Integration of the Real-Time Specification for Java in L4 Kaffe

Alexander Böttcher

Technische Universitaet Dresden
Faculty of Computer Science

Institute for System Architecture
Operating System Group

April 22, 2005

Responsible Professor: Prof. Dr. Hermann Härtig
Tutor: Dipl.-Inf. Ronald Aigner

Erklärung

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

Declaration

I declare to have written this work independently and without using unmentioned sources.

Dresden, April 22, 2005

Alexander Böttcher

Diese Seite wird ersetzt mit der Aufgabenstellung.

Contents

1 Introduction

In the last years Java applications entered into the world of real-time and embedded systems. However, some features of Java and of the Java Virtual Machine (JVM) complicate the development of predictable Java applications. The main problem is the automatic memory management by garbage collectors in order to release Java objects. These collections can lead to delays in the execution of a Java application, which are not acceptable for real-time tasks. Further, the language Java and the Java Virtual Machine specification do not support periodic tasks, access to hardware and asynchronous events generated outside the Java world as interrupts. These features are often required for productive usage in embedded and in real-time systems.

Therefore, the National Institute of Standards and Technology (NIST) produced a basic requirement document for a real-time Java extension [CR99]. The Real-Time specification for Java (RTSJ) [BBF$^+$00] and the Real-Time Core extension for the Java platform (RT Core) [J. 00] are concrete specifications that comply with this document.

For the microkernel based Dresden Real-Time Operating System (DROPS) a Java Virtual Machine, L4 Kaffe, is available that can execute Java byte code. This thesis focuses on the decision for a real-time Java specification and the evaluation of it in order to describe steps and to estimate the effort for its integration into L4 Kaffe. A further goal is the integration of selective parts and the representation of first results in the context of predictable execution of Java applications in L4 Kaffe on top of DROPS.

Organization of this Thesis

This thesis is organized as follows. In Chapter 2, I give an introduction to the operating system DROPS and the JVM L4 Kaffe (section 2.1 and 2.2). I discuss problems of Java in the context of predictable execution (Section 2.3) and how the RTSJ and RT-Core address these issues (Section 2.4). Further, I give reasons for my decision to use the RTSJ (Section 2.5) in this thesis.

In Chapter 3, I evaluate the aspects of the RTSJ and describe steps to integrate them into L4 Kaffe. In Chapter 4, I explain the implementation of few aspects. I present results of RTSJ in L4 Kaffe in the Chapter 5. Finally, I conclude this thesis with a summary and outline possible approaches for further investigations and implementations.

2 Fundamentals and Basic Considerations

In this chapter I present briefly the operating system Dresden Real-Time Operating System (DROPS) and L4 Kaffe, which are the bases of this diploma. Further, I outline some characteristics of Java and JVMs, which complicate the development of Java applications that are predictable in their execution. Afterward, I discuss two real-time specifications for Java, which address these issues.

2.1 DROPS

DROPS is a research project at the Technische Universitaet Dresden. The primary goal is to support applications with *Quality of Service* requirements. For this purpose, various real-time and time-sharing tasks run on top of a microkernel, which supports predictable execution of server and client applications.

2.1.1 Microkernel

L4 kernel family

The heart of DROPS is the Fiasco microkernel [Hoh98, Hoh02a]. Fiasco belongs to a family of L4 kernels like Pistachio and Hazelnut [L4K05]. The L4 *application binary interface* (ABI) describes the interface for these microkernels, which Jochen Liedtke initially designed [Lie96].

The microkernels provide isolation of tasks through separation of address spaces, execution instances known as threads, and communication between threads called *inter process communication* (IPC). All other required functionalities have to be implemented as services running as user level servers.

The intention of the L4 microkernel architecture is to force a better system design, to limit effects of errors caused by a service and to allow a good portability and maintainability through a tiny kernel. A single failure of a service like a network device driver cannot affect the kernel and other applications that are not depending on the driver.

Fiasco

Fiasco is developed and maintained at TECHNISCHE UNIVERSITAET DRESDEN and is written in C++ and assembler. It offers a minimal set of functionality provided due to the microkernel interface, which allows an efficient and tiny implementation of the kernel.

Further, Fiasco is a preemptible real-time capable microkernel with hard priorities for threads. These characteristics help to support short latencies between events and their handling. Fiasco uses *non-blocking* mechanisms as *locks*, and *compare and swap* operations for synchronization of kernel objects [HH01, Hoh02b]. Additionally, Fiasco supports periodic threads [Ste04], which are often required in real-time systems.

The microkernel interface L4 version 2 (L4 V2) is a stable interface in Fiasco. L4 X.0 and X.2 are experimental specifications for enhancement and adjustment of shortcomings of L4 V2. Fiasco implements the L4 V2, X.0 and X.2 microkernel interfaces and runs on the IA32, the IA64, and the ARM architectures. Additionally, a port of a user mode Fiasco on Linux, Fiasco-UX, is available for rapid prototyping of L4 applications [Ste02].

2.1.2 L4Env

The L4 Environment (L4Env) is a programming environment for applications on top of the L4 microkernel family [SRGTD03]. L4Env is developed as a part of DROPS and its intention is to provide a minimal set of functions, which developers often require for application development. The environment consists of services, such as protection of critical sections, page-fault handling, and resource management for memory and files. In contrast, in a monolithic operating system like Windows or Linux, these services are mainly provided by the operating system kernel.

Two C libraries DIETLIBC [die05] and OSKIT are provided for application development in DROPS in order to support and to simplify ports of UNIX application. They provide as frontend the classical C library headers and functions, but required functionality by the C libraries is supplied by services of the L4Env instead of the kernel in a monolithic operating system. The OSKIT library was the first provided C library, but is slowly replaced by the DIETLIBC.

2.2 L4 Kaffe

2.2.1 Java and JVM

Java is a set of technologies developed by Sun Microsystems [Sun05]. The language Java is a high level object oriented programming language, which uses implicit memory management, platform independent application programming interfaces (APIs) and dynamic loaded classes. Traditionally, a compiler translates the Java source code to hardware and operating system independent Java bytecode. A *Java Virtual Machine* (JVM) is necessary to execute Java bytecode [LY99].

In the approach of the GNU Compiler for the Java Programming Language (GCJ), Java source code or Java bytecode is translated into machine and operating system specific code, which can be executed without a separate JVM [gnu05b]. The execution speed of such applications is higher than of applications executed by a separate JVM.

A JVM is an abstract machine that executes the Java binary code on top of various operating systems. Therefore, a JVM is operating system specific, whereas the Java bytecode of a Java application can be executed without recompilation. Early JVMs only worked as pure *interpreter*. Today, they often use *just in time* (JIT) compilers that transform the Java binary code into machine code at runtime and execute it. These JIT compilers are similar to the GCJ approach, but the machine specific code is not optimized as by the GCJ and is only temporarily available during runtime of a JVM.

The object-oriented language Java uses implicit memory management to support the construction of robust applications. Garbage collection algorithms are necessary to reuse memory, which Java applications do not free explicitly as it is usually done in applications written in C or C++. The collectors have to scan for Java objects, which are not in use anymore and not attainable by other objects. Therefore, a JVM is able to reclaim memory of unused Java objects.

2.2.2 L4Kaffe

Kaffe is an open source implementation of the Java Virtual Machine Specification distributed under GPL 2 [kaf05]. It operates either as an *interpreter* or with JIT compilation. However, the complete Java API as specified by Sun Microsystems is not provided yet, because of the amount of it. Missing Java classes and methods are integrated by users of Kaffe if they require it or by the GNU Classpath project [gnu05a]. GNU Classpath has the goal to provide the essential libraries for Java freely available for various JVMs. Kaffe uses large parts of it as GCJ and many other JVMs do too.

Characteristics of L4 Kaffe

L4 Kaffe [Boe04] is a port of the JVM Kaffe to DROPS by using the L4Env and the L4Env adapted DietLibC. L4 Kaffe currently works as an *interpreter* and runs on Fiasco and Fiasco-UX with the kernel interface L4 V2 on IA32.

L4 Kaffe maps every Java thread to a native L4 thread. Synchronization in L4 Kaffe is realized with atomic *compare and swap* operations and the L4Env specific semaphore implementation. The L4Env servers *loader* and *exec* are used for loading L4 Kaffe. With the help of these servers L4 Kaffe is able to load shared native libraries, when Java applications require them.

Garbage Collectors

Kaffe provides two implementations of garbage collectors. The first is the incremental garbage collector KAFFE-GC [Bac99]. The second one uses the BOEHM GARBAGE COLLECTOR, which is described in [HJB05]. The Boehm collector is still being integrated into Kaffe. Therefore, it is not used in this diploma thesis.

In the following I will describe the basics of the garbage collector Kaffe-GC. Kaffe-GC has a separate garbage collector thread. If no memory is available for Java threads or a Java application explicitly requests a garbage collection, the garbage collector thread is executed. First of all, the garbage collector thread tries to allocate more memory from the operating system. If this action fails, the thread starts a garbage collection. It stops all Java threads to protect its internal data structures from concurrently executing Java threads. Following, it scans the native stack, the L4 thread stack, of each Java thread for references to Java objects. If the garbage collector thread detects references to Java objects, it marks them as reachable. That means, that they are still in use. Additionally, all found Java objects are scanned for other reachable Java objects that they reference. After this process, all unmarked Java objects are known not to be in use. Therefore, the garbage collector thread releases them. Finally, it resumes all stopped Java threads.

However, each Java object can have a `finalize()` method, which has to be executed before the object is released. Kaffe-GC puts such Java objects, which have a `finalize()` method, in a separate list. An separate *finalizer thread* iterates over this list and executes the `finalize()` methods of all Java objects. After all `finalize()` methods have been completed, the *finalizer thread* releases the corresponding Java objects.

2.3 Predictability of Java Applications

2.3.1 Effects of Automatic Memory Management

Some of the garbage collectors are not suited for real-time purposes, because they pause Java threads during the complete collection process and cause unpredictable execution delays.

Real-time capable collectors minimize these delays by permitting Java threads to preempt a running collection. Therefore, threads are able to run preferred to a garbage collection thread, for example because of a higher priority. Furthermore, they are predictable in their execution delays as long as they do not require additional memory during an ongoing garbage collection. Java threads have to mark the Java objects, which they are using during their execution, while a garbage collection is in process. The collector has to scan these objects for additional object references to find all Java objects that are in use. Read and write barriers are necessary for coordinating concurrent access to Java objects of running Java threads and of the garbage collector thread. The barriers introduce delays, which have to be predictable. They cause an overhead due to the synchronization between threads and the collector. However, the synchronization is necessary to keep the collection state of the garbage collector consistent.

Further, the maximum execution time is predictable for real-time capable collectors, for example of the collector described in [Bak92]. The operations for the garbage collection have a known maximum or constant execution time for instance for scanning, marking and queuing memory references in lists. They differ in complexity, for instance linear and quadric to the number of all existing references, but the main statement is that the worst case execution time of the collection is predictable for a known count of memory references.

The best solution to support real-time Java threads is not to interrupt or delay them by the garbage collector thread at all. However, the known garbage collection implementations cannot completely avoid delays. Therefore, in order to develop real-time Java applications it is necessary to estimate, how long a garbage collection can delay Java threads. The knowledge about delays due to the garbage collection is required to decide, whether a Java thread can finish his job before his deadline. Jobs in *hard real-time systems* always have to be finished before their deadline. In *soft real-time systems*, a percentage of the jobs have to be finished until the deadline. It is tolerable, when some jobs miss their deadline.

Additionally, the knowledge about the delays influences the maximum count of possible running Java threads, which have to meet their deadline. The application developer or an automatic service has to consider these delays for producing feasible schedules.

Conclusion

The automatic memory management is the main problem in order to support predictable execution of Java threads and applications. The Java application can be paused by a garbage collection at any points in time, which can be intolerable for real-time purposes. Additionally, the usage of real-time capable collectors and necessary barriers can result in slower execution time of threads due to barriers. Therefore, the collection time of the garbage collector can be longer than in cases when threads cannot preempt the garbage collector and barriers are not necessary for the garbage collector thread.

2.3.2 Minimizing Delays

After the consideration of the last section, I want to describe attempts to minimize or avoid delays by a garbage collection through specific design and behavior of Java applications. Additionally, I outline shortcomings of these attempts.

Preallocating of Java Objects

In order to support predictable execution, a Java application can allocate as much as possible Java objects in advance. This is helpful to minimize invocations of garbage collections during runtime. However, the Java language and JVMs report exceptions and errors by throwing Java objects of the class `java.lang.Throwable` and subclasses extending this class. These objects provide the stack traces of invoked methods, which require additional memory. This mechanism cannot be easily avoided to minimize the memory consumption, because it is widespread in the Java APIs.

Therefore, a developer has to assume that the number of Java object references can vary over time. Particularly for Java applications with large number of Java threads the count of references to objects can be badly estimated for a specific point in time. The number of these memory references directly affects the worst-case execution time of a garbage collection and the resulting delays of Java threads. Additionally, it is unknown how often no memory is available for each Java thread. It depends on the memory consumption for new Java objects by all Java threads. A developer can optimize this behavior of its own application, but normally he has no influence on the used Java API. Therefore, delays caused by garbage collections cannot be used for calculations of feasible schedules for Java threads, because admission for schedules has to be computed in advance.

It is impracticable to calculate the feasibility of schedules each time the count of Java objects changes. Therefore, another option is to specify and to assume a maximum count of Java objects for the complete runtime of the Java application. However, the utilization of the processor will be low and discontenting, when the associated worst-case execution time of the maximum count of Java objects is used, because the average execution time is mostly smaller.

Java Heap Size variation

A further aspect is a variable Java heap size of a JVM, when the JVM adapts it according to its memory consumption. Thereby, the maximum count of garbage collection cycles can change over time. An estimation of the resulting time of the delays for each Java thread is not exactly possible. However, this knowledge would be helpful to consider it for feasible schedules of Java threads.

Concurrent running Threads beside a Garbage Collection

Real-time garbage collection algorithms can support concurrently running Java threads during a garbage collection. If such a running Java thread requests memory and no memory is available, the thread will block until the end of the collection. Therefore, a Java thread can be delayed every time it requires memory, although it can run concurrently to a garbage collection.

Involving the Garbage Collector

Another attempt is to involve explicit garbage collection invocations in the design of a Java application. The application has to be split in parts of work, jobs, after which an application directly forces a garbage collection. During the execution of these jobs implicit garbage collection invocations by a JVM have to be avoided through adaptation of the Java heap size according to the memory consumption behavior of a Java application. By an explicit invocation a Java application knows when the garbage collection is executed and how long the collection takes for the worst case by measuring.

However, when multiple Java threads are used in an application, it is hard to estimate whether all threads have finished their work and whether a specific point in time is suited to start a garbage collection. Therefore, explicit synchronization of all Java threads would be necessary to provide such a point in time. If the jobs are of various size, Java threads will have to wait much of their time for other Java threads. Additionally, the memory consumption of all Java threads can hardly vary, when threads can run parallel on systems with multiple CPUs or multi-threading characteristics of one CPU. Therefore, the Java heap size has to be as large as the worst case memory consumption between two explicit garbage collector invocations. Otherwise, the JVM could implicitly start a garbage collection during an execution of a job.

Because of all mentioned reasons this attempt is improper for multiple Java threads and parallel execution of them. As well single threaded Java application cannot utilize the CPU with jobs of a Java applications because of often unnecessary explicit garbage collection invocations.

Conclusion

The consideration showed that the described attempts to support predictable execution of Java applications and real-time garbage collectors alone are not sufficient for real-time capable JVMs. Additional mechanisms are required. In Section 2.4, I describe two Java real-time specifications addressing these issues.

2.3.3 Further Aspects

The scheduling of Java threads is priority based. However, the Java Virtual Machine Specification does not describe a specific scheduling algorithm. Today, many JVM implementations conform to the scheduling of the Java Virtual Machine from Sun Microsystems. This JVM uses priorities ranging form one to ten. The higher the priority value is, the more preferred the threads are executed. However, the scheduling of threads with the same priority depends on the implementation of the JVM and the host operating system.

Additionally, the Java language and JVM support no external events like interrupts. Further, access to the underlying hardware is not supported, but often necessary for embedded and real-time devices. Partial solutions are possible using the *Java Native Interface*, which allows the integration of system specific libraries into a Java application [Lia99]. These system libraries are specific for each hardware architecture, operating system and software environment. Therefore, Java applications with specific requirements for system libraries will not be executable on other systems, if the requested libraries are not provided on these systems.

2.4 Real-Time Support in Java

The National Institute of Standards and Technology (NIST) produced a basic requirements document for a real-time Java extension [CR99]. The Real-Time specification for Java (RTSJ) [BBF+00] and the Real-Time Core extension for the Java platform (RT Core) [J. 00] comply with the document of NIST. Both, RTSJ and RT Core, define extensions of the Java API and describe modifications of the Java Virtual Machine.

The following two sections summarize the essential aspects of these two real-time specifications.

2.4.1 Overview of Enhancements in RTSJ

The Real-time for Java Expert Group (RTJEG) defined the Real-Time Specification for Java (RTSJ). This specification introduces an Application Programming Interface (API) to enable writing real-time Java applications. However, modifications and extensions to Java Virtual Machines are necessary to support and provide the API. For instance, the handling of Java threads has to be adapted to real-time threads.

One of the goals of RTJEG is not to restrict RTSJ to a specific Java environment, embedded, not embedded, or special development kits. Non real-time Java applications shall run without modifications in a RTSJ adapted JVM. Therefore, RTSJ introduces no syntactic extensions of the Java language for describing real-time constraints. All extensions are part of the real-time Java API or part of changes in the Java Virtual Machine (JVM). The specification allows various implementation decisions and possibilities, as long as the JVM conforms to the described behavior in RTSJ. However, the main goal is predictable execution, which may be expensive in sense of performance.

RTSJ introduces two real-time Java thread classes, which can allocate Java objects outside of the traditional Java heap. Particularly, one of the two real-time thread classes can be executed without influence by any garbage collection, because such threads only use Java objects outside of the Java heap. Further, RTSJ defines interfaces and classes for specifying real-time constraints. Additionally, a scheduling layer allows the selection of different schedulers separately for each Java thread. RTSJ does not restrict the selection of scheduling algorithms to a specific one. However, each RTSJ implementation has to provide at least one base scheduler. This base scheduler has to be priority-based and preemptive and must have at least 28 unique priorities. The provided interfaces and classes handle the creation, management, admittance, and the termination of real-time threads.

Garbage collectors are responsible for the automatic memory management of Java objects. RTSJ defines and proposes no special garbage collection algorithm, because the intention is to support varied real-time styles and systems. Furthermore, RTSJ introduces memory areas, which are not part of the automatic memory management. A Java application can handle the memory areas as resources, which are partly under control of the application. The specification defines a memory allocation mechanism, whereby Java objects can be situated outside of the memory managed by a garbage collector. Real-time threads use the memory areas by explicitly entering these areas. All following instantiations of Java objects allocate memory from the current memory area of a real-time thread. Additionally, a JVM has to count for each memory area the threads running in. If the last thread leaves a memory area, a JVM has to free the memory of the Java objects situated in this memory area.

Furthermore, the introduced API shall enable real-time threads to determine the effect on their execution time, when a garbage collection is necessary during their runtime. An implemented garbage collector in a JVM can provide information about execution delays of threads, if these real-time threads run in the influence area of the traditional Java heap, which the garbage collector manages. Another feature of the RTSJ is to support the access to physical memory. Therefore, RTSJ defines Java classes, which enable real-time threads to read from and write to specific areas of physical memory. These real-time threads can describe requested characteristics of the physical memory area they require, for instance alignment requirements. However, only if the host operating system supports access to physical memory for user level applications, the JVM can provide it also for Java applications.

Beyond, RTSJ requires the prevention of priority inversion. Priority inversion is a known scheduling phenomenon, which I describe in the following: A thread t_1 with a high priority requires a resource that a thread t_3 with a low priority already uses. Therefore, t_1 has to wait for t_3 until t_3 releases the resource. Additionally, a thread t_2 exists with a middle priority between the ones of t_1 and t_3. In this scenario, t_2 is executed due to its priority being higher than t_3's. The effect is, that t_2 prevents t_1 from running indirectly, although t_1 has a higher priority than t_2. This phenomenon is known as priority inversion. The protocols *priority ceiling* and *priority inheritance* describe mechanisms to avoid priority inversion.

RTSJ specifies that a JVM has to implement the *priority inheritance protocol* by default. In contrast, an implementation of the *priority ceiling protocol* is optional. RTSJ specifies that the policies, the different priority inversion mechanisms, shall be exchangeable at runtime. A Java application has to be able to set the policies for each Java monitor separately. *Monitors* are the synchronization mechanism of Java objects, which are used by the Java language and are represented by the keyword *synchronize*. These are the resources, at which priority inversion can occur.

Additionally, RTSJ specifies that monitors are not sufficient, when a non real-time thread shall not delay a real-time thread for an unspecific time in the case of synchronization on a commonly used object. The reason is that a garbage collector can stop non real-time threads during garbage collection. Therefore, real-time threads are also delayed until the garbage collection finishes, when they synchronize with non real-time threads. RTSJ defines a set of *wait-free queues* to avoid this problem. A JVM has to provide these classes for Java applications to support non-blocking thread synchronization.

RTSJ generalizes the Java event handling for asynchronous events. It defines classes to describe events and the logic that the JVM has to execute when events occur. Additionally, asynchronous events can specify scheduling constraints just as real-time threads. The JVM is responsible for scheduling the asynchronous events with their specified characteristics at appearance. Either Java threads, timers, or external events such as interrupts, can explicitly cause asynchronous Java events.

RTSJ also introduces support for *asynchronous transfer of control* (ATC). Real-time threads can be aborted or asynchronously change the execution path of other real-time threads if they explicitly indicate the support for ATC. Additionally, if real-time threads are in critical sections, the JVM is responsible for deferring ATC to the end of the critical section.

2.4.2 Overview of Enhancements in RT Core

The J Consortium's Real-Time Java Working Group defined the Real-Time Core Extension for the Java Platform (RT Core). RT Core clearly differentiates between real-time capable and non real-time capable Java Virtual Machines. Therefore, RT Core provides two separate APIs for real-time Java applications. The specification names non real-time Java applications and standard Java API *Baseline* and all real-time enhancements *Core*. A *Core Virtual Machine*, which also supports baseline Java code, is called *extended Baseline Virtual Machine*.

RT Core assumes that the Java Virtual Machine and the Java API are not appropriate for real-time applications in the original respectively classical form. Therefore, it strictly differentiates between *Baseline* and *Core*. A pure Core Virtual Machine only uses the defined real-time Core API and does not require the Baseline API. Only if the JVM supports Core and Baseline Java applications, the Baseline API is necessary. However, RT Core defines limited cooperation between both types of the Core API and the Baseline API.

RT Core specifies the development architecture for RT Core applications as a combination of development tools, Core and Baseline API, run-time environment and Java application code. The specification defines a Core class file as a Java 1.1 class file and a set of programming constraints. A Core verifier is responsible for validating the stringent rules, the constraints, mainly at compile time of the Java source file and partly at runtime.

The developers of Core real-time applications can use standard Baseline compilers as long as they follow the style guidelines and the stringent rules, which are described in detail in RT Core. The specification names the standard Java source code with instructions and invocations to Core classes as *stylized code source*. Additionally, RT Core specifies a *syntactic Core source file* format, which introduces extensions to the syntax of the Java language. A special core compiler is necessary to compile and verify this syntactic Core source file. The resulting code is standard Java byte code. Additionally, RT Core defines the keywords *stackable* and *baseline* for the syntactic Core source. The extension shall relieve the coding effort for developers. The keywords are replacements for long instructions and calls of classes and methods of the Core classes.

An *extended Baseline Virtual Machine* has to support two logical heaps, one for the Core objects and one for the Baseline objects. While the garbage collector manages the memory of the Baseline objects, the Core objects are under control of the developer. Each Core thread has an associated *allocation context*, a memory area, where Core objects are situated. The developers can explicitly release the memory of allocation contexts.

Further, RT Core specifies that Core classes and Core objects are identified by the way they are loaded and instantiated. Only a specialized Core class loader is responsible for loading Core classes. These Core classes have to validate rules as for instance explicit registration of Core objects in the `org.rtjwg.CoreRegistry` class. RT Core uses this registry class as central point to save characteristics of Core objects and Baseline visible methods.

The Core class loader and verifier requires the registry to validate interaction between Core and Baseline objects. Additionally, Java applications have to explicitly register Core objects, if they shall be visible and accessible by Baseline objects. Furthermore, a registration of each method is necessary, which has to interact with Baseline objects. Therefore, RT Core specifies that a JVM has to analyze the relationships of the accessed objects. Core objects are able to invoke all defined methods of other Core objects. On the other hand, Baseline objects can access and invoke only the registered Baseline methods of a Core object. The JVM has to provide different views to Core objects and methods for Core and Baseline objects. A partitioning protocol of RT Core defines all these rules and provides the views from a Core programmer's and from the Baseline programmer's perspective.

Additionally, RT Core specifies that the Core class loader has to manipulate loaded Core classes and references to standard Java classes at runtime. For instance, it has to replace references to classes and objects of `Error`, `Object`, `Throwable` and `Exception` of the package `java.lang` by appropriate classes and objects of the Core API. Furthermore, the Core Class Loader has to replace method implementations of `wait()`, `notify()`, `notifyAll()` and `getClass()` provided originally by `java.lang.Object` by methods of `org.rtjwg.CoreObject`. RT Core requires this replacement mechanism to clearly separate the standard Java classes and objects from the Core world.

Core threads have up to 128 Core priorities, additional to the ten Baseline priorities existing for the standard Java threads. Core threads must have a base and an active priority. The active priority describes the effective priority that the thread may inherit from another Java thread with a higher priority due to the priority inheritance protocol. The basic priority is the one with which a Java thread is executed normally.

Additionally, the Java keyword *synchronized* is constrained for Core objects. *synchronized* is necessary to get a monitor of a Java object, which protects a critical section against concurrent running threads in the same section. RT Core specifies that Core objects can only synchronize at their own monitor. The statement `synchronized(a)` is allowed only, when the Java object `a` is equal to the current Java object `this`. However, RT Core introduces special semaphore and mutex classes in addition to the Java monitors in the Java language.

Beyond, the specification introduces support for asynchronous transfer of control and asynchronous thread termination. RT Core provides stop and abort methods in `org.rtjwg.CoreTask`. However, threads in critical sections have to be executed to the end and monitors have to be left before transferring the execution path of the thread.

2.5 Specification Decision

Both real-time specifications, RTSJ and RT Core, address aspects of real-time extensions for Java applications, but in a different manner. They introduce various concepts of memory management independently from any garbage collection algorithm. Runtime checks validate the rules for accessing and assigning objects from and to memory areas from and to Core and Baseline objects respectively.

RT Core requires that all classes, which are used within the Core extension, have to extend from `org.rtjwg.CoreObject`. Therefore, standard Java classes cannot be reused without modifications. RTSJ introduce an additional API too, but it does not require changing the inheritance hierarchy of existing classes. Additionally, RT Core defines syntax extensions of the Java language. Special compilers are necessary to compile Java classes, and special verifiers have to validate the Core classes. In contrast, RTSJ requires no such rules as the explicit registration of objects. Therefore, no verifiers at compile time are necessary for RTSJ.

RT Core is appropriate for Java applications, which require only a small amount of standard Java classes. The estimation of the extension of a traditional JVM to an extended Baseline Core Virtual Machine is complex, because of the following points: Substitutions of methods of classes at loading time, code verification at compilation and optional special compilers for stylized source code cause a high effort and increase the complexity of a RT Core implementation. The various visibilities of methods in Java objects and the special registration of objects and the necessary checks at runtime affect the complexity of a JVM.

RTSJ is better appropriate for applications, which interact closely with the standard Java API. In addition, existing APIs and classes can be reused widely unmodified, because RTSJ defines no changes at the inheritance hierarchy of the essential Java classes. The memory area concept permits the separation of Java objects into regions with different lifetime, which are partly under control of the Java applications. They cannot explicitly release memory of Java objects but implicitly by leaving the appropriate memory area. The concept of memory areas is a homogeneous integration into the Java language, because the memory has not to be handled for each separate Java object instance explicitly.

After I had considered the two specifications, I decided to investigate the RTSJ in the diploma. My estimation is, that RTSJ is more appropriate than RT Core for supporting real-time Java applications, because of its detailed memory management concept. This concept allows the execution of threads independently of any garbage collection. A further reason for RTSJ is its good interoperability with existing Java applications. The effort for integration of RT Core is high, because of the reasons mentioned above. Further, I do not approve the extension of the Java language by specialized keywords, because it leads to Java source code that is only understood by specialized compilers.

2.6 Related Work

The reference implementation of RTSJ is provided by TimeSys [Tim05]. TimeSys introduced the first RTSJ-compliant JVM.

The aicas GmbH is a company from Germany delivering the JamaicaVM [aG05]. The JVM implements the RTSJ specification, provides features such as hard real-time support, a real-time garbage collector and static and dynamical loading of classes. Tools for analysis, compilation and transforming of Java code to read only memory (ROM) are provided.

This Ovm project develops a “framework for building programming language runtime systems” [OVM05]. Currently they develop a JVM, which supports RTSJ.

Mackinac is a project from Sun microsystems to integrate the RTSJ into its own JVM based on Sun’s Java Hotspot platform [Mac05].

JRate, Java Real-Time Extension [jRa05], is developed as extension of the GNU GCJ . GCJ is a part of the GNU Compiler Collection (GCC), which compiles Java code to native machine code. JRate introduces the support of RTSJ in GCJ.

3 Design and Evaluation

The basis of this diploma thesis is L4 Kaffe, a JVM running on top of the microkernel Fiasco and L4Env. In the following sections I describe the design requirements of the Real-Time Specification for Java (RTSJ) and outline the consequences and effort for integration into L4 Kaffe.

3.1 Real-Time Java Thread Classes

3.1.1 RealtimeThread and NoHeapRealtimeThread

RTSJ specifies two new real-time thread classes `RealtimeThread` and `NoHeapRealtimeThread` in the package `javax.realtime`. In contrast to a regular `Thread` class of the package `java.lang`, these real-time threads can be executed in memory areas outside the traditional Java heap. All created Java objects of a real-time thread are implicitly located in the current memory area of this real-time thread. The traditional Java heap is handled as a Java memory area, which is introduced by RTSJ and I describe in Section 3.2. In contrast to the `RealtimeThread` class, a `NoHeapRealtimeThread` may only operate with non-Java heap objects and may only be executed in Java memory areas outside the Java heap.

The intention of RTSJ is that a `NoHeapRealtimeThread` can run without interference of a garbage collection. The garbage collection does not have to stop `NoHeapRealtimeThreads` for scanning of references, because they are not permitted to use any Java objects located in the Java heap. Therefore, the execution characteristic of a `NoHeapRealtimeThread` is better predictable than for regular Java threads and `RealtimeThreads`. In contrast, the garbage collector can delay `RealtimeThreads` as often as a collection is necessary, because they can use references of Java objects within the Java heap.

However, references of Java objects located in the Java heap are accessible at runtime by `NoHeapRealtimeThreads`. For instance, a `RealtimeThread` can allocate Java objects in the Java heap and can put them into fields of Java objects shared with `NoHeapRealtimeThreads`. An access to such a field by a `NoHeapRealtimeThread` would violate the characteristic of this thread.

Therefore, RTSJ specifies that a JVM has to validate accesses and assignments of all `NoHeapRealtimeThreads`. The verifications during runtime are necessary to prevent the origination of invalid pointers to Java objects, which the garbage collector could have already released. The rules are described in detail in Section 3.2.2.

In order to support the real-time thread classes, L4 Kaffe has to distinguish the threads and their associated Java thread classes to be able to validate the memory access and assign rules. For this purpose, small modifications in L4 Kaffe and its Java thread handling are necessary, which are described in the implementation chapter. Additionally, the real-time Java thread classes `RealtimeThread` and `NoHeapRealtimeThread` have to be implemented.

Further, the complete handling, which is described in Section 3.2, of the Java memory areas is necessary to support the real-time thread classes. The detection of the location of a Java object must be realized in order that L4 Kaffe is able to perform the validation of the access and assignment rule for `NoHeapRealtimeThreads`, which are not allowed to handle references of Java heap objects. Therefore, I have to examine each Java byte code instruction handling with Java objects. Before L4 Kaffe executes such a instruction, L4 Kaffe has to detect the current thread type as well as the location of the used Java object.

If a thread is an instance of `NoHeapRealtimeThread` and it accesses a Java object situated in the Java heap, L4 Kaffe has to throw a `MemoryAccessError` as specified in RTSJ. This Java error indicates for a Java application that a thread violated a access rule. Java applications can catch such errors and can react to them, otherwise the Java application will be aborted.

3.1.2 Scheduling

RTSJ introduces interfaces and classes to support various scheduling policies and implementations. It requires a base scheduler, which has to be fixed-priority preemptive with at least 28 unique priority levels. Additionally, if the underlying operating system supports other scheduling algorithms, the JVM can provide it also for Java applications.

L4 Kaffe maps each Java thread directly to a native L4 thread. Therefore, the standard thread scheduler of the Fiasco kernel manages the scheduling of all Java threads. The Fiasco scheduler uses fixed priorities with 256 unique priorities. Therefore, the requested additional thread priorities can be provided directly to L4 Kaffe. 39 thread priorities are necessary, 10 for the standard Java threads, 28 for the real-time threads and 1 for the garbage collector and *finalizer thread*.

Additionally, RTSJ defines *schedulable Java objects* (SJO). These SJOs are represented in RTSJ by the real-time Java thread classes and the asynchronous event handler class, AEO. RTSJ specifies that the SJOs can be executed according to their characteristics as release time, priority and execution time. However, SJOs need not to be associated all the time to the same Java thread, for example in the case of the AEOs. RTSJ describes that many AEOs can exist within a Java application, but they are executed only, when the associated asynchronous event occurs. Therefore, RTSJ specifies that with each SJO an Java scheduler is associated, which has to schedule the SJO accordingly to the requested characteristics. If a SJO is not a Java thread, the scheduler has to attach one to this SJO.

The consequence for L4 Kaffe is that a management of the AEOs for a Java application is necessary. The AEOs have to be attached to Java threads, when they have to become executable. They must be scheduled accordingly to their characteristics such as release time. Therefore, a time service is essential, which notifies L4 Kaffe at specified points in time in order to schedule the AEOs. Additionally, the available set of Java threads has to be considered, which stand by for executing the AEOs.

3.1.3 Periodic Activities

RTSJ introduces support for periodic activities. *Schedulable Java objects* can be associated with characteristics as release time, worst case execution time, period and deadline. The `ReleaseParameter` class specifies the maximum execution time, the deadline, and Java handlers. If a SJO misses a deadline or overruns its maximum execution time, a JVM has to call associated Java handlers. A application developer can use the Java handlers to react to such events in order to change at runtime the behavior of their real-time application. However, the Java handlers are optional. If the underlying operating system supports the detection of deadline misses or worst case execution time overruns, the JVM can provide them.

Additionally, the subclass `PeriodicParameters` defines the period of a periodic SJO as well as the start time of the beginning of the first period. The class `AperiodicParameters` is required for SJO that become active at an unknown point in time, whereas the maximum execution time and the deadline relative to the unknown point in time has to be known. Its subclass `SporadicParameters` describes aperiodic schedulable objects with a minimum time between releases.

Fiasco supports for the kernel interface L4 V2 periodic L4 threads, which Udo Steinberg introduced within his diploma [Ste04]. The provided mechanisms were developed to support the *Quality-Assuring Scheduling* [HLR+01]. They are abstract and flexible to use them with fixed priority scheduling too. Therefore, I will describe the features of periodic threads in Fiasco in order to estimate, how they can be mapped to Java periodic threads.

A periodic L4 thread can have various execution parts within a period. These parts are called jobs. Further, a priority is associated with each job. Therefore, each job, within a period, can be executed at a different priority. Additionally, a time slice is associated with each job to describe the maximum time the job can run. If a job overruns its time slice or if a periodic thread is still executing at the end of a period, the kernel sends a message, a preemption IPC, to a dedicated thread. This thread is called preempter and receives notifications about time slice overruns or deadline misses of its associated threads. The deadline for L4 threads is the end of the period. A distinction between period end and deadline end within a period is not supported.

The Fiasco kernel does not validate whether a schedule of all currently existing periodic L4 threads is feasible or not. It only informs about occurring overruns of periods as well as misses of deadlines. Services at user level can validate feasibility in advance. They decide whether a thread is admitted to an existing schedule or not.

For the first steps of RTSJ integration, L4 Kaffe shall be able to set priorities and time slices according to the period and worst case execution time specified by Java applications. The period of a Java thread can be directly mapped to the L4 period of the associated native L4 thread. In L4 Kaffe the native L4 thread only requires one job with one time slice per period, because multiple jobs are not specified for Java threads within a period. Further versions of L4 Kaffe can support multiple jobs per period in oder to support the *Quality-Assuring Scheduling* through overloading the periodic Java classes, if this feature is requested. Additionally, the Java handler for overruns of the worst-case execution time can be directly associated with the preemption IPC of time slice overruns, because both have the same semantic.

RTSJ defines that the deadline of a periodic Java thread can be the end of a period or a fixed time relative to the first execution point in time, when the Java thread becomes running within a period. The second possibility is optional and can be provided, when the underlying operating system supports it. The periodic extension of Fiasco does not support this feature. Therefore, within this diploma this second possibility cannot be provided for L4 Kaffe. However, for L4 threads the deadline is equal to the end of a period. Therefore, the deadline preemption IPC can be mapped to the Java deadline miss handler. The restriction is that the Java deadline has to be the same as the end of the Java period. Deadlines unequal to the end of periods will be not supported in L4 Kaffe.

Additionally, in L4 Kaffe a special preempter thread is necessary, which receives preemption IPCs for all Java threads. L4 Kaffe has to associate the preempter thread with each periodic Java thread during creation of the Java thread. The responsibility of the preempter thread is to receive preemption IPCs and to deliver them to L4 Kaffe for further processing. The kernel only buffers one preemption IPC per thread. Therefore, the preemption thread has to be simple and fast in processing IPC messages to miss as few IPCs as possible.

3.1.4 Conclusion

The complete functionality of real-time threads and the handling of their scheduling are not realizable within this diploma thesis. Therefore, I will address only the following aspects, which I regard as essential.

L4 Kaffe shall provide the real-time Java thread classes. The estimated time for the integration of these classes into L4 Kaffe is about two weeks. This step contains writing the Java classes, introducing the distinction of the threads in the internal of L4 Kaffe and the adaptation of the thread creation process. The additional 28 priorities, which are required for the Java real-time threads, can be provided, whereas the introduction of the general concept of SJOs cannot be realized in the scope of this work.

Additionally, L4 Kaffe shall support the basics of periodic Java threads and a simple solution for mapping preemption IPCs to Java handlers. I estimate three weeks to integrate these points. They comprise extending the real-time Java classes, writing basics of the Java handler classes and adding support for periodic Java threads and Java handlers into L4 Kaffe.

An integration of the memory management of RTSJ is required, because of the decision to provide Java real-time threads. Therefore, in the following sections I evaluate the memory management of RTSJ. It is essential to support

NoHeapRealtimeThreads, which cannot be stopped by a garbage collection.

3.2 Java Memory Areas of RTSJ

3.2.1 Concept

RTSJ specifies the concept of Java memory areas as an alternative to the automatic memory management of the Java Virtual Machine. The Java memory areas exist beside the traditional Java heap. They are characterized by their lifetime and the location, where they situate the Java objects. Real-time Java threads can explicitly enter Java memory areas. Each following new instantiation of a Java object is implicitly situated in the current Java memory area of a thread.

In the following description, I use the terms *enter* and *leave* in the context of Java memory areas to clarify the internal behavior of the JVM. However, a Java application cannot explicitly leave a Java memory area by calling a Java method. In the RTSJ API exists no representation for such methods. A Java thread always leaves a Java memory area implicitly by finishing the execution of one of the methods specified by the RTSJ API. These methods are `executeInArea()`, `newInstance()`, `newArray()` and `enter()` in the `javax.realtime.MemoryArea` class. These methods receive an instance derived from `java.lang.Runnable` and call their `run` method. If the execution of a `run()` method is finished, the methods `executeInArea()`, `newInstance()`, `newArray()` and `enter()` return and leave the current Java memory area.

In Figure 3.1 the essential classes of the Java memory areas are shown. The `MemoryArea` class represents a uniform interface for entering and leaving Java memory areas (JMA). `HeapMemory` is an implementation of a JMA class that provides information about the Java heap, which is managed by the garbage collector. The classes `ImmortalMemory`, `ScopedMemory`, and `ImmortalPhysicalMemory` are JMAs, which are not under control of the garbage collector. All Java objects allocated in these JMAs are outside of the traditional Java heap.

The `ScopedMemory` class is a JMA whose lifetime depends on the number of threads running in it. Each real-time thread, which enters such a `ScopedMemory`, increments a counter and decrements it when this thread leaves a `ScopedMemory` area. If the counter value becomes zero, the JVM has to release the Java objects located in the `ScopedMemory`. `VTMemory` and `LTMemory` are implementations of `ScopedMemory`. For `LTMemory` the memory allocation time has to be linear to the requested memory size and may be variable for `VTMemory`.

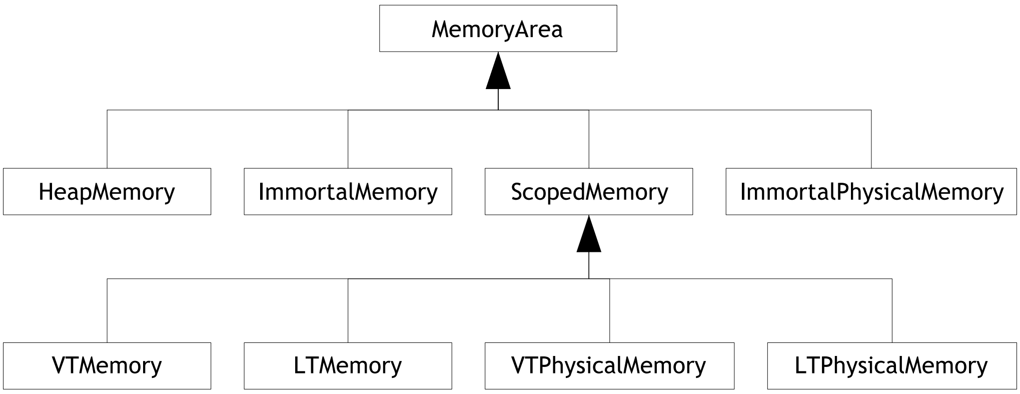


Figure 3.1: MemoryArea classes of the javax.realtime package

Java objects located in a JMA of the class `ImmortalMemory` live as long as the Java application, whereas objects located in `ImmortalPhysicalMemory` can live longer then the application. Further, the classes `ImmortalPhysicalMemory`, `LTPhysicalMemory`, and `VTPhysicalMemory` provide access to physical memory, which is managed by a instance of the `PhysicalMemoryManager` class. The described physical JMAs can specify characteristics, for instance alignment, direct memory access (DMA), or sharing with other applications. The JVM can provide these physical JMAs, if the underlying operating system supports it.

3.2.2 Access and Assignment Rules

The lifetime of `ScopedMemory` areas is limited. If the last thread leaves a `ScopedMemory`, a JVM releases the memory of the Java objects, which are located in this `ScopedMemory`. However, no other Java objects must have any reference to these released Java objects. Otherwise, the access to such invalid, dangling Java object references causes the failure of the JVM. Therefore, RTSJ specifies rules to restrict the free accesses and assignments to Java objects of different JMAs. In Table 3.1 the rules are shown, which restrict the handling of references from `ScopedMemory` areas to `ImmortalMemory` or to `HeapMemory`, because lifetimes of `ScopedMemorys` can be shorter then the lifetime of `HeapMemory` and of `ImmortalMemory`. The specified rules prevent the occurrence of dangling references to Java objects, which a JVM already released.

	Reference to Heap	Reference to Immortal	Reference to Scoped
Heap	Yes	Yes	No
Immortal	Yes	Yes	No
Scoped	Yes	Yes	Yes, if same, outer, or shared memory
Local Variable	Yes	Yes	Yes, if same, outer, or shared memory

Table 3.1: Memory rules as described in the RTSJ specification [BBF+00, p.73]

In order to avoid such dangling references in L4 Kaffe, I have to add these validation rules for each Java byte code instruction that handles Java object references. Therefore, a review of the instruction set of L4 Kaffe is necessary. Each time such a instruction is executed, the validation of the rules has to be performed. Kaffe defines the Java byte code instruction set for interpreter and JIT engine in one file, where I have to add these validations. The Java byte code instructions that have to be adapted will be shown in the implementation chapter of Section 4.3.

Additionally, L4 Kaffe has to determine the Java memory area of Java objects to perform the validations. The detection of the location by comparing the address of Java objects with the borders of all Java memory areas is a simple approach. Another approach is to associate a Java memory area with a Java object. In this solution, an additional reference to the Java object describing the Java memory area is necessary for each Java object. However, the memory of a Java object has to be accessed and used before the location can be detected. If a `NoHeapRealtimeThread` tries to determine the location of a Java object that is situated in the Java heap, the thread will violate the access rule of `NoHeapRealtimeThreads`. In this case, it is also possible that the garbage collector thread already released the Java object, so that the `NoHeapRealtimeThread` accesses invalid memory. Therefore, this second approach cannot be used without an extension, which Beebee and Rinard describe in [BR01] and I outline in the following.

Their idea is to manipulate the reference of a Java object to differentiate heap from non-heap references. The lowest bit of each reference of Java objects is set to identify Java heap references. This bit is not set for references of other Java objects, which are outside of the Java heap. Therefore, each reference has to be unmasked, before the JVM can use the reference to the Java object to access the Java memory area field in it. This solution is problematic for realization within L4 Kaffe, because all allocated memory for Java objects has to be 2-byte-aligned. This means that memory references have to be even, the lowest bit of the reference is zero. Unmasking of each Java object reference is necessary for each Java byte code instruction handling with object references. Additionally, the references have to be unmasked for all de-referencing instructions within the source code of L4 Kaffe and for all Java object references used in native Java libraries. This would require an extensive code review, for which I cannot estimate the required time and which is intractable in the scope of this work.

After the consideration of these approaches, I decided to use the detection of the location of a Java object by comparing the address of this object with the borders of all Java memory areas. However, optimizations will be necessary to minimize the time to find a location of a Java object in application scenarios with many Java memory areas.

3.2.3 Management and Nesting

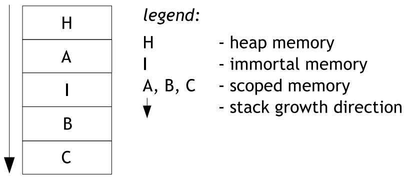


Figure 3.2: B and C are inner ScopedMemory areas of A; A and B are outer ScopedMemory areas of C

Real-time threads are able to enter Java memory areas nested, which means that an order of entered Java memory areas is established during runtime. This ordering has to be managed by a JVM in order that this JVM is able to find and set the previous Java memory area of a thread, when it leaves the current Java memory area. RTSJ recommends a scoped stack for each Java thread, which tracks the order. Additionally, RTSJ differs inner and outer `ScopedMemory` areas. A `ScopedMemory` area is an inner area, when it is pushed on top of another `ScopedMemory` area on the Java scoped stack of a Java thread. In Figure 3.2 an example Java scoped stack is illustrated, which shows the `ScopedMemory` areas A, B, C, the `HeapMemory` area H, and the `ImmortalMemory` area I. B and C are inner `ScopedMemory` areas of A. In contrast, A and B are outer `ScopedMemory` areas of C, because A and B were pushed before C on the scoped stack. Additionally, outer `ScopedMemory` areas are all areas that are not on a Java scoped stack as to the `ScopedMemory` areas on this stack.

Additionally, RTSJ defines restrictions of the nested usage of `ScopedMemory` areas. References of Java objects to `ScopedMemory` areas become invalid, when the last thread leaves a `ScopedMemory` area. This means, the lifetime of this area ends. Therefore, references to Java objects located in such a `ScopedMemory` area must not be in use by other Java objects located in outer `ScopedMemory` areas. Hence, RTSJ defines that references to Java objects of an outer `ScopedMemory` can be assigned only to Java objects of inner `ScopedMemory` areas that have a shorter lifetime.

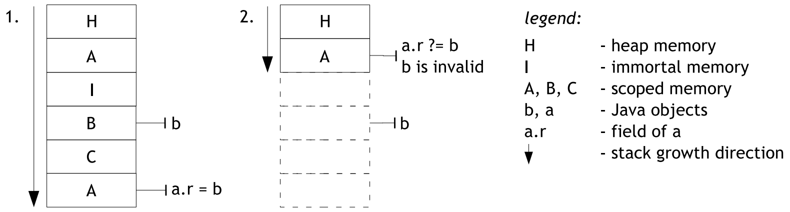


Figure 3.3: Emergence of dangling references on a Java scoped stack, when no Single Parent Rule is used.

However, references to Java objects of released `ScopedMemory` areas can still emerge, which I outline by an example in the following. RTSJ addresses this problem by the definition of the *Single Parent Rule*. But before I describe this rule in detail, I illustrate in the Figure 3.3 a scenario, which shows how this problem can occur. The following assumption is that a `ScopedMemory` can be put multiple times on a Java scoped stack, which is not permitted by the *Single Parent Rule*. In the figure, twice a `ScopedMemory` area `A` with a Java object `a` is on a Java scoped stack. It can obtain references to Java objects of outer `ScopedMemory` areas, for example to a Java object `b` saved in `a.r`. At one point in time the `ScopedMemory` area `A` and all other `ScopedMemory` areas, also `B`, are left until the thread is again in the first `ScopedMemory` area `A`. Now, the reference in `a.r` to the Java object `b` is invalid, because the `ScopedMemory B` is no longer on the Java scoped stack and the memory of `B` was released. An access to the Java object `b` will lead to failures in the JVM.

The same problem appears, when a `ScopedMemory` area is used on multiple scoped stacks by various threads and the order of entered `ScopedMemory` areas is different for this `ScopedMemory`. In Figure 3.4 two threads use a `ScopedMemory`

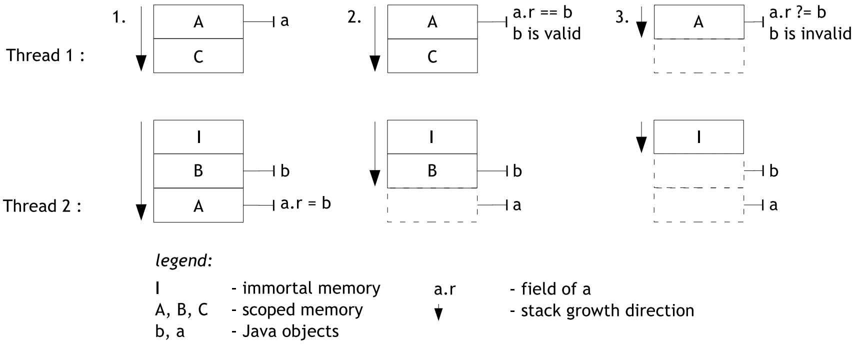


Figure 3.4: Emergence of dangling references on multiple Java scoped stacks, when no Single Parent Rule is used.

A. In the first subpicture the Java object b is assigned to the field a.r of a Java object a by thread 2. In the second subpicture thread 2 left the ScopedMemory A. The reference to b in a.r is valid, because B and A are still in use. In the third subpicture thread 2 left B and b becomes invalid. Additionally, thread 1 left C. If thread 1 uses the reference in a.r, it will access a invalid Java object. This access will lead to failures in the JVM.

Therefore, RTSJ defines the *Single Parent Rule* to prevent the usage of ScopedMemory areas multiple times on the same Java scoped stack and on various Java scoped stacks, if the order of entered ScopedMemory areas differs for a ScopedMemory area on these Java scoped stacks. A JVM uses this rule to detect cycles of entered ScopedMemory areas on Java scoped stacks. Therefore, RTSJ defines that each ScopedMemory area must have none or exactly one ScopedMemory area as parent, which can be another ScopedMemory area or the *primordial scope*[1]. If a Java thread tries to enter a ScopedMemory area, the parent rule is validated. The parent of this ScopedMemory area has to be equal to the last ScopedMemory area on the current Java scoped stack of a Java thread. If this does not hold, the *Single Parent Rule* is violated. L4 Kaffe has to prevent a Java thread from entering the ScopedMemory area by throwing a ScopedCycleException. A ScopedMemory area gets a parent, when L4 Kaffe pushes it the first time on a Java scoped stack. In contrast, a ScopedMemory area looses its parent, when L4 Kaffe pops the ScopedMemory area from a Java scoped stack and when it is not on any further scope stacks.

Additionally, each RealtimeThread can have multiple Java scoped stacks

[1] The phrase *primordial scope* encloses the ImmortalMemory and the traditional *Java heap* in RTSJ.

in order that these threads can place new Java objects in outer ScopedMemory areas. RTSJ defines that a JVM has to start a new Java scoped stack and to save the current, the old, Java scoped stack, when the methods executeInArea(), newInstance() and newArray() in javax.realtime.MemoryArea are executed. The JVM has to restore the old Java scoped stack after completion of the execution of the mentioned methods. Java threads are also able to nest calls to these three methods. Consequently, a management of multiple Java scoped stacks is necessary for each Java thread.

In Figure 3.5 an example of nested Java scoped stacks for one Java thread is illustrated. The first subpicture shows a current Java scoped stack of a Java thread. A ScopedMemory C is entered by the executeInArea() method, which causes a creation of a new Java scoped stack. The executeInArea(), newInstance() and newArray() methods can be only used with ScopedMemory areas, if these areas are on the current Java scoped stack. The second subpicture shows both stacks after calling the executeInArea() method. The stack with the dashed lines is inactive. Then a ScopedMemory D is entered by enter() and the resulting stacks are shown in subpicture 3. Now, the ImmortalMemory I is entered, which causes the creation of a new Java scoped stack. The ImmortalMemory area and HeapMemory area can be entered without restrictions in contrast to the ScopedMemory areas. The subpicture 4 shows the resulting stacks. In the end, the current Java thread leaves the ImmortalMemory I, the ScopedMemorys D, C, and B. The Java scoped stacks are released, when the called executeInArea() methods return.

For integration into L4 Kaffe, a Java scoped stack implementation is necessary. Further,

1. the L4 Kaffe thread data structure has to be extended by a reference to the current Java scoped stack of a Java thread.
2. Another reference for the management of currently not used Java scoped stacks is necessary.
3. The *Single Parent Rule* has to be validated for each ScopedMemory area pushed on the Java scoped stack.

3.2.4 Garbage collector interference

The garbage collector thread has to know, which Java objects located in Java memory areas have references to objects in the Java heap. During a garbage collection, it has to find all references to Java objects, which are managed by it,

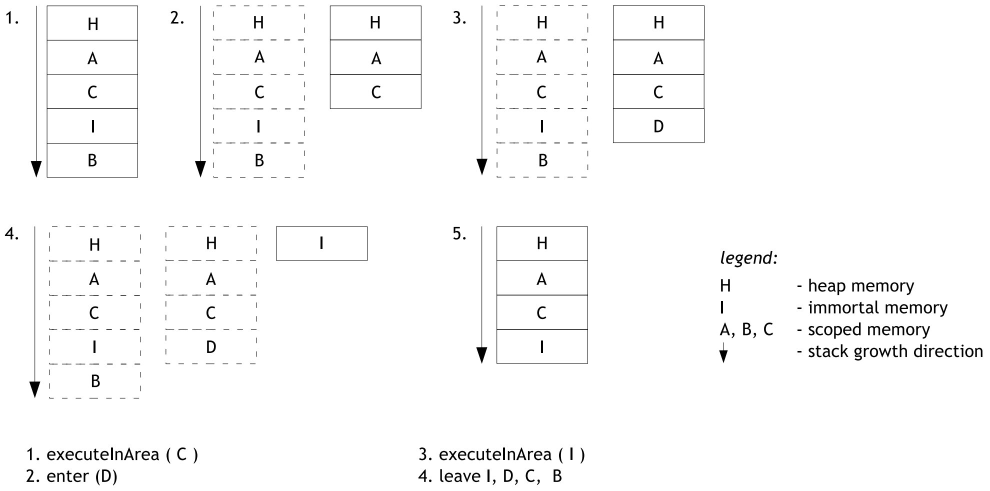


Figure 3.5: Example of multiple Java scoped stacks of a thread

in order to mark them as *in use* or *not in use*. Otherwise, the garbage collector could remove a Java object of the Java heap, which is only referenced by Java objects of Java memory areas. These invalid references to the Java heap would lead to errors in later execution of a JVM.

Therefore, the Java memory area implementation has to provide a function with which the garbage collector thread can scan the Java memory areas for Java heap references. If a reference to the Java heap is found, the garbage collector specific `mark()` routine will be called. This routine will mark the found Java object as reachable, which means it is still in use, through adding it to the garbage collector internal list of used objects. The garbage collector thread cannot call its own `mark()` routine for objects of a Java memory area, because specific data structures are not available for non Java heap objects.

When the scanning of the Java memory areas is finished, the normal garbage collection can be continued. Only `NoHeapRealtimeThreads` can run concurrently to the garbage collector thread. All other threads are stopped. Therefore, Java heap references cannot be added to Java objects of Java memory areas during a garbage collection, because `NoHeapRealtimeThreads` are not allowed to use them. All Java heap references are found between scanning the Java memory areas and the end of the normal garbage collection, so that no dangling pointers can originate.

3.2.5 Conclusion

The concept of the Java memory areas is a central point of the RTSJ, which is essential to support the real-time Java thread class `NoHeapRealtimeThread`. L4 threads associated with this Java thread class will not be stopped by a garbage collection in L4 Kaffe. Therefore, the execution characteristic of such a Java thread will be better predictable than for Java threads of the class `java.lang.Thread` and `javax.realtime.RealtimeThread`.

Therefore, L4 Kaffe has to support the complete memory management of the RTSJ. For this purpose, I have to integrate the following essential parts into L4 Kaffe:

1. the Java memory area classes `MemoryArea`, `ScopedMemory`, `HeapMemory`, and `ImmortalMemory`,
2. the Java scoped stack associated with each Java thread,
3. the *Single Parent Rule* to prevent cycles of used `ScopedMemory` areas and
4. the garbage collector adaptation.

Additionally, I have to review the Java byte code instructions to integrate the access and assignment rules of the Java memory areas and of `NoHeapRealtimeThreads`.

The estimated integration time for the Java classes, the Java scoped stack and the Single parent rule is six weeks and for the adaptation of the garbage collector is two weeks. Additionally, reviewing of the Java byte code and integration of the access and assignment rules will consume approximately two weeks. Further, the adaptation of the standard Java API of L4 Kaffe will consume two weeks in order that `NoHeapRealtimeThreads` can use the classes of this API without violations of the access and assignment rules.

3.3 Synchronization Enhancements

3.3.1 Requirements

RTSJ requires the avoidance of *priority inversion* for the keyword *synchronized*. In the Java language this keyword is the build in mechanism for mutual exclusion of Java threads, which share a critical section. A monitor is associated with each Java object, when methods or blocks in the Java class of a Java object are marked as *synchronized*. A JVM prevents Java threads from concurrent execution within these critical sections.

The *priority inheritance* protocol is specified as default *priority inversion* avoidance algorithm, the *priority ceiling* protocol is optional. Additionally, RTSJ requires that the default behavior of monitors has to be replaceable at runtime. Therefore, RTSJ introduces the abstract class `MonitorControl`, which manages the monitors for Java objects. These monitors can be set for each Java object separately through the `MonitorControl` class. The default system-wide monitor implementation has to be the *priority inheritance* protocol, which is represented by the `PriorityInheritance` class. Respectively, the `PriorityCeilingEmulation` class represents the monitor implementation of the *priority ceiling* protocol. The Java application can change the default behavior of all monitors by setting another *priority inversion* avoidance protocol in the `MonitorControl` class at runtime.

Further, RTSJ describes that the use of *synchronized* Java objects can lead to delays of a `NoHeapRealtimeThread`, when another thread, a `RealtimeThread`, or a `java.lang.Thread`, owns the monitor of the same Java object and the garbage collector stops that thread during a garbage collection. Therefore, RTSJ introduces the classes `WaitFreeWriteQueue`, `WaitFreeReadQueue`, and `WaitFreeDequeue`. These classes shall be used instead of the keyword *synchronized* for synchronization between `NoHeapRealtimeThreads` and `RealtimeThreads` as well as `java.lang.Thread`. The queues have an associated reader and writer thread, which can read from or write to the queue without blocking.

3.3.2 Locking Modifications

In L4 Kaffe, each Java object has to be extended by an additional reference to its associated monitor. If there is no reference, L4 Kaffe has to use the default monitor control object, which is accessible with the help of the `MonitorControl` class. Kaffe uses its own *locking scheme* for synchronization, which first tries to protect the Java object using atomic *compare and swap* operations. If another Java thread already runs in the Java object, Kaffe requires specific synchronization mechanisms provided by the operating system. In L4 Kaffe, it is the semaphore implementation of L4Env.

However, the semaphore implementation of L4Env provides no support for the prevention of *priority inversion*. Therefore, the following consideration shall outline problems of the current critical section implementation of L4 Kaffe and required characteristics of a semaphore implementation for L4 Kaffe.

A problem is the locking scheme of Kaffe, because it works independently of the additional semaphore implementation. If a thread t_1 acquires a monitor by atomic operations, the semaphore implementation is not involved, because the lock implementation of Kaffe tries to minimize the use of semaphores. If another thread t_2 tries to acquire the same monitor, the semaphore implementation queues t_2. Since t_1 is not registered in the semaphore implementation, the priority of t_1 and t_2 cannot be compared. Therefore, the priority of t_1 cannot be increased, when it becomes necessary. Therefore, the multi-level synchronization mechanism of Kaffe is unsuitable for avoiding *priority inversion*.

The conclusion is that only one level of synchronization is possible. It can be either a specific and close integration of Kaffe's locking scheme with the semaphore implementation of L4Env or a complete replacement that only uses the specific operating system synchronization mechanism. The first approach can lead to a bad portability of Kaffe, because it closely interacts with L4Env. In contrast, the second variant is better in terms of portability, but requires a comprehensive reengineering of the locking layer in Kaffe. Additionally, the second approach leaves the task of priority inversion avoidance in the responsibility of the operating system and its runtime environment, because they know about the internal activities and how priority inversion avoidance can be achieved. Therefore, a semaphore implementation has to provide an interface in order that L4 Kaffe can specify a *priority inversion* avoidance protocol for each critical section. Additionally, the protocol has to be replaceable at runtime for each critical section. RTSJ specifies for the `setMonitorControl()` method of the `javax.realtime.MonitorControl` class that a change of a *priority inversion* protocol of a monitor takes effect on the next attempt to lock this monitor after the completion of the `setMonitorControl()` method.

A semaphore implementation, which supports the *priority inversion* avoidance protocols, has to track the semaphores, which are used by each thread. If priorities of threads are changed in order to prevent *priority inversion* and if they have to be set back later, the semaphore implementation has to determine whether the current thread owns other semaphores. In such a case, the priority of the current thread has to be set to the highest priority of all threads, which are waiting for a semaphore that the current thread owns.

Additionally, RTSJ specifies that changing priorities at runtime by the Java application must not affect the correctness of any priority inversion avoidance algorithm. Therefore, the semaphore implementation has to handle priority changes at runtime of threads, which are enqueued to semaphores. This requires moving the waiting thread from its current waiting position within the queue to the position according to its new priority. This is necessary to guarantee that threads enter the semaphore in the correct order.

Additionally, setting of priorities by Java threads has to be synchronized with the execution of the semaphore thread, because it can lead to incorrectly set priorities. If a semaphore implementation sets the priority of a thread t_1 back to an older value and if another thread t_2 changes the priority of the thread t_1 at the same time, the resulting priority of t_1 has to be the one set by t_2. Synchronization is necessary to avoid such situations. Therefore, a semaphore implementation will have to synchronize itself, an additional form of synchronization beside the semaphore implementation will be necessary, or the semaphore thread will have to be the only one that can change Java priorities.

The mentioned problems in L4 Kaffe and of a semaphore implementation have to be considered in order to support the *priority inversion* avoidance protocols.

3.3.3 Conclusion

To support the requested extensions of RTSJ in L4 Kaffe, the following extensions and modifications are necessary:

1. Priority inheritance implementation in the L4 semaphore package;
2. An optional implementation of the priority ceiling protocol;
3. Support for changing priorities of threads already blocked on a semaphore;
4. Adaptations of Kaffe to use only the provided operating system synchronization mechanisms;
5. Support for interchangeable priority inversion avoidance implementations during runtime.

The tasks described above are complex. Therefore, I assume that the effort to implement the *priority inversion* avoidance algorithms and to integrate them into L4 Kaffe is high. Additionally, a solution shall not enormously slow down the critical section implementation in comparison to the current L4 semaphore implementation.

Because of the complexity, the estimated time for the integration is two to three months. The design and integration of the *priority inversion* avoidance algorithms and mechanisms in L4 Kaffe and L4Env are not realizable within this thesis, because of the limited time available for it.

3.4 Asynchronous Extensions

3.4.1 Overview

Java has rudimentary asynchronous support in form of the `interrupt()` method in the `java.lang.Thread` class, which sets a flag in the target thread. The target thread has to poll the flag to consume the event and has to reset it to become newly interruptible. If the target thread is blocked within the `wait()`, `sleep()`, or `join()` methods of `java.lang.Thread`, the JVM will throw a Java exception and the target thread will become unblocked. Now, the runnable thread has to handle the thrown Java exception.

RTSJ introduces support for real asynchronous interruption of Java threads. The class `javax.realtime.AsynchronouslyInterruptedException` (AIE) indicates for a JVM whether a method or a constructor supports asynchronous interruptions. A JVM can interrupt Java threads executing such methods or constructors immediately. However, if a thread runs in a *synchronized* method or *synchronized* block, the AIE will be delayed until the thread leaves the synchronized area. The delay is always necessary independently of the declaration of an AIE in a throw clause of methods or constructors.

The class `javax.realtime.AsyncEvent` describes events that can occur any time. These events can be caused by a Java application, by the Java Virtual Machine or by external events such as interrupts. Java applications can register `javax.realtime.AsyncEventHandler` for each asynchronous event, which the JVM has to execute in case of an event. These Java handlers are not bound to a specific Java thread. A JVM associates these handlers to a Java thread, when the asynchronous event occurs.

3.4.2 Event Handling

In L4, external events are characterized by synchronous IPC messages sent to L4 applications. L4 threads can only receive an IPC, when they are waiting for it. Therefore, in L4 Kaffe one or multiple L4 threads are necessary to receive external events sent by the operating system like timer events or interrupts. The L4 threads have to find registered Java event objects, which are bound to these external events. If such a Java event object exists, L4 Kaffe has to execute the associated Java handler. Therefore, in L4 Kaffe a central registration management for all asynchronous events is necessary.

Associated Java handlers of Java events cannot be executed in the context of the L4 thread, which received the external event. One reason is that some Java handlers can be explicitly bound to one Java thread that has to execute the specific Java handler. Additionally, a JVM cannot determine how long a Java handler is executed in advance. The L4 thread that executes the Java handler can miss IPCs, when the execution takes long and when sender threads of event IPCs use short timeouts. Therefore, the L4 threads receiving external events have to be able to accept the next IPC quickly. A solution is that these receiving L4 threads put messages about the external events in a list and then immediately wait for new IPCs. Other Java threads are responsible for consuming the events in the list. The threads have to find the event Java object for an external event as well as have to execute the associated Java handler for this event.

Java threads can also initiate asynchronous Java events. They can directly call the `fire()` method of a Java event object. In such a situation, seeking for the Java event object is not necessary, because the firing Java thread directly delivers the Java event object to the pool of Java threads that are responsible for the execution of the Java handlers.

3.4.3 Transfer of Control

RTSJ extends the behavior of the `interrupt()` method of the `java.lang.Thread` class by real asynchronous interruption. If a Java thread is interrupted while it is in execution within a method or constructor that declares an AIE in the throw clause of its method declaration, then the JVM has to asynchronously throw an AIE. However, if these methods or constructors are executed in the context of a *synchronized* method or block, the JVM has to delay the AIE until the critical section is left. RTSJ names all such areas, where AIEs have to be delayed, *deferred sections*.

To support the asynchronous transfer of control (ATC) mechanism, a JVM has to be able to detect whether a method or constructor declares an AIE. To detect this, L4 Kaffe has to scan the Java frame of the interrupted Java thread for such an AIE. The Java frame is put on the Java stack with each Java method invocation. If the AIE declaration is found on the Java frame, L4 Kaffe has to throw the AIE. However, if a Java thread is in a *deferred section*, the AIE has to be associated with the Java thread and has to be set pending. RTSJ specifies that a JVM has to throw the exception, when the Java thread returns from a *deferred section* into a *non-deferred section* or when the Java thread enters a *deferred section*. If the AIE is thrown, a JVM uses the normal exception handling mechanism of Java for seeking and finding a Java exception handler in one of the Java frames on the current Java stack of the Java thread.

If a Java exception handler for the AIE exception is found, then L4 Kaffe will execute this Java exception handler. Further, if an AIE exception is caught but not acknowledged within the Java exception handler, L4 Kaffe has to forward the AIE exception. RTSJ specifies that an AIE has to be explicitly acknowledged by calling the AIE method `happened()`. Therefore, in L4 Kaffe the exception handling has to be extended by the validation of the AIE state. If the AIE is not acknowledged and the current Java exception handler is left, then L4 Kaffe has to forward, to throw, the AIE exception until another Java exception handler is found for the AIE.

If L4 Kaffe has to interrupt a target Java thread because of an AIE, L4 Kaffe has to throw the AIE in the context of this target Java thread. Therefore, L4 Kaffe has to verify whether the target Java thread is in a *deferred section*. However, the target Java thread can be executed concurrently to the thread of L4 Kaffe that has to throw the AIE. Therefore, L4 Kaffe cannot reliably determine whether the target thread is in an *deferred section* and whether L4 Kaffe has to throw the Java exception.

A possible solution is that the target Java thread is the one to determine whether it is in a *deferred section* and whether it has to throw the AIE. The idea is that L4 Kaffe interrupts the native L4 thread of the target Java thread and hands the AIE to the L4 thread without throwing it. If the Java thread is not in a *deferred section*, the native L4 target thread throws the AIE. Otherwise, the AIE exception is saved and marked as pending and the old execution of the target Java thread is continued, where it was interrupted temporarily.

The L4 kernel provides a system-call `thread_ex_regs`, which can asynchronously interrupt a thread by setting it to a new instruction pointer. L4 Kaffe can change the execution path of the native L4 thread with the help of this system-call. After exchanging the instruction pointer, the native L4 thread has to save its thread status, such as registers, instruction pointer, and stack pointer. The saved thread state is necessary to restore the execution path of the L4 native thread at the interrupted instruction, if the AIE must not be thrown. After saving the thread state, the native L4 target thread has to determine whether the associated Java thread is in a *deferred section*.

Further, all critical sections within the virtual machine of L4 Kaffe have to be considered in the context of asynchronous transfer of control. They must not be interrupted and aborted, because they change status information of the JVM, which affect the stability of L4 Kaffe. Therefore, these internal critical sections have to be marked as *deferred sections* too.

3.4.4 Thread Termination

RTSJ specifies no special mechanism for asynchronous thread termination. Rather, it defines that asynchronous thread termination can be achieved by the usage of asynchronous transfer of control and the asynchronous exception handling.

3.4.5 Conclusion

The event handling mechanism can be realized by mapping L4 IPCs to Java event objects. The integration of the ATC concept into L4 Kaffe is complex, because all critical sections in L4 Kaffe have to be declared as deferred. Additionally, the complete handling of the AIE and its propagation by using the `thread_ex_regs` system-call has to be integrated.

The estimated time to integrate these tasks is about one to two months, because of their complexity. On the basis of these facts and the limited available time, these tasks cannot be realized within this thesis.

3.5 Time and Timers

3.5.1 Overview

RTSJ defines classes to describe points in time. The abstract class `HighResolutionTime` provides basic operations to compare points in time and to transform them into relative or absolute points in time. In RTSJ points in time are stored as a Java `long` value, which is a 64 bit signed number, in milliseconds and as a Java `int` value, which is a 32 bit signed number, in nanoseconds. Both values added represent the final time value. The subclasses `AbsoluteTime` and `RelativeTime` describe absolute and relative points in time. Absolute times are relative to midnight January 1st, 1970 GMT.

Additionally, RTSJ introduces timer classes, which allow a Java application to specify points in time, when the JVM shall wakeup threads or shall initiate asynchronous events. The abstract class `Timer` is a specialized asynchronous Java event that has to occur at one point in time, relative or absolute to a clock. The subclasses `OneShotTimer` and `PeriodicTimer` can be used to execute asynchronous events only once or periodically.

3.5.2 Relative and Absolute Points in Time

Fiasco provides the mapping of the internal kernel CPU time to L4 user level applications. L4 applications can use this time value to compute absolute points in time. This can be used for computing both, the start and release times of asynchronous Java events as well as for periodic Java threads.

The transformation of relative points in time in Java to the internal kernel CPU time is realizable with simple arithmetic operations. The transformation of the absolute Java time to the internal kernel CPU time requires the knowledge of the system date and time. The server *rtc* provides the seconds since 1970. Additionally, the server provides the C-library binding for `gettimeofday()` and `time()`. Therefore, L4 Kaffe can determine the current date and time, compare it to the absolute Java time and use this result, the difference, to compute the absolute time relative to the kernel CPU time.

3.5.3 Timer

L4Env provides a *timer* server, which programs the timer of the APIC of a CPU. The server provides for L4 threads timeouts with microseconds granularity. It programs for the next thread, which must be waken, the APIC timer and waits for the incoming interrupt IPC. The kernel generates this IPC, when the APIC timer raises an interrupt. The *timer* server responses to the waiting L4 threads, after receiving the interrupt IPC from the kernel.

L4 Kaffe can use this L4Env *timer* server for its Java timers. The point in time of a Java timer has to be sent to the L4Env *timer* server, so that it can program the APIC accordingly to the time when the interrupt shall be raised by the hardware. Therefore, the client API of the L4Env *timer* server has to be adapted, because it does not support the registration of a specific point in time. If the L4Env *timer* service receives an interrupt IPC from the kernel, it will send an IPC to the native L4 thread of L4 Kaffe, which requested to be notified at this point in time. This native L4 thread has to find the corresponding Java timer objects. If the Java timer objects exist, then Kaffe will execute the associated Java handlers accordingly to their release and schedule parameters.

Additionally, the POSIX signal *SIGALARM* can be used to implement timers in L4 Kaffe. RTSJ defines the `javax.realtime.POSIXSignalHandler` class, which provides signal bindings for the `javax.realtime.AsyncEvent` class, if POSIX signals are supported by the underlying operating system of a JVM. During this diploma an implementation of the POSIX signals was started in order to support them in the L4Env adapted DietLibC. This approach could not be considered within this diploma, because the POSIX signals were not available at the beginning of this thesis. However, with this implementation L4 Kaffe could support the `javax.realtime.POSIXSignalHandler` class and POSIX signals in Java.

3.5.4 Conclusion

The extension of the *timer* server is realizable in the scope of this work. It is a step after the integration of the asynchronous Java event concept in L4 Kaffe. The estimated time for the adaptation of the *timer* server and the integration in L4 Kaffe is two weeks, when the asynchronous event handling implementation in L4 Kaffe is finished.

4 Implementation

In this chapter, I describe aspects of the integration of RTSJ into L4 Kaffe. It provides information about the used mechanisms and characteristics of the current implementation. Additionally, problems are discussed, which appeared during the integration of RTSJ in L4 Kaffe.

4.1 Java Thread Data Structure

L4 Kaffe has to identify the various Java threads at runtime in order to validate the right access and assignment rules of Java objects and to let `NoHeapRealtimeThreads` execute during a garbage collection. A standard mechanism of the Java language is the verification of Java objects whether they are instances of a specific requested class. This variant can be used to distinguish the three Java thread classes `java.lang.Thread`, `javax.realtime.RealtimeThread` and `javax.realtime.NoHeapRealtimeThread`. Another solution is to add a thread type field flag into the internal thread structure representation of Kaffe.

An essential requirement for thread type distinction is that this operation proceeds fast: Each Java byte code instruction, which handles Java object references has to distinguish these three thread types. However, instance checking is slow in contrast to accessing a flag, because L4 Kaffe first has to determine the class of an object, and then it has to seek the complete inheritance class hierarchy for the requested class. Only accessing and checking a type field is faster, because the flag is directly accessible without seeking.

Therefore, I extended the Java thread structure in L4 Kaffe by a field, which indicates the type of the Java thread. Additionally, a reference to the associated scoped stack was necessary.

4.2 Garbage Collection

The Java thread type distinction is necessary in the thread layer of L4 Kaffe (drops-l4threads) too. Instances of `java.lang.Thread` and `javax.realtime.RealtimeThread` are stopped, when Kaffe has to execute a garbage collection. Only instances of `javax.realtime.NoHeapRealtimeThread` remain executable as requested by the RTSJ.

Every time a real-time thread enters a `ScopedMemory` area, a binary semaphore is acquired for a short critical section to protect the incrementing of the `ScopedMemory` area counter. If this counter is decreased to zero, Java objects have to be released within this critical section by the last Java thread. However, this process takes a certain time, because of the execution of the `finalize()` methods of each Java object. If a `RealtimeThread` is in such a short critical section, the garbage collector can stop it. If this thread is stopped, also `NoHeapRealtimeThreads` can be delayed, when they have to increment this counter within the same critical section.

To avoid this problem, I added an additional binary semaphore around the binary semaphore of the critical section. However, only `RealtimeThreads` and the garbage collector thread have to acquire this additional semaphore. In contrast, `NoHeapRealtimeThreads` skip the outer semaphore of the critical section and jump to the inner semaphore directly. The garbage collector has to acquire the outer semaphore, before it can stop instances of `java.lang.Thread` and `javax.realtime.RealtimeThread` to start the garbage collection. Therefore, all `RealtimeThreads` must release the outer semaphore before a garbage collection. When the garbage collector has acquired the outer binary semaphore, no other thread of `java.lang.Thread` or `javax.realtime.RealtimeThread` is in the critical section. Therefore, the garbage collector can stop these threads without delaying `NoHeapRealtimeThreads`. `NoHeapRealtimeThreads` has to acquire only the inner semaphore and not the outer semaphore, which the garbage collector holds during a collection.

The inner semaphore cannot be used by the garbage collector. If the garbage collector thread used the inner semaphore, this thread would share it with `NoHeapRealtimeThreads` and would be able to delay the `NoHeapRealtimeThreads` during a garbage collection.

4.3 Java Memory Areas

Memory in L4Env is handled as a resource. Therefore, an L4 application can request special characteristics for memory, like access to specified physical memory or location within the virtual address area of the running application. I used this feature to reserve an address region in the virtual address area, where all Java memory areas have to attach their memory to L4 Kaffe. The memory of the Java heap is located somewhere outside of this reserved memory address area. Therefore, L4 Kaffe can easily distinguish between references of Java objects from the Java heap and from Java memory areas. All references outside of the reserved region are interpreted as references to the Java heap. Java memory areas are managed using a statically allocated list. Each Java memory area within L4 Kaffe is identified by a number, `memID`. The memory layout of the Java memory areas is illustrated in Figure 4.1.

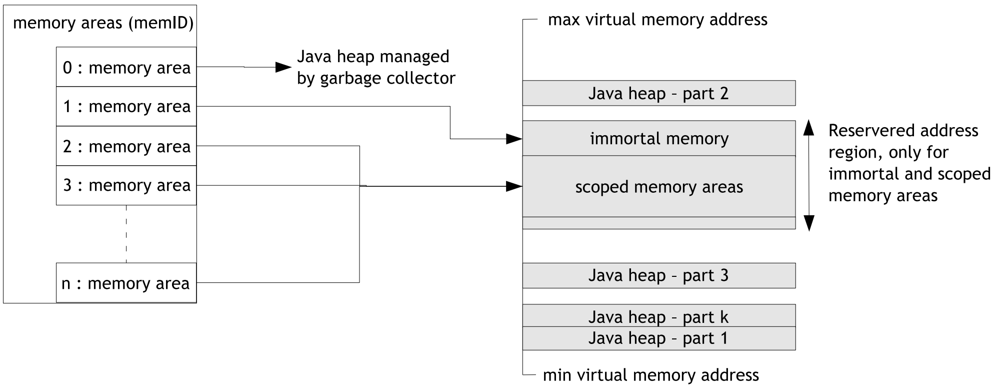


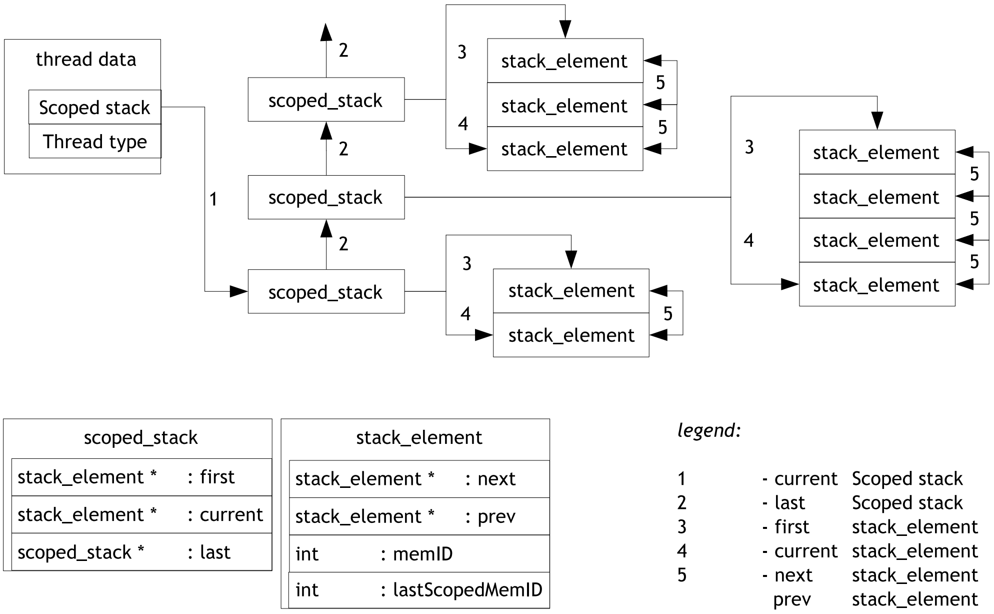
Figure 4.1: Internal Java memory area management in L4 Kaffe

The allocation scheme of the Java objects within the Java memory areas is the same for the `ImmortalMemory` area and `ScopedMemory` areas. Java objects are located in the memory areas successively, whereas each Java object has a reference to the successor Java object. The linkage of Java objects is necessary in order to search each of them for references to Java objects in the Java heap, when a garbage collection cycle is started.

4.3.1 Java Scoped_Stack Layout

Real-time Java threads can use Java memory areas nested. The Java scoped stack is required in order to manage the Java memory areas per Java thread. The management of the stack in L4 Kaffe was realized with two data structures to handle the multiple Java scoped stacks per Java thread and its entries within a Java scoped stack. In Figure 4.2 the interrelation between these structures is shown. Multiple `scoped_stacks` are organized in a single linked list, whereas the reference `last` in the `scoped_stack` data structure points to the scoped stack that was active before the current scoped stack was started. Additionally, each scoped stack has a reference to the first Java memory area of the scoped stack and to the current Java memory area.

The structure stack_element is associated with a Java memory area by the memory ID, memID. In lastScopedMemID the identifier of the last ScopedMemory area on the current scoped stack is saved for each stack_element. This identifier is used to avoid seeking for the last ScopedMemory area on the current stack in order to validate the *Single Parent Rule*, when a ScopedMemory area is pushed on the scoped stack.



scoped_stack	
stack_element *	: first
stack_element *	: current
scoped_stack *	: last

stack_element	
stack_element *	: next
stack_element *	: prev
int	: memID
int	: lastScopedMemID

legend:

1	- current	Scoped stack
2	- last	Scoped stack
3	- first	stack_element
4	- current	stack_element
5	- next	stack_element
	prev	stack_element

Figure 4.2: Scoped stack of Java threads in L4 Kaffe

4.3.2 Finalization of Java Objects in ScopedMemory Areas

The Java thread that leaves the ScopedMemory area as the last one is responsible for executing the finalize() methods of the Java objects of this ScopedMemory area immediately. I decided to implement it such that a Java developer knows when the finalize() methods will be executed and which Java thread executes these methods. This information is important in order to estimate possible delays of other real-time Java threads in the Java application, when they use the same ScopedMemory area. A ScopedMemory area can be reentered only, when the finalization of a ScopedMemory area is finished.

In contrast, the finalize() methods of the Java objects of the Java heap are executed by a special *finalizer thread*. I did not implemented the finalization for `ScopedMemory` areas as a separate real-time *finalizer thread*, because the finalization of the Java objects of a `ScopedMemory` area has to be executed with the same Java scoped stack entries before this area was left. Therefore, Java scoped stacks of the last real-time Java thread would have to be copied to provide the same Java scoped stack for a special real-time *finalizer thread*. Further, a developer could not easily determine when and how long this *finalizer thread* is executed, because the *finalizer thread* could still execute a finalization of another `ScopedMemory` area.

4.3.3 Java Bytecode Instructions

I analyzed the Java byte code instruction set of the Java Virtual Machine specification, because I had to find all instructions that handle Java object references. This task was necessary in order to add the access and assignment rules of Java objects of various memory areas as described in the Section 3.2.2 “Access and assignment rules”. Additionally, the validation of the adherence to the restriction of `NoHeapRealtimeThreads` had to be added. The restriction prohibits the usage of references of Java objects situated on the Java heap as described in Section 3.1.1.

I found 42 Java byte code instructions, which handle Java object references. They are listed in Table 4.1. These instructions are extended by the validation of the mentioned rules. Every time L4 Kaffe executes one of the listed instruction, the validation is performed. If L4 Kaffe detects a violation, it will throw the appropriate Java `javax.realtime.MemoryAccessError` with a short message about the violation reason.

aload	astore	ldc	invokevirtual	monitorenter
aaload	aastore	ldc_w	invokespecial	monitorexit
iaload	iastore	new	invokeinterface	—
faload	fastore	newarray	getstatic	ifnull
laload	lastore	anewarray	getfield	ifnonnull
daload	dastore	multinewarray	putstatic	—
baload	bastore	arraylength	putfield	areturn
caload	castore	checkcast	if_acmpeq	athrow
saload	sastore	instanceof	if_acmpne	—

Table 4.1: JVM instructions handling Java object references

If a Java thread leaves a `ScopedMemory` area and a Java exception or a Java error is pending, i.e., is not caught by the Java application, the Java object, which represents this Java exception or the Java error, will not always be delivered from the ScopedMemory area into the previous memory area of this Java thread. It is not delivered, when this Java object is situated in the ScopedMemory area, because this Java object becomes invalid outside its ScopedMemory area. In such a case, L4 Kaffe replaces the Java object of the Java exception or of the Java error by a javax.realtime.ThrowBoundaryError Java object that is located in the previous memory area. This behavior is required by the RTSJ specification.

4.4 Periodic Java Threads

In order to be able to run periodic Java threads, I implemented the Java classes ReleaseParameters and PeriodicParameters of the RTSJ, which were mentioned in Section 3.1.3. Additionally, the basics of the classes HighResolutionTime, AbsoluteTime, and RelativeTime mentioned in Section 3.5.1 are implemented in order to support the execution of periodic Java threads with release times of absolute and relative points in time. I realized the mapping of a Java point in time to the kernel CPU time and the transformation between absolute and relative points in time as described in Section 3.5.2.

Further, the periodic Java threads in L4 Kaffe uses only one time slice of its associated native L4 thread as described in Section 3.1.3. The priority of the time slice is set to the priority specified by the Java thread. Additionally, the mentioned restriction in Section 3.1.3 of periodic Java threads in L4 Kaffe must be considered, which means that deadlines unequal to the period end are not realized in L4 Kaffe. An example of a periodic Java thread is shown in Figure 4.3. In this example the Java thread overruns its cost, the worst case execution time e1, within the first interval and misses the deadline of the second interval e2.

Whenever a deadline is missed or a worst case execution time is exceeded by a Java thread, a Java handler has to be called, if a handler is associated with this Java thread. Therefore, I mapped the Java handlers of worst case execution time, wcet, overruns and of missed deadlines to the specific preemption IPCs sent by the Fiasco kernel. In L4 Kaffe a separate L4 native thread, the preempter, waits for preemption IPCs. If the preempter thread receives a preemption IPC, it determines the Java thread that caused the overrun of the wcet or the missed deadline. Then, the preempter puts a message including the reason and the causing Java thread into a ring buffer. Now, the preempter wakes a special waiting Java thread, I call it Java handler thread, and makes it ready-to-receive new preemption IPCs.

The Java handler thread is responsible for consuming the messages and executing the Java handlers associated with the causing Java thread of the preemption IPC. Depending on the implementation of the Java handler of a Java application, the causing Java thread might be interrupted in its execution or another action can be taken. Currently, only one Java handler thread is used. In the future, a pool of Java handlers shall be used and the count of the Java handler threads shall be adapted according to the appearance of such asynchronous events.

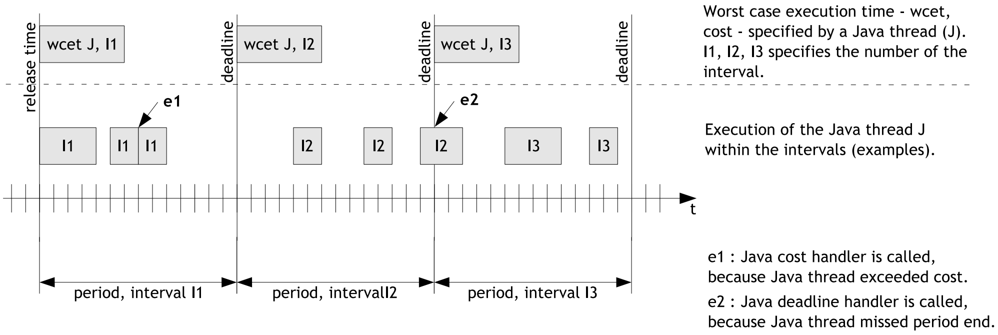

release time
wcet J, I1
deadline
wcet J, I2
deadline
wcet J, I3
deadline
Worst case execution time - wcet,
cost - specified by a Java thread (J).
I1, I2, I3 specifies the number of the
interval.
e1
e2
I1
I1
I1
I2
I2
I2
I3
I3
Execution of the Java thread J
within the intervals (examples).
t
period, interval I1
period, intervalI2
period, interval I3
e1 : Java cost handler is called,
because Java thread exceeded cost.
e2 : Java deadline handler is called,
because Java thread missed period end.

Figure 4.3: Example of a periodic Java thread overrunning the cost and missing the deadline

4.5 Class Loading

RTSJ defines that Java classes have to be accessible for every thread type. Therefore, a class loaded at runtime has to be situated in a Java memory area, where all threads can access it without restrictions. The Java heap as location for classes is not accessible for `NoHeapRealtimeThreads`. Java `ScopedMemory` areas are not adequate, because their lifetimes are limited and classes can exist longer then `ScopedMemory` areas. Therefore, the `ImmortalMemory` area is the only location, where all Java threads always have access.

If Kaffe allocates memory, it specifies a type for the requested memory. This type indicates the intended use of the memory. I used this characteristic to filter allocation requests for classes. In the header file `gc.h` Kaffe multiplexes all invocations for new memory. Therefore, I extended the old invocation for memory in this file. Now, first the types related to Java classes are compared to the requested type for memory allocation. If the type relates to memory for Java classes, a reference to memory within the `ImmortalMemory` is returned. Otherwise, the original invocation to the garbage collector managed memory is executed and a reference to memory situated in the Java heap is given.

A disadvantage of this solution is that Java classes cannot be unloaded at runtime as before.

A better solution could be to provide a special memory area only for Java classes. This memory area would require the same characteristic as the `ImmortalMemory` area, i.e., all threads can access the classes without restrictions. However, the classes should be unloadable. Therefore, the JVM has to determine whether classes are unused. The garbage collector already informs Kaffe, when a class is not in use anymore and that it can be unloaded. However, with the introduction of the `NoHeapRealtimeThreads` the garbage collector cannot find all references to classes, because it does not scan the stacks of the native L4 threads, which are associated with `NoHeapRealtimeThread` objects. Stopping them and scanning for the references to Java classes is not permitted. For this purpose, a list of used Java classes could be used, which tracks the used classes for each `NoHeapRealtimeThread`. They should be immediately accessible by Kaffe and the `NoHeapRealtimeThreads`. However, within this diploma no solution to this problem was found. To solve this, further research is necessary.

4.6 Modifications of the Standard Java API

During the integration of RTSJ in L4 Kaffe, I had to modify some classes of the Java API, which belong to Kaffe.

One problem is the usage of Java objects accessible by static fields of classes. Public static fields of a class can be read and written without access to an instance of a corresponding Java object. These fields are accessible from anywhere within the Java application. Each Java thread can access it without restrictions. However, with RTSJ some limitations for `NoHeapRealTimeThreads` are introduced. These threads cannot access Java objects situated in the Java heap. Therefore, each attempt to get references to Java objects of such a static class field by a `NoHeapRealtimeThread` causes a violation of the memory access rules.

Many of the classes of the standard Java API provide Java objects for Java applications by static fields, which have to be accessible for all Java threads. Therefore, I had to modify some classes of the standard Java API, which provide access to Java objects by static class fields. Now, these classes explicitly create Java objects within the `ImmortalMemory`.

However, this solution is not enforceable for all Java classes, because Java objects can have a short lifetime. Java objects within the `ImmortalMemory` have a lifetime as long as the Java application. Therefore, creation of Java objects within the `ImmortalMemory` is only advisable for Java objects that live as long as the Java application. However, if an application developer requires Java objects with short lifetimes, which are shared by different Java threads due to static class fields, then he must explicitly consider the mentioned restrictions in the class design.

Further, a similar problem appeared with Java classes managing a set of Java objects like `java.util.HashMap` or `java.util.HashTable`. The sets and their entries have to be situated in the `ImmortalMemory` area, when they shall be accessible by each Java thread. The Java API of L4 Kaffe uses these classes for managing Java threads, properties of the environment of Kaffe and of Java applications, loaded Java classes, and for many more. Classes have to be redesigned, when they are using such sets in the context of `NoHeapRealtimeThreads`. I modified only essential Java classes, which are required for starting L4 Kaffe and the Java application. For a production environment, additional modifications will be necessary.

4.7 State

In this thesis the basics of the memory management of RTSJ are integrated. Particularly, the `ScopedMemory` area management works with the access and assignment rules for Java objects. Additionally, the management of nested usage of `ScopedMemory` areas as well as the `ImmortalMemory` area is complete. However, the implementation of the Java classes of the memory areas and their methods is not complete, but the classes provide the essential methods to enter memory areas in various manners.

Both real-time Java thread classes `RealtimeThread` and `NoHeapRealtimeThread` are integrated into L4 Kaffe. However, their representation in the Java classes is not complete. Particularly, all methods related to admission for feasible schedules are not supported. Real-time Java threads can be executed periodically. Java handlers for detecting deadline misses and worst case execution time overruns are implemented. However, not the complete requested behavior as described in RTSJ for handling multiple misses and overruns is realized. In the current state of L4 Kaffe only periodic Java threads whose period end is equal to the deadline are supported.

The following aspects are not supported and realized within this diploma:

- Asynchronous Transfer of Control (ATC);
- Asynchronous events;
- The general concept of `javax.realtime.Schedulable` Java objects;
- Aperiodic or sporadic execution of Java threads;
- Priority inversion avoidance;
- Physical access to specific memory regions; and
- Scheduling related tasks such as admission of Java threads and feasibility analysis.

5 Evaluation

In this chapter I describe results of this diploma and remaining problems of the current RTSJ implementation of L4 Kaffe. The implementation of the RTSJ memory management and the real-time Java thread classes have not been thoroughly tested in regard to real-time capabilities. However, some tests are explained in the following.

I required much time for adaption of the standard Java API of L4 Kaffe to the RTSJ requirements. After I had added the access and assignment rules of the Java memory areas, many violations of these RTSJ rules were detected by L4 Kaffe, when it was started. Unfortunately, during the start phase the Java exception mechanism works only limited. Therefore, the messages of Java exceptions and their Java stack traces of the invoked methods were not complete. Additionally, a thrown Java exception of a detected violation can cause again new Java exceptions, which end in an infinite loop of nested exceptions. Therefore, the location of violations was not reliable and an intensive debugging was necessary, which consumed much time of this diploma.

In the following sections I provide a description of a test scenario and its results. These results are not representative, but they show that the right steps toward a better predictable L4 Kaffe have been done. Additionally, I describe a detected problem of RTSJ in L4 Kaffe, which can lead to delays of `NoHeapRealtimeThreads` by the garbage collector because of a explicit synchronization point between `ScopedMemory` areas and a garbage collection cycle.

5.1 Delay of NoHeapRealtimeThreads

`NoHeapRealtimeThreads`, NHRTs, are not stopped in L4 Kaffe during a garbage collection. However, one explicit synchronization point between NHRTs and the garbage collector thread exists. If a NHRT leaves a `ScopedMemory` area as last thread, it has to release the Java objects and reinitialize the `ScopedMemory` area. The garbage collector has to scan the Java objects in `ScopedMemory` areas, because these objects can have references to Java objects in the Java heap. When the NHRT leaves a `ScopedMemory` as last thread and the garbage collector thread scans it for Java heap references at the same time, the NHRT has to be delayed. Otherwise, the garbage collector thread would work with invalid references and that would lead to errors in L4 Kaffe.

Therefore, the critical section is the release and the reinitialization of a `ScopedMemory` area by the last thread and scanning one Java object of this area for Java heap references by the garbage collector thread. This critical section is protected by a semaphore. The last thread has to acquire it before it releases the memory of a `ScopedMemory` area and the garbage collector thread has to acquire it before it uses a Java object reference of this `ScopedMemory` area. Within this critical section the garbage collector thread extracts all references to Java heap objects of exactly one Java object. The garbage collector thread appends the located Java heap references to its internal list to analyze them later. After this, the garbage collector thread leaves the critical section. If the garbage collector thread has to scan more Java objects of this `ScopedMemory` area, it has to reacquire the semaphore.

The delay of a NHRT caused by this synchronization point depends on the time the garbage collector thread requires to extract all Java object references of one Java object. The number of these references depends on the used Java object and can differ according to each Java object. Some Java objects have no references to other Java objects, other Java objects have many. The delay caused by this synchronization point is not fix.

However, if an Java developer knows the Java objects with the most references to other Java objects used within a `ScopedMemory` area, an estimation of an upper bound of the delay can be made.

For this purpose following parts have to be measured:

- the time to extract all references of this known Java object;
- the time to switch from the NHRT to the garbage collector thread;
- the time to free the semaphore by the garbage collector thread; and
- the time to acquire the semaphore by the NHRT.

This measurement is specific for a Java object and a CPU, therefore I do not provide them here. This delay of a NHRT by a garbage collector thread can only appear, when a NHRT leaves a `ScopedMemory` area as last thread and the garbage collector thread owns the semaphore to scan one Java object of this area. However, RTSJ specifies that a NHRT must not be delayed by a garbage collection. Therefore, it is a remaining problem of the RTSJ implementation in L4 Kaffe.

5.2 Test Scenario

5.2.1 Structure of the Java Test Application

The intention of this test scenario is to show that a NHRT can be executed in a predictable form in contrast to other Java threads, which can be stopped and delayed by the garbage collector. Thus, I construct a scenario with two `RealtimeThreads`, RT_1 and RT_2, who allocate much memory. This behavior causes often garbage collection cycles. A NHRT is executed beside RT_1 and RT_2 to show that it is not affected by garbage collection invocations.

They perform the same task in order to allocate many Java objects. However, the NHRT is executed in a `ScopedMemory` area and RT_1 and RT_2 in the `HeapMemory`, which represents the traditional Java heap in RTSJ. The `ScopedMemory` for the NHRT is half of the size of the Java heap for RT_1 and RT_2, because the two threads share the Java heap. Further, the NHRT has to leave the `ScopedMemory` area at the end of its period in order to release the memory of the `ScopedMemory`. With the beginning of the next period it enters the `ScopedMemory` again. In contrast, RT_1 and RT_2 remain in the Java heap.

The three Java threads are executed in periodic mode to obtain informations about their execution behavior. Possible deadline misses indicate whether they were delayed during the execution. Additionally, the three threads have the same period, release time and priority. The deadline is the period end. The threads have to accomplish the same amount of work within a period beside the fact that the NHRT has to release the memory at the end of each period.

Further, with each of the three Java threads an identical *worst case execution time* (wcet) is associated, which they have to meet. The work within a period can be finished, if the Java threads are not delayed.

5.2.2 Discussion of Results of the Test Scenario

In table 5.1 an example execution of this scenario is shown. I accomplished the measurement on an Athlon XP 1800+ (Palomino), 1533 MHz, L1 cache of 128kb and L2 cache of 256kB.

The period of the three Java threads is 1500ms and the worst case execution time is constricted to 100ms. The times in the table are measured for 100 execution cycles of the period. The best time $t_{...,b}$, the worst time $t_{...,w}$ and the average time $t_{...,a}$ is presented for each measured time of the 100 cycles. Only 300ms are sufficient for the 3 threads with a wcet of 100ms. However, the period is chosen larger in order that other threads in L4 Kaffe can be executed, because their priority is lower than the priority of the three Java threads. These other threads are the garbage collector thread, the finalizer thread of the garbage collector, the preemption thread, and the Java handler thread for managing deadline misses of L4 Kaffe.

Firstly, in the table the consumed execution times t_c of the NHRT, RT_1 and RT_2 within a period are shown, which are measured by the Fiasco kernel. The t_c times describe the true execution time of the threads on the CPU within one period. Secondly, in the table the times t_j measured by the Java application are shown. The t_j times are taken after the Java thread became runnable at the start of a period and before the Java thread called the Java method to wait for the next period. The `currentTimeInMillis()` method of the `java.lang.System` class is used to measure t_j. The t_j times represent the noticed time by the Java application. It can include possible switches to other Java threads or interruptions by the garbage collector thread.

Additionally, the summarized waiting times w_h for memory requests for Java objects of the Java heap within one period by the Java threads RT_1 and RT_2 are shown. I added these times to show the reason why RT_1 and RT_2 missed their deadlines so often. In L4 Kaffe w_h is measured by using the time stamp counter of the CPU. w_h describe the waiting time before an allocation request is started until the request returned with a memory reference. Finally, in the table the number of deadline misses c_d are shown. The c_d numbers describe the deadline misses that appeared during the complete execution of the 100 cycles of the application for each of the three Java threads.

The t_c times of the NHRT are bigger than those of RT_1 and RT_2, because NHRT releases the memory of all Java objects of the `ScopedMemory` within one period. However, the t_j times differ hardly between the RT threads and NHRT. The reason is that RT_1 and RT_2 are delayed until a garbage collection is executed. If RT_1 requests memory from the Java heap and if not enough memory is available, a collection will be necessary. However, RT_2 is also stopped, which means that RT_2 cannot be executed and the t_j times increase. The same applies for RT_1, if RT_2 is stopped. Further, the w_h times show delays caused by garbage collection cycles. These delays are the reason of the high t_j

times of RT_1 and RT_2. For the NHRT the w_h times are zero, because the NHRT did not allocate memory from the Java heap.

description	NHRT	RT_1	RT_2
$t_{c,b}$	93	85	85
$t_{c,w}$	96	87	87
$t_{c,a}$	95	86	86
$t_{j,b}$	94	86	86
$t_{j,w}$	97	4065	4143
$t_{j,a}$	95	955	1489
$w_{h,max}$	0	3969	3977
$w_{h,min}$	0	2	2
$w_{h,a}$	0	864	1387
c_d	0	49	54

Table 5.1: Results of the test scenario in ms per period of 100 period cycles, period 1500ms, wcet 100ms

Additionally, the c_d numbers of RT_1 and RT_2 confirm the picture of the bad behavior of RT_1 and RT_2 in the terms of uniform and predictable execution. Within the test scenario no deadline was missed by the NHRT, because it was not delayed by other Java threads or by the garbage collector. The described problem of Section 5.1 and possible delays were not measured in this scenario.

The w_h times of RT_1 and RT_2 indicate the reason, why they miss so many deadlines. They wait long in order to obtain memory for Java objects, when a garbage collection was necessary. To allow better results for the both RT threads, the number of invocations of garbage collections has to be minimized. Additionally, w_h of RT_1 and RT_2 are so high, because the garbage collector thread has to iterate through each Java object situated in Java memory areas to scan it for Java heap references.

I think, a better solution is necessary in order to exclude Java objects that have no references to other objects situated in the Java heap. They could be managed in a separate list. However, the list has to be updated each time a Java heap reference is assigned to a Java object within these Java memory areas. Additionally, Java objects should be removed from the list, when they lost their last reference to objects situated in the Java heap. This attempt requires a detailed consideration. This is not done in this diploma.

These results are the first impressions of the current L4 Kaffe implementation. They show that with NHRT-threads and the memory management of RTSJ Java applications with better predictability can be supported by L4 Kaffe. Further investigations to advance L4 Kaffe towards a real-time capable JVM are necessary. For this purpose detailed measuring and appropriate configurations of L4 Kaffe are necessary.

6 Summary and Conclusion

One goal of this diploma was to evaluate the RTSJ in order to estimate how it can be integrated into L4 Kaffe. Therefore, in Chapter 2 I analyzed the parts of the RTSJ. On the one hand I described the extensions introduced by the API of the RTSJ and on the other hand the necessary extensions and modifications in the JVM L4 Kaffe. Additionally, I pointed out specific steps to integrate aspects of the RTSJ in L4 Kaffe and the relation of L4 Kaffe with RTSJ to services of L4Env on top of the microkernel Fiasco.

One result of the evaluation is that an introduction of the Java real-time threads of RTSJ also requires a complete implementation of the memory management of the RTSJ. The memory management is important to support `NoHeapRealtimeThreads`, which are not affected by the garbage collector. Therefore, another goal of the diploma was to implement the memory management of RTSJ and the real-time Java threads, in particular `NoHeapRealtimeThreads`, in L4 Kaffe. This goal is achieved, whereas the current state of the implementation does not support the complete functionality defined by the RTSJ API.

Additionally, the management of the Java memory areas, the Java scoped stack and restrictions of accesses and assignments between memory areas and their detection at runtime are realized. The real-time Java thread classes are introduced in L4 Kaffe, whereas the special feature is that the `NoHeapRealtimeThread` cannot be stopped by the garbage collector. Further, periodic real-time Java threads can be executed in L4 Kaffe, because of the direct support of periodic L4 threads by the microkernel Fiasco. The management of the Java handlers to detect deadline misses and overruns of the worst case execution time must be extended and optimized.

The implementation of the memory allocation for each Java object within the `ImmortalMemory` area and the `ScopedMemory` areas is simple in order to support uniform allocation time of Java objects. Further, the implementation supports no wishes or hints by the Java application as locality of special Java objects. However, better adapted allocation schemes for special purposes can be introduced in L4 Kaffe to support locality. Thereby, a Java application would have partial control about the location and alignment of special Java objects.

Further, support for accessing physical memory is not realized, because of the limited time within this diploma. However, L4Env provides all mechanisms to support it in L4 Kaffe, because memory is handled as a resource.

Another result of the evaluation is that I could not implement all aspects of RTSJ within this diploma. It was not possible to examine L4 Kaffe in order to decide whether it is real-time capable because of the limited time of the diploma. With the RTSJ integration and the supporting of the `NoHeapRealtimeThread` the execution of a Java thread in L4 Kaffe is better predictable as it is for a Java thread that can be stopped by the garbage collector. However, a detailed analysis of L4 Kaffe and measurements are necessary to find weak points in L4 Kaffe. A concrete application scenario would be helpful to detect deficiencies.

My time estimation for adapting the standard Java API of L4 KAFFE was not correct in order that the access and assignment rules are not violated by real-time threads. This part of this thesis consumed six weeks instead of the estimated two weeks. The remaining time estimation of the integrated parts of RTSJ in L4 Kaffe were correct.

Further, a result is that the critical section implementation of L4 Kaffe must be reworked to support priority inversion avoidance. Particularly, considerations are necessary in order to extend the L4Env semaphore implementation. I think, this an essential point to support L4 real-time applications generally and Java real-time applications in particular.

In my opinion a serious problem is the integration of asynchronous transfer of control into L4 Kaffe. All critical sections within L4 Kaffe beside the Java monitors have to be found and identified at runtime to support asynchronous interruption for Java applications. This detection is necessary for L4 Kaffe in order to reliably decide whether a thread is in such a critical, deferred section. Asynchronous transfer of control must be thoroughly analyzed, designed, and implemented to avoid deadlocks.

This implementation of RTSJ in L4 Kaffe supports the predictable execution of `NoHeapRealtimeThreads`, which can be used for *soft real-time systems* and *hard real-time systems*. RTSJ specifies instances of `RealtimeThreads` for usage in *soft real-time systems*. However, the current implementation in L4 Kaffe of `RealtimeThreads` is not real-time capable, because the garbage collector implementation of L4 Kaffe can stop these threads during a garbage collection. A real-time capable garbage collector or adaption of the current garbage collector of L4 Kaffe are necessary in order to support predictable execution of instances of `RealtimeThread`.

7 Outlook

I think, in L4 Kaffe the right steps toward real-time Java applications are done. However, a lot of work is necessary to support RTSJ completely. The primary task is to support a critical section implementation in L4Env or a special implementation for L4 Kaffe that implements the priority inheritance protocol and the priority ceiling protocol.

A further task is to provide the general concept of `javax.realtime.Schedulable` Java objects for L4 Kaffe. A `javax.realtime.Schedulable` Java object can be executed, but has not to be associated all the time with a Java thread. This concept is necessary to support and handle large sets of asynchronous Java events. Their Java handlers will be scheduled according to the requested behavior of the Java application.

Another interesting task is to bind asynchronous Java events to interrupt IPCs sent by the Fiasco kernel. With this characteristic and access to physical memory simple hardware drivers can be written in Java. This feature is not absolutely necessary for real-time abilities of Java, however they are useful in a production environment where Java application with access to hardware is used. L4Env and Fiasco provides all mechanisms to support such user-level Java drivers by providing access to physical memory and by sending interrupt IPCs to an L4 application.

Further, L4 Kaffe works only as an interpreter. The just in time compilation (JIT) for IA32 is available and used, for example with Linux. However, it is not tested for L4 Kaffe yet. If better execution speed is requested, it will be a task to test and to adapt the JIT of L4 Kaffe.

List of Figures

List of Tables

Bibliography

[aG05] aicas GmbH. Homepage of the acias GmbH. [WWW Document]. URL: `http://www.aicas.com`, 2005. Access time 03.2005.

[Bac99] Godmar Back. Garbage collector strategy of Kaffe. [WWW Document]. URL: `http://www.kaffe.org/doc/kaffe/FAQ.gcstrategy`, 1999. Access time 03.2005.

[Bak92] Henry G. Baker. The Treadmill: Real-Time Garbage Collection Without Motion Sickness. *ACM Sigplan Notices*, 27(3):66–70, March 1992.

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

[Boe04] Alexander Boettcher. Port of the Java Virtual Machine Kaffe to DROPS by using L4Env. [WWW Document]. URL: `http://wwwos.inf.tu-dresden.de/papers_ps/boettcher-beleg.pdf`, 2004. Access time 03.2005.

[BR01] William S. Beebee and Martin C. Rinard. An implementation of scoped memory for real-time java. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, pages 289–305. Springer-Verlag, 2001.

[CR99] Lisa Carnahan and Marcus Ruark. The requirements working group for real-time extensions for the java (tm) platform. Technical Report NIST Special Publication 500-243, NIST - National Institute of Standards and Technology, September 1999. Available from URL: `http://www.nist.gov/rt-java`.

[die05] dietlibc. Homepage of the Dietlibc Project. [WWW Document]. URL: `http://www.fefe.de/dietlibc`, 2005. Access time 03.2005.

[gnu05a] gnu.org. Homepage of the GNU Classpath project. [WWW Document]. URL: http://www.gnu.org/software/classpath, 2005. Access time 04.2005.

[gnu05b] gnu.org. Homepage of the GNU Compiler for the Java Programming Language, gcj. [WWW Document]. URL: http://www.gnu.org/software/gcc/java, 2005. Access time 04.2005.

[HH01] M. Hohmuth and H. Härtig. Pragmatic nonblocking synchronization for real-time systems. In *USENIX Annual Technical Conference*, Boston, MA, June 2001.

[HJB05] Mark Weiser Hans-J. Boehm, Alan J. Demers. Boehm-demers-weiser conservative garbage collector. [WWW Document]. URL: http://www.hpl.hp.com/personal/Hans_Boehm/gc, 1988-2005. Access time 03.2005.

[HLR+01] C.-J. Hamann, J. Löser, L. Reuther, S. Schönberg, J. Wolter, and H. Härtig. Quality Assuring Scheduling - Deploying Stochastic Behavior to Improve Resource Utilization. In *22nd IEEE Real-Time Systems Symposium (RTSS)*, London, UK, December 2001.

[Hoh98] Michael Hohmuth. The Fiasco kernel: Requirements definition. Technical Report TUD–FI–12, TU Dresden, December 1998. Available from URL: http://os.inf.tu-dresden.de/papers_ps/fiasco-spec.ps.gz.

[Hoh02a] Michael Hohmuth. The Fiasco kernel: System architecture. Technical Report TUD-FI02-06-Juli-2002, TU Dresden, 2002.

[Hoh02b] Michael Hohmuth. *Pragmatic nonblocking synchronization for real-time systems.* PhD thesis, TU Dresden, Fakultät Informatik, September 2002.

[J. 00] J. Consortium. Real-Time Core Extensions for the Java platform, September 2000. International J Consortium Specification, Revision 1.0.14. [WWW Document]. URL: http://www.jconsortium.org/rtjwg/. Access time 03.2005.

[jRa05] jRate. Homepage of the jRate. [WWW Document]. URL: http://jrate.sourceforge.net, 2005. Access time 03.2005.

[kaf05] kaffe.org. Homepage of Kaffe.org. [WWW Document]. URL: http://www.kaffe.org, 2005. Access time 03.2005.

[L4K05] L4Ka. Homepage of the L4Ka Project. [WWW Document]. URL: http://l4ka.org, 2005. Access time 03.2005.

[Lia99] Sheng Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[Lie96] J. Liedtke. L4 reference manual (486, Pentium, PPro). Arbeitspapiere der GMD No. 1021, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1996. Also Research Report RC 20549, IBM T. J. Watson Research Center, Yorktown Heights, NY, September 1996.

[LY99] Tim Lindholm and Frank Yellin. *The JavaTM Virtual Machine Specification*. Addison-Wesley, second Edition edition, 1999.

[Mac05] Project Mackinac. Homepage of the Project Mackinac of sun microsystems. [WWW Document]. URL: http://research.sun.com/projects/mackinac, 2005. Access time 03.2005.

[OVM05] OVM. Homepage of the Ovm Project. [WWW Document]. URL: http://www.ovmj.org, 2005. Access time 03.2005.

[SRGTD03] Operating Systems Research Group TU Dresden. L4Env - an Environment for L4 applications. [WWW Document]. URL: http://wwwos.inf.tu-dresden.de/l4env/doc/l4env-concept/l4env.pdf, 2003. Access time 03.2005.

[Ste02] Udo Steinberg. Fiasco μ-Kernel User-Mode Port. [WWW Document]. URL: http://wwwos.inf.tu-dresden.de/papers_ps/steinberg-beleg.pdf, 2002. Access time 03.2005.

[Ste04] Udo Steinberg. Quality-Assuring Scheduling in the Fiasco Microkernel. Master's thesis, TU Dresden, March 2004.

[Sun05] Sun. Homepage of the Java language and JVM of Sun Microsystem. [WWW Document]. URL: http://java.sun.com, 2005. Access time 03.2005.

[Tim05] TimeSys. Homepage of the TimeSys. [WWW Document]. URL: http://www.timesys.com, 2005. Access time 03.2005.