, except for the native method call, differ in how the method template is located. Native method calls are fundamentally differently structured.

Static Method Call

The static method call is equivalent to ordinary procedure calls. The static methods are collected inside the class template corresponding to the class where they are implemented. Operands of the static method call bytecode are the class index and the method index. The class is located by a class index (16 bits) in the class template table. The static method is found by the static method index (16 bits). Figure 3.6 extends the superhero program example to exemplify the method template location of a static method call. The static method call is depicted in Figure 3.7.

```
interface SuperPower { String superPowerName(); }

class SuperHero extends SuperPower {
  …
  public void useSuperPower() {…}
  public static int superHeroAmount() {…}
  String superPowerName() { return "Dimensional phasing";}

  public static void main(String[] argv) {
    …
    virtualMan.useSuperPower();             // virtual method call
    System.out.println(superHeroAmount());   // static method call
    String n=virtualMan.superPowerName());// interface method call
  }
}
```

Figure 3.6 *The example code contains a virtual method call, a static method call, and an interface method call in the* `main`*-method.*

```
java:     superHeroAmount();
bytecode:invokestatic <SuperHero index>16 <superHeroAmount>16
```

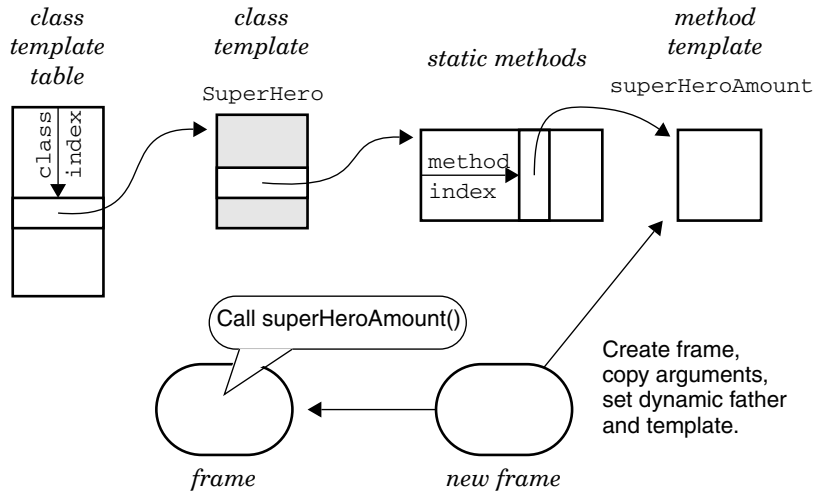


Figure 3.7 *The static method call is performed by locating the method template of the method, and then a new frame is allocated and initialised with arguments and a dynamic father to the caller.*

Virtual method call

The virtual method templates are reached via a virtual method table in the template of the object that receives the method call. The receiver is located on the stack. The index to the virtual method is stored as an operand of the bytecode. Figure 3.8 describes how the virtual method call in the program example is executed. The `useSuperPower`-method template is located and a frame is produced (see [KM93], Section 25.4.3, "Virtual binding", for a more detailed description of the design of the virtual method table).

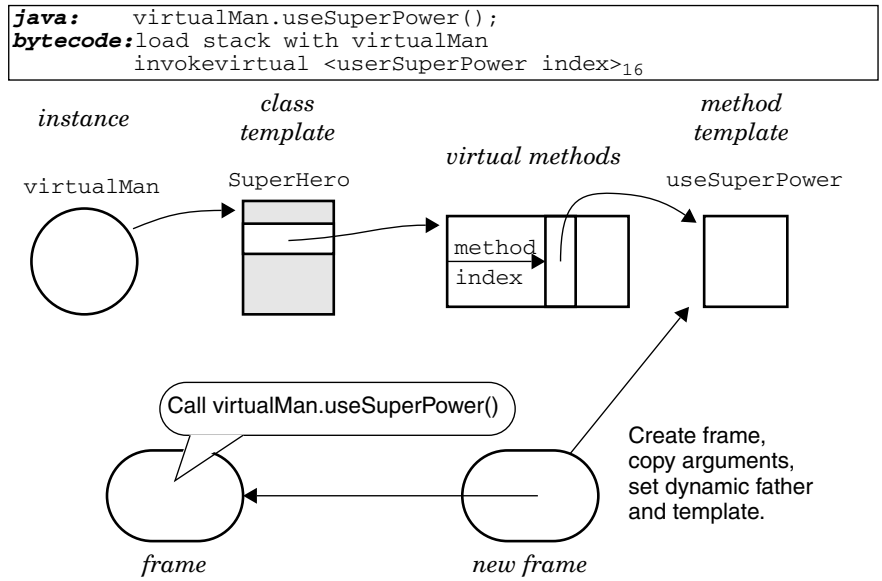


Figure 3.8 *The method template of the virtual method call,* `useSuperPower`*, is found via the object,* `virtualMan`*, which receives the method call.*

Interface method call

A class may implement many interfaces. All the interfaces are stored together with the implementations of the methods in the interfaces-array of the class. The receiver of the interface method searches through the interfaces-array in order to locate the correct interface and method template. All the interfaces of the superclasses are also included into the interfaces-array of the classes. Figure 3.9 illustrates the interface method template discovery.

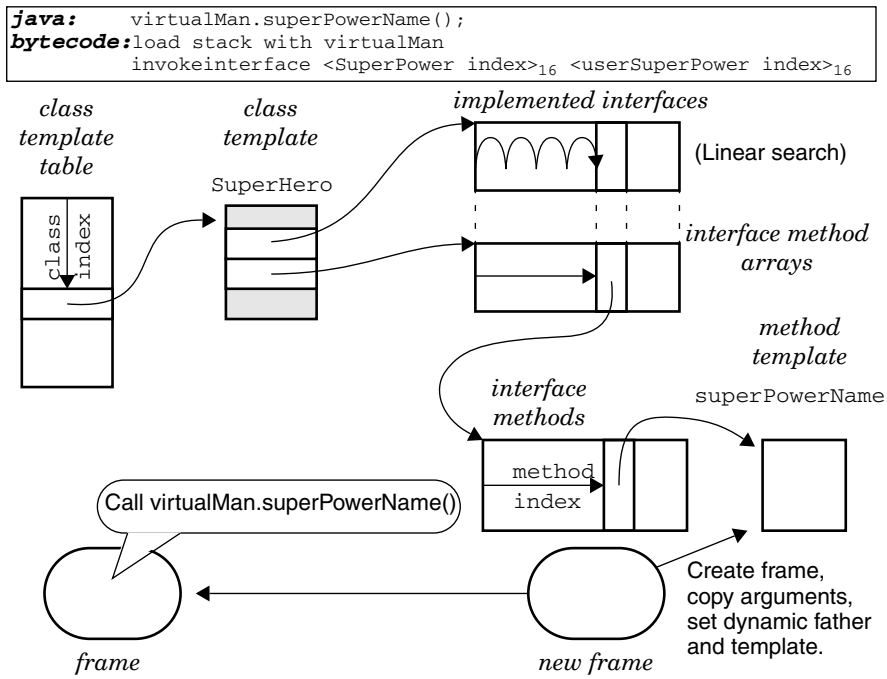


Figure 3.9 *Interface method templates are located through the interface and an offset to the method.*

Native method calls

Native method calls are declared in Java and written in C. They execute outside the scope of the Java Virtual Machine, but if they access the heap, the coding conventions for the machine (see [JNI99]) must be utilised to sustain correct functionality. We have developed a C-stub creation tool to support the programmer with an initial native method implementation according to the code conventions. Within the stub, arguments are popped from the caller's stack and delivered as arguments to the native method. References are registered in the garbage collector, and a default return value, if any, is declared.

The native methods store their frames on a stack that lies outside the supervision of the JVM. It is the same stack as the JVM utilises. The size of the stack and the worst-case execution time for the method have to be determined by the programmer.

No preemption is supported inside the native methods. The execution time of a native method counts as an execution time of a single bytecode during the WCET analysis. Time-consuming native methods can affect the analysis significantly, but the programmer can insert preemption points inside the native code to decrease the influence of native methods on the WCET analysis.

Synchronised methods

The synchronised methods are indicated by an access flag (see Appendix B) in the method template, to indicate that the method is a critical region

of code. The flag is examined before the method is executed. If the method is synchronised, the receiver's monitor, i.e. the lock, must be acquired before the method is executed. Synchronised static methods utilise the monitor contained in the class object.

Since only threads are able to utilise monitors, locks could be managed by threads and not by every object. This approach minimises the memory utilisation of locks by sustaining the impression that every object has a lock. Every thread could be fitted with a lock as they are created. This leads to another benefit. The monitor creation is, hopefully, outside the scope of the code that is relevant in real-time analyses and real-time applications do not have to include the monitor creation in the real-time loop. These thread-associated locks contain references to the locked objects, and how many times they are locked. Locked objects, on the other hand, refer to the lock of the thread that has locked them. As threads become inactive through a call to `wait`, the lock also refers to the waiting, inactive, thread. When other threads try to get the lock, they check if there is a thread that holds the lock and if that thread is inactive or active. If an inactive thread is holding the lock, another thread is free to take the lock. The waiting queue of the monitor is registered in the lock. More information about the memory efficient monitors can be found in [Blo00].

3.2.3 Runtime template hierarchy

The complete template hierarchy contains templates that describe all the objects on the heap, see Figure 3.10. In Java, three types of templates are

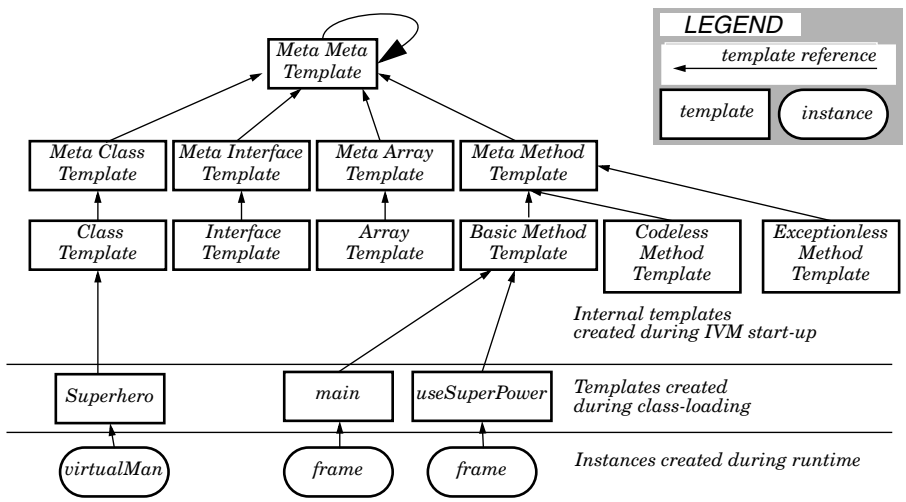


Figure 3.10 *The template hierarchy of the program example shows how the instances and the templates are connected.*

directly usable by the programmer: classes, interfaces, and arrays. The interfaces are simpler and smaller than ordinary classes, and take less memory. For example, they have no static methods. Arrays differ because they have propagated the object size from the template into the instances.

Arrays with the same dimension and type could utilise the same array template, regardless of their sizes. The array size is stored within the instance. If the size of an array would be described in the template, a new array template had to be created for every array size.

Method templates are divided into three types: ordinary methods, methods without code, and methods that do not catch exceptions. Methods that do not catch exception could be described with smaller method templates than ordinary methods because they do not have the need of a reference to an exception array. Codeless methods do not have any code reference either. The three different types of method templates are compared in Figure 3.11.

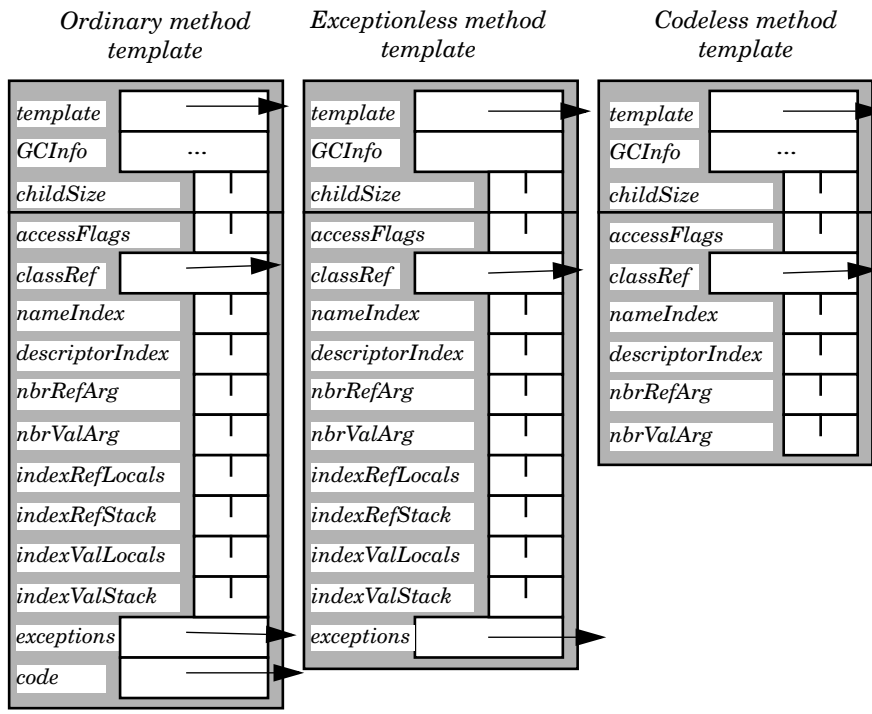


Figure 3.11 *The three types of method templates differ in size.*

3.2.4 Signal handling

After the execution of every bytecode, and during the execution of some long bytecodes, the machine checks if any external signal has occurred. Applications in Java may also explicitly check for signals. The machine handles signals concerning execution of an application: preemption, start, stop, termination, and step one bytecode. If a preemptive signal is pending, the scheduler performs a context switch. A graphical overview of the signal handling procedure is depicted in Figure 3.12.

Java applications are supported by a system method that forward system signals to the environment. Signals to Java applications are forwarded by a dispatcher to the designated thread. The dispatcher is a Java thread. If no dispatcher is present, the signals are dismissed by the machine. Typically, the signals to and from Java threads are asynchro-

nous method calls (see Figure 3.13). The identification of the thread is supplied inside the Java signal. An example of an asynchronous call from the machine to the surrounding system is exemplified in Figure 3.13.

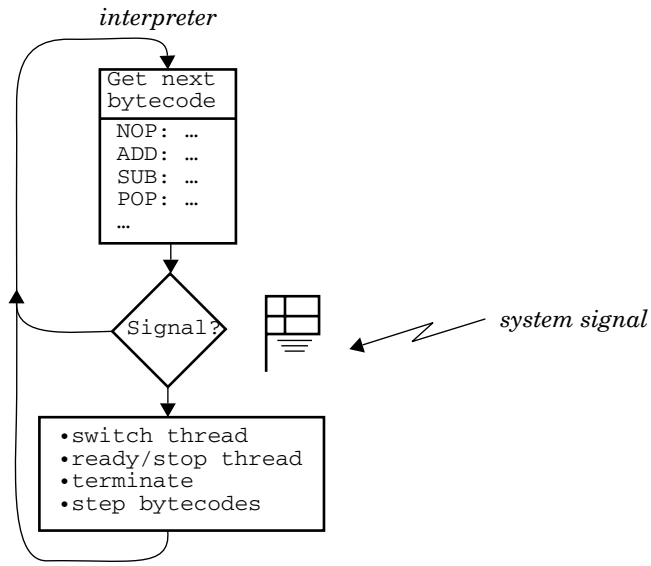


Figure 3.12 *After the execution of a bytecode, the machine checks whether the surrounding system has sent a signal to the machine or to a Java application.*

The signal check can often be implemented with a single machine code instruction. Preemption signals are set periodically by a timer interruption. The machine idles if there are no active threads available, and if there are threads that could possibly be reactivated. Otherwise, the application terminates. Signals to the machine while executing a native method are not handled until the native method is concluded. A native method affects the machine in the same way as the execution of a single bytecode.

The context switch procedure is dependent on the thread implementation. In the IVM, three different thread implementations have been implemented. They are described in detail in Section 3.2.7.

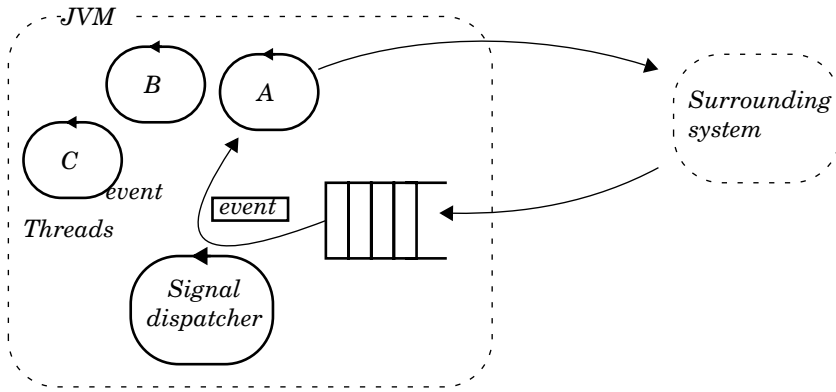


Figure 3.13 *Asynchronous method calls to the surrounding system contain information about the sender and arguments. The answer is converted to an event by the signal dispatcher, and sent to the sender.*

3.2.5 Context switch

The preemptive context switch transfers the execution point from a process to the scheduler. The scheduler determines the next process that continues execution, if there are any active processes. Otherwise, the scheduler idles. The basic process model is based on coroutines (see [WaDa71]). The word *process* is utilised to emphasize the coroutine aspect. *Threads* are utilised to express a similarity to the threads utilised in Java.

There are three methods associated with context switches: *initialise* a process, *call* a process, and *detach* a process. The scheduler *calls* processes that execute and the voluntary or preemptive context switch is performed by *detach*. Figure 3.14 visualises call, detach, and `initProcess`. Actually, every process in the system is modelled as a coroutine, even that of the scheduler.

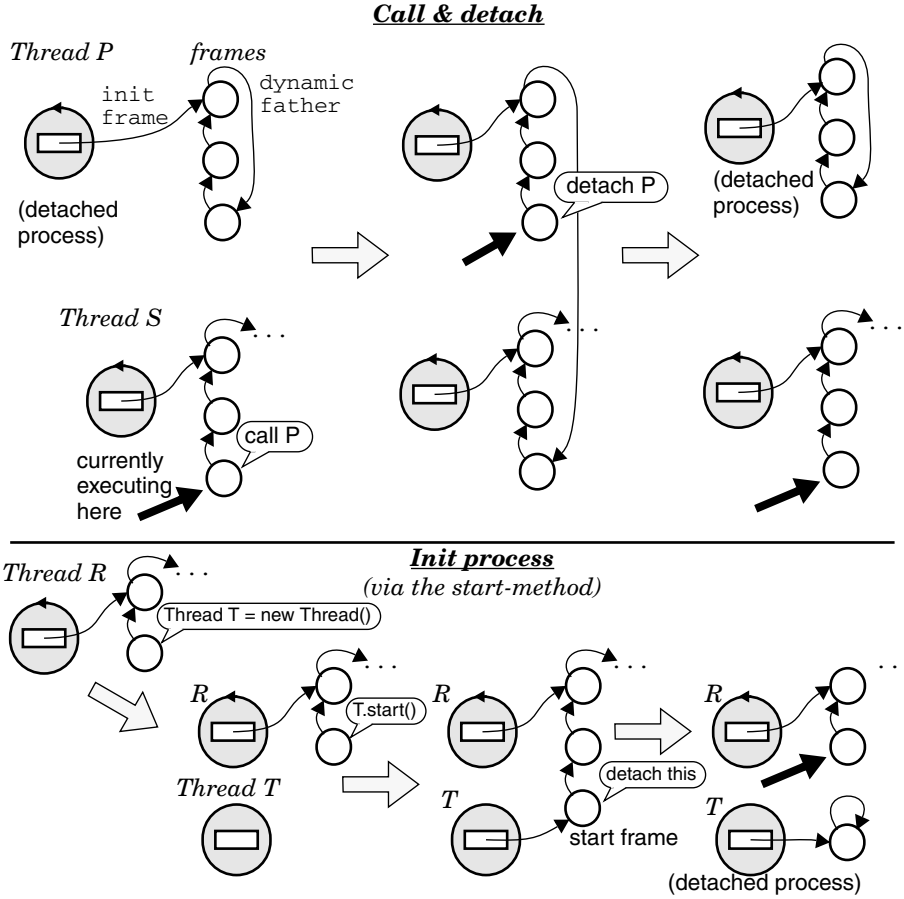


Figure 3.14 *The procedures of the coroutine methods (call, detach and initialise process) are depicted in the figure.*

When a process calls another process to continue execution, a reference to the frame of the caller is stored. When a process is finished, or when a process detaches itself, execution continues in the caller frame. In the figure, this is shown by changing the reference of the dynamic father of the initial frame. The process P that has been called sets its dynamic father to the frame where the call was performed, i.e. the top frame of the frame stack of process S. P continues execution. After a while, P is obliged to detach, either by a timer signal or voluntarily by itself. The detach method returns the execution to S and stores its own top frame as the dynamic father of its initial frame to indicate where execution shall continue after another call. The implementation of call and detach differ only in one aspect. Detached processes perform nothing if they are detached repeatedly.

The `initProcess`-method sets the frame reference in the process-object to the currently executing frame. It is utilised in a Java method that starts new processes, the `start`-method. The method invokes `initProcess` to set the current process to refer to the currently executing

method, i.e. start. This procedure is shown last in Figure 3.14. A new process is created named T. When T started from P from the method `T.start`, the `initFrame` reference of the process T is set to the currently executing frame, i.e. the `start`-frame, thus the initial frame of process T is the `start`-frame. Then the newly created process detaches itself. Execution continues in the calling frame, after the invocation of `start`. When the process T is allowed to execute, i.e. called by the scheduler, it invokes the `run`-method that is preferably overridden in a subclass. Otherwise, the default `run`-implementation in the `Thread`-class immediately returns and the `start`-method continues with the termination of the process.

There is a fourth coroutine method named *resume*. A resumed process continues execution as if it has been *called*. However, if the process is terminated or detached, execution continues after the first resume-invocation of the chain of resume-invocations. *Resume* is not utilised by the scheduler.

The identical call and detach methods have also been utilised in an implementation of the `Thread` API in J2ME written in C. With the same coroutine model, some parts of context switching may be written in C while other remain in Java. The Java variant is simpler to modify but slower than the C variant.

3.2.6 Process and scheduler structure

The scheduler of processes is implemented in Java and available for modification. Actually, the scheduler itself is executed by the machine like any other process. The machine only executes bytecodes regardless of the type of the process it executes. If an application only contains one process, there is no need for a scheduler or synchronisation between processes for that application. In this case, the application could be the only process that the JVM executes. However, for applications with many processes, a scheduler is created to manage the processes.

It is also possible to execute more schedulers than a single one concurrently in the IVM. This possibility exists because the process handling is implemented as an ordinary Java process. The only connection to the surrounding system is one operating system process in which the machine executes. Processes that execute as OS processes, *native processes*, are described like future work in Section 7.8.

The application environment — i.e. the loaded classes and their states — is also encapsulated. Different applications are able to execute concurrently on the same machine in completely separated environments. Even the same heap and garbage collector can be utilised different environments. Figure 3.15 shows three different environments and many processes co-existing in the same machine.

An application with real-time demands is instantiated in a system with only one environment and one scheduler to simplify the real-time analysis. Future work may cover a more general situation where more schedulers and machines are analysed in a real-time application.

Implementation of a coroutine scheduler

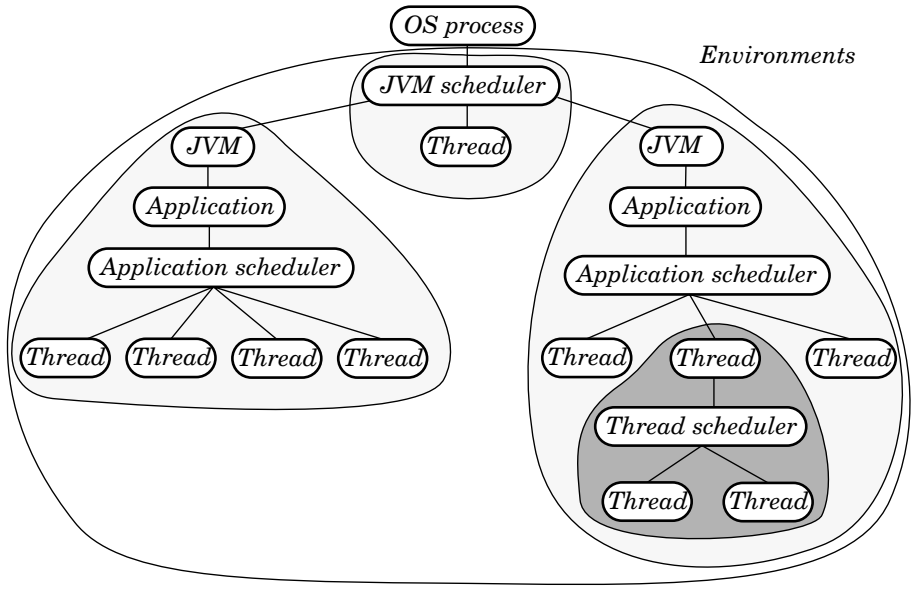


Figure 3.15 *Schedulers in the machine are implemented in Java and executes as Java threads. This simplifies the implementation and integration into applications. Many schedulers can co-exist on the same heap but with separate class environments.*

A simple implementation of a coroutine scheduler utilises a round-robin queue as the ready queue. The ready-queue is implemented by the classes `ThreadQueue`, `ThreadNotice`, and `QLinkage`. The scheduler is named `ThreadManager` and has to be started manually by a call of the method `runProcesses`. The context switch frequency is set by `orderPeriodicInterrupt`. Single interruptions are ordered with `orderInterrupt`, where the argument states when the interruption should occur. Processes are inserted into the ready queue when they are started. The coroutine methods are native and encapsulated into the class `BasicThread`. Figure 3.16 shows a class diagram over the classes related to scheduling. The classes `QueueLinkage`, `ThreadNotice`, and `ThreadQueue` cover the handling of queues.

The coroutine model has served as the foundation of a more elaborate Java variant, where different scheduling algorithms are implemented, e.g. earliest deadline first scheduling.

3.2.7 Exceptions

Exception handling is a mechanism to provide error handling in a controlled way. Unexpected situations are handled by exception-handlers that *catch* the exception within the code ranges of the handlers. If there is no handler of a matching type, the exception is passed over to the caller of the method. The caller then examines its handlers to see if it is able to catch the exception. If no frames catch the exception, the thread terminates. When an exception handler catches an exception, the program counter is set to the start of the code of the handler. The stacks are cleared and the reference to the exception is pushed on top of the reference stack.

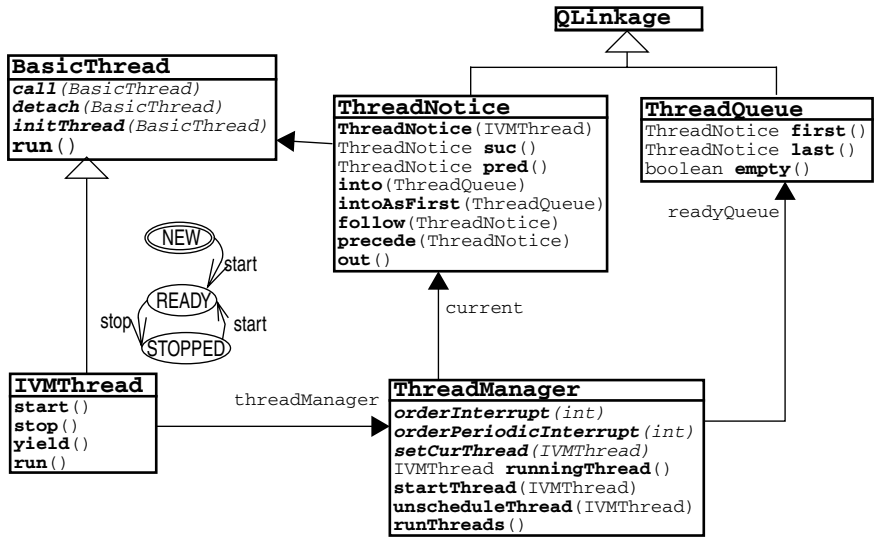


Figure 3.16 *The class diagram shows the relations of the thread, the thread manager, and queue classes. A thread can exist in three different states.*

The exception-handling algorithm can be expressed with the following pseudo code: (A detailed description of the exception handling mechanism in Java can be found in [JVM99], Section 3.10.)

```
loop until no more frames
    loop through all exception handling regions of the frame
        check if the exception can be caught within the current position of the code
            check if the type of the exception can be caught
                catch the exception (execution continues in the interpreter)
terminate the thread
let the scheduler decide the following execution
```

Since the exception handling mechanism is intended for unexpected situations in the software, it has been omitted from the scheduling analysis. However, Sven Gestegård Robertz shows in [Ges03] how exceptions can be utilised in memory critical real-time systems.

In the worst case when the exception is not caught and the thread terminates, the real-time analysis is not relevant since the thread terminates. If the exception is caught, it is relevant to calculate the WCET if it affects the continued execution of the real-time task. If the exception is caught outside the *control loop*, it is unlikely that the WCET affects the current real-time application. In those cases, the application should probably inform the application supervisor with alarm signals. It is interesting to study the effect of failures in real-time systems. However, failure recovery is outside the scope of this thesis and is left as future work.

3.2.8 Garbage collection

According to the work of Roger Henriksson on scheduling garbage collection in real-time applications ([Hen98]), it is important to provide the *memory allocation rate* of the high-priority threads, i.e. the threads that are obeying real-time requirements. The real-time garbage collector has

to be incremental to support preemption. Our machine supports various types of GC implementations, even incremental and real-time adapted ones. To support many different GC implementations, the garbage collectors and the machine utilise the garbage collector interface, GCI (see [GCI02]). The GCI extends the garbage collectors with thread safety and debug support.

A general GC algorithm determines which *live* objects that shall remain. Reachable objects are determined to be living since they can be utilised by the program. A more efficient approach would be to release objects as soon as they never will be accessed again. However, is difficult to determine at what point objects could be released in those cases. A common procedure to keep track of live object is to maintain a *live reference graph*. The live reference graph starts with a *root set* that contains live references. All objects that can be reached from the *roots* are elements in the live reference graph. Information about the reference locations inside different objects is stored in the templates of the objects.

The garbage collectors in IVM also have the possibility to add information in every object on the heap. The first bytes of every object are under the control of the GC. Some GC algorithms store a *handle location* in every object in order to make it possible to refer to the handle of the object. Other algorithms store the memory state of the object, for example, a *mark* field in mark-and-sweep algorithms. Some GC algorithms do not need information in every object. The GC-fields of the objects are adapted to the GCI to simplify the interchange of GC algorithm. Figure 3.17 shows an example of how the head of every object is extended with GC information for an incremental mark-and-sweep GC.

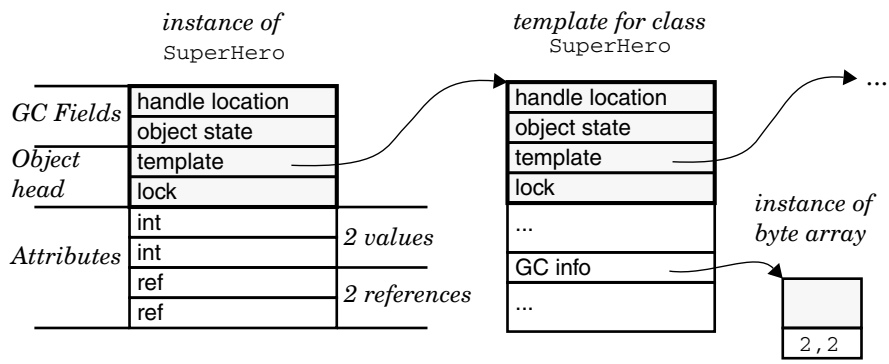


Figure 3.17 *The complete layout of an instance consists of attributes declared by the class of the instance, i.e. an object head, and GC fields.*

Our machine implements two different garbage collectors: a batch-copy GC, and an incremental mark-and-sweep GC that is prepared for real-time. For more information about the GC algorithms see [JL96].

A problematic feature of Java, concerning termination of objects, is the `finalize` method in the class `Object`. It has to be executed for every object before it is removed by the GC, and after the object is inaccessible from the application. Only the GC, or a similar algorithm, can decide if an object can be removed. Consequently, the GC instigates the call to the finaliser. The WCET analysis must then include the execution of the final-

iser, but it is omitted in our implementation in order to simplify the analysis. The analysis would be pessimistic if the finaliser is to schedule termination of objects during every period.

3.3 Real-time aspects

One major goal is to show that Java can be utilised in hard real-time systems without any introduction of code conventions or extra programming overhead. To approach this goal the following assumptions are assumed:

1. Existence of preemption points after the execution of every bytecode, and in some cases inside time-consuming bytecodes.
2. There are worst-case execution times for bytecodes, context switches, and the scheduler.
3. The GC is adapted to real-time.
4. The code is annotated to limit large data structures and recursive method calls.

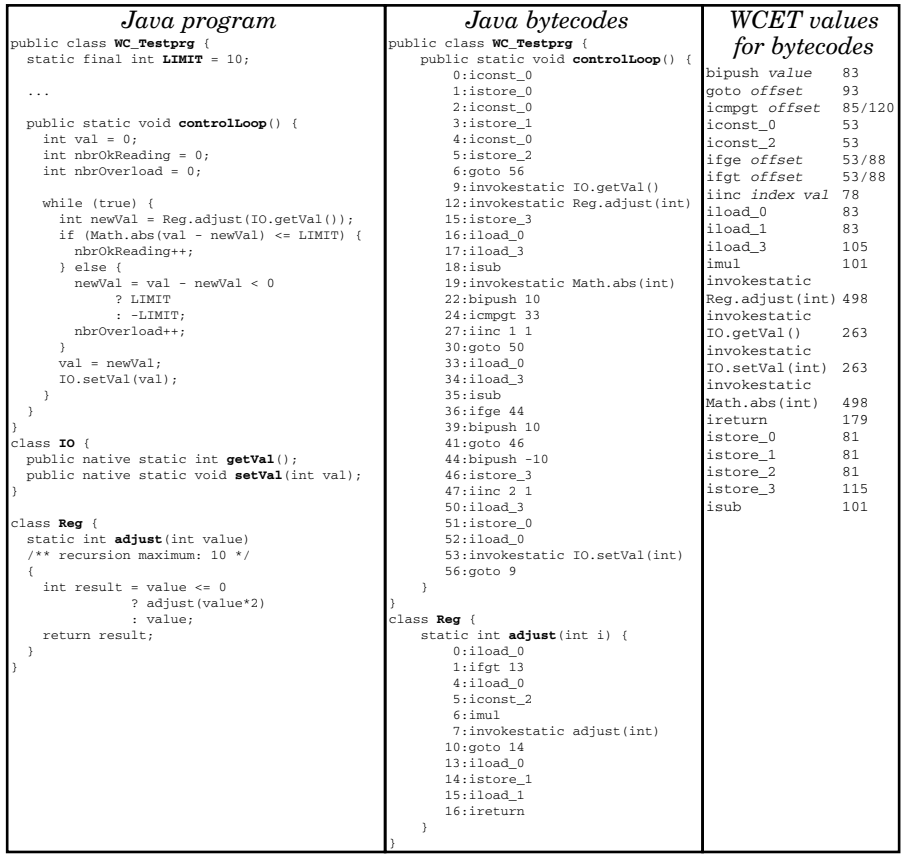
Based on those assumptions the scheduling analysis, the Worst-Case Live Memory (WCLM) analysis, and the Worst-Case Execution Time analysis could be performed. The following subsections describe the WCET and WCLM analysis based on the bytecodes. Traditional scheduling algorithms may be utilised as these analyses are performed.

3.3.1 WCET analysis

The worst-case execution time analysis determines the execution time of the various control-flow paths that a program may take. The duration of the most time-consuming path constitutes the WCET for either a program or a specific code sequence, e.g. a control loop. Indeterminable control paths (halting problem) have to be subjected to restrictions, if the analysis shall be able to deliver a result. The restrictions are noted as comments in the code. Those annotations are added to the classfile by a specific tool that extracts them from the source code. The WCET analysis is performed during class loading, and its procedure is described in detail in the thesis by Patrik Persson in [Per00]. To support the worst-case execution time analysis, every bytecode is given a worst-case execution time value that is obtained by a detailed study of the binary code (see Appendix A). The maximal time to perform a context switch also has to be included in the WCET analysis.

Figure 3.18 shows a simple control loop in Java, its bytecodes, and the fictitious WCET of the bytecodes. The WCET values in the figure are derived from a preliminary bytecode WCET analyser. The bytecodes that handle conditional jumps have two values. If the jump is performed, the higher value is utilised in the WCET analysis. Indeterminable loops and unlimited recursions are limited by annotations — comments that are

handled by a special tool. The basic blocks of the program and the WCET for bytecodes in the control loop are described in Figure 3.19.



Java program

```
public class WC_Testprg {
  static final int LIMIT = 10;

  ...

  public static void controlLoop() {
    int val = 0;
    int nbrOkReading = 0;
    int nbrOverload = 0;

    while (true) {
      int newVal = Reg.adjust(IO.getVal());
      if (Math.abs(val - newVal) <= LIMIT) {
        nbrOkReading++;
      } else {
        newVal = val - newVal < 0
             ? LIMIT
             : -LIMIT;
        nbrOverload++;
      }
      val = newVal;
      IO.setVal(val);
    }
  }
}
class IO {
  public native static int getVal();
  public native static void setVal(int val);
}

class Reg {
  static int adjust(int value)
  /** recursion maximum: 10 */
  {
    int result = value <= 0
               ? adjust(value*2)
               : value;
    return result;
  }
}
```

Java bytecodes

```
public class WC_Testprg {
    public static void controlLoop() {
         0:iconst_0
         1:istore_0
         2:iconst_0
         3:istore_1
         4:iconst_0
         5:istore_2
         6:goto 56
         9:invokestatic IO.getVal()
        12:invokestatic Reg.adjust(int)
        15:istore_3
        16:iload_0
        17:iload_3
        18:isub
        19:invokestatic Math.abs(int)
        22:bipush 10
        24:icmpgt 33
        27:iinc 1 1
        30:goto 50
        33:iload_0
        34:iload_3
        35:isub
        36:ifge 44
        39:bipush 10
        41:goto 46
        44:bipush -10
        46:istore_3
        47:iinc 2 1
        50:iload_3
        51:istore_0
        52:iload_0
        53:invokestatic IO.setVal(int)
        56:goto 9
    }
}
class Reg {
    static int adjust(int i) {
         0:iload_0
         1:ifgt 13
         4:iload_0
         5:iconst_2
         6:imul
         7:invokestatic adjust(int)
        10:goto 14
        13:iload_0
        14:istore_1
        15:iload_1
        16:ireturn
    }
}
```

WCET values for bytecodes

bipush *value*	83
goto *offset*	93
icmpgt *offset*	85/120
iconst_0	53
iconst_2	53
ifge *offset*	53/88
ifgt *offset*	53/88
iinc *index val*	78
iload_0	83
iload_1	83
iload_3	105
imul	101
invokestatic Reg.adjust(int)	498
invokestatic IO.getVal()	263
invokestatic IO.setVal(int)	263
invokestatic Math.abs(int)	498
ireturn	179
istore_0	81
istore_1	81
istore_2	81
istore_3	115
isub	101

Figure 3.18 *The program example is shown in terms of its bytecodes and their WCET.*

The class loader calculates the WCET for basic blocks and then selects the most time-consuming path for the final WCET calculation that is given to the scheduling analysis. The basic block of the program example is depicted in Figure 3.19 together with the WCET of the bytecodes. The recursive method `adjust` in the class `Reg` is annotated with a maximal recursion depth, i.e. 10 recursions. The programmer has to guarantee that this limit is never exceeded. The WCET calculation for the control loop in Figure 3.18 is described in Figure 3.20. The recursive method `adjust` is annotated with a maximum recursion depth of 10 recursions — the method is called at most ten times. The last call must terminate the

recursion. When calculating the WCET for the method, nine calls are recursive, and the last terminates the recursion.

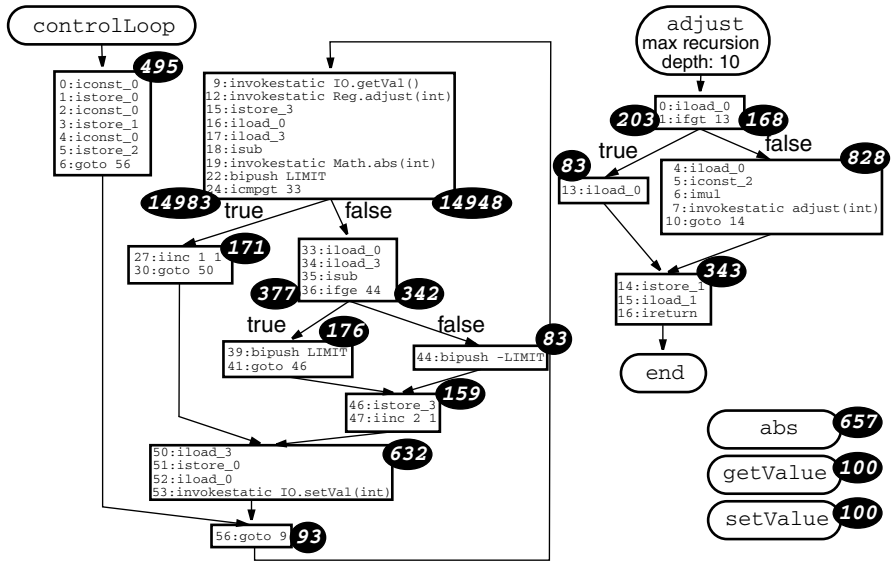


Figure 3.19 *The basic block structure of the program examples shows the control flow. The worst-case execution time of the main-loop is calculated for the scheduling analysis. The black circles hold the WCET for the blocks.*

$$\boldsymbol{T_{adjust\_recursion}} = T_{iload\_0} + T_{ifgt\_false} + T_{iload\_0} + T_{iconst\_2} + T_{imul} +$$
$$+T_{invokestatic\_adjust} + T_{goto} + T_{istore\_1} + T_{iload\_1} + T_{ireturn} =$$
$$=83+53+83+53+101+498+93+81+83+179 = \boldsymbol{1307}$$

$$\boldsymbol{T_{adjust\_return}} = T_{iload\_0} + T_{ifgt\_true} + T_{iload\_0} + T_{istore\_1} + T_{iload\_1} + T_{ireturn} =$$
$$=83+88+83+81+83+179 = \boldsymbol{597}$$

$$\boldsymbol{WCET_{adjust}} = 9*T_{adjust\_recursion} + T_{adjust\_return} = 9*1307+597 = \boldsymbol{12360}$$

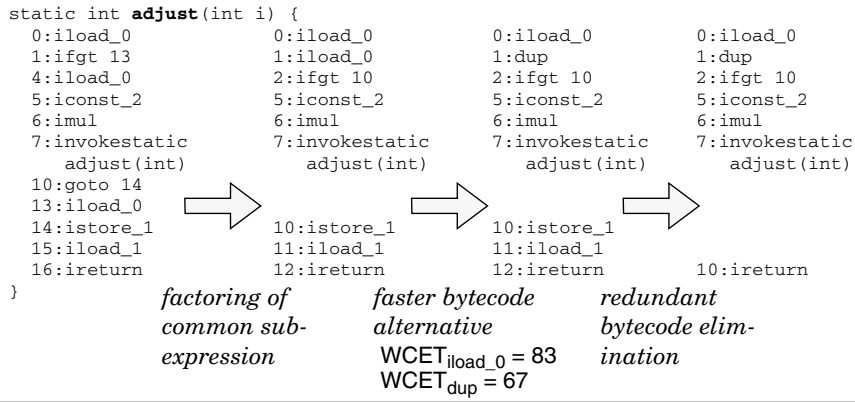
$$\boldsymbol{WCET_{control\_loop}} = T_{invokestatic\_getVal} + WCET_{getVal} + T_{invokestatic\_adjust} +$$
$$+WCET_{adjust} + T_{istore\_3} + T_{iload\_0} + T_{iload\_3} + T_{isub} + T_{invokestatic\_abs} +$$
$$+WCET_{abs}+T_{bipush}+T_{icmpgt\_false} + T_{iload\_0} + T_{iload\_3} + T_{isub} + T_{ifge\_true} + T_{bipush} +$$
$$+T_{goto} + T_{istore\_3}+T_{iinc} + T_{iload\_3} + T_{istore\_0} + T_{iload\_0} + +T_{invokestatic\_setVal} +$$
$$WCET_{setVal} + T_{goto} =$$
$$=263+100+498+12360+115+83+105+101+498+657+83+85+83+105+101+88+83+$$
$$+93+115+78+105+81+83+263+100+93 = 14948+377+176+193+632+93 = \boldsymbol{16419}$$

Figure 3.20 *The WCET calculation for the control loop depends on the longest control path of the code. Recursive methods must be limited by the programmer.*

The feedback of the time analysis supports the programmer in the design of the program. Well-considered algorithms and data structures that are adapted to the real-time behaviour of the program could be chosen. In many situations, the compiler is not aware of the real-time performance of the bytecodes. The generated bytecodes may not be optimal

for the application. In Figure 3.19, the bytecode in the control loop at index 6, `goto 56`, leads to another `goto`-bytecode that jumps to the bytecode at index 9, i.e. the bytecode directly after the first `goto`. The `goto` at index 6 could be removed. The method size would be three bytes smaller, and the WCET for the method would decrease.

Optimisations are relevant in code that is supposed to execute many times, or in our case, code that contribute to the WCET. In the example, the control loop will execute under hard real-time restrictions. Within the control loop there is a call to the method `adjust`. That method also calls itself recursively at most 9 more times. Optimisations in that method would probably result in significant changes to the execution times of the control loop. Figure 3.21 shows the how the bytecode may be optimised. First, the `iload_0` is moved from the basic blocks after the comparison, into the first block. Two bytecodes (`13:iload_0` and `14:goto 14`) are rendered unnecessary and removed. The execution of the `adjust` method would be faster and its size smaller. Second, the duplicated `iload_0` instructions could be replaced by `iload_0` and `dup`, if the `dup` is faster than the `iload_0` bytecode. This may reduce the execution time for every call to `adjust`. Both the average execution time and the WCET are lowered. Third, the `istore_0` and `iload_0` instructions are unnecessary. The returning value already resides on the stack and it does have to be not required to save it in a local variable.



Size (bytes): 17	13	13	11
WCET (ticks): 12360 (9*1307+597)	11523 (9*1214+597)	11363 (9*1198+581)	9723 (9*1034+417)

Figure 3.21 *Bytecode optimisations may reduce the method size and the WCET.*

3.3.2 WCLM analysis

The worst-case live memory analysis is performed during class loading. It determines the largest amount of allocated memory within a specific code sequence, or a program. If the hardware cannot offer enough free memory, the application cannot execute in a predictable fashion.

Some program constructions leave the WCLM indeterminable. In those cases, the programmer must support the analysis with restrictions,

e.g. lists, loops, and recursive method invocations must be limited. Figure 3.22 shows code whose memory consumption is indeterminable. The list in the code is annotated by the programmer. A special tool has been developed to extract the annotations and extend the classfiles with the extra information.

```
...
do {
   int n = SimpleInput.readInt();
} while (n<0 && n>10);
List l = new List(); /** WCLM:max size: 10 **/
for (int i = 0; i < n; i++) /** WCLM:maximum iterations: 10 **/
   l.add(new ListElement());
...
```

Figure 3.22 *The code contains memory allocations whose memory consumption cannot be determined without the support from the programmer via annotations.*

The number of elements in a list is indeterminable, in the general case, during analysis. In the example, a clever analyser could identify the maximum number of elements in the list because the program checks that the number of elements does not exceed ten. If the analyser cannot determine the exact types of the elements in the list, the WCLM is the maximal list size multiplied by the largest possible list element. This approximation may be pessimistic, but annotations may support the analyser to calculate the maximum size of the list with appropriate element sizes. Without the annotations, the analyser must assume that the largest possible element is utilised in the list. For example, if the list allows instances of `Object`, the largest possible class in the system, has to be assumed list elements. Feedback to the programmer supports the design of the list and the writing of annotations. A more elaborate description of the WCLM analysis can be found in the licentiate thesis [Per00] by Patrik Persson.

The WCLM analysis calculates the maximal possible amount of memory that can be allocated. Even method frames are included in the analysis. Displaying the memory utilisation during runtime may provide important information. The programmer may modify the program in order to reduce the memory utilisation and decrease the WCET for the most time-consuming control path.

3.3.3 Scheduling analysis

The scheduling analysis determines whether an application will keep all its deadlines at all times. Necessary requirements of the analysis are how often and how long threads execute. To simplify the analysis, threads are presumed to execute during their total worst-case execution time. Dynamic schedulers consider the actual execution time, but they introduce a risk where deadlines sometimes may be exceeded.

With values from the WCET analysis and thread periodicity, the scheduling analysis is able to determine if a system will fulfil its real-time requirements or not. High-priority threads and the GC thread are primarily considered in the analysis. Low-priority threads are supposed to be without real-time demands, but the scheduling analysis must take care of priority inheritance. The latency of a context switch is added when a high-

priority thread is interrupting a low-priority thread. To ensure schedulability of the GC the worst-case simultaneously live memory has to be calculated.

The execution time of the middle-priority GC thread depends on how much allocations the high-priority threads have performed. The actual GC work is delayed until the execution of the GC thread, i.e. after the execution of all high-priority threads. The scheduling analysis checks if the GC is schedulable (see [Hen98]). The maximal amount of work has to be determined in order to ensure correct GC timing.

3.4 Discussion

The described simple and homogenous memory utilisation provides simplicity at the cost of performance. It enables, however, the WCET and WCLM analysis that provide valuable information for the programmer in the design of real-time applications. Program code can be modified to utilise the memory more efficiently. Unexpected time-consuming and memory-consuming code sections can be identified.

Chapter 4

Classfile conversion

Classfiles must be converted into a format that is more suitable for interpretation. Direct interpretation of the classfiles would be slow because the classfiles uses symbolic references for identifier binding. To use these directly during runtime would mean extensive text matching and slow execution.

Besides improving performance, the classfile conversion verifies the program code in the classfile, i.e. makes sure it follows the format, and is not malignant. After conversion, the classfile has to be represented internally, with the runtime data structures described in Section 3.2.

The conversion procedure is the task of the *class loader*. After it has loaded and converted the classfile, the internal data structures of the class are handed to the interpreter that utilises them during interpretation. Figure 4.1 shows the three main parts of the class loader. *Class loading* refers to the process of locating and fetching classfiles. *Class linking*, on the other hand, verifies the class, creates an internal representation of the class, and resolves the bytecode into a format more suited for interpretation. It also matches references to other classes — it "links" them. During *class initialisation*, static code is executed, i.e. code that is executed only once before the class is utilised. All classes have to pass through the different phases of the class loader. The details of conversion are described in the following subsections.

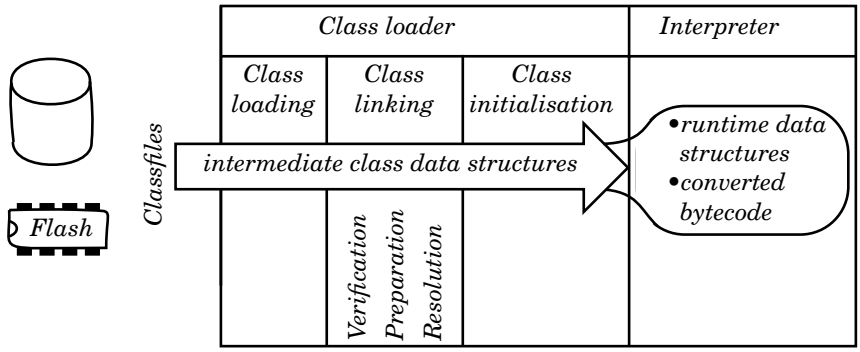


Figure 4.1 *The class loader converts the classfiles into internal runtime data structures that are more suited for interpretation.*

The contributions of this work are to elucidate the connection and to show how to integrate the Java virtual machine with the limitations of real-time embedded systems. This piece of work also introduces two techniques to reduce the size and the time a classfile spends in the class loader. These techniques are the *preconverted classfiles* and the *split machine*. Finally, the conversion of bytecodes is studied in detail.

4.1 Classfile conversion overview

Before a class can be utilised in execution, it must have passed the three major phases of the class conversion procedure according to the JVM Specification: loading, linking, and initialisation. In this piece of work, the phases will be referred to as class loading, class linking, and class initialisation in order to distinguish them from the more general concepts of loading, linking, and initialisation. This section gives an overview of the three phases. Among them, the most demanding phase, class linking, is described in detail in a later section. The structure of the classfiles is presented briefly, as a background for further discussions.

4.1.1 Conversion requirements

The class conversion procedure described in the Java Virtual Machine Specification [JVM99], pp. 48-54, is divided into the following sections and subsections that are performed sequentially for every class (the specification also describes how a multi-threaded class loader should work):

- **Loading** – refers to the process of finding classfiles.
- **Linking** – transforms the classfile into an internal representation. This process is divided into subsections:
  - **Verification** – ensures that the class is structurally correct, for instance, the bytecodes must be defined, operand types have to match the bytecode, and branch instructions have to land on the beginning of a bytecode and not in the middle of a bytecode.
  - **Preparation** – creates and sets static fields to default values.

  - **Resolution** – checks and converts symbolic references to an internal more efficient representation.
- **Initialisation** – consists of executing initialisations for static fields and static blocks.

Verification ensures that the program follows the JVM Specification and that the program does not make the machine crash. Verification is also an important component in order to make Java a secure programming language.

4.1.2 Classfile structure

The classfile contains all information that is necessary in order to link a class with a runtime system. Classfiles are similar to object files that have to be linked to form an executable program. The classfiles are thoroughly specified in the JVM specification, see [JVM99] chapter 4, "The Classfile Format", pp. 93–153. The classfile structure is briefly presented here as background information for the later discussions. All the references in the classfile are indices into its constant pool, where textual representations of the references are located. A classfile has the following contents:

- **Magic number** identifies this file as a classfile.
- **Versions** state the valid JVM versions that can execute this class.
- **Constant pool** contains all the constants, symbols, and symbol references to classes, and signatures[1] of methods, fields.
- **Access flags** show the class modifiers.
- **This class** identifies this class.
- **Superclass** identifies the superclass of this class.
- **Interfaces** list all the implemented interfaces.
- **Fields** describe all the attributes, static fields, and constants by name, type, field modifiers that are declared in this class.
- **Methods** describe all static and virtual methods, and interface methods by name, signature, method modifiers, and bytecode.
- **Attributes** contain extra and non-crucial information about the class.

Interfaces are also described by classfiles. However, interface methods and native methods do not contain a bytecode body. All other methods contain the bytecode, stack size, size of the local variable area, and exception handling information.

To illustrate the classfile format, Figure 4.2 shows a class and its classfile in a simplified form. Textual comparison is needed to resolve the symbols in the bytecode, for example, method bytecodes refer to the textual names of the methods via the constant pool. The elements of the constant pool are strings and symbolic references to classes, methods, and fields. A support element in the constant pool, called name-and-type, provides

1. A signature of a method contains type information but omits the modifiers. For example, the signature for `main` is: `([Ljava/lang/String;)V` – i.e. the argument is a string array (one dimension) and the method returns nothing (void).

methods and fields with more symbolic information. The methods are described with their code and information about the method name, access flags, types of arguments, and type of return value. In this example, the classfile is 372 bytes, of which 167 (45%) are spent by the constant pool. The methods constitute 71 bytes (19%). In the example, only those methods that are utilised in the bytecode have a method reference into the constant pool. Method descriptions refer to the textual fields directly.

```
public class SuperHero extends Hero {
    private int power;
    SuperHero(int power) { this.power = power; }
    public int power() { return power; }
    public boolean defeatObstacle(int difficulty) {
        return power() > difficulty;
    }
}
```

Magic number
Minor and major version
Constant pool
#1: Method class: 5=Hero name_and_type: 14=<<init> ()void>
#2: Field class: 4=SuperHero name_and_type: 15=<power int>
#3: Method class: 4=SuperHero name_and_type: 16=<power ()int>
#4: Class name: 17=SuperHero
#5: Class name: 18=Hero
#6: Utf8: "power"
#7: Utf8: "I"
#8: Utf8: "<init>"
#9: Utf8: "(I)V"
#10: Utf8: "Code"
#11: Utf8: "()I"
#12: Utf8: "defeatObstacle"
#13: Utf8: "(I)Z"
#14: NameAndType name: 8=<init>, signature: 19=()void
#15: NameAndType name: 6=power, signature: 7=int
#16: NameAndType name: 6=power, signature: 11=()int
#17: Utf8: "SuperHero"
#18: Utf8: "Hero"
#19: Utf8: "()V"

***Access flags**: public*
***This class**: 4=SuperHero super: 5=Hero*
***Interfaces** (count: 0):*

***Fields** (count: 1):*
Field name: 6=power private Signature: 7=int

***Methods** (count: 3):*

Method name:8="<init>" Signature: 9=(int)void
Attribute "Code", length:22, max_stack:2, max_locals:2
code_length:10
0: aload_0
1: invokespecial #1=<Method Hero.<init> ()void>
4: aload_0
5: iload_1
6: putfield #2=<Field SuperHero.power int>
9: return

Method name:6="power" public Signature: 11=()int
Attribute "Code", length:17, max_stack:1, max_locals:1
code_length:5
0: aload_0
1: getfield #2=<Field SuperHero.power int>
4: ireturn

Method name:12="defeatObstacle" public
Signature: 13=(int)boolean
Attribute "Code", length:26, max_stack:2, max_locals:2
code_length:14
0: aload_0
1: invokevirtual #3=<Method SuperHero.power ()int>
4: iload_1
5: if_icmple 12
8: iconst_1
9: goto 13
12: iconst_0
13: ireturn

***Attributes** (count: 0):*

Figure 4.2 *The figure contains a small Java program and its classfile representation. The <init> method is the name of the constructor. Types are coded with letters, e.g. I is int, V is void, and Z is boolean.*

4.1.3 Class loading

The task of the class-loading phase is to locate, load, and pass over the classfile to the next phase, i.e. class linking. There are four causes for class loading. First, in order to execute the `main`-method, the class that contains the `main`-method has to be loaded, the *main-class*. Second, there are some classes loaded by default. Third, during class conversion other classes may be referred to. The referred classes may be required during execution. Fourth, class loading may be initiated by the application during execution. The default classes are loaded by the JVM before the main-class. The main-class is the seed for further class loading due to its references that refer to other classes.

There are a number of different places to fetch classfiles. Table 4.1 shows examples of locations where classfiles are stored. Embedded systems often utilise the ROM or a network to store the classfiles. Desktop computers often utilise their local hard drive for classfile storage.

Classfile storage	*Suited for*
Network	desktop computers embedded systems in combination with classes in ROM
Local hard drive	desktop computers
ROM	embedded systems
"JVM"	rapid development, debug purposes

Table 4.1 *The classfiles may be stored in different places. It is the task of the class-loading phase to handle the different storage locations.*

The classfiles could be stored as a part of the JVM itself. This *static classfile* system circumvents problems that may arise with other types of storages — no file system is necessary. Static classfiles support debugging and rapid development. A drawback with the static classfiles approach is, however, that it requires the creation of an additional object file that has to be linked into the JVM. Another drawback is that static classfiles are located in memory together with the code of the JVM, which contradicts the requirement of efficient memory utilisation. During execution, the classes are represented twice: as static classfiles and as runtime templates.

All the classfiles may be compressed to save memory space. Decompression algorithms, in that case, must be included in the class loader. All the storage types can utilise compression.

Another requirement during the class conversion is to handle many threads that are loading the same class at the same time. These critical sections have to be synchronised according to the "Detailed Initialization Procedure" described in the JVM specification, see [JVM99], pp. 52–54.

4.1.4 Class linking

The linking phase links the classfiles into the runtime system by creation of corresponding class templates (preparation) and by converting the bytecodes into a form that is suitable for interpretation (resolution). Verification ensures that the classfiles are correct and non malignant. Java is a secure language and the verification phase is an important part to guarantee that.

Class linking is the most complex and largest part of the class loader. In an embedded system, it is relevant to analyse the temporary data structures that are utilised during class linking in order to examine the memory consumption. Class linking may be the most memory consuming part during the lifetime of an application. A detailed study of the tempo-

rary structures may reveal how the peaks of the memory utilisation may be reduced.

The class-linking phase prepares the classfile for execution by creating runtime data structures of the class and by converting bytecode to an interpretable form. The runtime class structures, i.e. class templates, are created with the aid of temporary data structures, mainly supporting the symbolic reference resolution. The following steps require temporary data structures, and they are performed for every classfile, during class linking:

1. Transfer the symbols of the classfile into global symbol tables.
2. Organise the class and generate class information, i.e. method tables, object layout, garbage collector information etc.
3. Convert the bytecode.

The bytecode conversion converts the references in the bytecode from symbolical references to direct ones, i.e. the textual resolution is converted into array indices. Other bytecode transformations are related to the control flow analyser.

All classes are stored in the *template table*. Their symbols, i.e. their names, are stored in another table, the *template symbol table*. Other symbols are stored in the *symbol table*. These tables are utilised to represent symbols with a unique index, instead of the complete symbol itself. For example, if the `java/lang/String` class template is located at index 30, the class template is referred to as the number 30. The symbol for the class is found in the 30:th element of the template symbol table. The advantage of global symbol tables is that identical symbols in different classfiles can be reused, which saves memory.

A significant deviation from the JVM specification in our approach is to load all the necessary classes before execution. The reason is to decrease the WCET in the real-time analysis by being able to exclude the class conversion. The class conversion, written in C, is difficult to analyse. However, if it was written in Java, the ordinary real-time analyser could be utilised even though the WCET for the class loader would become large.

The lazy class loading procedure, i.e. the loading of classes when they are needed, encounters problems in embedded systems, considering that the memory is limited and many classes cannot be loaded at the same time.

A problem with class loading during start-up is the bootstrap problem, where classes are needed before they are loaded, and they cannot be loaded because they are needed during class loading. This catch-22 situation is not addressed in the JVM specification. The lazy class loading is subject to a detailed class loading procedure (see [JVM99] pp. 158–170) in order to avoid other threads from loading the same class at the same time. Our implementation adopts a static class loading procedure in order to reduce memory consumption and complexity.

4.1.5 Class initialisation

The last step in the class conversion procedure is to execute the code in the static blocks of the class, i.e. class initialisers. Class initialisers have to be executed after the bytecode of the class has been converted but before the class is utilised, e.g. instantiated. The compiler collects all the static initialisers of a class into a class initialisation method. All the class initialisers in our machine are executed before the start of the application.

During the bytecode conversion, the static initialisers are identified. Method activations, frames, are created for them and stored in a frame list. Those frames are executed after the transformation of all classes and before the execution of the `main`-method. According to the JVM Specification ([JVM99], Section 2.17.4), there are two rules that determine the execution order of the initialisers. First, the order of the class initialisations is determined by the class hierarchy. The superclass must be initialised before the subclasses. Second, a class may not be utilised (e.g. instantiated) before the static initialisation is performed.

If the classes are converted in a hierarchical order (superclass down), the initialisation frame list is organised according to the first requirement. The second requirement is achieved by sorting the initialisation frames according to the utilisation order of the classes. The bytecode must be analysed to obtain the order before initialisation. An alternative is to execute the initialisers on the fly, i.e. lazy evaluation, and to check if a class is initialised as it is utilised. Figure 4.3 depicts the possible initialisation orders of four classes. Note that none of the possible execution orders is correct if the class `Hero` creates an instance of class `SuperHero` during the initialisation of the `Hero`-class. In that case, the execution order cannot be determined with the support of the JVM specification. The JVM should generate an error and terminate. The program (the `main`-method) can be executed after the initialisation phase, and this is marked by placing the `main`-frame last in the list.

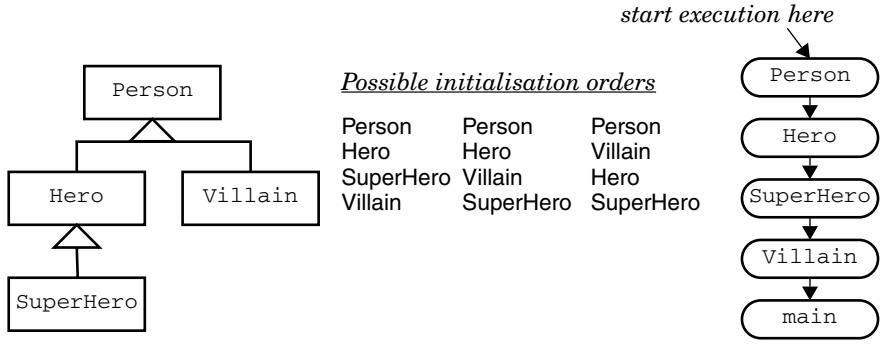


Figure 4.3 *The initialisation order of the four classes is determined by inheritance and class utilisation. The execution of the program (main) may commence after the initialisation.*

4.2 Class linking and memory utilisation

The limitations of embedded systems impose modifications on the memory utilisation of the class linking process. The class linking, with its temporary data structures, is memory intensive and has to be restricted in a system with limited memory. Class linking could be the most memory-consuming part during the lifetime of a program, because many temporary data structures are built to support verification, preparation, and initialisation.

The first, and often the largest, temporary data structures are utilised to hold the information in the constant pool (CP) of the classfile. All symbolic references from the rest of the classfile and from the bytecode are specified in the CP. In our solution, two temporary arrays hold the CP contents and their types — see Figure 4.4. The arrays hold as many entries as there are in the CP. The temporary CP arrays support the reference resolution process in the preparation and resolution phases. Since the temporary arrays allocate extensive amount of memory, it is important to release the arrays as soon as possible. As the information in the CP is transferred into other internal data structures, it may be released. The parsing of the constant pool is performed in three passes:

1. Parse the constant pool.
   - Transfer the constant pool of the classfile into the contents array and the constant pool type array.
   - Create symbols and store them in the global symbol table.
   - Count the number of strings, constants, methods, interface methods, fields, and references.
2. Transfer constants and create shadow templates.
   - Create two temporary arrays to hold string and value constants.
   - Create and insert strings in the string constants array.
   - Transfer value constants to the value constants array.
   - Create *shadow* templates, i.e. empty template objects that can be referred to, of classes and interfaces and store them in the template table. If the template already exists, it can be referred to directly. Insert the corresponding class symbol in the template symbol table.
3. Create a reference array to hold all the references to fields and methods.

Before a class can be loaded, space must be allocated for it. The *shadow* class representation incurs no extra memory overhead since it is utilised during the class loading of the shadow class. All the class representations are stored in the *template table*. Two additional internal tables contain all the symbols in the classes. They are the *symbol table* and the *template symbol table*. As a textual representation of a template is parsed, a shadow representation is created for it and added to the template table. The symbol of the template is stored in the template symbol table at the same index as the shadow representation.

Figure 4.4 shows how the CP of the classfile is parsed and how the constants are stored internally. Shadow templates are shown as empty boxes. The simplified CP of the Java program in the figure is transferred into a

type array and a contents array that are utilised to resolve the symbolic references in the classfile. The constants (declared as `static final` in the Java program) are stored in the constants arrays of the classfile. They are the value constant `MAX_power` and the string constant `motto`, and they are stored in the value constant array and the string constant array respectively. Offsets to the constants are stored in the contents array. Classes in the contents array are represented as indices into the class template table.

```
class SuperHero extends Hero implements SuperPower {
   int power;
   static final int MAX_power = 100;
   ...
   Villain getEnemy() { ... }
   static final String motto = "Right means Might";
   ...
}
```

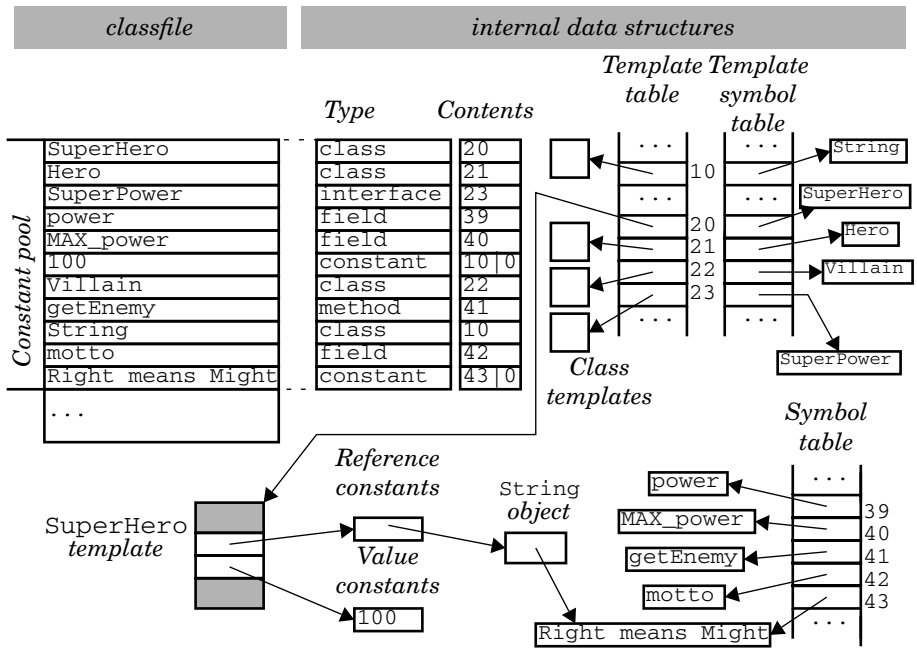


Figure 4.4 *The constant pool of the program example is stored in temporary arrays. Shadow templates (*`Hero`*,* `SuperPower`*, and* `Villain`*) are created for classes that are referred but not already loaded.*

During the last part of the preparation and during resolution, the parts that are dependent on information in other templates are resolved. The CP contents array is then superfluous and under the disposal of the garbage collector. The type array is only necessary during the parsing of the CP. If verification is supported, it cannot, however, be dropped until all the references to the CP have been checked to access the correct element type, e.g. unresolved method bytecodes contain a reference to a method description in the CP.

Besides the CP, there are two large sections in the classfile: the fields and the methods. The field section contains information about fields declared in the class, i.e. field attributes, field types, field offsets, and indirect references to names (indices into the symbol table). Field offsets to

the static fields and constants (they are accessed as static fields) are calculated and stored. Static fields are added to the constants arrays in the class template. Figure 4.5 shows the conversion of field information. The machine does not discriminate between the static fields and the constants; they are accessed in the same manner. Verification has to confirm that the usage of constants is correct.

```
class SuperHero {
   static final int MAX_power = 100;
   static final int MIN_power = 3;
   static final String motto ="Right means might!";
   static final String curse ="Jikes!";
   static int power;
   static float mood;
   static SuperHero master;
   ...
}
```

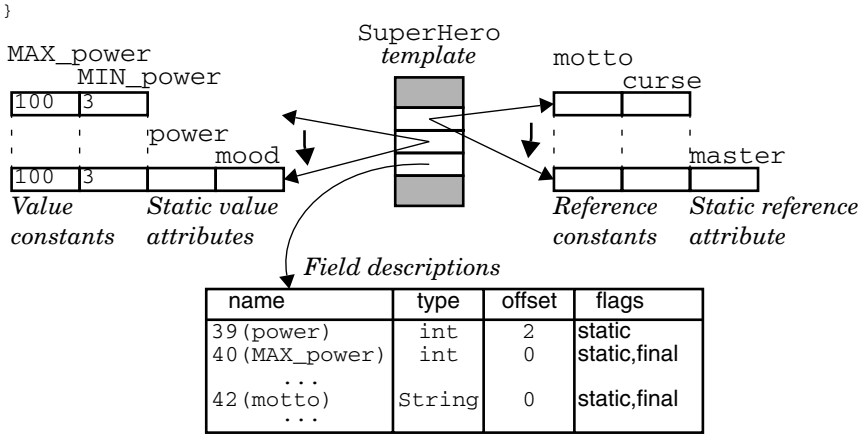


Figure 4.5 *The static fields in the classfile are added to the constant arrays. The machine does not discriminate between constants and static fields.*

All information about the methods in the classfile is stored in a temporary method array. The methods are separated into three different arrays to improve performance — the static methods array, the interface methods array, and the virtual methods array. First, the static methods are extracted at an early stage, but the interface methods cannot be identified and separated from the virtual methods until all the interfaces have been loaded. Figure 4.6 depicts the static method array creation. The two static methods `main` and `superHeroAmount` are extracted from the temporary methods array. The remaining methods are stored in the virtual and interface method array. Information about exception handling and the bytecodes are brought together with the methods.

When all the method templates have been sorted, the bytecode may be converted. All the class references in the bytecode are substituted with indices into the class template table.

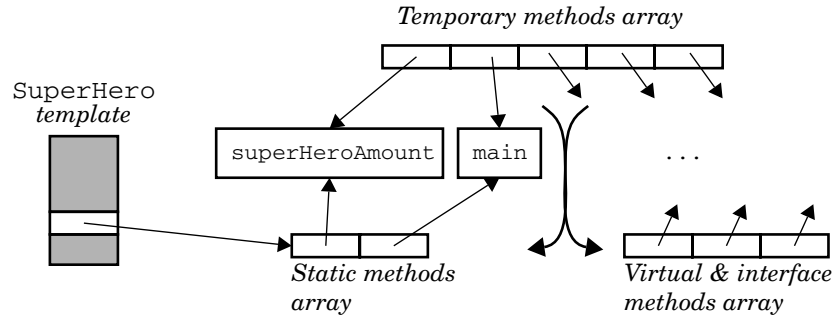


Figure 4.6 *All the methods in the classfile are converted to method templates that are stored in a temporary methods array. Static methods are extracted and put in the class template. Virtual methods and interface methods are separated after all interfaces are read.*

4.2.1 Deep and shallow template references

The template references can be divided into two different types, *shallow* and *deep* references. Shallow references refer to templates. They are called shallow since the referred classfile does not have to be loaded to be represented. It is sufficient to refer to an empty template. As the classfile is loaded, its already allocated template will be used. Deep template references, on the other hand, are utilised to access information inside a template. Thus, the accessed template has to be loaded and linked. An example of a shallow reference is a template's reference to the superclass. A deep reference is, for example, an access of a bytecode to a static field in a class template.

4.2.2 Finishing linking and memory utilisation

When all the necessary classes have been loaded, the last transformation phases commence, i.e. finishing linking and resolution. The last steps of class linking enable all deep template accesses, in the resolution phase, to be resolved without interference from further class loading. The tasks performed during this phase are performed one class at a time, in a hierarchical order (superclass first). The tasks are:

1. Calculation of offsets to attributes and object size.
2. Generation of garbage collector information.
3. Generation of the interface array.
4. Generation of the virtual method array.
5. Conversion of the deep references in the bytecode.

The different stages in class linking are depicted together with their requirements as a dependency graph in Figure 4.7. The requirements show how many templates are required to be loaded before the stated phase may commence. For example, the bytecode `new` that depends on a

class reference and the object size cannot be converted until the object size has been calculated, which requires that the current template is in the link phase and that its superclass templates have been linked.

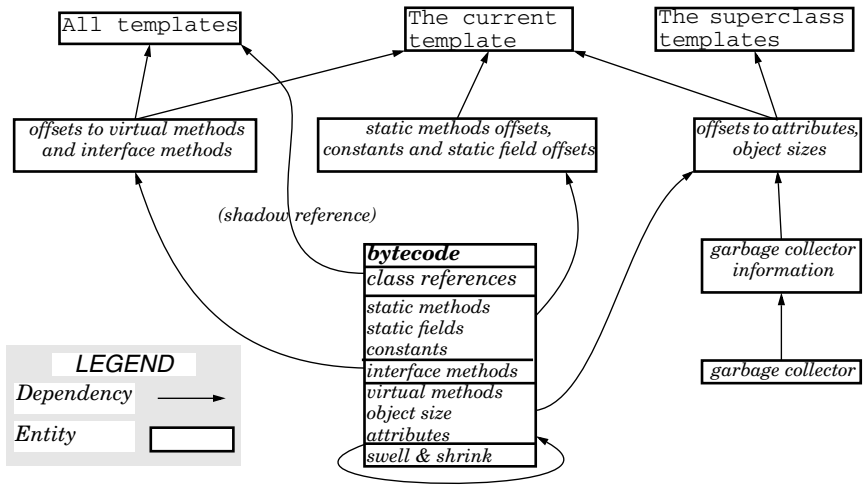


Figure 4.7 *The contents of a template are dependent on information in other templates as depicted in the graph. Furthermore, bytecodes depend on information in other templates and on the bytecode itself.*

The attribute offsets inside the instances are dependent on the attributes declared in template and the superclasses of the instance. Attributes declared in the template are added to the description of the instance by its superclass. After the offsets are calculated, the size can be determined.

The GC information can be generated when the object size and the attribute offsets have been calculated. The GC information of the superclass is copied and extended with reference locations of the template. If no new reference attributes are declared in the template, the GC information of the superclass may be referred to directly.

It is important to convert the attributes and the virtual methods before the final bytecode transformations. Some bytecodes are dependent on the information calculated in the previous phases.

The interface array consists of all interfaces implemented by the class, all the interfaces that the superclasses implement, and all the superinterfaces of the implemented interfaces. The number of interfaces is counted and represented in an array, together with the corresponding method arrays of those interfaces. Figure 4.8 shows an example of interface templates and interface arrays. In the class diagram, the interfaces are separated from the method arrays to make the picture clearer. They could be merged into a single array to make the class template smaller (one reference instead of two), but the arrays are separated here to make the example clearer. Some other examples in the thesis utilise the merged interface array.

Furthermore, Figure 4.8 shows a diagram of four interfaces named `iA`, `iB`, `iC`, and `iD`. Their runtime information is also displayed. An interface

contains information about the inherited interfaces and the declared interface methods. For example, interface `iD` implements `iB` and `iC` and declares methods `imC` and `imD`.

The two classes in Figure 4.9 implement the interfaces in Figure 4.8. Their runtime information concerning interfaces is displayed. If a class does not implement new variants of the inherited superclasses, the interface method arrays in its superclass can be reused. In the figure below, this is depicted by two references to a single interface method array. The figure depicts how the method, `imC`, is overridden by the interface `iD`. It would normally be meaningless to override methods in an interface hierarchy since they do not contain any code. However, this is allowed in Java.

As the interface methods are identified, they are removed from the temporary methods array that now consists solely of virtual methods. The order of the interfaces in the interface array may vary from class to class Every class may implement any number of interfaces in any order.

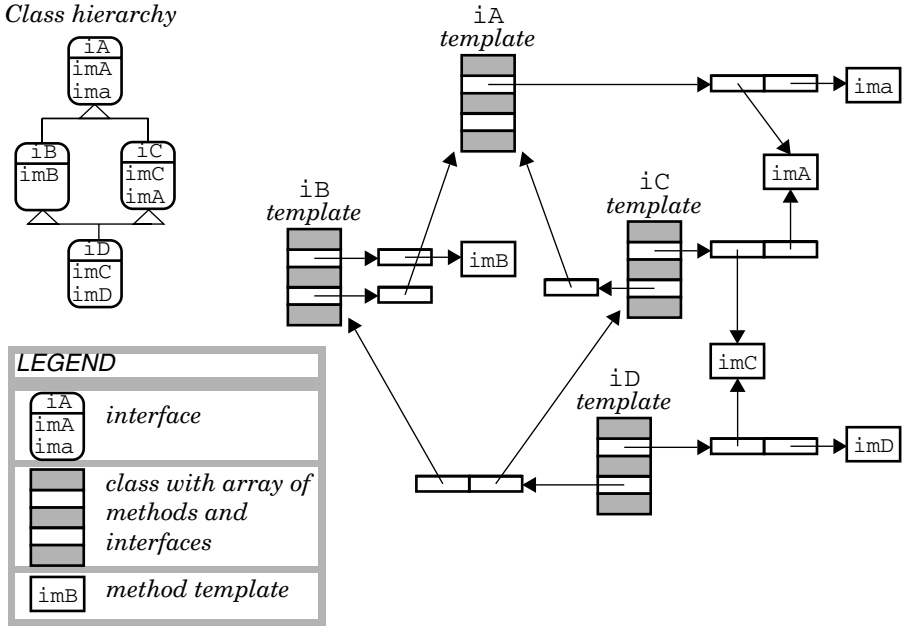


Figure 4.8 *The four interfaces and two classes reuse method descriptions and a method array in their runtime representation.*

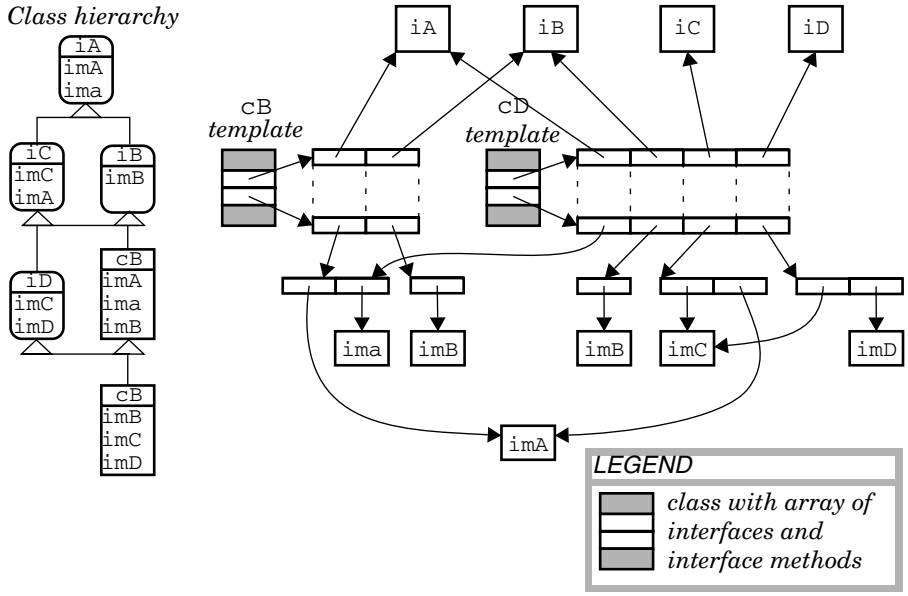


Figure 4.9 *When classes implement interfaces, all the interface methods placed in an interface array correspond to the interface. In some cases the same method templates can be reused.*

The virtual method array is based on the virtual method array of the superclass as well as the virtual methods implemented by the class. The new methods are appended to a copy of the virtual method array of the superclass. Overloaded methods replace the corresponding location in the method array. The procedure is exemplified in Figure 4.10. The virtual method array of the superclass is copied in order to maintain the same offsets to the methods in the subclass. Virtual methods have the same offsets independently of whether the instance is a superclass or a subclass. That improves the virtual method lookup time during runtime. In the figure, the two methods, `getName` and `doDeed`, declared in the superclass `Person`, are inherited by all the other subclasses. The `doDeed` method is overridden by the two direct subclasses. However, the good superheroes inherit the `doDeed` method defined in the `Hero` class, while the evil villains implement a `doDeed` method more suited for evil purposes.

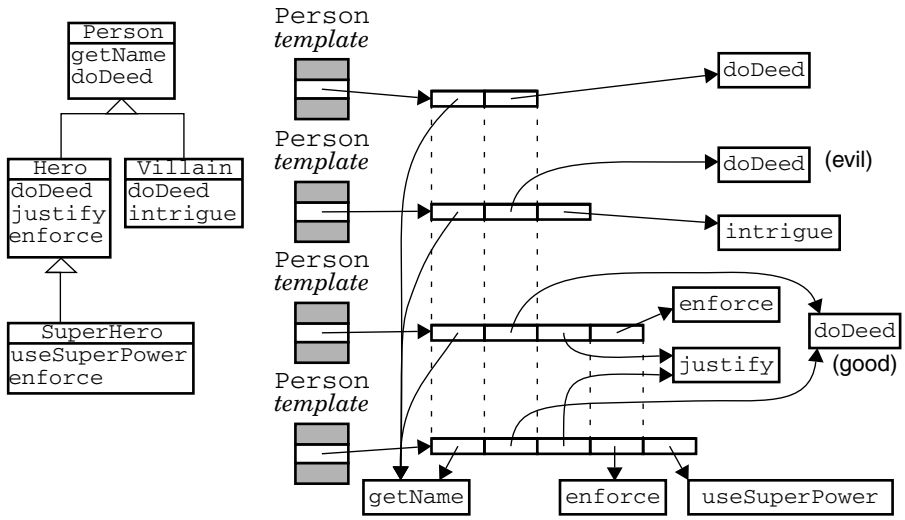


Figure 4.10 *The structure of virtual methods is dependent on inheritance.*

When all the methods are in place, the bytecodes are transformed into its final form during the resolution phase. All the operands that contain deep references are converted into internal references. For more information about the internal reference conversion see Section 4.2.5, "Detailed reference analysis" on page 96.

4.2.3 Memory allocation during class conversion

The temporary data structures during class loading, linking, and initialisation occupy a significant amount of memory. An analysis of these data structures is required to determine the memory consumption, and to provide support for actions in order to decrease the memory consumption. The class conversion procedure may be the most memory-consuming phase during a lifetime of an application. Consequently, it is important to analyse the behaviour of the class loader. The following temporary memory data structures are allocated, for every loaded class, during class initialisation:

- Constant pool contents array
- Constant pool type array
- Reference constants
- Value constants
- Implemented interfaces
- Implemented methods
- Implemented virtual and interface methods
- Class initialisation frame list

The following data structures are created and maintained for every loaded class:

- Static methods

- Virtual methods
- Reference constants and static reference field array
- Value constants and static value field array
- Interface method array and corresponding interfaces
- *Field description array*
- *Reference array*

The following data structures are global to all loaded classes (in one environment):

- Class template table
- *Class template symbol table*
- *Symbol table*

Data structures written in italics could be removed if dynamic class loading and reflection were not supported.

The data structures are depicted on a time axis in Figure 4.11. The figure shows when memory is allocated and utilised during class conversion. The lifetimes are divided into four sections, which represent steps in the linking process. The first section is valid when a classfile is loaded. The second section represents the moment the class hierarchy has been loaded. If the classes are loaded according to the class hierarchy, i.e. with the superclass before its subclasses, there is no difference between the first and second sections. The third section is entered when all classes in the application have been loaded. The fourth section shows when the bytecode is transformed. Data structures that are utilised during interpretation are marked in the last column.

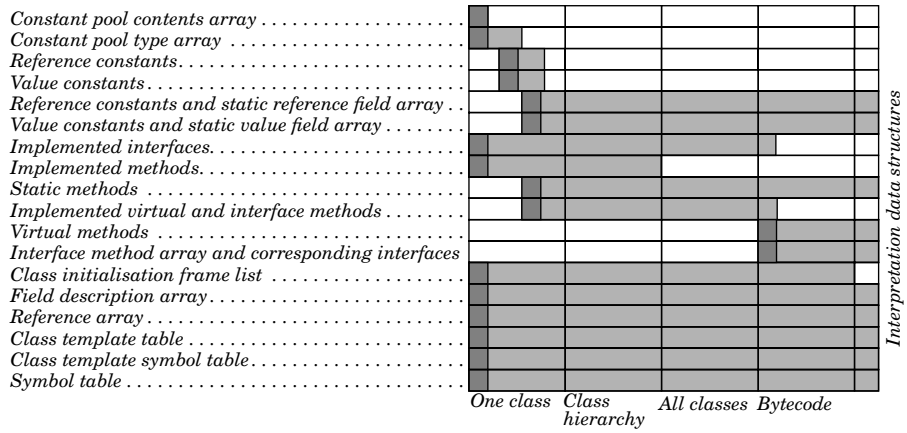


Figure 4.11 *The lifetime of data structures are marked in the diagram. Creation is marked in a darker hue. The data structures marked in the last column are utilised during runtime.*

In Figure 4.11 the conclusions can be drawn that the most long-lived temporary data structures are the constant pool contents array, the implemented interfaces, the implemented virtual and interface methods, as well as the class initialisation frame list. The implemented interfaces, and the implemented virtual and interface methods are substituted with the virtual methods and the interface method array and corresponding

interfaces, as soon as all the classes are loaded. The latter arrays could be constructed when all the necessary interfaces are loaded. If interfaces are loaded before other classes, the lifespan of the temporary data structures concerning interface methods could be shortened. The virtual methods cannot, however, be calculated before the necessary interfaces are loaded and all the superclasses are linked. The memory consumption of these temporary data structures is approximately of the same size as the final representations of those structures.

The class initialisation frame list is utilised after the conversion of the bytecodes. In order to minimise the initialisation list, an early attempt to convert the bytecode could be performed. The references of bytecodes to the constant pool are listed in Table 4.2. As the elements in the constant pool are utilised, they could be removed. For example, static fields in the class are calculated as the classfile is parsed and they are utilised by some bytecodes. The elements in the constant pool that describe those static fields are utilised by some bytecodes. A preliminary bytecode conversion of "static" bytecodes, referring to static fields in the class, could be performed to decrease the size of the constant pool. The constant pool could be copied to a new constant pool without the description of static fields. Bytecodes referring to static fields in other classes can be converted if the other classes have been linked. Traversing the bytecodes many times would slow the overall classfile conversion process. The size of the class loader would not be affected considerably.

Constant pool element	***Bytecode group***	***Utilisation***	***Removal***
Constants (values and strings)	Constants	Bytecode	After bytecode transformation.
Field references	Field accesses	Bytecode	After bytecode transformation.
Method references	Method accesses	Bytecode	After bytecode transformation.
Interface method references	Method accesses	Bytecode, interface array	After bytecode transformation.
Name and type	—	From other constant pool elements.	After parsing of fields and methods.
Utf8 – textual references	—	String constants are referred from the bytecode, other constant pool elements, from the classfile	String constants — after bytecode transformation. After utilisation from the classfile

Table 4.2 *Parts of the constant pool may be removed at an early stage in the classfile conversion in order to save memory.*

In Figure 4.11, it can be concluded that a suitable situation for preliminary bytecode conversion is after the linking of the current class, or after the linking of a complete superclass hierarchy.

JVM stack

The execution of the machine itself utilises memory for its frames on a stack. Some sections of the class linking code are recursive and require significant amounts of stack memory. However, class loading in a hierarchical order minimises the memory consumption of the stack.

Default classes
The Java requires a number of default classes to be loaded. These are derived from the JVM specification and indirectly via the API. The classes that are necessary in the machine are about 50 exceptions that the machine may produce during execution, for example, `NegativeArraySizeException`, `NullPointerException`, `IllegalAccessError`, and `OutOfMemoryError`.

Necessary classes that are indirectly accessed are `Object`, `String`, and `Class`. To what extent these classes access other classes is dependent on their implementation. In J2SE, `Object` refers directly or indirectly to 251 other classes, of which 15 are exceptions utilised by the machine. In J2ME, `Object` directly or indirectly refers to 50 other classes — 15 of them are exceptions utilised by the machine.

4.2.4 JVM start-up classes

Before the designated classfile is loaded, initial class templates have to be created. The top of the class hierarchy (`Object`) has to be loaded before inheritance can be calculated. However, `Object` requires other classes before it can be loaded. This conflict is solved by "manual" creation of shadow class templates with some information. For example, the size of instances of `Object` is put into its shadow class template. The value is overwritten as the class is loaded.

4.2.5 Detailed reference analysis

The accesses to information inside templates (deep accesses) and shallow references are located in the class itself and in the bytecode. Shallow references are resolved by substitution of the symbolic reference to a template index in the global template table (indirect reference), or to a direct reference to the template itself. Deep accesses are converted into indices to the different class template structures. For instance, a virtual method is located by a method index into the virtual method array. The final location of the virtual methods must have been calculated before the resolution of virtual method invocations. Figure 4.12 shows all references and accesses to other templates from a classfile.

Shallow references do not have to actually result in the class to be loaded. An empty class object could be allocated and loaded at a later stage. The memory allocation during class conversion is dealt with in Section 4.2.3. The shallow references in a classfile are references to the superclass, this class, i.e. a reference to itself, and the interfaces that the class implements.

During bytecode resolution, the bytecode in the classfile requires information from other classes. The information in the class template dependent on information inside another class, is offsets to the virtual methods,

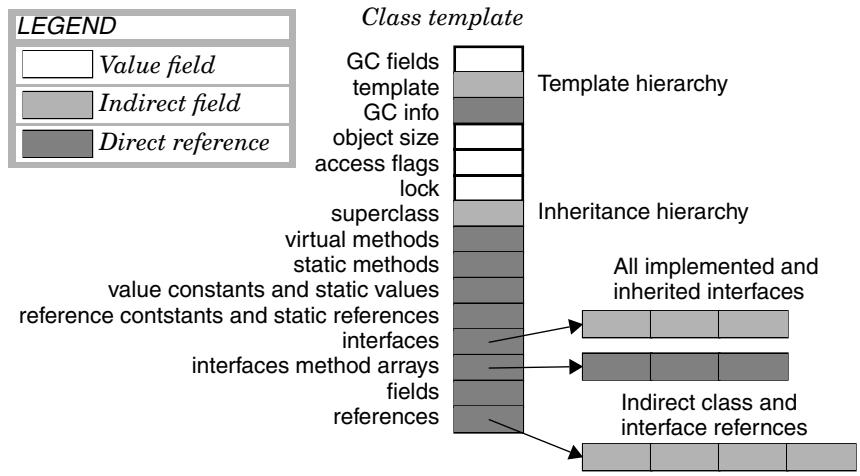


Figure 4.12 *The figure depicts where accesses and references to other templates occur in a class template. References from the bytecode only access non-bytecode information, thus enabling bytecode resolution in one class at a time.*

offsets to the interface methods, the object size, the attribute offsets, and the garbage collection information. Accesses to information inside a class are also found in the bytecode. The class information requested concerns static fields and attributes, constants as well as static, virtual, and interface methods. This information is actually dependent on the JVM implementation. Table 4.3 shows the bytecodes with references, and how they relate to information in other classes. Access to information inside a class inside only to non-bytecode information. This limitation prevents deadlock in the bytecode resolution phase. It is possible to resolve the bytecode in one class at a time.

Bytecode reference group	***Bytecode example***	***Information type***
static fields	`getstatic`	deep – offset to static fields in class
attributes	`putfield`	deep – offset to attributes in object
virtual method invocation	`invokevirtual`	deep – offset in virtual method table
static method invocation	`invokestatic`	shallow – index to the class and deep – offset in static method table
interface method invocation	`invokeinterface`	shallow – interface index and deep – reference to method descriptions
type check	`checkcast`	shallow – index to class
object creation	`new`	shallow – index to class (during conversion) (and object size (deep) during execution)

Table 4.3 *Some bytecodes require information located in other classes.*

4.2.6 Verification

The verification part is thoroughly specified in the JVM Specification in order to avoid malignant or malformed classfiles. It is divided into the following four passes:

1. **Classfile structure check**. The contents of the classfile must be recognizable and correctly formed.
2. **Further classfile analysis**:
   2.1. **Type check of symbolic references**, e.g. the superclass reference has to be of a class type. This pass does not check the symbolic references in the bytecode, nor does it check if the symbols can be resolved.
   2.2. **Semantic check on keyword final**. This pass also ensures that final classes are not superclasses of other classes and that final methods are not overridden.
3. **Data and control flow analysis**. Every method is checked for:
   3.1. **Method end**. The code must not end in the middle of a bytecode; neither must it fall off the end of the code.
   3.2. **Type check of operands**. Every bytecode must utilise operands of the correct type. The stack, the local variables, and the symbolic references have to contain the corresponding operand type.
   3.3. **Branch check**. Every branch in the code must land within the method on a new bytecode. Exception handlers have to be correctly limited.
   3.4. **Overflow and underflow check**. Accesses to local and stack variables must be within the stated limits.
   3.5. **Stack outlook**. For every position in the code, the stack has to contain the same types independent on which code path taken to reach the code.
4. **Last checks**:
   4.1. **Symbolic type consistency check**. The symbolic types must correspond to something real. This check may be delayed until execution (lazy evaluation).
   4.2. **Access check**. Every method call and every field access has to be accepted according to the access modifiers, e.g. public, package public, protected, and private.

The complete verification requires execution time and memory. The size of the interpreter also is increased if the verifier is added. One step of the verification process is to ensure that the bytecodes operate on operands of the correct types. The verification process must simulate the operand stack to perform this check and execute every possible code sequence in order to assure that the control flow is type safe. This type control could be simplified if the bytecodes themselves contained the type information about their operands.

4.2.7 Discussion

To enable classfile conversion in embedded systems, it is important to study where and how the memory is utilised in the class loader. The memory constraint emphasises the idea of loading one class at a time. Since the linking of subclasses depends on superclasses, it is desirable to identify the superclass hierarchy at an early stage of the class-loading phase. Otherwise, temporary data structures would be kept for an unnecessary long time. If the superclasses are loaded, the loading of subclasses can continue with class linking. The linking depends on all the references in the class template as well as on all the accesses to other templates. As they are resolved, temporary data structures may be released. If the requirements of the JVM omit reflection and dynamic class loading, other data structures may be released, and memory may be gained.

4.3 Loading converted classfiles

An optimisation of the linking procedure would be to load already converted classfiles, i.e. classfiles that are represented in the internal format of the JVM. There are two different types of converted classes: the *preconverted classfiles*, and the *heap image*, depending on how the symbols in them are handled. Table 4.4 shows the different types. Classfiles are also listed in the table.

Class type	*Class references*	*Linking*
Classfiles	strings	complex, see Section 4.1.4.
Preconverted classfiles	indices	convert indices to references, create class templates
Heap image	references	copy image to RAM

Table 4.4 *The two different types of converted classfiles are simpler and much faster to integrate into a runtime system than ordinary classfiles.*

The preconverted classfiles constructs a unique number to each class. A class reference is represented as the number of the class. During linking, all classes are loaded and all the direct references are resolved, i.e. the numbers of the classes are converted into real references.

A variant is to load an image of the heap, containing all the converted classes, instead of all the classfiles separately. All direct references are already calculated, which necessitates that the heap image is located exactly at the same location in memory as it was after having been created. If the heap is located at another memory location, every reference in the heap image has to be recalculated, i.e. an addition of an offset to every reference. The updated references reflect the physical locations in the memory. The offset is the difference between the physical location of the heap and the presumed heap location of the heap image. To support the localisation of the references, the class image must contain information of the whereabouts of the references. Since every reference is stored inside

objects, the problem is to identify all the objects, and update the references in them. The references in an object are described in the GC information that is stored in the template of the object. Together with the object size and the location of the references in the object, the objects in the image could be updated sequentially. This problem is the classical issue of relocation of references during linking.

The heap image contains the converted objects, the size of the heap, a root stack, and in some cases, depending on the garbage-collecting algorithm, an object handler array. The root set contains references to live objects in the heap. All live objects can be found, directly or indirectly by the help pf these root references. Some garbage-collecting algorithms utilise an object handler array. All objects are represented in the object array, and all references refer to objects indirectly via the handler array. The root stack and the object handler array are normally located outside the heap. Figure 4.13 shows a description of an image.

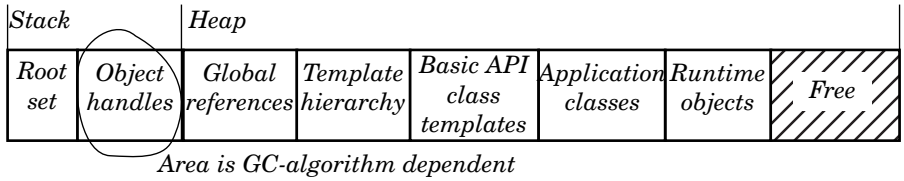


Figure 4.13 *The class image contains preconverted classes, and all other objects on the heap. The object handles are utilised by some garbage collector algorithms, and the roots contain all the roots that the garbage collector needs to locate all the live objects on the heap.*

When converted classes are utilised, it is necessary to record the class environment variant where the classes have been generated. When other classes are loaded, the linker must determine if the environment is suited for the classes. If a class is created within another API, attributes and methods could conflict with the utilised environment. Together with the converted classfiles, a *working class set* versioning system has to support the verification. However, the working class set and the preconverted classfiles are not considered crucial for the project goal. Still, they are planned as future work.

The heap image, in contrast to the preconverted classfiles, also requires a specific version of the JVM. The object layout and the garbage collector must not differ in the image or in the JVM that utilises the image.

A combination of ordinary classfiles, preconverted files, and a heap image, is possible. For example, the heap image could consist of initial global tables and the internal template hierarchy. Frequently utilised classes, e.g. the classes in the API, could be loaded and stored as preconverted classfiles. Other classfiles could be converted normally.

4.4 Split machine

A step towards executing Java on small devices is to split the JVM into one class loader and one interpreter, where the interpreter is located in a

little node computer and the class loader resides in a more powerful supervision computer. The interpreter does not need to load, link, or even initialise the classes since those tasks may be performed on the supervisor computer. However, a small linking step remains in order to install classes in the node. If the only connection with the node is via the supervisor, verification may be omitted in the node. The supervisor is responsible for verifying the code. These networks may be found in industry where many smaller nodes are connected to a supervisor computer, e.g. SCADA [SCADA] (see Figure 4.14).

If the nodes do not contain a necessary class, they require it from the supervisor. It could be possible to throw away not utilised classes and methods to save memory in the node. If they are needed, they are fetched from the supervisor. This procedure would of course lead to significant delays, but in a well-trimmed system, it could be possible to only have the "control loop code" and the interpreter in the node, which would decrease the memory utilisation significantly.

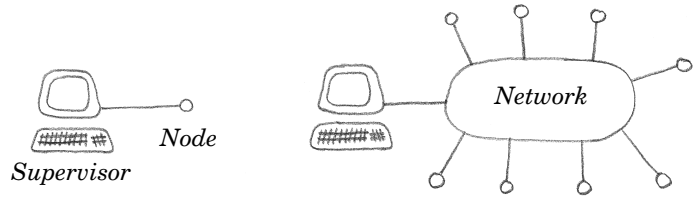


Figure 4.14 *The configuration of the split machine places the class loader in the supervisor computer and only the interpreter in the node. The class loader can support many nodes.*

Since the node is behind a secure supervisor, a number of insecure methods could be applied. Verification could be omitted in the nodes. A specialised instruction set and a specialised interpreter could be designed for a node.

The split machine incurs a different memory model from a homogeneous JVM, since the applications reside in two different memory spaces. The following sections examine the memory model and the communication protocol of the supervisor and its nodes.

4.4.1 Memory model

An important reason for splitting the machine is to reduce the memory need of the nodes. It is assumed that the supervisor has a large memory in comparison with the nodes, which could consist of inexpensive single chip systems with simple processors and small memory. To achieve this end, nodes execute an interpreter, and the supervisor takes over the rest, i.e. the class loader.

Only the classes that are utilised are necessary in the nodes. Missing classes can be required from the supervisor. Even parts of classes can be loaded as they are needed, e.g. methods. Unutilised classes and methods could be removed in the nodes. In principle, the nodes can remove all classes and methods because those are requested when they are needed. Static fields have to be stored in the supervisor before they can be

removed in the node. The supervisor can contain one implementation of a class even though many nodes are utilising it. However, the static fields are uniquely represented for every node that utilises a class. A node could be represented in the supervisor as a thread with its own working environment as explained in Figure 3.15.

As objects are sent to the nodes, the references of the objects must be converted by the class loader in the node. It is assumed that the supervisor does not know the outlook of the node's heap. The references therefore must be set by the node.

Other memory issues are to omit the symbols in the node and the field and reference descriptions from every class template. Symbols and field descriptions are only relevant if reflection and dynamic class loading is required. The supervisor could perform those tasks.

The nodes utilise a stripped environment as well as reduced class descriptions, with focus on runtime instead of maintainability. Other optimisations the split machine enables are that interpreters can be modified to suit the hardware architecture of a specific node, and that the bytecode instruction set can be modified in order to suit the memory-limited environment of a node. The size of the interpreter could be reduced if some bytecodes that are unnecessary for small systems were omitted.

4.4.2 Communication between supervisor and node

The communication between the supervisor and the node consists of object transfers. The supervisor sends the classes and other objects that are required by or requested of the node. When the node removes classes and objects, due to an almost exhausted memory, their static values are sent to and stored in the supervisor. When these removed classes or stored static objects are required, they are again transferred to the node.

When objects are sent from the supervisor to the node, they have to be linked into the runtime system of the node. The object transmission consists of four parts:

1. Locate the reference from where the object will be referred.
2. Create the object.
3. Set its value parts.
4. Set references.

The first point (locate the reference position) assumes that the location can be found in the node — the object containing the reference, for example, may not reside in the node. To simplify matters, the supervisor restricts the references to keep track on to the templates that are loaded in the node, and not on all the roots in the node. It is difficult for the supervisor to determine all the actual runtime roots in the node. One root is, however, always determinable — the environment reference that contains the template tables.

Before a sent object is created in the node, its corresponding class must be present. The supervisor keeps track on all the classes that have been sent to the node, as well as the state of each class (*skeleton* or *fleshy*). If the node removes the contents of a class template from the runtime sys-

tem, the supervisor must be informed that the removed class is only represented with a skeleton class. The state of the static fields is also sent from the node to the supervisor for storage. Templates cannot be completely removed since they contain information that is necessary for the garbage collector. As the object is created in the node, the object is linked to the template hierarchy so the object may be exposed to the GC.

After the object has been created, the value parts are sent to the node. The values are sent directly as a memory image of the object, where all the references are set to null. The supervisor and the node must have an equal object representation. If they differ, the supervisor must adapt the object design to the node. An alternative to the object image is to attach type information together with the values, so that the linker of the node can set the correct attributes that are received from the supervisor.

The last step is to set the references. Only references to existing objects may be set. The supervisor is responsible of keeping track of the objects that are installed in the node, and to keep track of all references in the node that have not been set due to the lack of objects.

An example of how three objects are transferred from the supervisor to the node is described in Figure 4.15. The program declares static fields, attributes, and how the created objects are connected. A static variable (`ourHero`) refers to an object (`captainJava`) in the object graph. The object therefore can be reached via the template table and sent to the node. Objects are transferred one at a time. Another object has a reference (`sidekick` in object `captainJava`) that cannot be set directly after it has been sent away. The referred object does not yet exist in the node. The supervisor must remember the references that are not set. After the referred object has been sent, the reference is set.

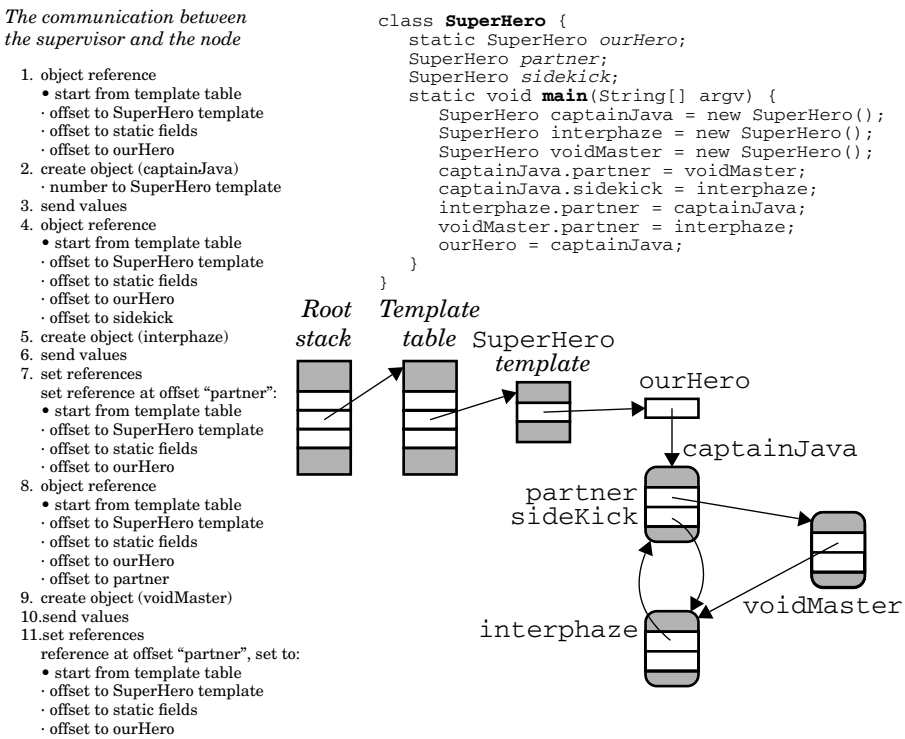


Figure 4.15 *Three SuperHero-objects are sent from a supervisor to a node.*

The internal part of the template hierarchy can also be sent. However, the automatic memory management cannot be activated until a minimal set of templates and objects are in place. The meta meta template and its garbage collector information is essential. The meta templates of the GC-information object also must be in place, together with the GC-information for the meta templates, in other words, if the GC-information is a byte array, the template of the byte array and the meta array template must also be loaded. When all the templates and the GC-information objects are loaded, the garbage collection may commence.

An optimisation to the communication is an installation of a global reference in the node. It is set by the supervisor and utilised as a cache of the last reference position. The reference position description could be shorter, which increases communication performance.

Another communication modification is to keep the static fields in the node at all times, in order to make the supervisor less complex. The node must in any case inform the supervisor about the skeleton classes.

4.5 Bytecode conversion

The specified bytecodes (see [JVM99]) are not suited for memory-limited real-time applications. The conversion of classfile bytecodes has to consider the real-time requirements, more compact bytecodes, and perform-

ance-increasing bytecodes. The real-time requirements imply an exact garbage collector and its need of well-defined element types of the stack. Some of the bytecodes utilise operands that are unnecessary large. Memory could be saved if these bytecodes were represented with smaller operand sizes. The performance-increasing bytecodes are new versions of those containing symbolic references. They are resolved into direct or indirect references. The following subsections deal with the different bytecode conversion aspects.

4.5.1 Symbolic reference resolution

The original bytecodes in the JVM specification [JVM99] describes bytecodes with symbolic reference operands. Textual comparison has to be utilised in order to determine the correct reference. Bytecodes with symbolic reference operands could be replaced with indirect references that are resolved in a determinable number of indirections. Real-time applications rely on techniques that are limited in time. Performance also would increase compared to textual resolution during runtime.

Shallow references to classes and interfaces are replaced by indices into the template table. Static fields are replaced by their corresponding indices into the static field array. Object attributes are accessed by their offset into the object. Methods are accessed by the method indices into the static, virtual, and interface method arrays. Reference constants are replaced by an index into the symbol table, where the corresponding symbol is to be found.

The bytecode groups with symbolic references are listed in Table 4.5 together with notes about the type of reference.

Bytecode group	***Example***	***Operand reference***	***Resolution***
Constants	`ldc, ldc_w, ldc2_w`	symbolic location to constant	Represent location as index into one constants array.
Attribute access	`getfield, putfield`	symbolic attribute reference	Replace reference with offset to attribute.
Static field access	`getstatic, putstatic`	symbolic class reference and symbolic static field reference	Replace class reference with index to class. Replace field reference with index to field. May increase bytecode size.
Virtual method calls	`invokevirtual, invokespecial`	symbolic method reference	Replace method reference with index to method.
Static / interface method calls	`invokestatic, invokeinterface`	symbolic method reference and symbolic class/ interface reference	Replace class/interface reference with index to class/ interface. Replace method reference with index to method.
Object creation	`new, anewarray, multinewarray`	symbolic class reference	Replace class reference with index to class.
Type	`instanceof, checkcast`	symbolic class reference	Replace class reference with index to class.
Exception	`athrow`	symbolic class reference	Replace class reference with index to class.

Table 4.5 *Bytecodes that access information in a class must be resolved. Symbolic, i.e. textual, references are replaced with indices into tables. Methods, attributes, and fields are represented as indices.*

The operands of the original bytecodes are indices into the constant pool of the classfile. When they are replaced with other indices, they may be too small to express the new index. For example, a constant is accessed with the bytecode `ldc`, which has an operand size of one byte. Only 255 indices may be expressed with this bytecode. However, if the corresponding symbol is located in the symbol table at an index that is larger than 255, the operand has to be extended. This *bytecode swelling* has to be handled by a control flow analyser.

The string constants have to be created before they can be used. This is performed as the classfile is loaded. The string instance is stored in a symbol table. The index to the string in that array is stored as an operand in those bytecodes.

4.5.2 Memory limitation bytecode changes

Embedded systems require efficient memory utilisation. There are a number of ways to reduce the memory consumption with bytecode modifications. The interpreter can be reduced in size if bytecodes are omitted, and the bytecodes can be made smaller, making the code smaller, in a system where the address space is limited. The number of allowed classes in

an application also affects the bytecodes. Some bytecodes are introduced to increase the performance. They may be expressed by other bytecodes, thus reducing the instruction set. Bytecodes could be removed until a minimal instruction set remains.

A limited heap leads to a limited address space, which incur less and smaller bytecodes. If the JVM is limited to maximum 255 classes, they may be referred indirectly by a single byte index into the template table. Many bytecodes utilise two bytes when referring to a class. For example, `new` refers to a class template. The original bytecode utilises two bytes to access the class template. There are five bytecodes with class, interface, or array references of two bytes. They are: `new`, `anewarray`, `multianewarray`, `checkcast`, and `instanceof`.

If the address space is limited, the methods cannot be as large as some bytecodes can handle. In an embedded system, the address space may be limited to, for example, 2^{16} bytes. In this case, the address space is fully covered by two bytes. Bytecodes that utilise more than two bytes in order to express an address may be reduced to three bytes. Some bytecodes have two variants, a 2-byte address operand, and a 4-byte address operand. The latter may be removed completely. For example, the bytecode `goto_w` and `jsr_w` have a 4-byte address operand.

By reducing the number of bytecodes, the memory required for the interpreter is also reduced as well. The remaining bytecodes, in some cases, may have to be more general, while in some cases larger than their original variants. Table 4.6 shows some typical bytecode modifications and how they affect the JVM at large.

Bytecode	***Modification***	***Consequence***
`putfield` op_2 `getfield` op_2	Shrink operand size from 2 to 1 byte	The maximal number of attribute positions in an object is reduced from 2^{16} to 2^8.
`putstatic` op_2 `getstatic` op_2	Use operands as class- and static field indices	The maximal number of static fields in a class is reduced from 2^{16} to 2^8 and the maximal number of classes is reduced from 2^{16} to $2^{8.}$
`wide <`*TYPE*`>load` op_2 `wide <`*TYPE*`>store` op_2	Remove bytecode	The maximal number of local variables cannot be larger than 2^8.
`invokevirtual` op_2	Shrink operand size from 2 to 1 byte	The maximal number of virtual methods in a class is reduced from 2^{16} to 2^8.
`invokestatic` op_2	Use operands as class- and method indices	The maximal number of static methods implemented by a class is reduced from 2^{16} to 2^8 and the maximal number of classes is reduced from 2^{16} to $2^{8.}$
`invokeinterface` op_4	Shrink operand size from 4 to 2 bytes	Effects are similar to `invokestatic`.
`<`*TYPE*`>const_<`*VALUE*`>`	Remove	Replace bytecode with one byte larger `bipush` op_1 (push constant integer) bytecode and add type conversion bytecode when needed. The bytecode will increase but the size of interpreter decreases.

Table 4.6 *Modifications of the bytecode instruction set may affect the size of the interpreter or the bytecodes.*

Bytecode	*Modification*	*Consequence*
<*TYPE*>load_<*VALUE*> <*TYPE*>store_<*VALUE*>	Remove bytecode	Replace bytecodes with <*TYPE*>load op_1 variant.

Table 4.6 *Modifications of the bytecode instruction set may affect the size of the interpreter or the bytecodes.*

4.5.3 Real-time bytecode changes

The real-time requirements impose modifications to the bytecode instruction set. The exact real-time garbage collector must always have control over all of the references in the machine. In particular, the stack has to be typed. Our solution to meet this end has been to split the stack into a value stack and a reference stack (see [Mag84], [KMMN91], [KML00]). Bytecodes that operate on both value and reference types have to be duplicated for each stack. For example, in the JVM specification, `pop` would operate on the top element of the Java stack. Regarding the split stack, it is necessary to specify if the value stack or the reference stack should be popped. The bytecode has to be converted to `pop_val` (value stack) or `pop_ref` (reference stack). The concerned stack-related and type neutral bytecodes are presented in Table 4.7, together with their new representations due to the split stack. Some bytecodes affect both stacks. Those bytecodes could be expressed as new bytecodes or they could be

described as a sequence of previously defined bytecodes. Such sequences are shown in the "Equivalent bytecode sequence" column.

Original bytecode	*Value stack*	*Reference stack*	*Reference and value stack*	
			New bytecode	*Equivalent bytecode sequence*
dup	**dup_val**	**dup_ref**		
pop	**pop_val**	**pop_ref**		
pop2	**pop2_val**	**pop2_ref**	**pop_val_ref**	pop_val, pop_ref
swap	**swap_val**	**swap_ref**		
dup_x1	**dup_x1_val** dup_val	**dup_x1_ref** dup_ref		
dup_x2	**dup_x2_val** dup_x1_val dup_val	**dup_x2_ref** dup_x1_ref dup_ref		
dup2	**dup2_val**	**dup2_ref**	**dup_val_ref**	dup_val, dup_ref
dup2_x1	**dup2_x1_val** dup_x1_val dup2_val	**dup2_x1_ref** dup_x1_ref dup2_ref	**dup_x1_val_ref** **dup_x1_ref_val**	dup_x1_val, dup_ref dup_x1_ref, dup_val
dup2_x2	**dup2_x2_val** dup2_x1_val dup2_val	**dup2_x2_ref** dup2_x1_ref dup2_ref	**dup_x2_val_ref** **dup_x2_ref_val** **dup_x1_val_x1_ref**	dup_x2_val, dup_ref dup_x2_ref, dup_val dup_x1_val, dup_x1_ref
putfield	**putfield_val**	**putfield_ref**		
getfield	**getfield_val**	**getfield_ref**		
putstatic	**putstatic_val**	**putstatic_ref**		
getstatic	**getstatic_val**	**getstatic_ref**		
ldc	**ldc_val**	**ldc_ref**		
ldc_w	**ldc_w_val**	**ldc_w_ref**		
ldc2_w	**ldc2_w_val**	**ldc2_w_ref**		

Table 4.7 *The table shows all original and type neutral stack related bytecodes that have to be converted due to the split stack. The representations of new bytecodes are shown. They are tagged with _ref or _val if they are related to the reference stack or to the value stack, respectively. Bytecodes typed in bold letters indicate additions to the instruction set.*

The instruction set is expanded with 39 new bytecodes that replace the 16 original bytecodes. In the IVM, the original bytecodes are reused, leaving the instruction set with 23 new bytecodes. It is possible to reduce the instruction set expansion from 23 to 16 new bytecodes if equivalent bytecode sequences are allowed, i.e. the new bytecodes are expressed as a sequence of "old" bytecodes. They are only allowed if control flow analysis is included into the converter, since the equivalent bytecode sequences increases the size of the method.

Some bytecodes have non-trivial worst-case execution times. These are:

- `invokeinterface` (linear search for class)
- `anewarray` (dependent on number of entries)
- `athrow` (difficult to estimate)
- `checkcast`, `instanceof` (proportional to deepest hierarchy depth)
- `multianewarray` (proportional to dimensions multiplied by depth)
- `lookupswitch` (search through alternatives).

4.6 Control flow analysis

The control flow analysis determines the type of the internal bytecodes and performs bytecode optimisations. Two different control flow analyses are identified: a thorough one for the split machine, and a simple one for the homogeneous machine. The consequences of the simple conversion are that it is smaller, it decreases conversion time, and it requires less temporary memory, but it omits bytecode optimisations. The thorough implementation constructs a control flow graph that is utilised in a bytecode optimiser. The optimiser requires temporary data structures for the control flow analysis. To put it simply, optimisations require memory. The control flow structure overhead could be minimised if the methods were converted one by one. However, that approach removes some optimisations, for example, method inlining. The simple control flow analyser modifies the bytecodes in the original methods, without moving them to another offset. No temporary method structures, or basic blocks are necessary, but the bytecode cannot be shrunk or swollen. Some bytecodes must increase in size if certain conditions are fulfilled. These bytecodes cannot be executed, because a bytecode is not allowed to expand in size in the converter, without control flow analysis.

Chapter 5

Results

The IVM has primarily been developed as a research platform. However, the demand of Java in embedded systems has resulted in many porting projects to enable the Java technology on many platforms. The research subjects have to be concluded in future work.

5.1 Target platforms

The IVM has been utilised in a number of projects and ported to different computer systems. The projects have mainly been related to the studying of the influence of Java in concurrent embedded systems. In one project the IVM was shipped with a product. The computer systems that the IVM has been ported to are listed in Table 5.1.

Target					***Configuration***		***OS***	***Reference***
System	***Processor***	***Clock (MHz)***	***Memory***		***Split***	***Single***		
			Flash	***RAM***				
ATMEL Evaluation board	AVR ATMega107L	8	128kb	32kb	X			[STK300]
Evaluation board, Bluecell	ARM				X	X		
Evaluation board	ARM 7	25		512kb	X	X	eCos	
Koala	68332	16	1Mb	1Mb		X		
Khepera	68000	16	256kb	64kb		X		
Palm V	DragonBall, MC68253	16	2Mb	34kb		X	PalmOS v.3.1.1	

Table 5.1 *The target platforms for the IVM.*

Target					*Configuration*		*OS*	*Reference*
System	*Processor*	*Clock (MHz)*	*Memory*		*Split*	*Single*		
			Flash	*RAM*				
Controller for IRB (industry robot)	PPC	25				X		
ABB AC800M controller	PPC 860	25		8Mb		X		
Ericsson R320						X	Enea RTOS	
iPaq	Intel StrongARM	206	32	32		X	Pocket PC	
Intel XScale	XScale	333–766	0	2Mb		X	libc	

Table 5.1 *The target platforms for the IVM.*

The projects in which the IVM has been utilised are listed below:

- Research project on ABB Corporate Research, autumn 1999. The IVM was ported to a development board from ATMEL as a preparatory study of Java.
- Research project on Ericsson, autumn 1999. The IVM was ported to a low-end mobile phone. Initiatory evaluation of Java technology in mobile phones was performed.
- Master thesis, winter 1999. The task of the master thesis work was to port the IVM to a palmtop (see [Palm99]).
- Student project, autumn 1999. The IVM was ported to the AVR development board for real-time control of a servo. The IVM was split to fit it into the small memory of the development board.
- Research project for Bluecell, autumn 2000. The IVM was ported to an ARM based computer system developed by Bluecell as an initial evaluation of the IVM as a component in a product.
- Product development for Bluecell and TAC, spring 2001. Bluecell implemented Java midlets for supervision of the control systems of TAC. The Java programs were transported wirelessly and they were based on the J2ME API and MIDP. The wireless network layer was written in C. The IVM was utilised as the JVM in the final product.
- Master thesis on ABB, autumn 2001. The IVM was integrated into the control system of ABB to support Java programs (see [TAHG02]).
- Research project, summer 2001. The IVM was almost successfully ported to a Khepera robot.
- Research project with Computer Science in Lund, Department of Control in Lund, and DIKU, Department of Computer Science at the University of Copenhagen, spring 2002. The project targeted

dynamic Java code exchange between semi autonomous systems, i.e. a control computer (Pentium and GNU/Linux) and a Koala robot [Bon02]. The communication was via Bluetooth. The Bluetooth stack was written in Java together with the control code of the Koala. The control computer also ran Java with the IVM to control a beam.

- Research on XScale, spring 2003. Thread scheduling was adapted to the different clock frequencies the XScale to minimise the power consumption (see [FG02] for more information about power consumption).

Chapter 6,

Related work

In the diverse market of Java virtual machines, there are some related to embedded systems and others are related to real-time. The API has to be adapted to embedded systems and real-time. Two competing real-time specifications have emerged in addition to our approach: the real-time specification for Java (see [RTSJ00]) and the Real-Time Core Extensions (see [JCRE]) from the J-Consortium. Another method to enable Java in real-time embedded systems is to convert the Java program to C, and add real-time garbage collection, real-time threads, and real-time scheduling. In our approach, the real-time concepts are concealed from the programmer as much as possible, while the real-time API specifications are explicitly expressed in an API.

The related work in this section describes the real-time API specifications, the embedded APIs, real-time JVMs for embedded systems, and Java-to-C conversion in real-time systems.

6.1 Java Real-Time API Specification

Real-time Java work has been focused on further specifications of the virtual machine and the API. The concepts that need to be specified more are threads, scheduling, the memory management, and the garbage collector.

Real-Time Specification for Java
The real-time Java expert group[1] has written a Real-Time Specification for Java (see [RTSJ00]), i.e. an API for Java where real-time issues are

1. The expert group consists of representatives from different companies and universities, e.g. Sun Microsystem Laboratories, Lucent, Mitsubishi Electric, Carnegie Mellon, and the University of York. The group has three parts: a technical interpretation committee, an advisory team, and the original specification development core team. More information about the specification and the group can be found at www.rtj.org.

addressed. Real-time threads are introduced with more specified execution behaviour than ordinary Java threads. Other covered topics are scheduling, memory management, synchronisation, and timers.

The difference between our approach and the solution provided by the real-time specification is that time critical threads utilise manual memory management instead of automatic memory management. The programmer is burdened with memory management and exposed to the traditional errors, i.e. dangling pointers and memory leaks. Our approach avoids these problems by maintaining automatic memory management for time critical threads. The garbage collector is scheduled as an ordinary thread. The RTSJ introduces complex concepts and lowers the simplicity of Java. For example, four new memory areas are introduced: heap memory, immortal memory, scope memory, and physical memory, where our solution utilises only one, the heap.

6.2 Java platform

There are many standardised Java APIs targeted for different system requirements and demands, ranging from enterprise computers to smart cards. An API that is called Java 2 Micro Edition, J2ME, covers the embedded systems. This API is further divided into subsections to suit specific areas of embedded systems. The other Java platforms are the enterprise edition (J2EE), the standard edition (J2SE), and Java Card. Figure 6.1 shows the different platforms. In the smaller APIs, some specifications are removed from the virtual machine. The virtual machine for the J2ME/CLDC configuration is designed for resource-constrained targets. It requires 160kb-512kb to execute. Suns implementation, the Kilo[1] virtual machine, KVM, is derived from the Spotless research project. The Card VM implements only the fundamental functionality of the JVM Specification. It omits dynamic class loading, the Security Manager, finalisation, threads, cloning, access control in Java packages, and garbage collection (an optional object deletion mechanism is offered). Only the following types are supported: `boolean`, `byte`, `short`, and `int` (optional).

1. It was so named because its memory budget is measured in kilobytes (whereas desktop systems are measured in megabytes).

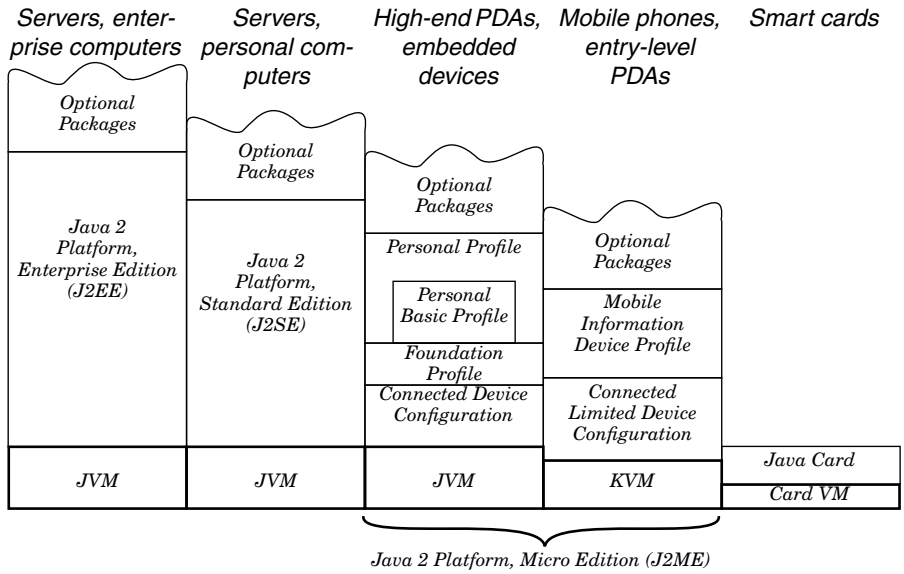


Figure 6.1 *The Java architecture consists of an API and a JVM. There are various sizes of the APIs targeted for different systems.*

Java 2 Platform, Micro Edition

The J2ME architecture is composed of a virtual machine and a minimal set of class libraries that are designed to provide the functionality for a particular range of devices, sharing similar characteristics. Currently, there are two J2ME configurations: the Connected Limited Device Configuration (CLDC), and the Connected Device Configuration (CDC). CLDC is smaller than CDC and designed for devices with slow processors, limited network, and limited memory, for example, mobile phones and PDAs. Further, it addresses core Java libraries (java.lang and java.util), input and output, networking, security, and internationalisation. The following areas, though, are not addressed: application management (installing, launching, and deleting), user interface, event handling, and interaction between the user and the application. In addition to this, the float and double data types are not supported and error handling is limited. CLDC can be utilised on systems with a minimal memory of 128kb to 512kb. CDC targets devices with a minimum memory of 2Mb — for example, high-end PDAs and communicators.

On top of the fundamental API, it is possible to add optional packages, or profiles. For example, in the CLDC there is a Mobile Information Device Profile (MIDP) that is designed for mobile phones. It contains a user interface, network connectivity, local data storage, and application management. CDC, on the other hand, contains profiles with more functionality. The foundation profile is the basis of the personal profile, PP, (support for full GUI, and applets), and personal basis profile, PBP, which is a subset of PP. Other optional packages are developed for Bluetooth, web services, wireless messaging, multimedia, and database connectivity.

The CLDC relaxes the JVM Specification at the following points: no floating-point data types (float and double), no Java native interface (native code is linked directly into the JVM), no Java level class loaders, no reflection, no thread groups or daemon groups, no finalisation of class instances, no weak references, and limitations on error handling.

6.3 Java virtual machines for embedded systems

On Sun Microsystems many embedded virtual machines for Java have been developed. The embedded API is specified and a real-time specification has been developed. The embedded virtual machines from Sun are considered in a separate subsection, while other embedded JVM implementations are listed in another subsection. Existing real-time JVMs are listed and described in a following subsection.

6.3.1 Embedded JVMs from Sun

The JVMs from Sun that addresses the embedded systems have been developed from research platforms (JavaInJava [JIJ98], and Spotless) to the commercial variants (KVM and CLDC HotSpot Implementation VM). At this date, no official attempt to establish real-time Java in embedded systems has been performed by Sun.

Spotless system
The Spotless system is a JVM and a class library implemented by Sun Microsystems Laboratories, suited for small systems, e.g. palmtops and embedded systems [JDP]. It manages dynamic class loading, the complete bytecode set, garbage collection, and multi-threading. The goal of the Spotless project is to create the smallest possible "complete" JVM. The Java Card and Embedded Java are subsets of Java. They do not support, for example, dynamic class loading, garbage collection. The creators identify program criteria for consumer device manufacturers as portability, and fast learning curve for developers, which is well in line with the philosophy of standard Java.

The functionality of the Spotless is a straightforward JVM without memory-consuming optimisations like just-in-time compilation. However, quick bytecodes are included to increase the performance of the runtime system. The major memory consumption is, however, the classfiles and the classfile library. A small subset of the standard non-graphical classes has been implemented without redundant methods and classes that are seldom utilised. The number of exceptions is kept at a minimum. Folding them together has reduced dependencies between classes. The Spotless JVM starts with nothing and add only what is necessary, rather than starting with the complete JDK and removing what is not needed.

The internal class structure is similar to the IVM constitution. However, the garbage collectors do not seem to be incremental, and the thread switching seems to depend only on bytecode counting. The native methods are included in the machine instead of implementing the large Java Native Interface [JNI99].

The first target system of the Spotless was a Palm. The size of the Spotless JVM is 30kb - 50kb depending on platform and debug information. It performs 30% - 80% of the JVM in JDK 1.1 (without JIT compiler). The size of the needed heap is tens of kilobytes. The code is and consists of 14000 lines distributed on 25 C/C++ files. The code is written to be readable (well commented) and easy to understand.

It is unclear if it is possible to download Java programs that are not known beforehand, for example, Java MIDlets.

KVM

The first commercial JVM for embedded systems developed by Sun is the KVM. The size of KVM is between 40 - 80kb depending on compiler options and target platform. A normal application would use about 256kb, of which the heap is half (128kb).

A pre-verification tool that is typically run on another machine before the classfile arrives to the VM in the embedded system supports verification. Classfiles are given an extra "stackmap" attribute to support fast, and memory effective verification (5% larger classfiles).

A "ROMiser" tool for converting classfiles into a format that is directly linkable in the virtual machine has been developed, reducing the start-up times considerably. The tool is named Java Code Compact, JCC. The result is a C code file that is compiled and linked together with the KVM.

CLDC HotSpot Implementation Virtual Machine

The latest addition to the JVM from Sun is the CLDC HotSpot Implementation Virtual Machine, which incorporates a JIT compiler. The optimisations execute an order of magnitude faster (10 - 20 times) than the interpreted variant. The compiled code uses four to eight times as much space as the original bytecodes. Only the most frequently used parts of the application, i.e. the *hotspots*, are compiled to keep the memory consumption at a low level. A statistical profiler keeps track on the number of times a method is executed. The basic optimisations of this one-pass JIT compiler are: constant folding, constant propagation, and look peeling. The target platforms are 12-32MHz processors with at least 512kb memory, i.e. mid- to high-end mobile phones.

Java Card VM

There exists a Java Virtual Machine specification for small-memory embedded devices (see [JCVMS]), e.g. smart cards, together with a shrunk API, and a small runtime system. The typical application is an 8-bit or 16-bit processor architectures with 1.2kb RAM, 16kb EEPROM, and 32kb ROM. The Java Card Application Interface (see [JCAPI]) is a small subset of the java.lang package with added functionality for security and communication between applications, Java Card Applets. The runtime system is specified in the Java Card Runtime Environment Specification (see [JCRE]). It explains how to utilise Java Card Applets and how they can interact with each other.

The JCVM Specification restricts the applications to be single threaded, thus removing some functionality of the language and invalidating the java.lang.Thread class and related classes. Dynamic class load-

ing and garbage collection is not supported. Finalisation of objects is omitted. Further, the security manager in the J2SE API is removed. An applet *firewall* is introduced to keep objects unreachable from other applications. Cloning is removed, and native methods are not permitted. The following types are removed: char, double, float, and long. Integers could optionally be removed. Only one-dimensional arrays are supported. However, exception is fully supported. Only a few exceptions and errors of the virtual machine are not included from the J2SE specifications.

Classfiles are considered to large to fit into the environment of Java Card. Instead, a Java Card Converter collects classes in a package into the Connected Applet format (CAP). Symbols, i.e. the CAP interface, in the CAP are placed in an *export* file. Linking of many packages is resolved with the aid of the export files. CAP files are loaded into a terminal, typically a "powerful" computer that prepares the applet that is executed by the virtual machine of the "card" computer.

Since the target platform has a limited address space, many bytecodes are unnecessary. Every class is limited to 256 static methods and 256 virtual methods (2 different types * 128 methods per type). Every class may only be instantiated 255 times, and the maximum length of a method is 32767 bytes.

These restrictions are well in line with those of the IVM. However, the aim of the IVM is to add multi-threading and garbage collection to the same target platforms as those of Java Card.

6.3.2 Embedded virtual machines

Many Java virtual machines, like the KVM, are targeted towards embedded systems. Typical features of those are that some features of the JVM Specification are not supported to make the machine smaller. The supported API is often a subset of the J2SE API or the J2ME/CLDC PAI. Applications tend to focus on concurrent systems, and not hard real-time systems.

SimpleRTJ
The simpleRTJ (see [SRTJ]) is a Java architecture for concurrent memory limited embedded system without a RTOS. The footprint of the system can be 17-19kb with almost all features of Java. Floating-point types are optionally included in the runtime environment. The classes of a Java application are bundled together by the classlinker and linked with native code and the simpleRTJ source files to form a native code binary image that can be put in the embedded system. The JNI is not supported. Many classes from the standard libraries are supported. SimpleRTJ is developed by RTJ Computing Pty. Ltd. in Perth, Australia.

WABA
An attempt to migrate Java for embedded systems is the open source Waba virtual machine (see [WABA]). The Waba classfile and bytecode format are strict subsets of the classfile and bytecode format supported by Java. The primary target platform for Waba has been PalmOS, but it has been ported to many different operating systems.

The Waba language, virtual machine, and classfile format were designed to be for small devices. Features that would use substantial amounts of memory or that were deemed unnecessary for small devices were omitted from the design of the Waba language and platform, e.g. exceptions, floating-point arithmetic, and limited address space (32kb). Waba has a rewritten subset of the core functionality of the J2ME/CLDC API, with some simple user interface routines. Waba programs may run in a Java environment, however, the contrary is not always the possible.

Waba was designed for small, usually mobile, devices. Waba virtual machines are available that are under 72.3kb in size (including foundation classes) and that run programs in less than 10K of memory. The VM takes 40KB of executable code on Motorola 68K processors, and about double that on a Pentium.

A more powerful Waba variant, Superwaba (293kb with classes and VM) executes faster and contains a larger subset of Java.

6.3.3 Real-time Java Virtual Machines

These machines are focused on achieving hard real-time behaviour.

6.3.4 JamaicaVM

This JVM implementation is made by Aicas real-time in Karlsruhe (see [SW]). Their JVM is suited for hard real-time applications in embedded systems. The *program builder* takes a set of classfiles and produces a C source code file that is compiled by a C compiler to an object file that is linked together with other Jamaica VM files into a single executable. The builder can run in a *smart* mode that omits symbols, and unnecessary code, i.e. this selective compilation removes code that cannot be utilised during runtime, at the expense of reflection, and dynamic class loading. The classfiles may be transformed completely to C code, or to memory efficient intermediate internal bytecodes that are interpreted. The C code executes faster than the bytecodes, however, it is more spacious. The bytecodes are linked with an interpreter to form an executable. The Jamaica VM identifies classfiles as the largest memory consumer. Compaction of classfiles reduces the size of classfiles up to 50%, and smart linking up to 90%. Just-in time compilation is completely omitted since it takes to much time to convert the code, which results in a pessimistic real-time analysis.

Jamaica VM supports J2SE (however, not all the classes have been implemented). Native methods are interfaced by the relatively memory consumptive JNI 1.2 (added optionally) or by their own and more compact Jamaica Binary Interface, JBI.

A simple “HelloWorld” program (62 Java bytecodes) compiles to a 150kb executable on a PowerPC. If smart linking is activated, the application shrinks to 130kb, of which the VM is 120kb. The RAM utilisation for is 260kb, of which 1,8kb are utilised as a heap by the application.

The real-time is supported by an exact and incremental real-time garbage collection and profiling tools. All threads are real-time threads. Any higher priority threads are guaranteed to preempt lower priority threads within a fixed worst-case delay. The profiling tool determines the worst-

case execution is given for any code, by counting the number of executed bytecodes in every method. The memory analyser determines the memory consumption. It finds the amount of memory that is actually used by the application and how long time the maximal time the GC will interrupt the application. A larger heap reduces the, the smaller worst-case GC execution interruption of the garbage collector. The exact size of the heap and the number of threads, but also the classfile API affects the size of the executable. The parameters are given to the Jamaica compiler. Threads are preemptied after a set number of intermediate instructions, i.e. a more efficient form of bytecodes. It makes the preemption mechanism simple to port. Native methods do not affect the real-time behaviour. We assume that the native code may be preemptied in the same fashion as the compiled Java code.

6.3.5 PERC

Kelvin Nilsen and his company NewMonics put aside the Java certification process from Sun, in favour own ideas. NewMonic's Java environment is named PERC and it consists of a development environment besides the JVM and the classfiles (J2SE). PERC can handle hard real-time with the support of an incremental, copying, and exact real-time GC. The systems consist of approximately 50000 lines of code. Optimising ahead-of-time and just-in-time compilation is supported. The development environment supports debugging, simulation, and performance tuning, e.g. maximum number of heap allocation regions, and CPU time dedication to threads. ROMised bytecodes are also supported. It seem that the target architectures (ARM, Intel X86, XScale, ARM, MIPS, and 68K) are in the more advanced end of the embedded systems market.

6.3.6 JBed

A full-featured real-time JVM for embedded system is the JBed from Esmertec (see [JBed], and [JBed99]). JBed supports both ahead-of-time compilation and classfile conversion during runtime. Real-time is achieved by compilation to the platform before execution. A step towards the real-time specification for Java has been taken with *Tasks*. They are a subset of ordinary threads, but add real-time characteristics. A task may be specified to complete within a certain deadline, or utilise a specified amount of CPU time. The scheduler implements an earliest deadline first scheduling algorithm [Pil00].

J2ME/CLDC 1.0 and MIDP 2.0 are supported. JBed, in its static form, and the CLDC classfiles occupy 210kb on an ARM7 code.

6.3.7 Kertasarie

A german JVM implementation, called Kertasarie, claims to handle real-time in embedded systems. The Kertasarie VM implements its own native threads based on the OS threading model (green threads). Priority inheritance is implemented to avoid priority inversion. The machine occupies typically 60-80kb and the API occupies about 200kb (ROM or RAM).

Another 100-200kb RAM is needed during execution. Classes may be "preloaded" to increase the start-up time. Many parts of the machine are modularised and can be selected or removed from a VM. Hard real- time issues have yet to be proved for the Kertasarie JVM.

6.3.8 Summary

Spotless and JavaInJava is remarkably simple. The idea of describing a JVM in Java and then generating C/C++ code of that code is appealing. Object layouts could be suited for the platform, e.g. a fast but memory consumptive 8-bytes aligned offset to every attribute, or a compact byte alignment of object attributes. However, the real-time aspects are lacking in Spotless, but Spotless would be well suited for real-time modifications.

PERC is complex. The real-time aspects are complex and require extensive knowledge of the runtime system. The simplicity of Java is lost.

• VMs from Sun are targeted for embedded systems without real-time requirements. They leave a large embedded system segment between 1-8MHz, 32kbROM, and 10-32kb RAM. The Java Card VM is too limited to provide high-level features. However, there it is relevant to the industry to be able to write high-level code for that segment.

6.4 Java to C compilation

Many real-time applications are written in C (or a "safe" subset of C). To achieve real-time functionality, concepts of threads, scheduler, and predictable execution time and predictable memory utilisation are added. Java provides a standardised way to utilise threads. Scheduler implementation, however, is not standardised in Java. Memory management is standardised in Java with the garbage collector. Memory analysis tools have to be added to guarantee the memory utilisation.

A Java to C compiler could enable the benefits of Java to the programmer while maintaining the traditional real-time language community. This approach is utilised in the JamaicaVM, and the PERC system described in Section 6.3.4, and Section 6.3.5. Another approach is the Java2C compiler written by Anders Nilsson, [NE01]. Java2C converts the Java code into C and adds garbage collection and a predictable kernel, i.e. threads and scheduler. A hard real-time kernel has been developed by Torbjörn Ekman [Ek00], for the AVR processor. The binary code had to be analysed by hand to produce the WCET. The GC implements the Garbage Collector Interface, GCI, which enables different garbage collectors to be utilised without changing the generated C code.

A preliminary performance estimation of the Java2C execution and the IVM reveals 5 to 10 times faster execution of the generated C code.

Chapter 7

Future work and conclusions

In this work, a foundation has been laid for further work. Even though the principal objectives, i.e. merging real-time and high-level programming languages, have not been proven, no obstacles have appeared to prevent the merge either. Future work, however, will conclude the integration after more work has been done on the WCET analysis.

Many other interesting topics related to the IVM have appeared during the implementation of the IVM. The IVM serves as a good foundation for further explorations into the real-time domain. The following subjects are discussed as future work: how to adapt the IVM to real-time, more general real-time issues, and optimisations of the IVM. Interpretation and compilation that run together in embedded systems could be studied to determine if the benefits form the domains are possible to join. During the development of the IVM, many interesting topics were discovered. Code replacement during execution of real-time applications would be an interesting continuation of the real-time IVM. The Meta Virtual Machine, MVM, could be created to describe a JVM and to generate the machines. The minimal language is a spin-off to describe a language that is portable by itself. At an architecture level, the split machine presents an intriguing situation where nodes could communicate with each other.

7.1 Real-time adaptations

To achieve predictable behaviour of the virtual machine, the following parts must be implemented:

- Predictable bytecodes – all bytecodes in the machine must have a WCET.
- Code execution analyser – the control flow must be analysed together with the memory utilisation.
- Predictable scheduler – the scheduler must be predictable and implement a predictable scheduling algorithm.

- Real-time garbage collector – the garbage collector must be predictable.

The three first parts have been implemented in the IVM. However, the current garbage collector is not completely adapted to real-time. Real-time analysis remains to be implemented before WCET can be determined for the bytecodes and the scheduler. The code execution analyser is implemented but not integrated into the machine.

7.2 Real-time code replacement

Preliminary work has been conducted to replace code during execution in the IVM. With a real-time virtual machine, as stated in the previous section, the code replacement could be performed during execution of real-time programs. Two approaches of code replacement are considered. First, the fine-granular approach replaces segments of code by, for example, adding a "backpack" of the new code, and inserting a method call where the code is replaced (see [BM83]). The other approach is to exchange complete modules, plug-ins (see [JDP]).

Code replacement requires another computer and a network to communicate the new code to the real-time machine. The network does not have to be deterministic in the sense that information is transmitted within a determinable period. However, together with a real-time machine, the performance may probably be increased with a deterministic network.

Another related issue is to study the real-time behaviour of a system that could download binary code as well as bytecodes.

7.3 Interpretation and compilation co-operation

Interpreted code and complied code could co-exist in the same application. Time critical code sections could be compiled and the other code could be interpreted to save space. For example, in a real-time application, a control loop could be compiled to increase the performance of the control system. The other code could be left interpreted. However, the compilation would be performed offline and not in a JIT fashion because JIT compilation often require vast amount of memory. Interpretation is often burdened with slow performance, but it often has smaller method sizes, and benefits from portability. Compiled code often requires more space, performs better, and is difficult to port to other systems.

The Java2C-compiler developed by Anders Nilsson (see [NE01]) is especially interesting to study in the relation to the IVM since the object design is the same, and the interface to the garbage collector has been developed in co-operation.

7.4 Optimisations

There are many sections of the IVM code that should be optimised for both speed and size. Development of the machine has been concentrated upon basic functionality, i.e. to get the machine to work. When optimisations have been performed, the size has always been considered before speed.

The following list is a selection of optimisations that are interesting to implement and study:

- Exception optimisations – the machine require many exceptions that are similar in structure. Since they are similar, they could be handled differently to decrease the memory consumption. For example, exception classes could be created, as they are needed, to save memory. Real-time related exception classes should be generated before the entrance of the real-time loop. In the split machine, the necessary exceptions should be loaded on demand.
- Compressed file systems – the classfiles consume significant amount of memory. Compression could reduce the size of the classfiles significantly. Many Java systems have already adopted compression to their classfiles (see [SRTJ] and [WABA]).
- Remove arrays with zero elements – currently in the IVM there are many zero sized arrays. Memory could be saved if these zero sized arrays would be removed by, for instance, a check everywhere an array is accessed. Measurements should be performed to show how much is gained by removal of zero size arrays.
- Organise global data structures – the final size of global tables is determined when all the classes have been loaded. To support tables with variable size, they could be implemented as a linked list, or as small arrays linked together. However, the tables are extensively utilised during class conversion that renders linear search algorithms inefficient. To improve the table access, it could be implemented as a hashtable, or as another performance increasing data structure. The penalty would be larger memory consumption.
- Bytecodes optimisations – since it is possible to modify the internal bytecodes arbitrarily, it would be interesting to study variations of the bytecode instruction set, or a combination of bytecodes and binary code. Platform specific bytecodes could utilise hardware more efficient and a reduced instruction set reduces the size of the virtual machine.
- Interface method optimisations – analyse the interface methods to find a faster bytecode variant for real-time embedded systems. The class linking could try to create interface arrays where the interfaces are located at the same offsets — the interface array could be extended to hold all the interfaces. Every interface would then acquire a unique offset that is utilised to locate its method array independently of which class template is utilised. Another approach to improve the performance of interface method calls is to *guess* the interface location in the interface method array by adding a guess

operand to the interface bytecode. A good guess is the last interface index that was found. That approach, however, has as bad WCET behaviour as the worst case is a linear search for the method array. As bytecodes are changed in this approach, they must not reside in ROM. It would be interesting to study different approaches to the interface method localisation.

- Inlined class templates – the information in the class template is consists of many references to other data structures, e.g. the virtual method table, and the constant table. These data structures could be inlined in the class template itself to increase performance. Memory consumption may even decrease with inlined data structures because the number of objects, and object heads, is smaller.
- Other optimisations techniques – ordinary optimisation techniques should be implemented and evaluated. For example, caches for field and methods, constant propagation etc.

7.5 Measurements

Performance and memory measurements should be conducted in depth on the IVM. Interpretation is not as fast as compiled code, but it may be more memory efficient. The performance bottlenecks and the benefits should be pinpointed. Different target platforms and the two machine variants should be utilised in the tests.

An in depth study of the performance (speed and memory) of the virtual machine would make clear the bottlenecks and the advantages with interpretation. For example, how much performance is lost by the memory efficient virtual method calls in the bytecode, or how much memory is gained by removal of floating-point types and long integer types (32- and 64-bits). How much space is saved by the symbol table?

Since the memory requirement of the IVM is crucial, it is interesting to get a grasp of the classfile sizes and the sizes of the contents of the classfiles. The classes of the following APIs should studied: J2SE, J2EE, and J2ME. More information about the APIs can be found in [J2SE], [J2EE] and, [J2ME], respectively. The measurements on the classfiles concern size of constant pool, size of fields, methods etc. How many symbols can be reused? How much of the symbols are only utilised as reference and descriptive purposes? How much is gained if only the necessary parts remain?

The sizes of the data structures in the machine should be measured in detail. The split machine should also be taken into account. Its internal structure differs from the homogeneous machine. Ideally, only the necessary data structures should reside in the split machine.

7.6 Meta virtual machine

A meta virtual machine, MTM, describes and generates virtual machines. There are many different requirements for different platforms. In the IVM, the port specific information is collected in a module that is modified

according to the specific platform. However, there always arise new demands and desires that are not thought of in the original design. It is desirable to test and evaluate new ideas quickly. For example, object layout could implement four-byte aligned data, which is desirable in 32-bits architectures, or one-byte aligned attributes in 8-bits processors.

New ideas could be implemented and tested without thorough knowledge of the virtual machine code. A future project idea is to implement a meta virtual machine and use Java as the description language (see [JIJ98]). To further describe the machine in a more expressive way is to utilise a symbol language, e.g. the notation language described in [Ive98.3]. This section discusses and presents design ideas for a MVM implementation in Java.

7.6.1 Template and class structure

It is essential to describe the internal structure of the JVM. This could be done in MVM as Java classes. The internal data structures that have to be described are the template structure of the IVM. It consists of:

- Classes
- Arrays
- Interfaces
- Primitive classes
- Instances of classes, arrays, interfaces, and primitive classes
- Methods
- Method activations (or instances)

The layout of the classes in the generated machine should be described by classes in the MVM. The attributes in a class of a generated machine should be described as attributes in the MVM.

7.6.2 Object layout

The object layout of a virtual machine should depend on the underlying processor architecture. In addition, different requirements such as memory utilisation, performance, and hardware considerations, e.g. mapping of an object to hardware port, may affect the object layout. The MVM could describe the different object layouts in an object-oriented fashion. Figure 7.1 describes three different object layouts: objects with four-bytes aligned attributes, one-byte packed attributes, and an object layout that maps a hardware port. To map a hardware port directly may avoid translation overhead, as data in one representation must be converted to data that suits another representation. For example, an IP-packet may be mapped directly onto a Java object.

Some processors utilise different reference sizes, e.g. 16-bits references or 24-bits references. If the address space is four-bytes aligned, there is only need for 14-bits to fully cover a 16-bits address space. Reference sizes

should be reflected in the object layout. In Figure 7.1, 2-bytes and 4-bytes references are also taken into consideration.

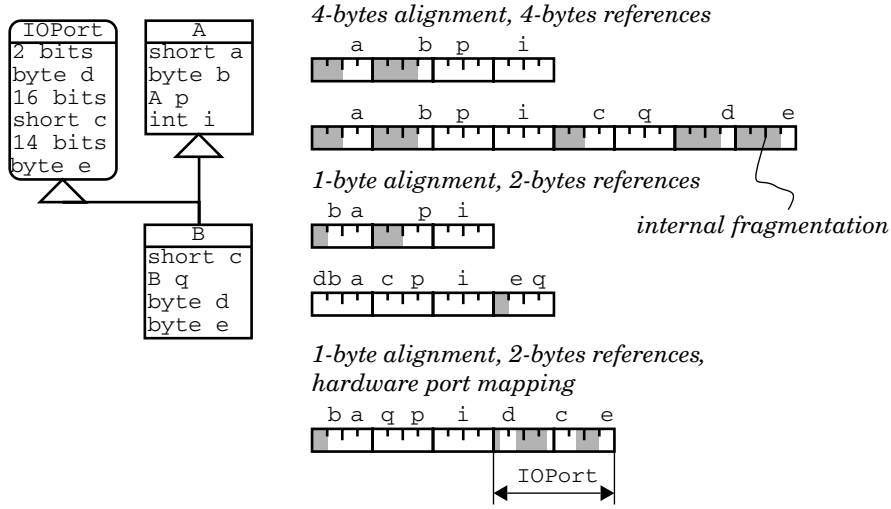


Figure 7.1 *Different object designs would easily be expressed in the meta virtual machine. The figure contains three different object designs. The variants are: four-bytes alignment, one-byte alignment, and hardware specific considerations.*

Other requirements may also influence the object design. In a system with a small memory, a non-moving garbage collector could be considered, even if it results in fragmentation. An object consists of an object head, e.g. GC-information, a template reference, and synchronisation information. If the memory is limited to 256 memory blocks, a reference could be expressed with only one byte. In those systems, an object head could consist of three bytes: template reference, GC-information, and synchronisation information, e.g. an index to a monitor pool, or a direct reference to a monitor object. If the block size is four bytes, the remaining byte could be an attribute of reference type, a boolean type, or a byte type. The complete memory area would then be 1kb. A more realistic block size would be eight bytes, resulting in a 2kb heap. The VM could be equipped with a minimal interpreter optimised for an address space covered by 8-bit addresses. Drastic modifications like the minimal object design could be performed with the MVM. The effects could directly be compared with other alternative object designs.

7.6.3 Code generation

The MVM could be designed to generate C code for those segments that would benefit from faster execution. Primarily, the MVM could be executed as a Java program. However, a minimal amount of C code generation from the MVM code is necessary to bootstrap the generated virtual machines. As the core code for the virtual machine is generated, the rest could execute as bytecodes. One advantage of a two-language representation of the same description is that a comparison would reveal bottlenecks.

The generation of the core VM code could also be expressed in the MVM by adding descriptions of the code generation of the core. In the generated VM, the interpreter's bytecodes could be specially designed for a specific instruction set, e.g. a limited address space, or a digital signal processor. Different interpretation algorithms could be expressed without too much work. Optimisations, such as inlining, could be added. Virtual machines could easily be generated and compared. Another code generation example is how the interpreter should be expressed in C code. One direct approach is to implement the interpreter as a switch statement, where each case alternative corresponds to a bytecode. There exist more efficient interpretation methods, e.g. by storing addresses as bytecodes, where the bytecode is implemented as a code sequence and located by a simple jump-instruction (the bytecode).

Another code generation consideration is to map the stack and the local variables on processor registers to increase performance. Bytecodes have been mapped to registers in a JIT compiler (see [ACL98]).

The interface to the code generation should be interesting to generalise. The design of an interface that accommodates the different requirements is an interesting problem.

7.6.4 Summary

A Java virtual machine generator could be a solution to test and compare the different requirements and optimisations that has emerged during the development of the IVM. Platform specific requirements could be obliged in a safer and more controlled way. A test suite could generate comparison tests of the generated machines. To raise the abstraction layer from the current C-code of the IVM to Java would result in code that is more secure and reduce the debugging time significantly. Most of the debugging sessions of the IVM revolve around errors in the hand coded garbage collector interface.

The interface to the machine description should be open and sufficiently simple to implement modifications. The meta virtual machine addresses these problems.

7.7 The minimal language

Another approach to platform independence is to create a bootstrap language that is implemented in the language itself and that require as small platform support as possible. The platform specific parts should be simple to port to any other language that is supported by the platform. When those initial parts have been compiled, the bootstrap procedure begins. The language should be able to build itself.

A direct approach would be to express the language on basic assembly level operations, for example, load from memory, store, and perform a calculation.

7.8 Real-time issues

Schedulers in C and Java
It would be interesting to study the difference between schedulers implemented in Java and in C. Java schedulers benefit from a Java interface to the scheduler that enables simple scheduling modifications. Schedulers in C, should be faster, but must be compiled together with the machine. The performance, time of context switching, and the size should be compared. Threads would also be interesting to implement in Java and in C.

A scheduler is responsible for the execution in its environment. Other schedulers may be instantiated inside the environment of another scheduler. The combination of many schedulers in one application would be studied in deeper detail. Communication between real-time environments within one application has not been studied in detail, to my knowledge.

Native threads co-existing with Java threads
It should be interesting to study threads in Java that are mapped to the thread handling routines of the underlying operating system. Benefits from, for example, a real-time kernel could be integrated into the VM. A combination of native threads and threads handled by Java should be interesting to study in detail.

Native threads require a stack for every thread. The WCLM size of the stacks is relevant to determine. A study of the native thread memory utilisation could be integrated into the memory analyser, if the code is annotated.

Predictable C-stack sizes
The size of the C-stack may vary during execution, especially if native methods are utilised. A predictable VM must be able to keep track on its WCLM of the C-stack. If native methods do not allocate dynamic memory, or call other methods outside the machine, the memory consumption of the C-stack can be determinable by profiling. To enable recursive native method calls, a more thorough analysis has to be performed to determine the WCLM for the C-stack. With some annotations about the memory behaviour in the native C-methods, a memory profiler can provide the WCLM.

Different context switching points
Currently the context switch may be performed after the execution of a bytecode. Other context switching points would be interesting to study, for example, after each line of source code ([SIM89]), after the execution of a method invocation and backward jump, or in specific context switching bytecodes that are inserted by the class loader. The execution of a given number of bytecodes before a context switch would enable the benefits of RISC architectures. The registers in the processor do only need to be written back before a context switch.

To allow preemption everywhere during execution, even in the middle of a bytecode, is another interesting approach. The IVM could serve as a test bench for preemption, and it could be compared with other context switching alternatives.

Real-time application debugging
Real-time applications often tend to be more complex to debug than ordinary applications. One problem is often to reproduce the error. With the IVM, a debugging context switch could be performed after every bytecode to increase the predictability of the multi-threaded program. This extreme thread switching could also put pressure on the functionality of the application. Some real-time errors could be forced to appear and repeated with this kind of extreme context switching.

Memory efficient synchronisation
The Java Language Specification states that every object should have a lock. However, in embedded systems, the locks take considerable memory space, and not all the locks are utilised during runtime. An idea is to circumvent the unnecessary memory consumption is to give the impression that every object has a lock. Only the necessary objects are equipped with locks. This can be achieved in many ways:

- Lazy-evaluation – create locks as they are needed. This approach is time-consuming, and burdens real-time applications.
- Lock pool – create a limited amount of locks that are reused. This is time efficient since all locks are created during the start of an application. However, the amount of locks may be difficult to determine.
- Static analysis – the application is analysed before runtime and the necessary locks are created. The ability to download new classes is prohibited with static analysis. Code outside the analysis may utilise objects as locks that are not determined as locks by the analyser.
- Dynamic analysis – give every thread its lock that the thread utilises to lock objects (see [Blo00]).

It would be interesting to study the efficiency and the memory consumption for the different approaches.

WCET analysis
The determination of the worst-case execution time for a Java program should be performed by addition of all the WCETs for the bytecodes in the most time-consuming execution path of the program. The bytecodes execution times are calculated on a deterministic processor by adding the binary code for each bytecode. The WCET of a bytecode must also incorporate the execution of read-and-locate the next bytecode.

After the execution every bytecode, the interpreter checks if there is a pending context switch, and if so, the active thread is rescheduled. The WCET for the scheduler has also to be added in the scheduling analysis. The scheduling analysis determines if the application is schedulable.

Real-time garbage collector
The exact and incremental RTGC is scheduled as a thread. Higher priority threads can interrupt its execution. Lower priority threads are not considered time critical. They are executed after the high-priority threads have had their memory allocations managed by the GC-thread. The mem-

ory management of low-priority threads are performed incrementally as they occur in the code, while high-priority threads only performs minimal memory management work when they are running.

Real-time analysis feedback to the programmer
The real-time analysis could be included in a tool to provide a programmer with feedback of the real-time analysis. The execution time of a code sequence could be shown and utilised in a scheduling analysis. A real-time expansion of an existing incremental development tool would be preferable, e.g. eclipse (see [Eclipse]) or applab (see [Bja97]).

The tool should also show the worst-case live memory. The program has to be annotated with *memory comments* to support the WCLM analysis. As the annotations are changed, the memory analysis is performed again.

Periodic jitter
The preemptive context switches in the IVM are performed only after the execution of a bytecode. The time to finish the bytecode execution imposes an extra time overhead to take into account during scheduling analysis and in the control loop. The occurrences of the periodic jitter for two threads are depicted in Figure 7.2. The figure also shows a presumed distribution of jitter times, where the worst-case jitter time marks the execution of the longest bytecode.

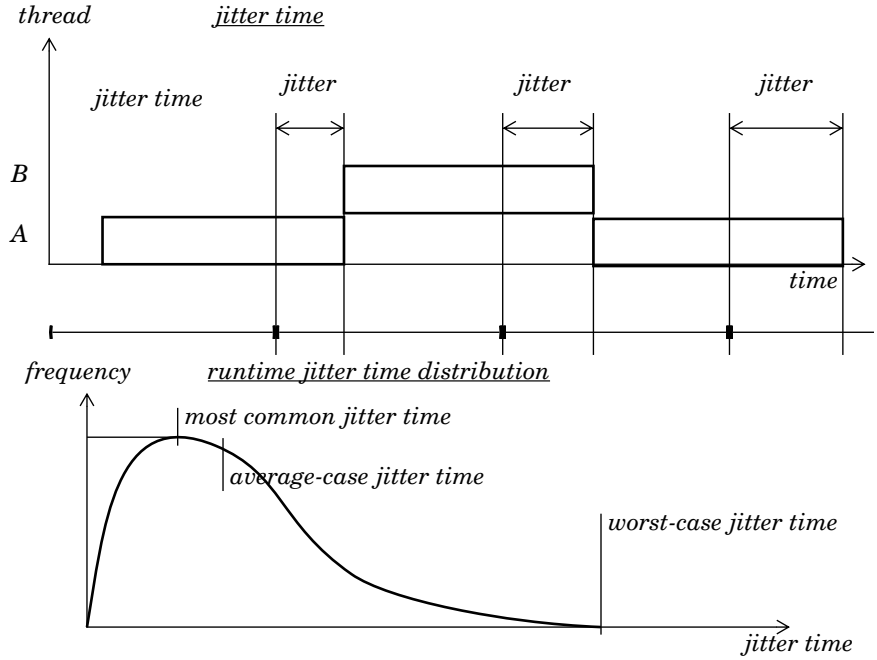


Figure 7.2 *Threads have to finish their currently executing bytecode before a context switch can occur.*

7.9 Communication between nodes

The nodes in a split machine can communicate with each other, and not only with the supervisor, since they have access to the same network. The co-operation of many nodes may result in semi automated computer system architecture. Inter node communication would also be a step towards ubiquitous computing. Java can serve as a powerful interface language between nodes. A project in this direction is the Java code transfer project over Bluetooth (see [Bon02]) where the IVM was run in a control computer and a Koala robot. Bytecodes were transmitted to the control computer that later ran the downloaded program. The system architecture in semi-autonomous systems would be interesting to study in deeper detail.

7.10 Conclusions

This thesis constitutes a foundation on which further work can be implemented to empirically prove that real-time and embedded systems can achieve the desirable consequences of high-level object-oriented programming languages without significant modifications or adaptations. Only some parameters for the GC are required, for example, memory allocation speed of high-priority threads, together with annotations in the code, to reach this end, however, it still has to be proven. The approach to implement a virtual machine on top of the hardware platform seems to be a good balance for safety critical systems. They have to rely on statically verified code. In safety critical systems, every compilation would require a new time-consuming verification of the compiled code — even small changes of the source code could result in major changes in the binary code. A VM could be a good tradeoff. It can be compiled and verified once for the platform. All the programs that are executed by the VM run in a safe environment.

A major contribution of the IVM is to open the runtime system from native C code. Threading and garbage collection are possible to modify and even adapt to an already existing runtime system. Integration of the IVM into another system is simplified through this possibility to reach the internals of the IVM from native C code. No other VM enables this possibility. Other JVMs require at least a specific memory area for the machine and they cannot cooperate with a GC or a scheduler that already operates in the underlying runtime system. Most common is a JVM that require complete control of the hardware platform — an unthinkable sacrifice for many applications.

Even though the goals of the project have not been empirically proven, no obstacles have emerged during the implementation this far. WCET and WCLM analysis remain to be calculated and verified, and the complex integration of a RTGC and a predictable RT kernel into the IVM is left as future work. Together with the RT analyser and a scheduling analyser, the system would be complete to prove the goal of modern high-level programming languages in RT embedded systems. All the parts have been developed and verified separately, but they all have to be collected

together in one application- The IVM serves as the link to all the mentioned parts.

References

[ABW90] N.C. Audsley, A. Burns and A.J Wellings. Hard Real-Time Scheduling: The Deadline Monotonic Approach, *Proceedings of 8th IEEE Workshop on RealTime Operating Systems and Software*, Atlanta, GA, USA, pp127-132, 1990.

[ACL98] Ali-Reza Adl-Tabatabai, Michal Cierniek, Guei-Yuan Lueh, Visgesh M. Parikh, James M. Stichnoth. Fast Effective Code Generation in a Just-In-Time Java Compiler, *SIGPLAN '98*, ACM 0-89791-987-4/98/0006, Montral, Canada, 1998.

[AD97] O. Agesen, D. Detlefs. Finding References in Java Stacks. *OOPSLA'97 Workshop on Garbage Collection and Memory Management*, Atlanta, Georgia, October, 1997.

[AT36] Alan Turing. On computable numbers, with an application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society*, Series 2, 42 (1936), pp 230-265.

[Big98] L. A. Bigagli. *Real-Time Java – A pragmatic Approach*, Master thesis, LU-CS-EX:98-12, Dept. of Computer Science, Lund Institute of Technology, Lund, 1998.

[Bja97] Elizabeth Bjarnason. *Interactive Tool Support for Domain-Specific Languages*, Licentiate thesis, Department of Computer Science, Lund University, Lund, 1997.

[Blo00] A. Blomdell. Efficient Java Monitors, *Technical Report ISRN LUTFD2/TFRT--7593--SE*, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, 2001.

[BM83] Boris Magusson. *Code-Objects: a support for incremental compilation*, LUTFD2/(TFCS-3005)/(1-15)(1983) & LUNFD6/(NFCS-3005)(1-15)/(1983), Department of Computer Science, Lund Institute of Technology, August, 1983.

[BMRSS96] Fran Buschmann, Regine Meunier, Hans Rohnert, Pater Sommerlad, Michael Stal. *A System of Patterns – Pattern–Oriented Software Architecture*, John Wiley & Sons Ltd. 1996.

[Bon02] Mads Bondo Dydensborg. *Implementation of Dynaminc code and state transfer with Java via Bluetooth*, Department of Computer Science, University of Copenhagen, July 2002.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, Philip Wadler. Making the future safe for the past: Adding Genericity to the Java

Programming Language, In *OOPSLA Proceedings*, October 1998.

[CAN91] Bosch, Postfach 50, D-700 Stuttgart L. *CAN Specification*, version 2.0 edition 1991.

[C78] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs New Jersey, 1978.

[C++91] Bjarne Stroustrup, *The C++ programming language (2nd ed.)*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1991

[CE00] A. Cervin and J. Eker: Feedback Scheduling of Control Tasks, In *Proceedings of the 39th IEEE Conference on Decision and Control*, Sidney, Australia, December 2000.

[Der74] M. L. Dertouzous. Control Robotics: The Procedural Control of Physical Processes, *Information Processing 74*, North-Holland Publishing Company, 1974.

[DNM68] O.J. Dahl, K. Nygaard, B. Myrhaug. Simula 67 Common Base Language, *Technical Report, Publ. no. S-2, Norwegian Computing Center*, Oslo, 1968.

[Eclipse] http://www.eclipse.org/org/index.html

[EHÅ00] Johan Eker, Per Hagander, Karl-Erik Årzén. A Feedback Scheduler For Realtime Controller Tasks, *Proceedings of IFAC Control Engineering Practice 2000*.

[Ek00] Torbjörn Ekman *A hard real-time kernel with automatic memory management for tiny embedded devices*, Master thesis, Department of Computer Science, Lund Institute of Technology, Lund University, Lund, 2000.

[FG02] Flavius Gruian. *Energy-Centric Scheduling for Real-Time Systems*, Doctorate thesis, Department of Computer Science, Lund Institute of Technology, Lund Univerisy, Lund, Sweden, December 2002.

[GCI02] Anders Ive, Anders Blomdell, Torbjörn Ekman, Roger Henriksson, Anders Nilsson, Klas Nilsson, Sven Robertz–Gestegård, Garbage Collector Interface, *Proceesings of NWPER'02 – Nordic Workshop on Programming Environment Research*, Copenhagen, August, 2002.

[Ges03] Sven Gestegård Robertz. *Flexible automatic memory management for real-time and embedded systems*, Licenciate thesis, Dept. of Computer Science, Lund University, May 2003.

[GHJV95] E. Gamma, T. Helm. R. Johnson, J. Vlissides. *Design Patterns: Abstraction and Reuse of Object Oriented Design*, Addison–Wesley, Reading, 1995.

[GR83] Adele Goldberg, David Robson. *Smalltalk-80: The Language and its Implementation*, Addison-Wesley Publishing Company, 1983.

[Hen98] R. Henriksson. *Scheduling Garbage Collection in Embedded Systems*, Doctorate thesis, Dept. of Computer Science, Lund University, July 1998.

[HePa96] John L. Hennessy, David Patterson. *Computer Architecture A Quantative Approach*, Morgan–Kaufmann, second edition, 1996.

[Ive98.1] A. Ive. *Adaption of a Mailbox to a Real-Time Garbage Collector*, Technical report, LU-CS-TR:2002-225, Deptartment of Computer Science, Lund Institute of Technology, Lund University, Lund, January, 1998.

[Ive98.2] A. Ive. Runtime Performance Evaluation of Embedded Software, *Proceedings of NWPER'98 – Nordic Workshop on Programming Environment Research*, 1998.

[Ive98.3] Software Architecture Notations. *Software Architecture - An overview of the state-of-the Art*, University of Karlskrona/Ronneby, Department of Computer Science and Business Administration, Ronneby, april, 1998.

[J2EE] Shannon, Bill. *The Java 2 Platform Enterprise Edition Specification, v 1.3*, Sun Microsystems, Inc., August 2001.

[J2ME] J2ME CLDC API 1.0, Sun Microsystems, Inc., 2000.

[J2SE] Java 2 Platform, Standard Edition, v 1.4.1 API Specification, Sun Microsystems, Inc., 2002.

[JBed] Jbed ME Java technology for Small Handheld Devices. http://www.esmertec.com/download/pdf/Jbed_ME_White_Paper.pdf

[JBed99] JBED: Java for Real-Time Systems, *Dr. Dobb's Journal*, November, 1999.

[JCAPI] Sun Microsystems, Inc. *Java Card 2.2 Application Programming Interface Specification*, September 23, 2002.

[JCRE] Sun Microsystems, Inc. *Java Card 2.2 Runtime Environment Specification*, May 13, 2002.

[JCVMS] Sun Microsystems, Inc. *Java Card 2.2 Virtual Machine Specification*, May 13, 2002.

[JDP] The JDrums project, http://www.ida.liu.se/~jengu/jdrums/

[JIJ98] Antero Taivalsaari. *Implementing a Java Virtual Machine in the Java Programming Language*, Techical report TR-98-64, Sun Microsystems Laboratories Inc., march 1998.

[JLS00] K. Arnold, J. Gosling, D. Holmes, *Java Language Specification Third Edition*, The Java series, Addison-Wesley, June, 2000

[JL96] Richard Jones, Rafael Lins. *Garbage Collection – Algorithms for Automatic Dynamic Memory Management*, ISBN 0 471 94148 4, John Wiley & Sons Ltd, 1996.

[JNI99] S. Liang. *The Java Native Interface Programmer's Guide and Specification*, The Java Series, Addiston-Wesley, 1999.

[JVM99] T. Lindholm, F. Yellin. *The Java Virtual Machine Specification Second Edition*, The Java series, Addison-Wesley, 1999.

[HS02] The Java HotSpot Virtual Machine, v1.4.1, d2, Sun Microsystems Inc., technical paper, September, 2002

[KMMN91] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pederson, Kristen Nygaard. *Object oriented Programming in the Beta Language*, Matematisk Institut, Aarhus Universitet, draft, Septemper, 1991.

[KML00] Kelvin D. Nilsen. Simanta Mitra, Steven J. Lee. *Method for efficient soft real-time execution of portable byte code computer programs*, United States Patent 6,081,665, June, 2000.

[KM93] J. L. Knudsen, O. L. Madsen. Language Implementation. In *Object-Oriented Environments – The Mjølner approach*, edited by J. L. Knudsen & al., Prentice-Hall International Ltd., 1993.

[LR80] B. W. Lampson, D. D. Redell. Experiences with Processes and Monitors in Mesa, *Communications of the ACM*, Vol. 23, No. 2, 1980.

[Lan00] M. Landqvist. *Porting and Evaluation of an Embedded Java Virtual Machine on Palm OS*, Master thesis, LU-CS-EX:2000-4, Dept.of Computer Science, Lund Institute of Technology, Lund, February, 2000.

[Lin51] Carl Von Linné. *Linnaeus' Philosophia Botanica*, ISBN 0198501226, Oxford University Press, June, 2003.

[LL73] C. L. Lui, J. W. Layland. *Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment*, Journal of the ACM, Vol. 20, No 1, 1973.

[Lun99] T. Lundqvist. *A Static Timing Analysis Method for Programs on High-Performance Processors*, Licentiate thesis, Dept. of Computer Engineering, Chalmers University of Technology, Göteborg, 1999.

[Mag84] Boris Magnusson. *Contributions to execution environment design applied to Simula*, Doctorate thesis, Department of Computer Science, Lund Institute of Technology, Lund 1984.

[NE01] Anders Nilsson, Torbjörn Ekman. Deterministic Java in Tiny Embedded Systems, Processdings of ARTES01, Department of Computer Science, Lund, 2001.
Lund University, Sweden.

[Nør99] Tor Nørretranders. *Märk världen*, Albert Bonniers Förlag AB, ISBN: 9100570702, Sverige.

[Palm99] Neil Rhodes, Julie McKeehan. *Palm Programming – The Develope's Guide*, O'Reilly & Associates, Inc, January 1999.

[Per00] P. Persson. *Predicting Time and Memory Demands of Object-oriented Programs*, Licentiate thesis, Dept. of Computer Science, Lund University, 2000.

[PERC02] *Differentiating Features of the PERC Virtual Machine*, White paper, Newmonics Inc., August 2002.

[Pil00] Markus Pilz. *Earliest Deadline First Scheduling for Real-Time Java*, Paper at Embedded System Conference Europe 2000.

[Rez98] Ali-Reza Adl-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, James M. Stichnoth. Fast, Effective Code Generation in a Just-In-Time Java Compiler, *SIGPLAN'98*, Montreal, Canada, ACM 0-89791.987-4/98/00006.

[RTCE00] J Consortium. *International J Consortium Specification, Real-Time Core Extension*, Revision 1.0.14, 2000.

[RTSJ00] G. Bollella, B. Brosgol, S. Furr, D. Hardin, P. Dibble, J. Gosling, M. Turnbull. *The Real-Time Specification for Java*, Addison-Wesley, 2000.

[RTC00] J Consortium. *International J Consortium Specification – Real-Time Core Extensions*, Version 1.0.14, September, 2000, Available at http://www.j-consortium.com/rtjwg/rtce.1.0.14.pdf

[SCADA] http://www.ab.com/abjournal/nov1999/departments/prodfocus/scadaarch.html

[SIM89] *Data Processing - Programming Languages - SIMULA*, Swedish Standard SS 63 61 14 (1987), available through ANSI.

[Spotless] Antero Taivalsaari, Bill Bush, Doug Simon. *The Spotless System: Implementing a Java System for the Palm Connected Organizer*, Techical report TR-99-73, Sun Microsystems Laboratories Inc., february 1999.

[SLR90] L. Sha, R. Rajkumar, J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization, *IEEE Transactions on Computers*, Vol. 39, No 9, 1990.

[SRTJ] http://rtjcom.com/files/simpleRTJ-TechInfo.PDF

[STL95] Stepanov, A.A., Lee M. The Standard Template Library, Tech. Rep. HL-95-11(R.1), Hewlett-Packard Laboratories, Palo Alto, California, Feb. 1995.

[STK300] *STK300 Complete development system for the Atmel ATmega128 microcontrollers*, Datasheet, Kanda Systems Inc., http://www.kanda.com/datasheet/STK300wb.pdf

[SW] Fridtjof Siebert, Andy Walter. *JamaicaVM -- User Documentation, The Virtual Machine for Real-time and Embedded Systems.* http://www.aicas.com/jamaica/doc/html/index.html

[TAHG02] T. Andræ, H. Gustavsson. *Extended Support for Java in Control Systems*, Master thesis, Department of Computer Science, Lund Institute of Technology, Lund University, February, 2002.

[Ung84] D. M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm, *ACM SIGPLAN Notices*, 19(5):157-167, April 1984.

[WABA] Guilherme C. Hazan. *Frequently Asked Questions*. http://www.superwaba.com.br/faq.asp#wabaxsw

[WaDa71] Arne Wang and Ole-Johan Dahl. Coroutine sequencing int block structure environment, *BIT 11*, (1971).

Appendix A

Bytecode execution time calculation

The WCET calculation of the bytecodes in the IVM are analysed by hand. The execution time of the binary code produced by a bytecode, is summarised. No considerations of how caches changes the execution time are included. The following figure shows an examples of a bytecode, their binary code and their execution time calculation.

Pseudo Code:

```
case IADD: // push(pop() + pop());
  int val2 = pop();
  int val1 = pop();
  int res  = val1 + val2;
  push(res);
  inc pc;
break;
```

C-code:

```
case IADD:
  top--;
  stack[top-1] += stack[top];
  pc++;
break;
```

Expanded C-code:

```
case 96:
  ((*ivm->ap)->valueStackTop) -= 1;
  ((*((uint32**)ivm->ap))[((*ivm->ap)->valueStackTop)-1]) =
    (uint32)(
      ((int32)((*((uint32**)ivm->ap))
              [((*ivm->ap)->valueStackTop)-1]))
    + ((int32)((*((uint32**)ivm->ap))
              [((*ivm->ap)->valueStackTop)]))
    );
  ((* ivm->ap )->pc)++;
break;
```

Optimised 68000 assembler:

```
L117:
  movel a5@(4),a0     //
  movel a0@,a0        //
  subqw #1,a0@(20)    // int val2 = pop();
  movel a5@(4),a0     //      ivm->ap
  movel a0@,a0        //     *
  clrl d0
  movew a0@(20),d0    //   ((        )->vST
  lsll #2,d0
  movel a0@(d0:l),d1  //ivm->ap[vST]
  addl d1,a0@(-4,d0:l)// + ivm->ap[vST-1]
  movel a5@(4),a0     //ivm->ap
  movel a0@,a0
  jra L459
  .even
...
L459:
  addqw #1,a0@(16)    //->pc++
```

Hand optimised 68000 assembler: Execution Time (cycles):

```
L117:
  subqw #1,a0@(20)            12
  clrl d0                     2
  movew a0@(20),d0            12
  lsll #2,d0                  2
  movel a0@(d0:l),d1          16
  addl d1,a0@(-4,d0:l)        18
  jra L459                    1
  .even
...
L459:
  addqw #1,a0@(16)            12
```

Appendix B

Access flags

Modifier	*Value*	*Applicable to*			
		class	***field***	***method***	
ACC_PUBLIC	0x0001	•	•	•	According to the JVM Specification
ACC_PRIVATE	0x0002		•	•	
ACC_PROTECTED	0x0004		•	•	
ACC_STATIC	0x0008		•	•	
ACC_FINAL	0x0010	•	•	•	
ACC_SYNCHRONIZED	0x0020			•	
ACC_SUPER	0x0020	•			
ACC_VOLATILE	0x0040		•		
ACC_TRANSIENT	0x0080		•		
ACC_NATIVE	0x0100			•	
ACC_INTERFACE	0x0200	•			
ACC_ABSTRACT	0x0400	•		•	
ACC_STRICT	0x0800			•	
ACC_SHADOW	0x0002	•			IVM Specific
ACC_INTERMEDIATE	0x0040			•	
ACC_PRECOMPACT	0x1000				
ACC_INTERFACEMARK	0x2000				
ACC_INIT	0x80				

Table 0.1

Appendix C

Method efficiency

The number of pointer dereferences, for the three different method calls, to find a method description is shown in Table 0.2. The purpose of the table is to briefly show the overhead introduced in the IVM compared to efficient C code. Optimisation techniques, as method inlining, could drastically reduce the overhead of the search for the method description. Every reference in the IVM is counted as two actual pointer dereferences. Offsets and indices are counted as one dereference.

Method call	***Number of indirection steps***		***Remarks***
	IVM	***"Efficient" C***	
static	9	0	The method pointer is located in the code.
virtual	9	2	
interface	7 + C*N	2 + N*D	N is the size of the number of interfaces in the interface array. C and D is the time compare two interface references, and to run the search algorithm.

Table 0.2 *The table shows the number of indirection steps to find the method description in the IVM and in an "efficient" solution.*

Interface method descriptions lookups are dependent on the unpredictable location of the interface in the interface array. In the worst case, the complete interface array has to be searched before the correct method array is found. This unpredictable behaviour could be circumvented with the implementation of a hash table interface lookup. This, for example, is implemented in the hotspot engine from Javasoft (see [HS02]).
Different method optimisation techniques that are relevant to the IVM, are presented in Section 7.4.

Appendix D

Exceptions and memory utilisation

Some exceptions are generated by the JVM itself. Others are related to specific bytecodes, while some span over several bytecodes. The supported API often adds more exceptions that are always necessary. About 50 exceptions and errors necessary for the JVM are depicted in Figure 0.1 together with basic exceptions from the API. Java applications may later add more exceptions, but those in the figure are necessary for every application.

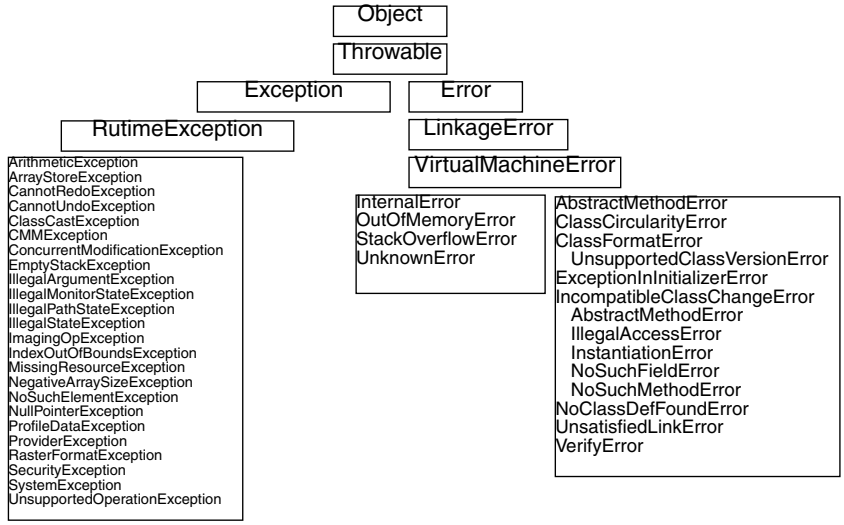


Figure 0.1 *The exceptions and errors necessary for the JVM are described in about 50 classes.*

The classes of the exceptions and errors occupy much memory. A sensible optimisation would be to represent each exception as a number instead of a complete class. The class could then be constructed during runtime if the memory is not exhausted. The exception could then be instantiated from the class. In the IVM, all classes are loaded prior to the main-class.