Port of the Java Virtual Machine Kaffe to DROPS by using L4Env

Alexander Böttcher
ab764283@os.inf.tu-dresden.de

Technische Universität Dresden

Faculty of Computer Science
Operating System Group

July 2004

Contents

1 Motivation **3**

2 Fundamentals **4**
- 2.1 Dresden Real-Time Operating System 4
  - 2.1.1 Microkernel 4
  - 2.1.2 L4Env - an environment on top of the L4 kernel 5
- 2.2 Java and Java Virtual Machines 6
  - 2.2.1 Architecture 6
  - 2.2.2 Available Java Virtual Machines 7
  - 2.2.3 Kaffe 8
- 2.3 Decision for Kaffe 9

3 Design **10**
- 3.1 Concepts 10
- 3.2 Threads 10
  - 3.2.1 Thread mappings 11
  - 3.2.2 Adaptation of the L4Env thread library to Kaffe 13
  - 3.2.3 Garbage Collector 14
  - 3.2.4 Priorities and Scheduling 15
- 3.3 Critical sections 16
- 3.4 Java exception handling 17
  - 3.4.1 Null object and null pointer handling 18
  - 3.4.2 Arithmetic exception handling 20
- 3.5 Memory and files 20
- 3.6 Java native interface (JNI) and shared native libraries 21

4 Implementation **22**
- 4.1 Kaffes configuration and build system 22
- 4.2 Thread data structure management 22
- 4.3 Delaying of newly created threads 23
- 4.4 Suspending and resuming threads 23

4.4.1 Stopping all threads 24
4.4.2 Resuming all threads 25
4.5 Semaphore problems with timing constraints 26
4.5.1 Deadlocks caused by semaphore_down_timed 26
4.5.2 Incorrect admissions to enter a semaphore 27
4.5.3 Solution for getting a semaphore with timeouts 27
4.6 Files 28
4.7 Native libraries 28
4.8 Error handling 29

5 Evaluation 30
5.1 Measuring methods 30
5.1.1 Test application: matrix multiplication 30
5.1.2 Test application: consumer and producer 31
5.1.3 Selective JVM functions 31
5.2 Test scenarios and environments 32
5.3 Resource usage 34
5.4 Garbage Collector 35
5.5 Code size, modifications and maintenance 37

6 Conclusion 40
6.1 Further tasks 40
6.2 Summary 41

Chapter 1

Motivation

Java became in the last years a popular language. The object-oriented language eases the development of user-level applications by hiding operating system and hardware specific features. The hardware independent Java binary code allows the compact distribution of Java applications for different system architectures without recompilation. The execution environment - the Java Virtual Machine - is responsible to run the binary code on a specific hardware and operating system. Therefore, the JVM as system specific part has to be adapted to a new operating system to execute Java applications.

An existing Java Virtual Machine (JVM) shall be ported to L4/Fiasco. The intention is to allow Java applications the entry to the L4 world. The L4 Environment and a Java Virtual Machine are the base of this work. The goal is to run Java applications as “native” L4 tasks. The port shall also be the base for future analysis of predictability in the ported Java Virtual Machine.

In this paper, I reviewed some freely available JVM and describe my decision to port the Java Virtual Machine Kaffe to L4/Fiasco. In the next step I show, how the functionality provided by Kaffe to Java applications can be mapped/adapted to L4 services. In the implementation chapter I describe, how the adaptation was achieved and which problems arised. Additionally I compare Kaffe on L4, L4Linux and Linux on the basis of two Java test applications.

Chapter 2

Fundamentals

2.1 Dresden Real-Time Operating System

The Dresden Real-Time Operating System (DROPS) is a research project at the TU Dresden. The primary goal is to support application with Quality of Service requirements. Various server and application programs run on top of a microkernel, whereby they are divided into real-time and time-sharing components.

2.1.1 Microkernel

The heart of the operating system is the microkernel Fiasco [Hohmuth, 1998, Hohmuth, 2002] that offers a minimal set of functionality. The kernel provides isolation through separation of address spaces, execution instances known as threads and communication between threads called inter process communication (IPC). All other required functionality has to be implemented as service running at user level in the non-privileged mode of the CPU.

Fiasco belongs to a family of L4 kernels like Pistachio and Hazelnut developed at the University of Karlsruhe [L4Ka, 2004]. The L4 ABI describes the interface for the microkernels, which was initially designed by Jochen Liedtke [Liedtke, 1996].

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

2.1.2 L4Env - an environment on top of the L4 kernel

The L4 Environment (L4Env) is a programming environment [L4Env, 2003] for applications on top of the L4 microkernel family. L4Env is developed as 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, which a monolithic operating system would provide such as protection of critical sections, page fault handling, resource management of memory and files.

Threads

The L4Env thread library provides a basic abstraction to native L4 threads. It is a basis for native L4Env applications and a higher-level thread abstraction. It provides functions to create and destroy threads, supports shutdown callbacks, manages thread local data and priority and stack handling. It simplifies the usage of native L4 threads by allocating the memory for the stacks from the data space manager and provides thread IDs independent from the native L4 threads.

Semaphores and Locks

The L4Env provides a task local counting semaphore and lock implementation for protecting critical sections whereby the lock package uses the semaphore package. In case of no contention on the semaphore the thread, which wants to enter the critical section, modifies atomically the semaphore counter. If the critical section is not free and a thread cannot enter, than it calls per IPC the task local semaphore thread.

The separate semaphore thread in each address space serializes the multiple attempts to enter a critical section. The kernel delays the calling thread until the semaphore thread is ready to receive the IPC. The semaphore thread then decides to queue the calling thread in the waiting list. If a thread releases the semaphore, the semaphore thread will send an IPC to the next waiting thread in the queue and this thread enters the critical section.

Each IPC is usable with timing constraints, which limits the time for sending or receiving a IPC to another thread. The semaphore library uses this IPC feature to provide critical sections with timeouts. If the attempt to enter cannot be satisfied on time, the kernel will cancel the IPC and will not allow the calling thread to enter the critical section.

C libraries

Currently two different C libraries are available for application development. The Oskit [Oskit, 2002] was developed at the University of Utah and adapted for the usage by L4 applications. Currently the Dietlibc [Dietlibc, 2004] is also adapted to the L4Env, because it is smaller and more flexible to handle then the Oskit. The Dietlibc supports the known POSIX functions for file handling like open, read, write, close. A naming service is responsible for resolving files and paths and allows the usage of different file providers, which the L4 virtual file system (L4VFS) provides.

Both C libraries provide the memory management for applications by supporting the usage of malloc and free. They hide the concept of data spaces used by the L4Env [Aron et al., 2001]. A data space is a container that can contain different types of memory. The data space manager (dm_phys) owns the available physical memory. The region mapper (l4rm) manages an address space and is responsible for allocating virtual memory regions of the address space and to invoke the responsible data space manager to resolve page faults.

The C library map functions like printf, fprintf to the Log respectively the Con server to provide outputs. The two packages log and con are responsible for console input/output. The log server provides a basic textual console output. Con is a graphical console server that provides virtual consoles that it multiplexes to one visible graphic screen.

2.2 Java and Java Virtual Machines

2.2.1 Architecture

Java is a set of technologies developed by SUN Microsystems. The language Java is a high level object oriented programming language that a compiler translates to hardware and operating system independent bytecode. The language and the bytecode are developed to support portable code without recompilation of the source code on top of different hardware architectures and operating systems.

The language uses implicit memory management, platform independent application programming interfaces (APIs) and dynamically loaded code/classes. The necessary environment to execute the bytecode is the Java Virtual Machine (JVM) [Lindholm, Yellin] - an abstract machine that

executes the binary code on top of an operating system by using the specific services of that platform. First versions only worked as pure interpreter and were known as not performant. Today the JVMs often use compilers that transform the binary code into machine code and then execute it, known as just in time compilation (JIT).

Furthermore, Java is well known and used because of its high level concepts and programming abstractions for application development of platform-independent components (J2EE). A detailed description about the internal of a JVM can be found in [Venners, 1999] and about J2EE at [SUN, 2004].

2.2.2 Available Java Virtual Machines

A wide variety of Java Virtual Machines (JVM) were and are developed for different purposes [Friedman, 2002]. In the next sections I describe some of them, free and non-free implementations. For the port we needed an open source variant, so that we can freely distribute it with DROPS.

GNU Compiler for Java - GCJ

GCJ is a part of the GNU Compiler Collection (GCC) that is open source and widely used especially with Linux distributions. GCJ is an optimizing compiler for the Java programming language, which can compile

- Java source code directly to native machine code,
- Java source code to Java bytecode (class files) and
- Java bytecode to native machine code.

The GCJ uses a runtime library that provides the core class libraries, a garbage collector and a bytecode interpreter for mixed compiled and interpreted applications. The compiled applications have to be linked against the runtime library.

The documentation of the GCJ describes that in order to port the GCJ, the thread layer, the file handling layer and the signal layer [GCJ, 2004] must be modified. Additionally some modifications at the core GCC for the exception handling and a mutex and condition variable implementation will be necessary.

Jikes Research Virtual Machine

The Jalapeo project [Alpern et al.] was renamed to the Jikes Research Virtual Machine (RVM) in October 2001, because since it is available as open source. The IBM T.J. Watson Research Center [IBM, 2004] develops RVM.

The virtual machine is implemented in the Java programming language and is thought for academic and research purposes. The intention is to provide a flexible testbed for prototyping new virtual machine technologies. To build the Jikes RVM another VM is needed to run the two Jikes RVM compilers on Jikes RVM class files, because the entire RVM and its compilers are written in Java. Blackdown, Kaffe and the JVM from SUN can be used for that purpose.

The also known and freely available Jikes compiler [Jikes, 2004] is a Java source code to Java bytecode compiler and must not be mistaken with the above mentioned RVM compilers. It is a separate project and can be used, but has not, to compile the bytecode for the RVM.

JVM by SUN and Blackdown

SUN Microsystems distributes its Java development kit and the virtual machine under its own license. The Blackdown team is a group responsible for porting Sun's Java software to Linux and has licensed the source code.

SableVM and LaTTe

The Sable Research Group at the McGill University develops the SableVM, which is a bytecode interpreter written in C [SableVM, 2004]. LaTTe is developed at the Seoul National University [LaTTe, 2004], which is a virtual machine with a Just In Time (JIT) compiler. The main goal is to improve the performance of the JIT compiler and it was initial based on Kaffe 0.9.2, but meanwhile is widely modified. Both VMs are still in research and development state.

2.2.3 Kaffe

Kaffe is an open source implementation of a Java Virtual Machine distributed under GPL. It is implemented in the programming language C and supports an impressive long list of different system platforms [Kaffe.org, 2004]. Kaffe operates as Interpreter or as JIT(Just in Time) compiler. It uses

clearly separated architecture and operating system specific files and directories. Kaffe provides interfaces for the thread library, the critical section implementation, the garbage collector and partially for the file handling. The intention behind the interfaces is to support ports to new platforms, whereby the core of the virtual machine — the class loader and the execution engine — do not have to be modified.

2.3 Decision for Kaffe

I came to the decision to use Kaffe for the port to DROPS by using L4Env, because it is clearly structured and it uses interfaces for platform depend services. I can use the L4Env services directly without special emulations of libraries or services. I decided not to use the GCJ, because adaptation to the GCC would be necessary. The Jikes RVM intention is to be a testbed and another JVM would be necessary to bootstrap Jikes RVM. SableVM and LaTTe are interesting and potential candidates, but because they are still under development I did not decide to use them. SUN's VM respectively the Blackdown VM could not be used because of their licensing issues.

Chapter 3

Design

3.1 Concepts

A Java Virtual Machine (JVM) is a runtime environment for executing programs in the Java class format. The JVM offers services and libraries, which the operating system usually provides. Input and output of data, thread handling and critical section protection are some of them. Typically the JVM itself is not an operating system, so it acts as intermediary. The JVM uses the specific services and mechanism of the operating system or has to simulate these parts, which the operating system does not support or provide directly. In addition, the JVM itself requires the specific services of the operating system to protect its own data structures, to use memory, files and threads. In the next sections I will discuss how the JVM can use and/or can map the provided and required mechanism by using the L4Env services and which additional solutions are necessary.

3.2 Threads

The Java language directly supports the concept of threads, therefore the JVM has to handle threads whether the operating system provides thread support or not. The Virtual Machine has to map the Java threads to native threads, which the operating system offers directly to applications, in our case the L4 threads. I will use the term of native threads to describe in general the possible mappings of the threads. In the following considerations, I only discuss the case of a Java Virtual Machine that uses only one task per Java application. Usually a JVM uses only one task. Kaffe has no support to distribute a single Java application over many tasks. An interesting attempt to use more than one task by a JVM is published in [Back, 2000].

If the operating system supports threads directly, an one to one mapping of Java threads to native threads is possible. Otherwise, the JVM has to map all Java threads to one native thread respectively to the one execution instance of the task, known in Java as 'green threads'. If an user-level thread library is available on the operating system, than the JVM can use this instead of an own thread implementation. Also a thread mapping m:n is possible. These variants are discussed in the next sections.

3.2.1 Thread mappings

Green and User-level threads

Green threads respectively user-level threads are useful for platforms without support for threads by the operating system. With that approach, the JVM can switch fast between threads, because no system call is necessary. A system call causes the switching to the kernel, the heart of the operating system. Many load/store operations are necessary from and to the memory, whereby the CPU maybe has to replace cache entries now used by the kernel, previously by the application. The gap between the high frequency of a modern CPU, the low one of the memory and the necessary operations to store/load information to/from the memory worsen the situation. Partially the large caches of modern CPUs compensate the problem.

However, there are some problems to consider. A JVM respectively the library of the user-level threads has to limit blocking calls to the operating system. The operating system does not know other threads of the application, which might be runnable. Additional a timing mechanism in user-mode is necessary to interrupt user-level threads, which are non-cooperative and consume too much execution time, so that the JVM respectively the user-level thread library can interrupt the threads. On systems with more than one processing unit, this user-level thread handling limits the maximal parallelism, because the kernel does not know that the JVM uses an own threading system. The kernel cannot distribute the user-level threads over all available processing units. Additional the management of the threads, priorities and scheduling must be implemented, which is normally done by the kernel or by one of its services.

Green threads are the solution for operating systems without support for native threads. I have not choose this approach for the port of Kaffe, because the Fiasco kernel directly implements threads and the L4 thread library

offers additional support.

Partly mapping to native threads

I think mapping of some (more) Java threads to some (less) native threads can be used for optimizations, for example, when system calls are expensive. If threads often call the operating system or services, which cause a long waiting period, than the JVM can map these Java threads directly to native threads. The service respectively the operating system blocks the thread until the job is finished. All other threads can process jobs that do not use blocking calls. If switching between them is necessary then the JVM will do it in user-mode to reduce the number of system calls.

The problem is to determine the threads, which often use blocking calls. If that cannot be done in advance, then a dynamical adaptation and mapping of Java to native threads is necessary. It is not always possible to determine, whether a function call of a service uses a blocking system call. Additionally user-level thread management is necessary. The JVM also should know how many threads could really run in parallel, to choose an appropriate mapping to support maximal scalability.

This approach is complex and has to be researched further in respect to useful advantages, such as performance or resource usage. Kaffe has no direct support for such a scenario. Therefore, some modifications on the core JVM would be necessary. For this initial port of Kaffe to DROPS, this approach would exceed the available time.

Direct mapping

Modern operating systems support the thread concept directly. The kernel is responsible for the management of processes and threads. The scalability of an application is not limited due to the usage of user level threads on multiprocessor systems. Switching between threads is not as fast as in user-mode but acceptable. No separate thread handling is necessary, managing wait queues or run queues; priority and scheduling decision are part of the kernel and/or of a service. Direct Java to native thread mapping reduces the complexity of the JVM. Parts like different scheduling algorithms can be done by special services of the system environment.

3.2.2 Adaptation of the L4Env thread library to Kaffe

Kaffe directly supports different thread implementations by defining an abstract interface. The interface is a data structure with some function pointers, which the used thread system has to implement. The expected functionalities are

- to create and destroy threads,
- to change priorities of threads,
- to map the data structures from java to native threads,
- to stop and resume all threads,
- to determine whether a pointer is on the stack of a thread or not and
- to determine the stack range and the remaining space on the stack.

The thread library of the L4Env provides the basic functionality for the thread handling. A problem is the creation of new threads, because in Kaffe the newly created threads have to wait until the creating thread stored the runtime information in its internal data structures. The data structure can be prepared before the create call, but the identification of the new thread is not known until it becomes runnable. The L4Env thread package does not support a previous reservation of thread identifications or the creation of suspended threads.

Kaffe uses a function `jthread_current` to ask the thread implementation about information of the actual running thread. The function seeks the intern data structures for the thread identification. In the case that the identification of the newly created thread is not stored in the data structure yet, the function will not find the requested information and it will throw an exception.

I introduce a function, which the JVM has to call instead of the original start function to suspend the created thread. The new thread has to wait until the original thread acknowledges the storage of the new thread identification. After the notification the new thread becomes runnable and can invoke the original start function.

Kaffe itself provides for Unix/Linux platforms a green thread implementation, called unix-jthread, and a version for the usage with the pthread library, called unix-pthread. I decided to not use these implementations to

adapt to DROPS, because L4Env does not provide a pthread library and the green-thread implementation would not use the full abilities of the microkernel like support for real time threads. The flexible facility of Kaffe to adapt new thread systems allows me to use the thread library of the L4Env directly.

Resuming and stopping threads is necessary for the garbage collector. The determination whether a pointer is on the stack or not is necessary for Kaffe‘s own critical section implementation. Kaffe uses the stack range knowledge of each thread to avoid stack overflows. Additionally the garbage collector uses the knowledge to scan the stack for Java object references. I will discuss these requirements in the following sections.

3.2.3 Garbage Collector

The JVM allocates memory for instances of classes with instructions like `new`, `newarray` and others [Lindholm, Yellin], but no instruction explicitly releases the memory. The Java VM specification does not describe how memory is recycled respectively that this is necessary, but because no computer has infinite memory JVMs uses garbage collectors that are responsible for the detection and freeing of memory of unused Java objects.

Venner describes in [Venners, 1999] garbage collection algorithms and [Back, 1999] describes Kaffe‘s algorithm more specific. Kaffe uses a tracing garbage collector that follows the graph of object references starting with root nodes. The collector marks the located objects. After the complete trace, Kaffe can free unmarked objects, because they are unreachable.

Kaffe’s garbage collector has to stop all running threads beside itself. It scans the stack of these threads to find references to Java objects. The garabage collector stops the threads, because it does not protect its internal data structures for concurrent access. A garbage collection is therefore an expensive operation in terms of performance. Worse is the fact that the stopped threads cannot do anything like receiving events from the GUI, packets from the network and other jobs, where informations can be lost when they are not processed on time.

For systems with one processing unit and a priority based thread scheduler, stopping and resuming threads can be achieved easily. The garbage collector receives a higher priority than the other JVM threads to ensure no other thread can run. That means, that the garbage collector must not use functions that block him, because then the operating system can schedule

another Java thread, which must not run. This solution does not work on systems with more than one processing unit, like Symmetric Multi Processing (SMP) or Hyperthreading. Therefore, another approach will be necessary.

The system call `l4_thread_ex_regs` can set a running thread to a new address of execution - initiated by another thread. The garbage collector use `l4_thread_ex_regs` to set the threads to a function that saves the state of a thread and calls a sleep function. After it has finished, the garbage collector uses the system call to invoke another function, which wakes up all threads, restores the state and restarts the execution.

3.2.4 Priorities and Scheduling

Java threads use priorities to privilege some threads in execution. Priorities are described by numbers and a higher number means a higher priority. Kaffe lets the schedule of threads to the OS respectively the thread implementation. Two constants of the "`java.lang.Thread`" package define the number of available priorities. Currently SUN's JVM provides 10 Java priorities, therefore Kaffe also use 10 for compatibility reasons. In general, more Java priorities are possible, for example for real time threads. Java threads use 10 Java priorities, the native threads for the garbage collector and the finalizer require as highest prior threads another one, so for Kaffe 11 native L4 priority levels are necessary.

Some operating systems offer insufficient priorities for mapping Java thread priorities directly to native thread priorities. They have to map a couple of java priorities to only one priority of the operating system and therefore scheduling is not done correctly as expected, shown in [Pinilla et al., 2003] for SUN's JVM. A possible solution was published in [Pinilla et al., 2003].

Fiasco supports 128 priorities. Therefore, a direct mapping between Java and native thread priorities is possible. The JVM provides fixed priority scheduling, whereby threads with equal priority are scheduled in round-robin fashion. Fiasco schedules threads with the same algorithm. No additional adaptation will be necessary.

3.3 Critical sections

Kaffe has several layers to the locking/protecting abstractions. The JVM has to protect internal data structures, because of concurrently executed threads, to avoid race conditions and to ensure correct state of the VM. Furthermore, the Java language itself supports the usage of monitors to protect critical sections, which are marked by the keyword `synchronized`. This keyword can be associated with methods or blocks.

```
class SynchronizationExample{
 public synchronized void a(){
  ...
 }
 public void b(Object a){
  synchronized(a){
   ...
  }
 }
}
```

If a thread tries to invoke a method or a block that is `synchronized`, it has to obtain the monitor for this object. Then the thread can execute the method and will release the monitor after it finished the execution of the method.

Kaffe implements the several layers of locking by its own "Fast Locking Scheme" [Baker et al., 2000] and the support of the provided synchronization and protection primitives of the operating system. Kaffe tries to minimize the usage of the operating system specific primitive, internally called "heavy locks", by using atomic compare-and-swap operations. Only when another thread holds the lock or a compare-and-swap operation fails, Kaffe acquires a "heavy lock". Kaffe offers two interfaces to implement the synchronization and protection primitives, a semaphore interface and a mutex/condition variable interface.

The pthread library uses mutex/condition variables for synchronization and protection. A mutex protects shared data resources and only allows one thread to access the resource i.e. to enter the critical section. Mutexes allow synchronization by controlling access to data. Condition variables are used to synchronize threads depending on a value of data. If threads require a certain value to continue their execution, then they will have to continually comparing the value. If it is required in a critical section then the lock will

have to be freed after each comparison, so that other threads can enter the critical section and maybe fulfill the condition. To prevent polling, a programmer can use condition variables and associated locks in the pthread library. If a thread waits for a condition variable then the responsible function will automatically and atomically unlock the associated mutex variable and lock it again, when another thread will signal the fulfilled condition.

The Semaphore interface needs a binary semaphore implementation with the ability to enter a semaphore with time constraints. L4Env provides a Semaphore package with these features. L4Env has no support for the pthread library and no support for condition variables, therefore I implement this semaphore interface to provide the required synchronization/protection primitive.

3.4 Java exception handling

The Java language uses exceptions to signal an error or an unexpected situation. A Java program or the JVM can generate exceptions. For instance, when the program detects an illegal situation or the JVM has not enough memory. The Java application can catch such Java exceptions if it expect them. The following abstract example illustrates such a situation:

```
public void calc(int a, b){
   ...
   try{
     add(a,b);
   }catch(InvalidValueException){
     System.out.println("Only positive values are supported");
     return;
   }
   ...
}
private void add(int a, int b) throws InvalidValueException{
 if ( a < 0 || b < 0) throw new InvalidValueException();
 else ...
}
```

The JVM handles the Java exceptions at runtime by backward seeking for a “catch” handler in the trace of invoked methods. If a handler exists, than the JVM executes the “catch” block, otherwise the JVM stops the thread with an error message.

Special handlings for Java exceptions are necessary, which cause a software exception in the CPU. Errors like dividing by zero or illegal - unmapped - memory accesses as stack overflows and null pointer accesses are two of them. The CPU forces the causing thread into the kernel and the kernel can identify the thread and its state, which is described by the used registers, by the instruction pointer and the stack pointer. Kaffe provides two functions, which the thread implementation has to call when a null pointer access or an arithmetic exception occurs. The two functions then implement the correct Java exception handling. Therefore, some mechanisms are necessary to allow the kernel or services, which are responsible for such exceptions, to notify the JVM. In the following two chapters, I describe how this is achieved.

3.4.1 Null object and null pointer handling

Java has a special type, known as `null` object, which means that no instance of a class is used by that reference. If a java program tries to use the null object reference then the JVM will raise a `null pointer` exception. The JVM specification [Lindholm, Yellin] does not specify the value of the `null` object and how the JVM realizes that mechanism of detecting such null objects accesses at runtime. The operation `aconst_null` [Lindholm, Yellin] of the JVM creates such an object reference and puts it on top of the Java stack.

Kaffe implements that mechanism by mapping the `null` object references to the virtual memory address zero. The JVM Kaffe expects that the kernel never maps the first page of the address space and the usage causes a page fault. On Unix/Linux Kaffe registers itself for the signal “SIGSEGV” and installs a signal handler. If a Java application uses a object that is `null`, the application will make a access to the first page. The CPU detects the access with the unmapped page and forces the causing thread in the Unix/Linux kernel. The kernel determines the reason and sends a signal to Kaffe’s signal handler. Kaffe is now able to start the Java exception handling for the `null` object access.

In the case of our microkernel Fiasco, each L4 thread is associated with a pager thread that the kernel informs about page faults per IPC. The pager is responsible for getting and mapping the requested page. If it is not possible, because the address belongs to an unused area, the pager will show an error message and the causing thread will not be runnable.

An additional pager

To handle the `null` pointer accesses, the JVM could install a specialized pager that knows about the special handling. Page faults to other pages than the first it would forward per IPC to a standard pager like the one of the L4RM service.

This approach has a big disadvantage. Every forwarding of a "normal" page fault causes an additional IPC to the standard pager and this is the normal case for a pager. Page faults to the first page are specific exceptions to determine uninitialized pointers. Therefore, I think, a little modification of the L4RM pager is better then the performance penalty of this solution.

Enhancement of the L4RM pager

To handle the page fault at the first page it is necessary to extend the functionality of the pager. Kaffe could register its own routine that knows about the specialized meaning and the pager calls the routine in the case of an unresolvable page fault. Kaffe provides such a function that looks for the last executed Java instruction. Kaffe as a (virtual) machine with an own program counter can detect i.e. knows the position in the Java program, where the CPU caused the exception. Therefore, the instruction to which the Java program counter points is responsible for the access to the null object. Consequently, the JVM generates the Java null pointer exception for the thread. A Java application is able to catch the null pointer exception - the JVM does not enforce the termination of the Java application in this case.

Optimization for debugging

The JVM cannot decide whether a Java application or a bug in its implementationan caused the null pointer access. For a better differentiation between accesses to null objects and illegal null pointers it should be possibly to map the null object to another unmapped page. The function that handles the unresolved page faults can then determine whether it is a problem of the currently executed Java application, a bug in the JVM or one of the native libraries.

3.4.2 Arithmetic exception handling

Dividing by zero cause a software exception in the CPU. That exception handling is not complex as the page fault handling. Similar to the null pointer exception handling Kaffe provides a routine that has to be called in case of arithmetic software exceptions. For that purpose, Fiasco supports the registration of a routine in the LIDT (Local Interrupt Description Table), which the CPU invokes in the case of an exception. On the x86 architecture the exceptions 0 and 16 signal arithmetic exceptions. For each newly created thread a registration in the LIDT for both exceptions is necessary.

3.5 Memory and files

Kaffe directly depends on functions provided by a C library like `open`, `read`, `malloc`, `fprintf`, and so on. The L4Env has two C libraries, Dietlibc and Oskit, whereby the Dietlibc was under development when I started to port Kaffe. Even so, I decided to use the Dietlibc, because the Dietlibc intention is to provide many small functional parts, which an application can use and link only functionality it requires. Other reasons are the better support for using files with the Posix functions and that the Dietlibc will replace the Oskit C library in the future.

Kaffe's memory usage behavior can be easily configured with parameters. These parameters describe initial heap size, maximal heap size and the size of heap increments. The C library `malloc` function allocates memory dynamically from the heap. Kaffe mainly uses fprintf and friends for console outputs. The Dietlibc maps them to the write function at the file descriptors "stdout" and "stderr". A server or library providing the files "stdout", "stderr" and "stdin" has to map these file descriptors to the services, which are responsible for that.

Kaffe tries to ease the port for systems with no complete C library by providing a thin indirection layer. The JVM encapsulates calls to the C library functions with own functions, which are prefixed with "K" like `KOPEN`, `KREAD`, `KFSTAT` or `KSOCKET`. The operating system specific part of the Kaffe code then has to implement these functions when they differ from the standard behavior or have a different syntax - alternatively the original posix call can be used.

3.6 Java native interface (JNI) and shared native libraries

Java classes can also use native, machine specific, libraries of the operating system they are running on. The Java Native Interface (JNI) [Liang] is a specification that describes the usage of libraries written in the language C. A Java program can specify a method as native and the name of the native library that implements the method as shown in the example.

```
class NativeLibraryExample{
 static {
  System.loadLibrary("example");
 }
 native void helloC(void);
}
```

The JVM seeks for the native library when an application calls the method, whereby the real library name depends on the operating system. For Linux it will be “libexample.so”, for Windows “example.dll”.

Kaffe knows the symbol for a C function not until a Java application invokes a native Java method. Therefore, Kaffe requires the help of the L4Env to load machine specific libraries at runtime, to link them into the VM address space and to find the C functions that Kaffe has to call.

It is also possible to link the native libraries statically to the executable of Kaffe, but additional efforts are necessary. The core of JVM does not know, does not use the symbols of the functions. Therefore, the linker excludes them from the executable. In this case, the Kaffe’s building environment would have to extract all symbols of the native libraries. Then it would have to generate a C source code file and to link them to Kaffe, which lists and uses the symbols. Now the JVM would be able to find the symbols at runtime.

The L4Env provides the service “L4Exec”, which is an ELF interpreter that the “Loader” service uses to start L4 application at runtime. In addition, both services are able to load libraries dynamically at runtime. The services provide the necessary functionality and therefore an adaptation is possible to Kaffe’s interface for loading native libraries.

Chapter 4

Implementation

In this chapter, I describe some of the steps that were necessary to port Kaffe. The port works for the IA32 hardware architecture and the L4v2 ABI. I use in the context of stopping and resuming threads assembler to save and restore the state of the threads. Therefore, modifications will be necessary for L4Env based systems on other hardware architectures.

4.1 Kaffes configuration and build system

I decided to use the configuration and build system of Kaffe, because it is aware of different implementations of threading systems and operating systems. Each implementation has its own paths in the directory hierarchy and so I introduced for the DROPS port the two necessary paths:

```
config\i386\drops
kaffe\kaffevm\systems\drops-l4threads
```

Kaffe has to be compiled and linked with the cross compiling option, otherwise the build system guesses that is for the currently running system, in our case Linux.

4.2 Thread data structure management

Each specific thread implementation has to define the `jthread` data structures, which Kaffe uses to describe Java threads. The data structure is a wrapper that has to consist of the `threadData` data structure and additional thread implementation specific information. The core of Kaffe uses the `threadData` to describe per Java thread data, which are independent of the underlying native threads.

The management of the `jthread` data structures is part of the specific thread implementation. Kaffe uses the function `jthread_current` to acquire the `jthread` data for the currently running thread. Additionally Kaffe requires a facility to get all thread data structures selectively or successively. The garbage collector requires the successively facility to invoke for each native thread the function to scan the native stack to find used Java objects - implemented by the function `jthread_walkLiveThreads`.

I decided to manage the `jthread` data structures in a list, which is double linked by pointers. The L4Env thread library has the facility to register thread local data, which the package can acquire easily (fast) back. I use it to prevent searching the (maybe long) list of threads by assigning the Java thread data structure `jthread` directly to the native thread. `jthread_current` uses the local thread data feature directly to provide the requested data structure. If a thread ends and the thread implementation has to remove it from the list it will find the thread quickly by the L4Env thread local data feature. Additional the predecessor and the successor of the thread has to be updated about the removal of the termed thread and about their new neighbors. Without the double linked list, the thread implementation would have to scan for the predecessor from the start of the list.

4.3 Delaying of newly created threads

Newly created thread starts its execution by getting a binary semaphore. The critical section is not free and the semaphore is hold until the thread that initiated the creation of the new thread has stored the runtime information. The semaphore prevents the JVM asking for the Java thread data structure before it is available.

4.4 Suspending and resuming threads

The L4Env thread library does not support stopping and resuming the execution of threads. Therefore, a solution with the help of the system call `l4_thread_ex_regs` is necessary. The system call disrupts the normal execution of a thread. The original execution has to continue later, therefore it is necessary to save the state of a thread. The language C does not provide support for saving and restoring registers of a CPU. I have to use

the x86 specific instructions of the assembler language.

When I stated the port, we discussed many variants, how the stopping and resuming of threads can be achieved. We came to the conclusion that the garbage collector thread has to wait until all threads have saved their state and the last thread has to notify the garbage collector thread. Otherwise, the resuming of the threads is not safe, because the garbage collector cannot decide whether all threads have already saved their states. The function to resume threads assumes that the state of each thread is saved.

In the first attempts, I suggested the usage of semaphores to avoid race conditions and to synchronize the execution of the threads in the context of the garbage collector thread. During the implementation phase, I noticed that it is not possible. The Java threads can be involved in a receive IPC from the semaphore thread, which will not response before a critical section is free. If the garbage collector thread suspends a thread, the kernel will cancel the IPC. In the meantime, a critical section can be freed by another thread, which is not suspended yet. Now the semaphore thread tries to send an IPC to the already suspended thread. The semaphore threads will block until the IPC is successful. Therefore, the usage of semaphores in this case caused deadlocks, because the suspended thread can be resumed only after the garbage collector finishes and the garbage collector cannot finish before the semaphore thread answers. A simple IPC send and receive is sufficient for the notification about all stopped threads, so I do not use the semaphores in the context of the garbage collector thread.

4.4.1 Stopping all threads

A thread invokes the garbage collector thread when it could not get memory for a new object from the Java heap. The garbage collector calls the function `jthread_suspendall` of the thread interface. The function iterates through the double linked list of `jthread` thread data structures. For each thread, the function invokes the `l4_thread_ex_regs` to set the instruction pointer to the `_l4threads_suspendall` function.

`_l4threads_suspendall` first reserves space for the return address on the stack by pushing a place holder address onto it. In the second step, `_l4threads_suspendall` saves the state of the thread by pushing the EFLAGS and the eight general purpose registers onto the stack. Then a shared counter is incremented atomically. Each thread compares the counter with the number of all threads that have to be stopped, which

the garbage collector has set before in a global variable. If the thread is not the last one, then it will call a function to sleep forever. Otherwise the thread informs the garbage collector thread per IPC about the successfully completion of stopping all threads and then it also sleeps forever.

The garbage collector thread waits after setting all threads to the `_l4threads_suspendall` address for the notification IPC of the last thread as mentioned above. After receiving the IPC, all threads beside the garbage collector have stopped and they have saved their state on the stack. Additional the garbage collector thread saves the old stack pointer and the old instruction pointer for each stopped thread in the specific `jthread` data structure, because the `l4_thread_ex_regs` returns them after its invocation. The stopped threads do not know where their instruction pointer was before they entered `_l4threads_suspendall`. Now the garbage collector thread is able to scan safely the stack of the stopped threads for Java object references and to handle the garbage collection.

Additionally the garbage collector thread checks that it accepts only the notification IPC of a thread from its own address space. Moreover the threads must verify that the IPC was really sent respectively received and not canceled or aborted, because for example another service used the `l4_thread_ex_regs` system call.

4.4.2 Resuming all threads

After the garbage collector finished his work, it calls the `jthread_resumeall` function that the thread interface defines. Again, the garbage collector thread iterates the list of all threads and prepares them for resuming at their interrupted execution point.

First, the garbage collector thread replaces the mentioned place holder address on the stack by the saved old instruction address, which will be used as return address. Then it calculates the stack pointer address for each thread, so that the address points to the last saved general purpose register. Now the garbage collector uses the `l4_thread_ex_regs` system call to set the stopped threads to a new instruction pointer marked as `_l4threads_resumepoint` and to the calculated stack pointer.

`_l4threads_resumepoint` can now restore the state of the concrete thread by copying the saved registers and the EFLAGS from the stack back to the CPU registers, because the stack pointer is correctly set. Then the

thread can simply return, because the real return address replaced the place holder address.

4.5 Semaphore problems with timing constraints

The implementation of the semaphore interface of Kaffe by using the semaphore library of L4Env made no problems, because I could directly use the functions `semaphore_down`, `semaphore_up` and `semaphore_down_timed` of L4Env. The timing values used by Kaffe and `semaphore_down_timed` are both in milliseconds. One timing value of Kaffe is different to the one of the L4Env semaphore. Kaffe interprets a timeout of zero as (possible) infinite waiting and the L4Env semaphore implementation interprets it as a single attempt to get the semaphore.

Problems caused the existing implementation of the `semaphore_down_timed` of the L4Env. The attempt to enter a critical section with timing constraints caused deadlocks or permitted threads to enter a critical section when the section was not free. The following scenarios describe the reasons, which caused the problems.

4.5.1 Deadlocks caused by semaphore_down_timed

The original implementation of `semaphore_down_timed` used an IPC call with timing constraints to communicate with the semaphore thread and thereby to enter the critical section. The semaphore thread enqueued the calling thread when the critical section was not free. If a timeout occured, the waiting thread called the semaphore thread again to inform about the event and to induce the dequeuing of itself. The L4Env semaphore implementation used a normal `semaphore_up` for that purpose. The `semaphore_up` call caused the semaphore thread to dequeue the first waiting thread. The semaphore thread then tried to wake up the dequeued thread, assuming that it wanted to enter the critical section. If no other thread tried in the meantime to enter the same critical section, the dequeued thread was the same which caused the dequeuing. But, this thread with the timeout did not wait any longer for any IPC from the semaphore. The `semaphore_up` implementation was not the right instrument to inform the semaphore thread about the aborted attempt with a time out.

4.5.2 Incorrect admissions to enter a semaphore

The reason for the above deadlock was also responsible for entering of non-free critical sections. Assume two threads try to enter a critical section that is not free. The first thread uses a `semaphore_down` without timing constraints. It is enqueued by the semaphore thread. Now the second thread tries to enter the critical section with a timing constraint and is enqueued at the second position in the queue. The IPC of the second thread was canceled by a time out and so it informes the semaphore thread about the event using `semaphore_up`. The semaphore thread then dequeues the first waiting thread, wakes it up and allows it to enter the critical section. The first thread enters the non-free critical section and the semaphore thread did not dequeue and remove the second thread that timed out.

4.5.3 Solution for getting a semaphore with timeouts

The scenarios show that the semaphore thread must know about the different attempts to enter a semaphore — whether it is with or without a timing constraint — to ensure a correct behavior. Moreover, the semaphore thread has to mark the queued threads that entered with timing constraints. If the semaphore thread tries to wakeup such marked threads, then it will use only one attempt to send an IPC. If the thread is not ready to receive the IPC, the semaphore will assume that the attempt to enter the semaphore was canceled because of a timeout.

I introduced two new identifiers for an IPC call to the semaphore thread. "BLOCKTIMED" means that a thread tries to enter a critical section with timing constraints and "RELEASETIMED" informs the semaphore thread about a canceled attempt. A thread will only inform the semaphore thread about a cancellation when it happens in the receive phase of the calling thread. Otherwise, it is not necessary, because the semaphore thread did not receive the IPC and does not know of the attempt. I used the ability to register a pointer for thread local data to mark and to easily find a thread in the queue to avoid scanning the whole list. If a thread has a local data pointer than it will be the one to the waiting queue entry for the thread, which was called with timing constraints. If the semaphore thread tries to wake up a thread that was canceled, but the information about the cancellation, the IPC, has not arrived at the semaphore thread yet, then the wakeup IPC would fail and the semaphore thread will wake up another thread. The semaphore thread detects and ignores IPC arriving too late, because the thread local data pointer is already set to NULL by the previous failed wake

up attempt.

4.6 Files

At the start of the port I decided to use the ability of the indirection layer to provide files for the JVM. Kaffe requires only its standard class library - rt.jar, and the java class of the application. I statically linked these two files to the JVM executable with the help of GNU's OBJCOPY. Additionally I modified `KOPEN` and `KREAD` so that they were able to read the files from the executable of Kaffe.

Meanwhile the Dietlibc implementation for the Posix file functions is available. Therefore, I adapted `KOPEN` and `KREAD` to the normal posix file functions. For testing and providing the files, I use the simple file server of the L4VFS package that also links the provided files statically to the server's executable as I did first with the executable of Kaffe.

Additional, Kaffe requires environment variables for the path settings, which describe where Kaffe can find its libraries (jar files), especially rt.jar. The Dietlibc does not provide the function `getenv`, therefore I use a rudimental implementation. My implementation of `getenv` tries to find and open a file "kaffepath.env", where the user of Kaffe can specify paths settings, for example "BOOTCLASSPATH=rt.jar:kjc.jar". If the file is not available, Kaffe uses the standard settings of the paths.

4.7 Native libraries

Kaffe has to be started with the "Loader" service to use the ability of dynamically linking libraries. Frank Mehnert enhanced the "Loader" and "L4exec" service by the facility to support scanning the native libraries for a function at runtime. With the enhancement, I was able to use the interface of the "Loader" for the native library functions in Kaffe. At the moment, the file providers used by the Dietlibc do not support the generic file provider interface (generic_fprov), which the "Loader" and "L4exec" service require to open and to read files. Therefore, other file providers than the L4VFS servers have to provide the native libraries.

The configuration system of Kaffe allows compiling the core and basic libraries for the JVM as shared ones. At the moment the configuration process

makes trouble, because the configuration does not know about the system DROPS. Therefore, Kaffe's executable can only be linked statically.

4.8 Error handling

As described in 3.4.2 Kaffe can registrate its arithmetic exception handler for the software exceptions 0 and 16 in the LIDT. The thread, which causes a software exception by dividing by zero, is set by the kernel to the address of Kaffe's arithmetic exception handler. The kernel returns to the causing thread and then, Kaffe is able to perform the Java exception handling - searching for the corresponding Java `catch` handler in the executed Java binary code.

For the Java null pointer Java exception I had to modify the L4RM pager. I extended it to registrate a address by the executed application, which it has to use when a page fault on the first page occurs. The L4RM pager uses this address to set the causing thread by the system call `l4thread_ex_regs` to the registrated address. The causing thread then executes the null pointer Java exception handler, which searches for the corresponding Java `catch` handler.

Chapter 5

Evaluation

In this chapter, I show the methods and the results of the measuring of some aspects of Kaffe. First, I describe in general the used measuring methods and test applications. In the second part, I discuss the resource usage, the garbage collector invocations and the results for the Java test applications.

5.1 Measuring methods

Java test applications are necessary to evaluate the JVM. Kaffe itself is a runtime environment and provides functionality for Java applications. It is difficult to find / to produce Java applications, which consider all characteristics that affect the performance of Kaffe. Different Java applications can influence diverse aspects of the JVM and more or less affect the results. Therefore, I describe consecutively the intention of the test Java applications.

I use the time stamp counter of the CPU to measure durations of the execution time of the Java applications. Additional, I evaluate the consumed time of selective functions, which Kaffe often uses and I had to implement. In the case of the selective functions I use only one Java thread. With this limitation I try to reduce the probability that a Java thread interrupts another one, which would adulterate the results.

5.1.1 Test application: matrix multiplication

The first Java Application multiplies 5x5 matrices with each other and performs 65536 multiplications. The application distributes the multiplications over a certain amount of threads, whereby the number of threads can be configured at startup of the Java application.

The intention of the application is to produce many operations like summations and multiplications, many method invocations and bad resource usage. The application provokes the invocation of the garbage collector by wasteful resource usage. I use “wasteful” in terms of using the Java class `java.lang.Long` instead of the native type `long`. It causes for the resulting matrices the allocation of new objects at the Java heap, which the Java application only uses for the following multiplication and then not afterwards.

5.1.2 Test application: consumer and producer

The second Java Application is a consumer/producer scenario. The producer generates elements (a product) and deposits it in a buffer. The consumer takes elements from the buffer (consumes it). Getting and putting elements to the buffer have to be synchronized to avoid overflows of the buffer caused by the producer and concurrent extractions of elements by several consumers.

The Java application uses 30 consumers and 30 producers, whereby each of them is a separate Java thread. All of them try to access the buffer 5000 times, the producer to put an element in the buffer and the consumer to get one. The methods of the buffer Java class are `synchronized`, so that only one Thread can get or put an Element from/to the buffer. The buffer is a ring buffer and the available places for the elements can be configured at the start of the Java application.

The intention of the application is to measure the influence of synchronization efforts, which the JVM has to take for concurrent access to Java objects. The scenario stresses the semaphore implementation and causes additionally many thread switching.

5.1.3 Selective JVM functions

In the first step, I identified mainly four C functions, which Kaffe often uses during the normal execution of classes and I had to implement.

- `jthread_on_current_stack`
- `jthread_stackcheck`
- `jthread_extract_stack`
- `jthread_current`

`jthread_on_current_stack` determines, whether a location of an address is on the stack of the current running thread. `jthread_stackcheck` checks, whether enough space is on the actual stack of the running thread. `jthread_extract_stack` determines the stack range of the running thread, which the garbage collector has to scan. `jthread_current` returns the associated local data pointer of the native thread. The address points to the `jthread_t` data structure, which describes a Java thread used by the JVM.

5.2 Test scenarios and environments

For evaluation, I decided to compare the execution of Kaffe in three different environments, native Linux 2.6.7, L4Linux 2.2 on Fiasco and L4Env on Fiasco. I name the different Kaffe execution environments in the diagrams to this port as L4Env, native Linux as Linux and Kaffe on Fiasco as L4Linux.

The test environment was an Athlon XP 1800+ (1533Mhz = 133Mhz x 11.5), 256 DDR RAM (133MHz), Board MSI KT3 Ultra (VIA KT 333). I use L4Linux 2.2 with a ram disk, which includes the native Kaffe. Fiasco, all services and the ram disk are located at an ext2fs partition on the harddisk. GRUB loads it during computer startup in memory.

I wrote some C functions to take measurements with the time stamp counter, which are in the directory `kaffe\kaffevm\systems\drops-l4threads` in the two include files `mess.h` and `mess_types.h`. I add the measure points to the source code of Kaffe for the native Linux and for the L4Env version.

The start measure point is the first instruction in the main function of Kaffe, which is the same for Linux and L4Env. The end measure point is in the function `jthread_exit`. This function exists separately for each threading system. As end measure point I use the location in these functions, where the last non-daemon thread exits a Java application. The JVM terminates the execution of a Java application as described in the Java specification, if the last non-deamon thread exists.

I measured the execution time of the matrix application. One to 56 threads calculate the 65536 matrix multiplications, whereby the steps from 1 to 16 threads are in steps of power by 2 and from 16 to 56 threads in steps of 8. The results in figure 5.1 shows that Kaffe on Linux and L4Linux executes the Java application faster than the L4Env

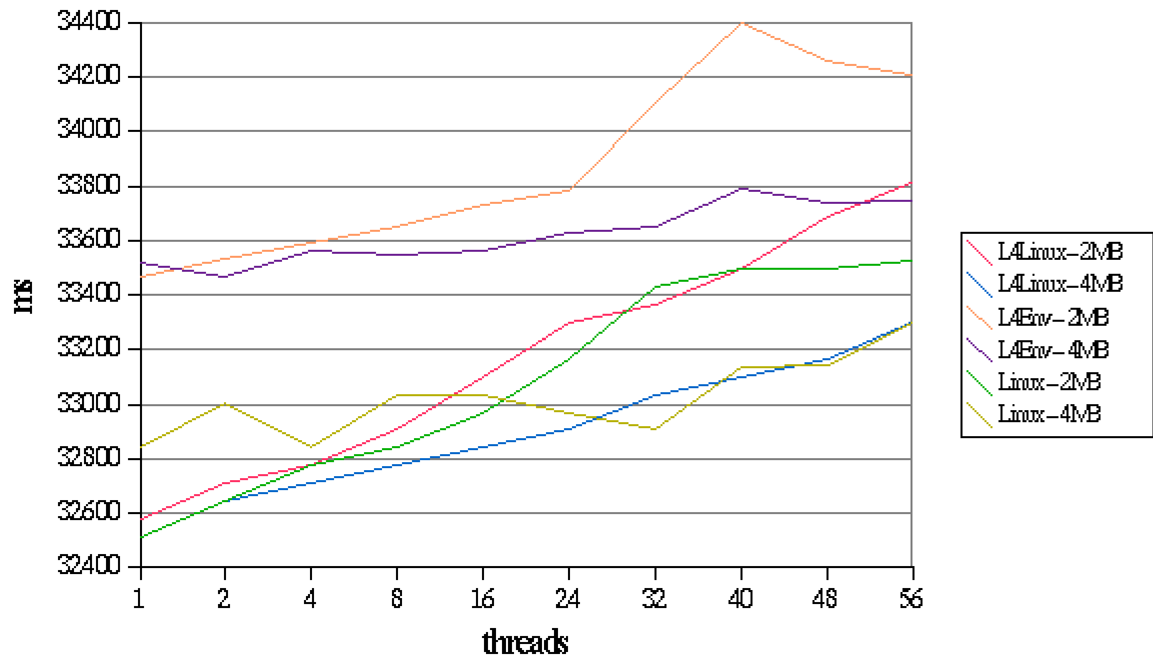

Figure 5.1: matrix application, execution time, Athlon XP 1800+(1533MHz)

version. The reason I lead back to the four `jthread` functions mentioned above. The table 5.1 shows the best and the average time of the execution time of these functions. The invocations of the L4Env version are two to three times slower then the one of Linux. The Linux `jthread_stackcheck` invokes `jthread_on_currentstack` and it invokes `jthread_current`, therefore the invocations of these functions are so high. The measured tacts for `jthread_current` and `jthread_on_currentstack` are not directly comparable, because the L4Env implementation of these functions not call one another. But, for `jthread_stackcheck` and `jthread_extract_stack` it is possible. The last column shows the time difference for the `jthread_stackcheck`, which is mainly is the reason for the time difference of the complete execution time between Kaffe on L4Env and Linux.

The figure 5.2 shows the results for the producer and consumer scenario. Kaffe on L4Linux requires about one dimension more time then the L4Env version for a small buffer size. The gap between L4Env and L4Linux becomes smaller with more buffers, but it is still a factor of two (buffer size = 50, L4Env circa 2.7s, L4Linux circa 7s). The results heavily depend on

Function jthreads*	Kaffe	Invocations	Total tacts	Noninterdependence invocations	Min tacts	Average tacts	Tact difference between L4Env and Linux divided by frequence (in ms)
_current	L4Env	3.549	7,71e+05	3.549	00	217	—
	Linux	2,35e+07	1,23e+09	3.802	39	52	
_on_current-stack	L4Env	4.621	1,07e+06	4.621	112	232	—
	Linux	2,35e+07	1,27e+09	4.180	42	54	
_stackcheck	L4Env	2,35e+07	2,95e+09	2,35e+07	110	126	1046
	Linux	2,35e+07	1,35e+09	2,35e+07	44	57	
_extractstack	L4Env	148	3,20e+05	148	697	2161	0
	Linux	126	2,70e+04	126	26	215	

Table 5.1: Matrix application, 1 thread, Java heap 2 MB, comparison between jthread functions, Athlon XP 1800+(1533MHz)

the schedule of the threads and the semaphore invocations, which are better for L4Env than for L4Linux shown in figure 5.3. I cannot explain, why the semaphore invocations are so much higher for Kaffe on Linux and L4Linux then for the L4Env version. I guess it has to do with the schedule of the threads. I also tried to measure the behavior on Linux, but the results for all measure points (different buffer size) fluctuate heavily between 4 seconds and 30 seconds. Therefore, I would have to describe the Linux results as scatter plot in figure 5.2 and thus it is not shown.

5.3 Resource usage

Kaffe can be configured at startup to use a maximal amount of memory for the Java heap, the initial heap size and the size of heap increments for expansion. The minimal heap size, which worked in my test cases, was 1.5 Mbytes. Below it, the JVM stops with the error “Internal error: caught an unexpected exception”. Therefore, I used for the tests as minimal heap memory 2 MB.

Kaffe’s memory amount for Linux I determine with “`ps o user,pid,vsize,fname`”. The table 5.2 shows the vmsize for

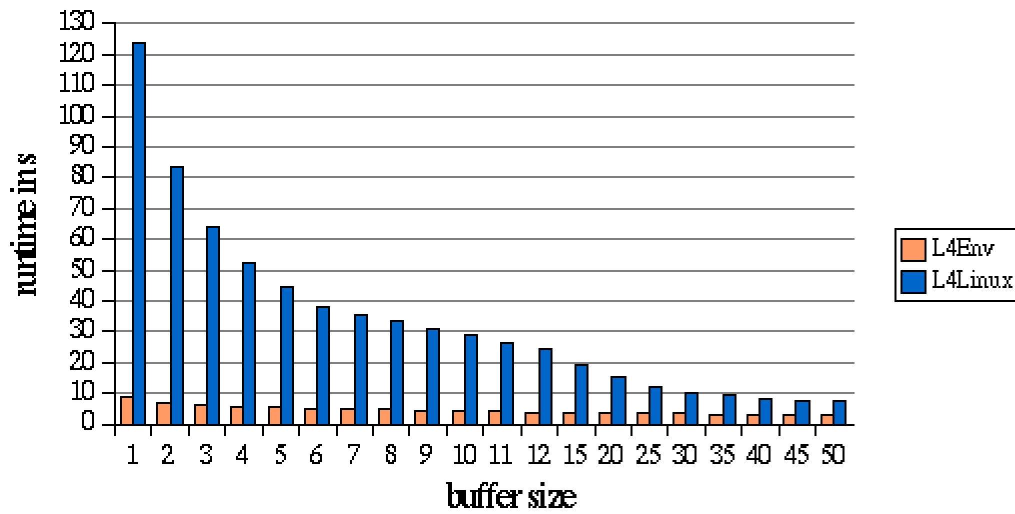

Figure 5.2: producer / consumer scenario, execution time, Athlon XP 1800+(1533MHz)

both applications. For the matrix application the minimum amount of memory is 5.5 MB plus the used memory for the heap. Each additional thread adds the amount of about 64 Kbytes, because this is the standard stack size for the native threads. The producer/consumer application requires about 11.5 Mbytes. This scenario uses three classes and more functionality from the Java thread package than the matrix scenario like synchronization. This does not explain why the memory amount is about 4 MB higher than for the matrix application.

5.4 Garbage Collector

The JVM invokes the garbage collector when no memory is available. Kaffe calls the function

- `jthread_suspendall` to stop all Java threads and
- `jthread_unsuspendall` to resume their execution.

If the garbage collector has to stop n threads, then the following invocations are necessary:

- n times the systemcall `l4_thread_ex_regs`,
- one receive IPC by the garbage collector,

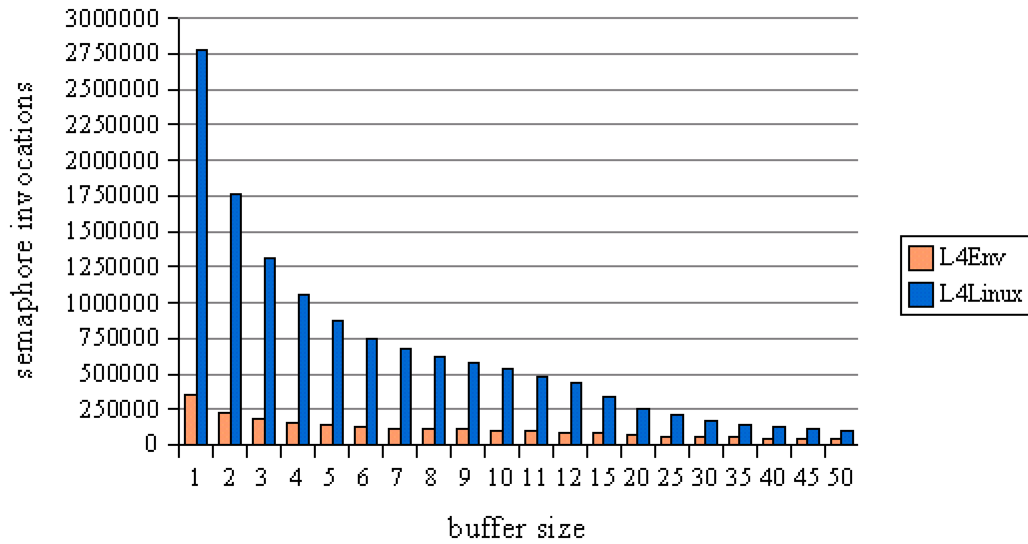

Figure 5.3: producer / consumer scenario, semaphore invocations, Athlon XP 1800+(1533MHz)

- one send IPC by the last stopped thread.

For resuming all threads

- n times the systemcall `l4_thread_ex_regs`

is necessary. The time of the duration of a `l4_thread_ex_regs` system call and an IPC depends mainly on the used hardware. I measured the total duration from the start to the end of the matrix Java application shown in figure 5.4. L4Env Kaffe calls the garbage collector more often, but the absolute required time was less. Figure 5.5 shows the consumed time per garbage collector invocation. I think the reason for the better values for L4Env are the stopping and resuming of the Java threads. In L4Linux each Linux thread is a separate L4 task. Therefore, a higher overhead in address switching is necessary.

Invocation	Linux threads	VMsize in kB
./java -ms2M -mx2M Matrix 1	5	7.612
./java -ms4M -mx4M Matrix 1	5	9.680
./java Matrix 1	5	11.308
./java -ms2M -mx2M Matrix 4	9	7.816
./java -ms4M -mx4M Matrix 4	9	9.884
./java Matrix 4	9	11.512
./java -ms2M -mx2M ProducerConsumer 1	64	11.628
./java -ms4M -mx4M ProducerConsumer 1	64	13.696
./java ProducerConsumer 1	64	15.324
./java -ms2M -mx2M ProducerConsumer 4	64	11.628
./java -ms4M -mx4M ProducerConsumer 4	64	13.696
./java ProducerConsumer 4	64	15.324

Table 5.2: Memory usage of Kaffe on Linux ascertain with ps, Athlon XP 1800+(1533MHz)

5.5 Code size, modifications and maintenance

About 3000 lines of source code were necessary to adapt the L4Env environment to Kaffe (without the code for measurement). I modified the core of Kaffe to prevent a race condition by adding a semaphore and I had to include for the "Loader" and "L4Exec" service a include file to be able to resolve native functions at runtime. No other changes at the core source code of the JVM were necessary, because of the interfaces for the threading and semaphore implementation of Kaffe. The implementations of this port for Kaffe are an additional operating system and an additional threading system, which are clearly separated from the source code of the other implementations.

At least the files "kaffevm", "rt.jar" and "libloader.s.so" are necessary to run Kaffe at DROPS. The size of these files (circa):

```
kaffevm         2300 kBytes - JVM binary
rt.jar           450 kBytes - Java library
libloader.s.so   600 kBytes - Loader native library
```

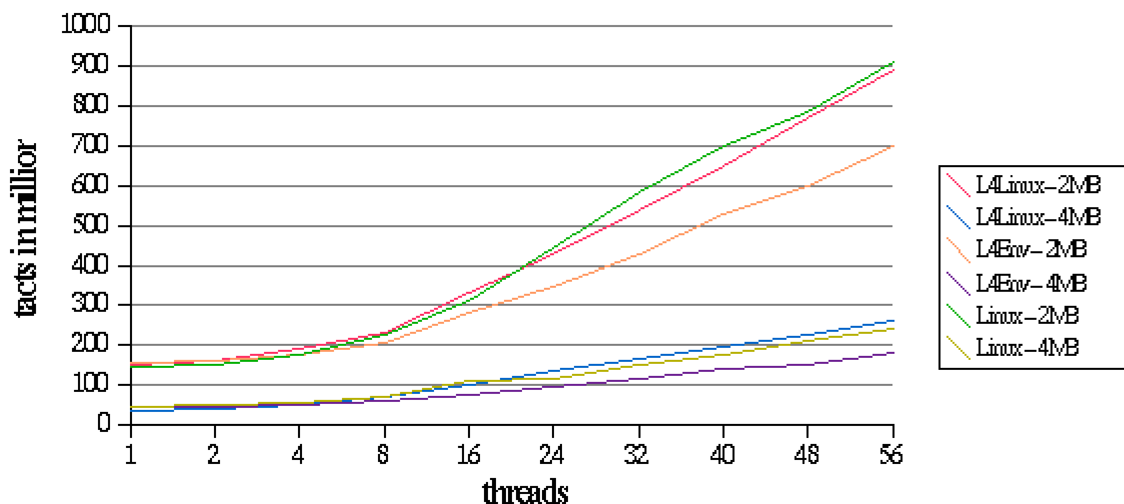

Figure 5.4: matrix application, complete execution time of the garbage collector thread, Athlon XP 1800+(1533MHz)

The basic library “rt.jar” is a minimal version, which only provides the basic classes to run simple Java applications. Additional, beside the kernel “fiasco” and the basic services “sigma0” and “rmgr’ the following services of the L4Env environment are necessary: “names”, “dm_phys”, “rtc”, “simple_ts”, “name_server”, “l4exec”, “loader”, “simple_file_server” and “tftp”. Optional the “con” and “term_con” services are useful to show console outputs from Java applications.

During the port I decided to use the build system of Kaffe. I had some trouble to use it solely with L4Env without modifications. Therefore, I use the build system of Kaffe to compile the basic parts of Kaffe as libraries. For the resulting binary of L4Env Kaffe I use the DROPS build system with the Kaffe libraries. The integration into the build system of DROPS is almost finished.

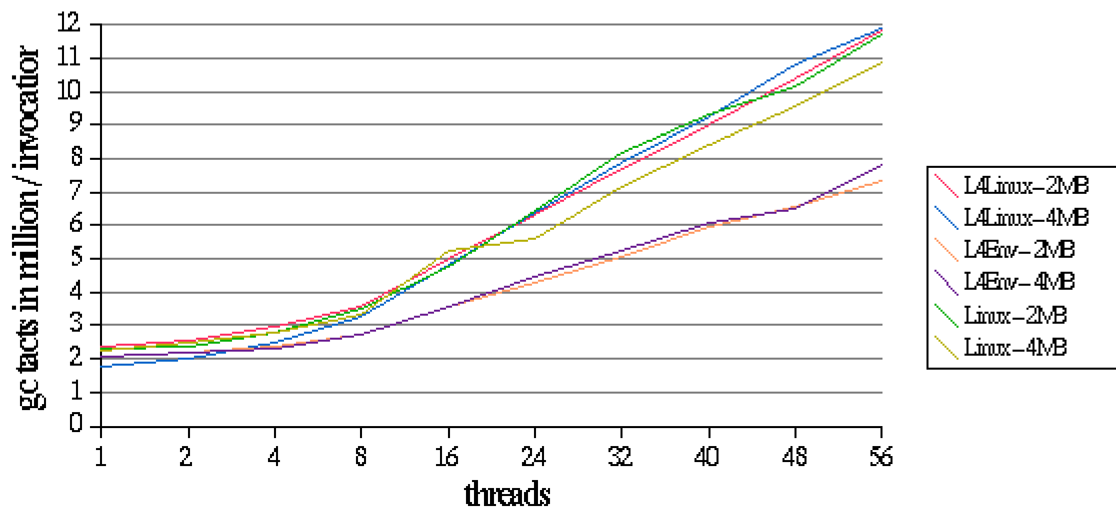

Figure 5.5: matrix application, execution time of the garbage collector per invocation, Athlon XP 1800+(1533MHz)

Chapter 6

Conclusion

6.1 Further tasks

The users often use Java in the context with Graphical User Interfaces (GUI), like AWT and Swing. Additional, Java is often used with Servlet containers like Apache Tomcat and EJB (Enterprise Java Bean) component containers like JBoss.

For DROPS exists a real-time window environment named DOpE. The adaptation to DOpE could allow the usage of GUI Java applications. I do not know, whether DOpE provides all functionality required by GUI Java applications, but this analysis and implementation is an interesting task.

The container for EJB and Servlets requires mainly an implementation for the socket interface, which has to use the existent IP stack, named FLIPS, and the corresponding network card drivers. In the context of a diploma a student already implements the socket interface for the Dietlibc, therefore it could be usable in the near future for Kaffe.

A complex task is the analysis of Kaffe to determine, whether it is possible to support realtime Java applications and how this could be achieved. It would be necessary to identify the critical parts in term of predictability like the garbage collector and others. The Java community already researches this area for a long time and a good start is the study of their Real-Time Specification [Bollella et al., 2000] for Java.

6.2 Summary

The main goal of this work was to port a JVM to DROPS by the usage of L4Env, which has been achieved. Especial that Java threads run as real L4Threads, classes and native libraries can be reloaded at runtime, which makes Kaffe useable. The core of the JVM works, but the big Java API is widely untested and can cause errors, which have to be found and fixed. Parts of the Java API, which make use of unimplemented native functions partly due the Dietlibc and the port, will not run. In the future the missing functionality will be added.

Some extensions to the L4Env services like the semaphore library (deadlock problem), the L4Thread library (stack pointer issues), the Loader/L4Exec service (symbol finding at runtime) and the L4RM pager (null pointer handling) were necessary. The port of Kaffe to DROPS was possible due the good support by the maintainers of these services and packages.

Finally, a special thank goes to my tutors Ronald Aigner and Martin Pohlack, who instructed and supported me. Additional, due the work on Kaffe I really learned to understand some aspects of Java and operating systems, especial microkernels.

Bibliography

[Alpern et al.] Bowen Alpern et al.. *The Jalapeo Virtual Machine*. In: IBM System Journal, 39:1, 2000, p. 211-238.

[Aron et al., 2001] Mohit Aron, Yoonho Park, Trent Jaeger et al.. *The SawMill Framework for VM Diversity*. In: Proceedings of the 6th Australasian Computer Architecture Conference, Gold Coast, Australia, 29.01.-30.01.2001 [WWW Document]. URL: ftp://ftp.cse.unsw.edu.au/pub/users/disy/papers/Aron_PJLED_01.ps.gz (State 02.06.2004).

[Friedman, 2002] David K. Friedman and David A. Wheeler. *Java Implementations*. [WWW Document]. URL: http://www.dwheeler.com/java-imp.html (State: 28.05.2004).

[GCJ, 2004] GCJ Documentation [WWW Document]. URL: http://gcc.gnu.org/java/docs.html (State: 28.05.2004).

[Hohmuth, 1998] Michael Hohmuth. *The Fiasco kernel: Requirements definition*. Technical Report TUD-FI98-12, TU Dresden, 1998 [WWW Document]. URL: http://os.inf.tu-dresden.de/papers_ps/fiasco-spec.ps.gz (State: 02.06.2004).

[Hohmuth, 2002] Michael Hohmuth. *The Fiasco kernel: System architecture*. Technical Report TUDFI02-06-Juli-2002, TU Dresden, 2002.

[IBM, 2004] Homepage of IBM Watson Research Center [WWW Document]. URL: http://www.watson.ibm.com/ (State: 28.05.2004).

[Jikes, 2004] Homepage of Jikes [WWW Document]. URL: http://www-124.ibm.com/developerworks/oss/jikes/ (State: 28.05.2004).

[Jikes RVM, 2004] Homepage of Jikes Research Virtual Machine (RVM) [WWW Document]. URL: http://www-124.ibm.com/developerworks/oss/jikesrvm/ (State: 28.05.2004).

[Kaffe.org, 2004] Homepage of Kaffe.org [WWW Document]. URL: http://www.kaffe.org/ (State: 28.05.2004).

[L4Env, 2003] Operating Systems Research Group Technische Univeritt Dresden. *L4Env - an Environment for L4 Applications*. [WWW Document]. URL: hhttp://wwwos.inf.tu-dresden.de/l4env/doc/l4env-concept/l4env.pdf (State: 02.06.2004).

[LaTTe, 2004] Homepage of LaTTe [WWW Document]. URL: http://latte.snu.ac.kr/ (State: 02.06.2004).

[Liedtke, 1996] Jochen Liedtke. *Towards real -kernels: The inefficient, inflexible first generation inspired development of the vastly improved second generation, which may yet support a variety of operating systems*. In: Communications of the ACM, 39:9, 1996, p. 70-77.

[Oechsle, 2001] Rainer Oechsle. *IParallele Programmierung mit Java Threads*. Fachbuchverlag Leipzig, Mnchen, 2001.

[SableVM, 2004] Homepage of the SableVM Project [WWW Document]. URL: http://www.sablevm.org and http://www.sable.mcgill.ca (State: 06.2004).

[Venners, 1999] Bill Venners. *Inside the Java 2 Virtual Machine*. McGraw-Hill Book Company, New York, 1999.

[Pinilla et al., 2003] Ruben Pinilla, Marisa Gil. *JVM: platform independent vs. performance independent*. In: ACM SIGOPS Operating Systems Review, volume: 37, Number 2, April 2003.

[Pinilla et al., 2003] Ruben Pinilla, Marisa Gil. *ULT: A Java threads model for platform independent execution*. In: ACM SIGOPS Operating Systems Review, volume: 37, Number 4, October 2003.

[L4Ka, 2004] Homepage of the L4Ka Project [WWW Document]. URL: http://l4ka.org State: 06.2004.

[Oskit, 2002] Homepage of the Oskit Project [WWW Document]. URL: http://www.cs.utah.edu/flux/oskit State 03.2002.

[Dietlibc, 2004] Homepage of the Dietlibc Project [WWW Document]. URL: http://www.fefe.de/dietlibc

[SUN, 2004] Homepage of the Java language and JVM of SUN Microsystem [WWW Document]. URL: http://java.sun.com State 06.2004.

[Back, 2000] Godmar Back, Wilson C. Hsieh, Jay Lepreau. *Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java.* In: 4th Symposium on Operating Systems Design & Implementation, 2000. URL: http://www.stanford.edu/ gback/thesis/kaffeos-phd-thesis.pdf State 06.2004

[Back, 1999] Godmar Back *Garbage collector strategy of Kaffe* [WWW Document]. URL: http://www.kaffe.org/doc/kaffe/FAQ.gcstrategy state 06.2004

[Baker et al., 2000] Jason Baker, Alexandre Oliva, Patrick Tullmann *A brief summary of locking in Kaffe* [WWW Document]. URL: http://www.kaffe.org/doc/kaffe/FAQ.locks state 06.2004

[Lindholm, Yellin] Tim Lindholm, Frank Yellin *The JavaTM Virtual Machine Specification, Second Edition*

[Liang] Sheng Liang *The JavaTM Native Interface - Programmer's Guide and Specification*

[Bollella et al., 2000] Greg Bollella, Ben Brosgol, Seve Furr, David Hardin, Peter Dibble, James Gosling, Mark Turnbull *The Real-Time Specification for Java* Addison-Wesley, 2000.