Martin Maas

A JVM for the Barrelfish Operating System

Computer Science Tripos, Part II

Queens' College, University of Cambridge

2010-2011

Proforma

Name:	**Martin Maas**
College:	**Queens' College**
Project Title:	**A JVM for the Barrelfish Operating System**
Examination:	**Computer Science Tripos, Part II**
Year:	**2010-2011**
Word Count:	**11,802**
Project Originator:	**Dr Tim Harris**
Supervisor:	**Dr Ross McIlroy**

With the sanction of both my Director of Studies and my Overseers, the Appendix exceeds the normal length of 15 pages.

Original Aims of the Project: Barrelfish is a research OS developed at ETH Zurich and MSR. The aim of this project was to create a JVM for Barrelfish. This includes the implementation and evaluation of a Java Bytecode interpreter and enabling it to run on the system. Extensions of the project would explore ways to execute the interpreter on multiple cores, contrasting a shared-memory approach and a distributed systems approach.

Work Completed: I completed all the work I set out to do, including the extensions. I implemented a working JVM that supports 198 out of the 201 Java Bytecode instructions and runs a significant set of real-world programs (including many benchmarks from the Java Grande Benchmark suite). The JVM runs on Linux as well as Barrelfish and has been extended to run parallel applications on both systems. On Barrelfish, it supports both a shared-memory approach and a distributed approach.

Special Difficulties: The Barrelfish version I was originally using proved to be unsuitable for the project since it did not work on any hardware available to me and contained bugs that prevented the distributed JVM from working. This difficulty was resolved when a new version became available in March 2011. This problem led to delays in the project (such as having to change to different APIs), but did not prevent it from being completed successfully.

Declaration of Originality

I, Martin Maas of Queens' College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed Date

Contents

Acknowledgements

Thanks is owed to everyone who supported me in this project. First and foremost, I would like to thank my supervisor, **Dr Ross McIlroy**, for his guidance and support throughout the entire project. Special thanks also goes to **Dr Tim Harris** for supervising the early stages of this project and supporting it throughout.

I would like to thank **Dr Anil Madhavapeddy** and **Malte Schwarzkopf** for setting me up with access to the test machines at the Systems Research Group and supporting me in running my experiments.

1 Introduction

This chapter describes the Barrelfish operating system and how it fits into the current research landscape. It explains the motivation to implement a JVM for this system and reviews related work.

1.1 Overview

Barrelfish [12] is a research operating system for future many-core architectures, developed at Microsoft Research Cambridge and ETH Zurich. This dissertation describes the implementation of a Java Virtual Machine (JVM) [32] for Barrelfish, with a particular focus on the design decisions for such a system.

The work completed encompasses a feature-reduced Java Bytecode interpreter for Linux and Barrelfish. It runs several real applications, including benchmarks from the JGF benchmark suite [23]. The system has been extended to run across multiple cores on Barrelfish, contrasting a shared memory approach with a distributed systems approach. The implementation fulfils the requirements of the project proposal, including all main extensions.

1.2 Background

1.2.1 The many-core revolution

The past years have seen tremendous changes in hardware for commodity computer systems: Core counts are increasing and systems are becoming more diverse as new types of caches, interconnects and coprocessors (such as GPUs or programmable network adapters [1]) appear. There is evidence that these developments put challenges on future operating systems [11, 13]. General-purpose OS's will not only have to scale for large numbers of cores but also support an increasing range of different system configurations, including heterogeneous architectures (such as the Cell CPU [44]) and NUMA architectures (Figure 1.2).

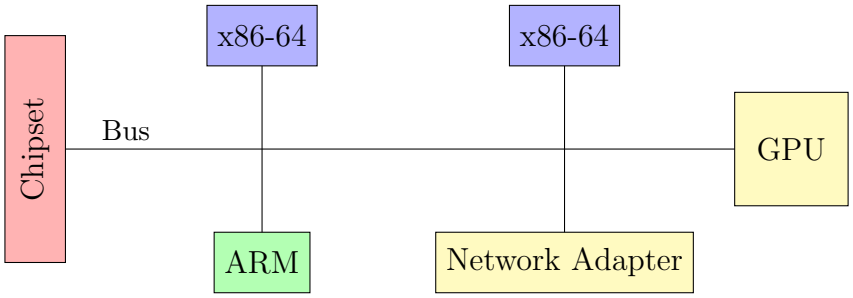

Figure 1.1: A heterogeneous architecture

While there are indicators that traditional operating system designs may scale well for current hardware even as the number of cores increases [21], there is a sense that future hardware will require fundamentally new OS structures. Mattson et al. [35] claim that the cost of cache coherence protocols prevents scaling to ever larger number of cores and that cores should instead communicate via message passing (such as the Intel SCC [34] or Beehive [45]). However, current operating systems are not structured to handle such architectures. Moreover, optimisations to make software scale to large numbers of cores (such as RCU data structures [37], address ranges [20] and approaches from HPC) are often hardware-specific and do not generalise well to a wide range of hardware.

Several OS designs have been proposed to deal with these problems, including Barrelfish, Corey [20], fos [40], Helios [43] and Tessellation [33]. There is also work on run-time environments, programming models and compilers, aiming to make software scale to large numbers of cores and heterogeneous systems.

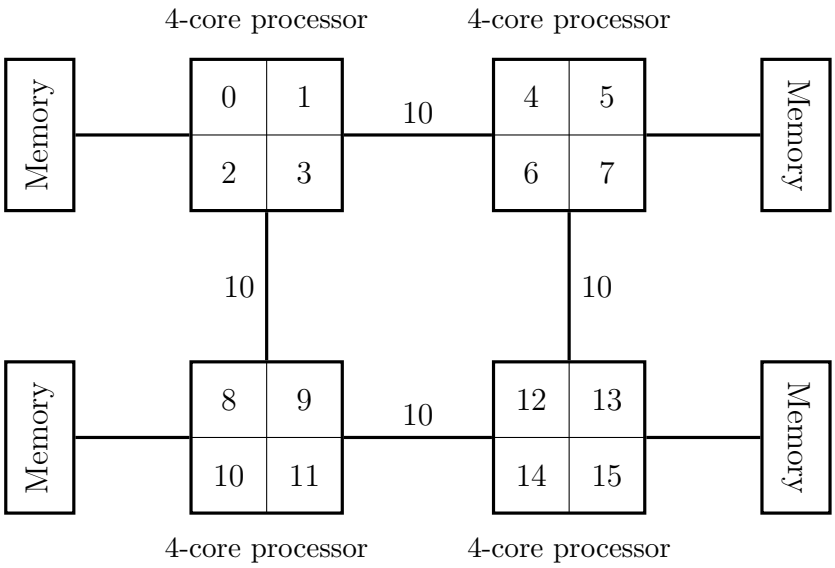

Figure 1.2: A NUMA architecture

1.2.2 The Barrelfish operating system

Barrelfish is an operating system based on the multikernel model [12], a design that treats many-core systems as a network of independent nodes communicating via message-passing (Figure 1.3). By doing so, it can exploit message-passing hardware while maintaining compatibility with shared memory architectures. State is replicated rather than shared, and traditional OS functionality such as memory management, I/O or power management is implemented as services running on different OS nodes. Barrelfish also provides mechanisms to make the OS independent from the underlying hardware and therefore allows it to run on heterogeneous architectures and adapt to a wide range of system configurations.

Inter-core communication is made explicit and implemented as a light-weight message passing library. Threads are provided via a POSIX-like programming model. While Barrelfish provides implementations of libraries such as `libc` that hide the distributed nature of the system for certain system calls (e.g. `malloc`), the programmer has to be aware of the distributed nature of the system and adopt an event-based software design for communication between cores.

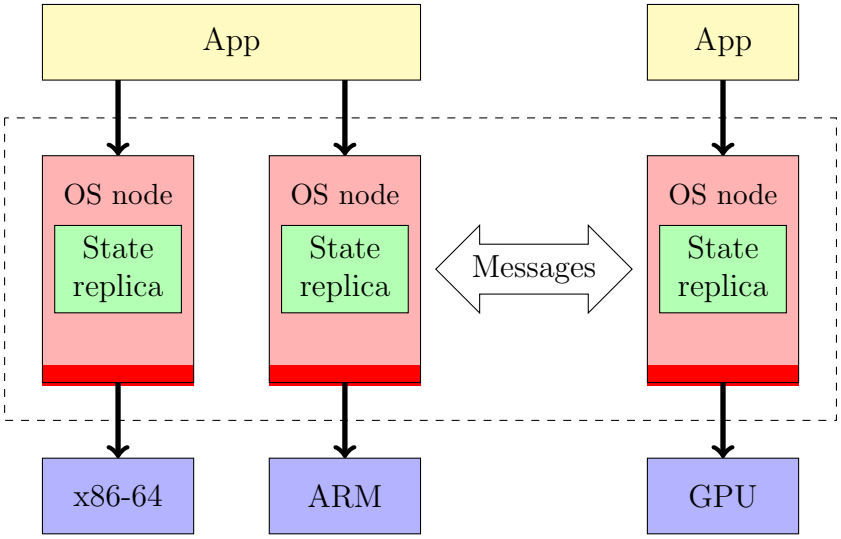

Figure 1.3: Overview of the Multikernel Model (adapted from [12])

1.3 Motivation

Operating systems like Barrelfish can profit from higher-level run-time environments such as the Java Virtual Machine, since such models have several advantages over low level languages like C:

- **Single-system image**: The run-time environment hides the distributed nature of the system. This allows it to run software that has not explicitly been developed to run on heterogeneous many-core systems. McIlroy et al. [36] demonstrated that a JVM is a suitable abstraction for such systems.
- **Transparent migration of threads between cores**: Migration of threads between heterogeneous cores is difficult in the current model, since it requires code to be compiled for each instruction set and state to be translated between the different architectures. In a JVM, it is sufficient to provide a run-time environment for each core type, since code and state of the program are hardware-independent.
- **Optimisations**: The JVM provides high-level information (such as class structures) that can be used to optimise the code, e.g. by using this information for scheduling decisions or adaptive recompilation [10].
- **Extensibility**: Java can be extended using annotations and language extensions, which makes it possible to gather additional information

from the programmer in order to improve scalability to many-core architectures. A project which demonstrates how Java can be extended for parallel computing is the Titanium project [50].

Some of these advantages also apply to parallel programming models such as Cilk [17]. However, Java has several advantages over these models: It is a general-purpose programming language and is widely adopted for both client and server applications. This is important since Barrelfish targets commodity systems rather than HPC scenarios. The high adoption also means that there are stable benchmarks, development tools and libraries available. Compared to the CLR [38], Java has the advantage of having excellent open source implementations and being more prevalent in scientific research [16].

For these reasons, it is desirable to investigate the possibility of bringing a JVM to Barrelfish. It could be used as basis for further research in fields such as JVM design for many-core systems or many-core scheduling. It could also be extended to run higher-level frameworks such as Hadoop [48] or CIEL [41] on Barrelfish.

1.4 Project Description

While the task of bringing a JVM to Barrelfish can be approached from different angles, this project focuses on design decisions for such a JVM. It contrasts a shared-memory approach with a completely distributed approach, similar to the one used by Barrelfish. This allows investigating design decisions for future hardware, which may only provide message passing instead of shared memory.

As the limited scope of the project will not allow to create a JVM that can perform competitively with industry-standard projects, I chose to build a proof-of-concept prototype that allows the evaluation of design decisions by taking performance measurements, comparing them to industry-standard JVMs and estimating performance impacts on current and future hardware.

1.5 Related Work

While the project is related to the work on many-core operating systems, parallel programming models and language run-times cited in the previous section, there is some research which is directly relevant for the project.

1.5.1 Distributed JVMs

Since 1998, there has been an extensive amount of publications on distributed JVMs for clusters [2], and projects have chosen different approaches to run Java programs distributed across multiple machines: One of the first solutions was Sun's own RMI framework [26], which implemented RPCs directly in Java. Other approaches include a custom JVM running distributed across multiple servers (cJVM [9]) and a monolithic JVM running on a distributed computing platform with distributed shared memory (Kaffemik [8]).

While this work is similar to the Barrelfish JVM, the trade-offs on a cluster are very different to those in a multi-core machine. For example, message passing is orders of magnitude cheaper and a lower error-tolerance is required. The distributed approach chosen by the Barrelfish JVM resembles cJVM while the shared memory version is similar to a JVM using distributed shared memory. The project draws inspirations from previous work in this field.

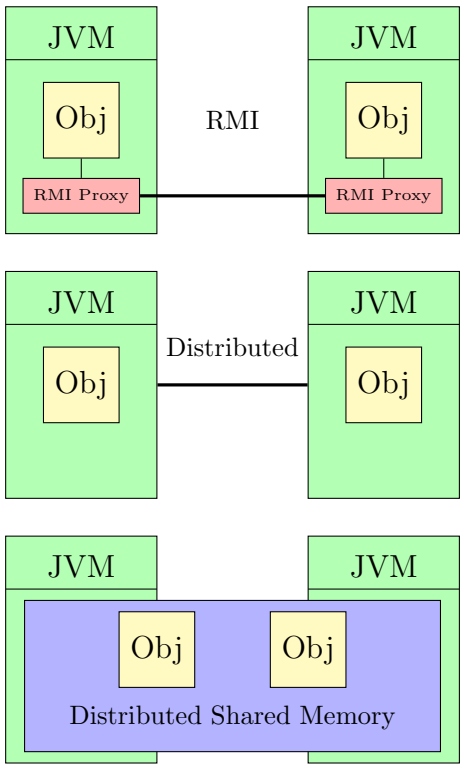

Figure 1.4: Different approaches to distributed JVMs

1.5.2 Java on many-core systems

There has been research on making Java more effective on many-core systems. One example is Kilim [46], a lightweight framework that allows Java threads on a single machine to communicate via message-passing rather than shared memory, an approach similar to that of Barrelfish. However, this system runs within Java on a traditional operating system while the Barrelfish JVM is effectively running on a distributed system with unusual characteristics.

1.5.3 JVMs for heterogeneous systems

Hera-JVM [36] implements a JVM for the heterogeneous Cell microprocessor, providing a single-system image and transparently migrating threads between cores. There are also projects that make special-purpose coprocessors available from within Java, such as JCUDA [49]. These systems differ from my project in that the Barrelfish JVM uses the underlying operating system to handle heterogenity and communication between cores.

2

Preparation

This chapter gives an introduction to the Java Virtual Machine and Barrelfish and presents material that had to be understood before any practical work could commence. It then shows a requirements analysis to determine the goals of the project and how to achieve them.

2.1 The Java Virtual Machine

The Java Virtual Machine is an abstract model of computation that executes Java Bytecode, an instruction set resembling intermediate code in a compiler. It was conceived at Sun in the 1990s and various implementations exist, such as Sun's own HotSpot JVM [7] and Jikes [16]. The JVM model is stack-based, type-aware (enabling introspection), provides garbage-collection and allows for native function calls. It is based around Java's class model and its notion of methods, fields and class instances.

The JVM is specified in "The Java Virtual Machine Specification" [32] and any JVM will have to adhere to these specifications. Familiarisation with this material was therefore crucial for the project. While a complete description of the JVM is beyond the scope of this dissertation, this section describes some core aspects.

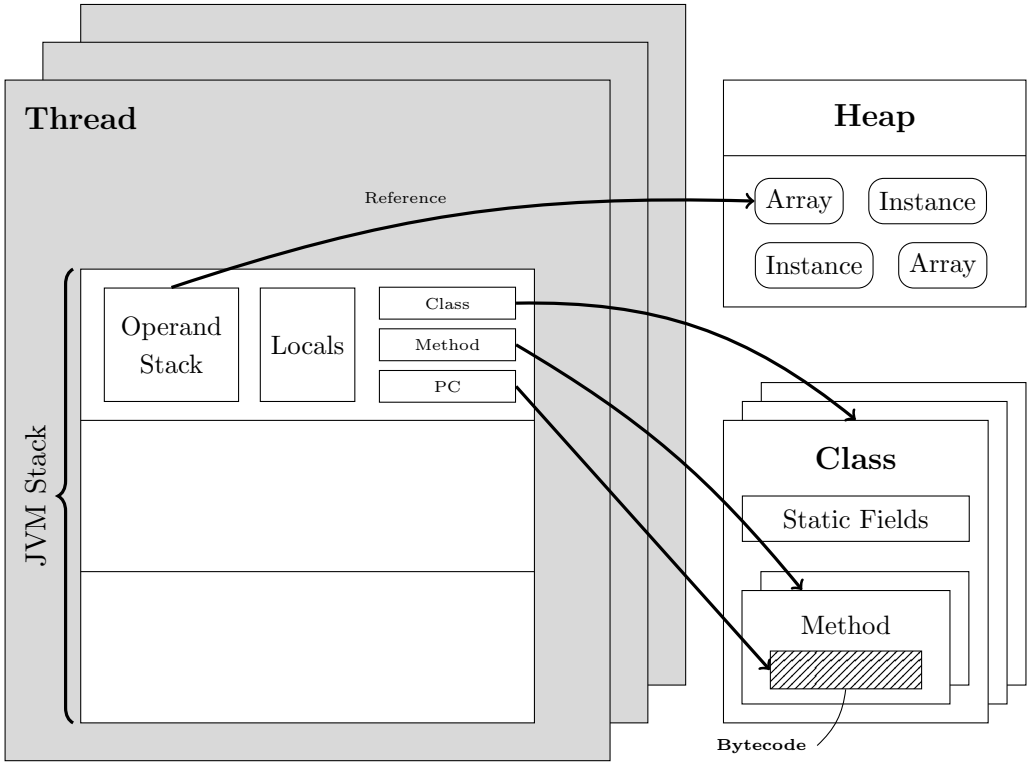

Figure 2.1: Overview of the Java Virtual Machine

2.1.1 Java Bytecode

The execution of a Java program is defined by its *Bytecode*. Each instruction reads or writes values at the top of an operand stack and results are put back onto this stack. Each method call has its own operand stack and the maximum size of this stack is pre-computed by the compiler. Stack entries are typed and Java provides basic types such as `int`, `double` or `long`.

Method calls require the allocation of a new frame on a thread-local JVM stack, where each frame contains an operand stack, a set of local variables and a program counter (PC). At each step of the computation, the instruction at the PC is executed and updates the state of the machine.

In addition to this local state, the JVM provides a global heap, which is indexed by references, a special basic type. Heap entries can hold arrays or class instances and programs can allocate entries on the heap, but not delete them (this is done by a garbage collector).

```
public static int fib(int);
  Code:
    0:   iload_0
    1:   iconst_1
    2:   if_icmpgt       7
    5:   iconst_1
    6:   ireturn
    7:   iload_0
    8:   iconst_1
    9:   isub
    10:  invokestatic    #2; //Method fib:(I)I
    13:  iload_0
    14:  iconst_2
    15:  isub
    16:  invokestatic    #2; //Method fib:(I)I
    19:  iadd
    20:  ireturn
```

Listing 2.1: A method calculating Fibonacci numbers

2.1.2 Constant pool

Program data such as code, constants and references to other classes, methods or fields is stored in a constant pool, a per-class list of typed entries. Java Bytecode instructions such as `new`, `getfield` or `invokevirtual` contain an index into the constant pool of the current class (the class that defines the current method) which is then used to, for example, resolve a class reference or obtain a string constant.

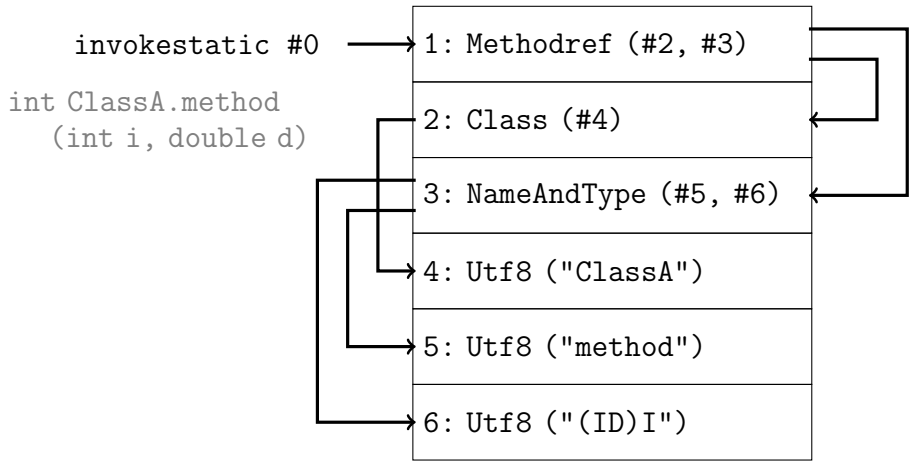

Figure 2.2: Resolving a method reference from the constant pool

2.1.3 Class files

Java programs are represented as `.class` files. Each class file belongs to a particular class (including nested or anonymous classes) and defines the class's properties, methods, fields and constant pool. Classes are dynamically loaded and linked at run-time.

2.1.4 Class library

The Java class library is a set of classes that provides a wide range of functionality, including access to certain features of the JVM, such as thread management or string handling. This is usually implemented via native method calls.

2.2 The Barrelfish operating system

Barrelfish was introduced in 2009 as a reference implementation of the Multikernel model [12]. Since then, the team has published several snapshot releases as open source. While this project was first based on the December 2009 snapshot, I changed to the March 2011 snapshot as soon as it became available.

In order to conduct the project, I had to gain knowledge about the system. Unfortunately, Barrelfish is not very well-documented. While research papers and technical reports gave a high-level overview, many details had to be extracted from the source code. This section gives an overview of these findings and material necessary to understand the project.

2.2.1 The Multikernel Model

Barrelfish runs an instance of the OS on each of the system's cores. Each of the instances contains the following core components (Figure 2.3):

CPU Driver: This is Barrelfish's equivalent to a traditional kernel. It schedules and mediates core access of user-level processes, handles interrupts and provides local inter-process communication and low-level primitives for inter-core signalling (e.g. inter-processor interrupts). The CPU driver is lightweight and abstracts away little, while hiding the underlying hardware

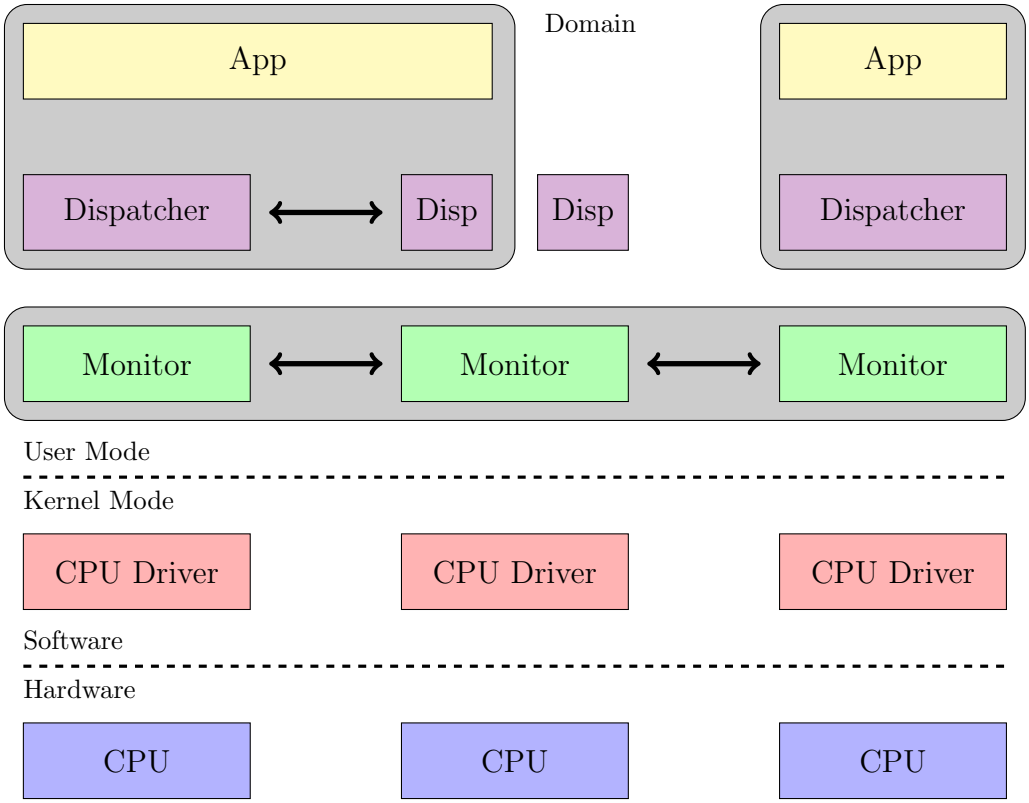

Figure 2.3: Interactions between Barrelfish's core components

from the rest of the system. This allows Barrelfish to support heterogeneous systems, since the CPU driver exposes an ABI that is (mostly) independent from the underlying architecture of the core.

Monitor: The monitor runs in user-mode and together, the monitors across all cores coordinate to provide most traditional OS functionality, such as memory management, spanning domains between cores and managing timers. Monitors communicate with each other via inter-core communication. Global OS state (such as memory mappings) is replicated between the monitors and kept consistent using agreement protocols.

Dispatchers: Each core runs one or more dispatchers. These are user-level thread schedulers that are up-called by the CPU driver to perform the scheduling for one particular process. Since processes in Barrelfish can span multiple cores, they may have multiple dispatchers associated with them, one per core on which the process is running. Together, these dispatchers form the "process domain". Dispatchers are responsible for spawning threads on the different cores of a domain, performing user-level scheduling and managing

thread synchronisation (e.g. waking up threads on remote cores). Just like monitors, they communicate via message passing and replicate per-process state across the domain's cores.

In addition to these core components, cores may run drivers (e.g. a network stack), applications (e.g. a web server) and system services (e.g. a memory server). Communication between these services works via message passing as well. In order to allow connecting to a specific service, Barrelfish provides a global nameservice.

2.2.2 Inter-core Communication

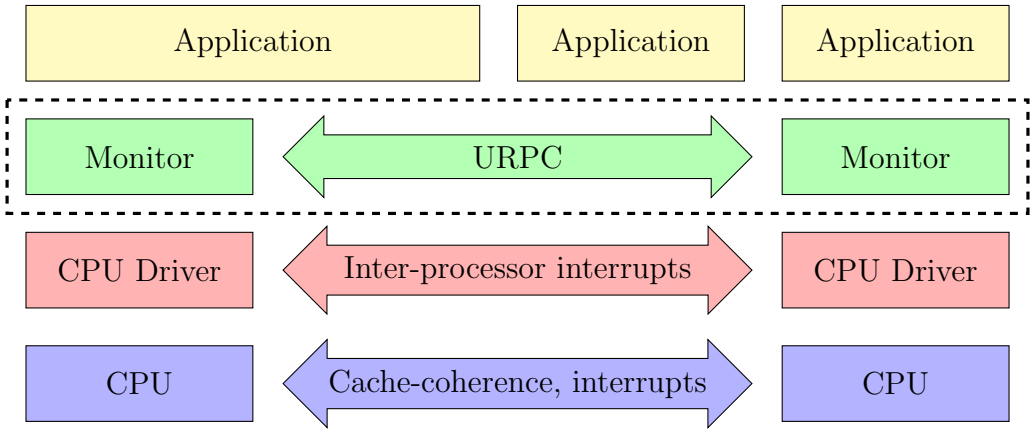

Figure 2.4: Inter-core communication mechanisms (adapted from [12])

The low-level support for inter-core communication is hardware-dependent and implemented in the CPU driver. On a system that supports message passing in hardware, Barrelfish will use those capabilities, while on a shared-memory system, it will use the cache coherence protocol. This works by using a cache line for communication between two cores, where one core is writing a message to the cache line and the other core is polling on the cache line's last word.

Based on these primitives, the system provides a user-level RPC (URPC) mechanism that does not require a system call. Together with intra-core communication, which is handled by the CPU driver, Barrelfish exposes a common message-passing interface to the user via a library. This interface provides methods to poll for messages, set up handler methods, connect to other cores and send messages to them.

```
// A name and description of the interface
interface hello "A flounder interface" {
  // A message from the client to the server
  call hello(uint32 number);

  // A response from the server to the client
  response answer(string text);
};
```

Listing 2.2: An example Flounder interface

RPC interfaces are defined using Flounder, an interface description language developed for Barrelfish. The code is compiled into source and header files that are added to the application. Listing 2.2 gives an example of a Flounder interface and Figure 2.5 demonstrates how it is used.

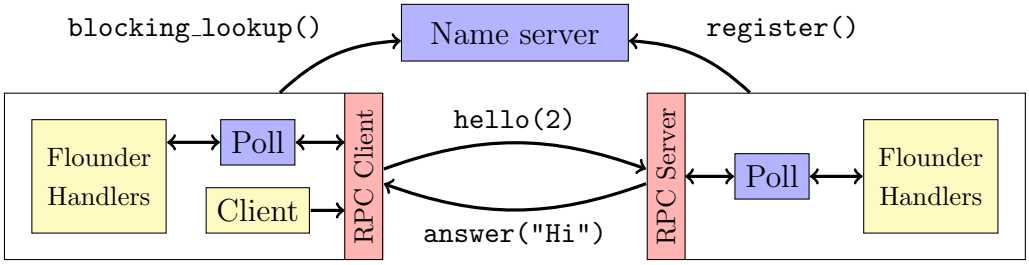

Figure 2.5: Use of a Flounder RPC interfaces for inter-core communication

2.2.3 Shared Memory Support

Spanning a domain between different cores on a system that provides shared memory support allows the threads on different cores to access a shared virtual address space. This works by replicating a hardware-independent representation of the page table across the dispatchers on different cores. From a user's perspective, nothing has to be done except for setting up the domain and running threads on remote cores.

2.2.4 Platforms and Build system

Barrelfish was originally developed for x86-64 systems, but has now been ported to x86, ARM, SCC [34] and Beehive [45]. Barrelfish uses regular ELF executables and can therefore be built using the standard x86-64 gcc compiler.

```
[ build application {
    target = "jvm",
    cFiles = [ "jvm.c", "heap.c", "linker.c", "loader.c" ],
    addCFlags = [ "-DHEAP_MANAGED", "-DSTACK_MANAGED" ],
    addLibraries = [ "timer", "bench", "msun" ],
    flounderClients = [ "jvm" ],
        flounderServers = [ "jvm" ]
} ]
```

Listing 2.3: An example Hakefile

Barrelfish uses a custom build system by the name of Hake. It was written in Haskell and generates a Makefile for Barrelfish using a set of *Hakefiles*. These files specify sources, compiler definitions, flounder interfaces and additional parameters. This makes writing applications for Barrelfish as simple as writing any C program (subject to using the Barrelfish libraries).

2.2.5 Booting Barrelfish and running applications

```
timeout 0

title   Barrelfish
root    (nd)
kernel  /x86_64/sbin/elver loglevel=4
module  /x86_64/sbin/cpu loglevel=4
module  /x86_64/sbin/init

# Domains spawned by init
module  /x86_64/sbin/mem_serv
module  /x86_64/sbin/monitor
...

# General user domains
module /x86_64/sbin/jvm core=0 jvm-node0
module /x86_64/sbin/jvm core=1 jvm-node1
module  /x86_64/sbin/serial
module  /x86_64/sbin/fish
```

Listing 2.4: A menu.lst file for booting Barrelfish

To boot Barrelfish, all its components have to be specified in a menu.lst file which is supplied to the GRUB bootloader [19]. Applications are added as modules. The reason for this is that Barrelfish does not support disk-based file systems at this point, which means that all data has to be loaded by the bootloader.

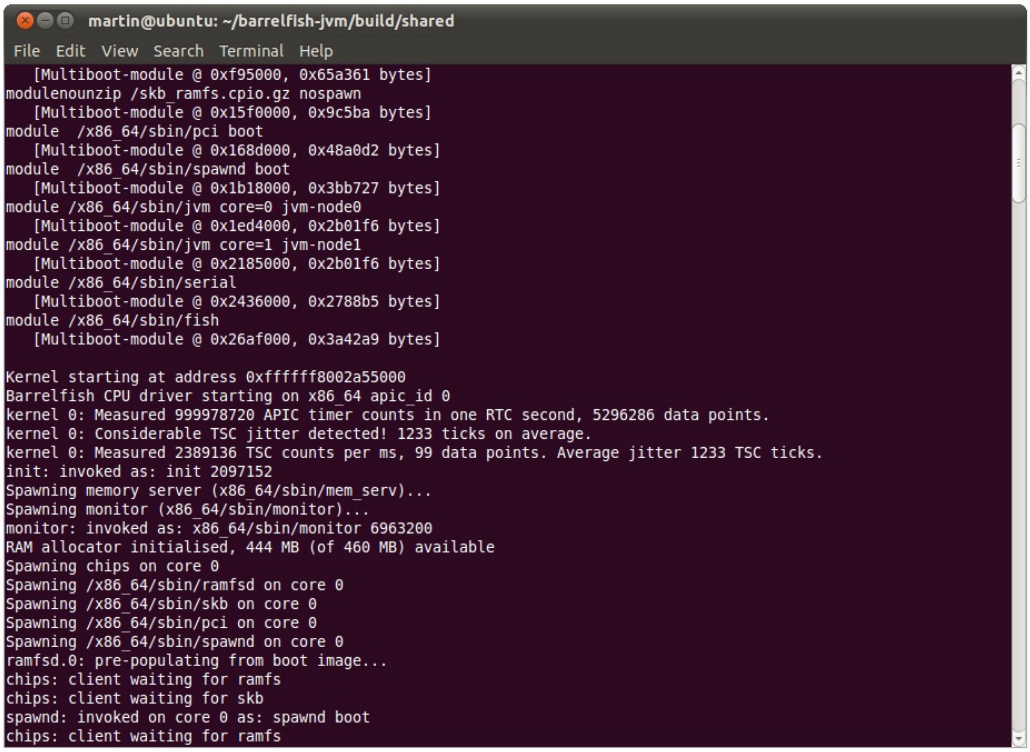

Figure 2.6: Barrelfish running within QEMU

2.3 Requirements Analysis

The implementation of a JVM is a complex and open-ended problem and it was clear from the beginning that it was impossible to implement a fully-featured JVM. To guide the implementation and evaluation of the project, it was therefore necessary to conduct a requirements analysis to limit its scope. This resulted in a set of requirements that captures the core objectives.

2.3.1 Required JVM features

While it would have been possible to select an arbitrary set of Java features as requirements, a better approach is to determine these features based on the programs the JVM will have to execute.

Since the main goal of the project is the evaluation of design decisions, the JVM will have to run programs that enable this evaluation. I therefore selected two representative benchmarks from the Java Grande Benchmark

Suite [23]. This suite is the standard in Java benchmarking and covers a wide range of Java features, making it a good choice for this purpose. I chose the following benchmarks:

- **JGFHeapSortBenchSizeA (sequential)**: A single-threaded benchmark that sorts a set of integers using heap sort. This covers most core JVM features and a subset of the classes from the Java Class Library.
- **JGFSparseMatmultBenchSizeA (parallel)**: A parallel benchmark that multiplies a compressed representation of a sparse matrix with a vector. This covers features to run multi-threaded applications.

By manual inspection, I extracted the minimum set of Java features that are required to execute these benchmarks. The full set of features is given in Appendix A, while Table 2.1 contains a brief overview. The set of features mentioned in the project proposal is a subset of this.

#	Description
J-1	Loading and linking classes (supporting inheritance)
J-2	Executing basic programs (supporting arithmetic, control transfer)
J-3	Static method calls and static field access
J-4	Creating instances (heap), field access and virtual method calls
J-5	Creating and manipulating arrays on the heap
J-6	Supporting native method calls and system features
J-7	String handling (including command line arguments)
J-8	Spawning and joining threads, synchronisation primitives
J-9	Basic classes from the class library (e.g. `java.util.HashMap`)

Table 2.1: Summary of requirements (full list in Appendix A)

2.3.2 Further requirements

In addition to supporting J-1 to J-9, the JVM will have to run on both Linux and Barrelfish. This is captured by the following requirements, once again limiting the scope of the JVM to executing a limited set of programs:

- **C-1**: The JVM correctly runs a set $S_{C\text{-}1}$ of programs on Linux. Correctness is defined as "giving the same output as the HotSpot JVM up to varying values such as time measurements, formatting of numbers, or small floating-point variations".

- **C-2**: The JVM correctly runs a set $S_{C\text{-}2}$ of programs on a single node on Barrelfish (definition of correctness as before).

Due to the size and complexity of the project, the task of comparing a shared-memory approach with a distributed approach was considered an extension, which leads to the following optional requirements:

- **E-2**: The JVM correctly runs a set $S_{E\text{-}1}$ of programs on multiple Barrelfish nodes, using a shared memory approach.
- **E-2**: The JVM correctly runs a set $S_{E\text{-}2}$ of programs on multiple Barrelfish nodes, using a distributed approach where each core runs an instance of the JVM and no state is shared between these cores.

The sets of programs that are used to capture these requirements includes the benchmarks from the previous section and a unit testing framework to facilitate testing on Linux. Table 2.2 shows how these programs relate to the different sets of programs above.

	$S_{C\text{-}1}$	$S_{C\text{-}2}$	$S_{E\text{-}1}$	$S_{E\text{-}2}$
JGFHeapSortBenchSizeA (sequential)	✓	✓	×	×
JGFSparseMatmultBenchSizeA (parallel)	✓	✓	✓	✓
Unit testing framework (j2meunit)	✓	×	×	×

Table 2.2: Overview of required programs

2.3.3 Dependencies

Based on these requirements, it was possible to determine the dependencies between them. This gave rise to a graph (Figure 2.9) of how these features depend on each other, which is helpful to guide the implementation.

2.4 Development Process

Since many details and trade-offs were unknown at the beginning of the project, I used an iterative, test-driven development process, inspired by the spiral model [18]. Features were implemented one at a time, in an order that is a topological sort of the dependency graph (Figure 2.9). For each feature, I performed the following steps (Figure 2.7):

- **Definition**: Writing an initial test to capture the requirements. This was normally a Java program that made use of the feature.
- **Design**: Analysing the requirements and developing a design that allows the new feature to fit into the existing structure, applying foresight to avoid impeding the implementation of later iterations.
- **Implementation**: Implementing the feature and refactoring existing code, if necessary.
- **Testing**: More extensive testing of the feature, including corner cases. Occasionally, performance measurements were taken to evaluate whether the performance was sufficient.

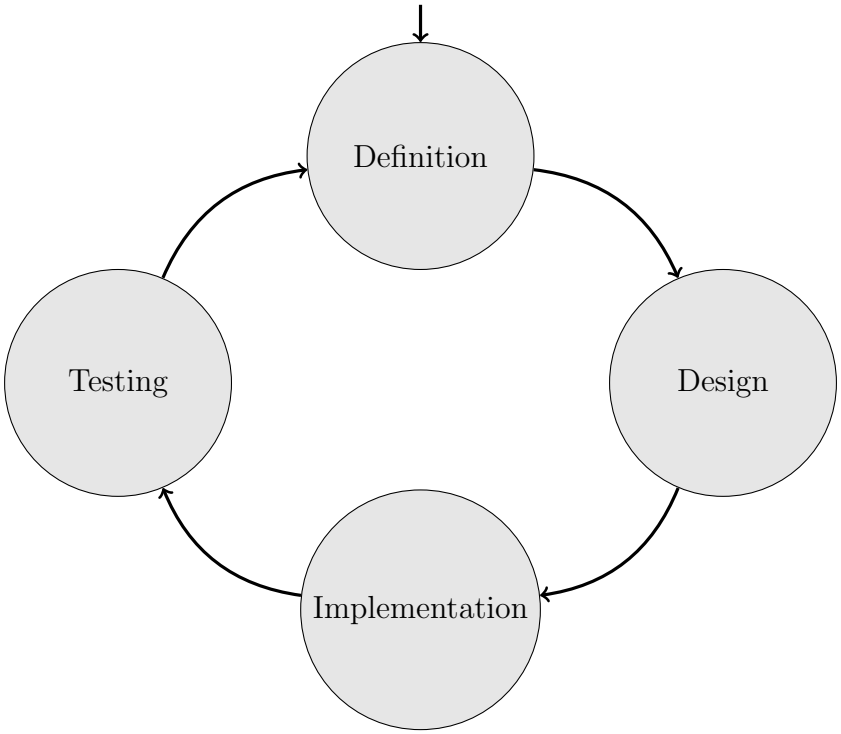

Figure 2.7: The development process

2.4.1 Testing strategy

The goal of testing was to give evidence that the developed JVM fulfils the requirements presented in the previous section. I employed different kinds of tests to ensure the correctness of the JVM:

- **Regression tests**: Define goals for the test-driven approach, usually Java programs that require an unimplemented feature.
- **Unit tests**: Confirm that the main components of the software fulfil their contract and work for a range of inputs (black-box and white-box testing).
- **Conformance tests**: Check that the JVM correctly runs a range of Java programs by manual inspection of execution traces and comparing the output to running the same program on a reference implementation (OpenJDK [6]).
- **Integration tests**: A suite of integration tests written in Java, confirming the correct execution of complete Java programs. Many of these tests can be taken from the Java Grande Benchmark suite since most benchmarks provide validation facilities.
- **Stress tests**: Run real-world benchmarks (Java Grande) of increasing size to determine how the JVM copes with high loads.

2.4.2 Adaptive approach

This project represents, to an extend, open-ended research. It was therefore necessary to adapt to intermediate results and measurements and try out different approaches. This dissertation will ignore many experimental parts that were written during the project, but will hint at them where appropriate. In the spirit of Barrelfish, the project was guided by learning about design decisions for JVMs, rather than implementing an end-user system.

2.5 Development Environment

The following development environment was used throughout the project:

- **Languages and compilers**: The JVM was written in C and Java. On Barrelfish, it makes use of the Barrelfish tool chain, including Flounder and Hake.
- **Versioning and Backup infrastructure**: I used a local git repository, regularly pushing changes to a git repository on the PWF as backup.

- **Build system**: My build system was based on Makefiles. There are three possible builds of the JVM: "Linux", "Barrelfish (Shared)" and "Barrelfish (Distributed)". Each of the builds collects all necessary files from the source directories, copies them into a separate directory structure and runs the platform's appropriate build tools (e.g. Hake for Barrelfish).
- **Running Barrelfish**: I used a local instance of QEMU during most of the development. Performance measurements were taken on the SRG's test machine `tigger` (a 48-core system).

Coding conventions were adapted from Barrelfish, using Doxygen-style comments for documentation.

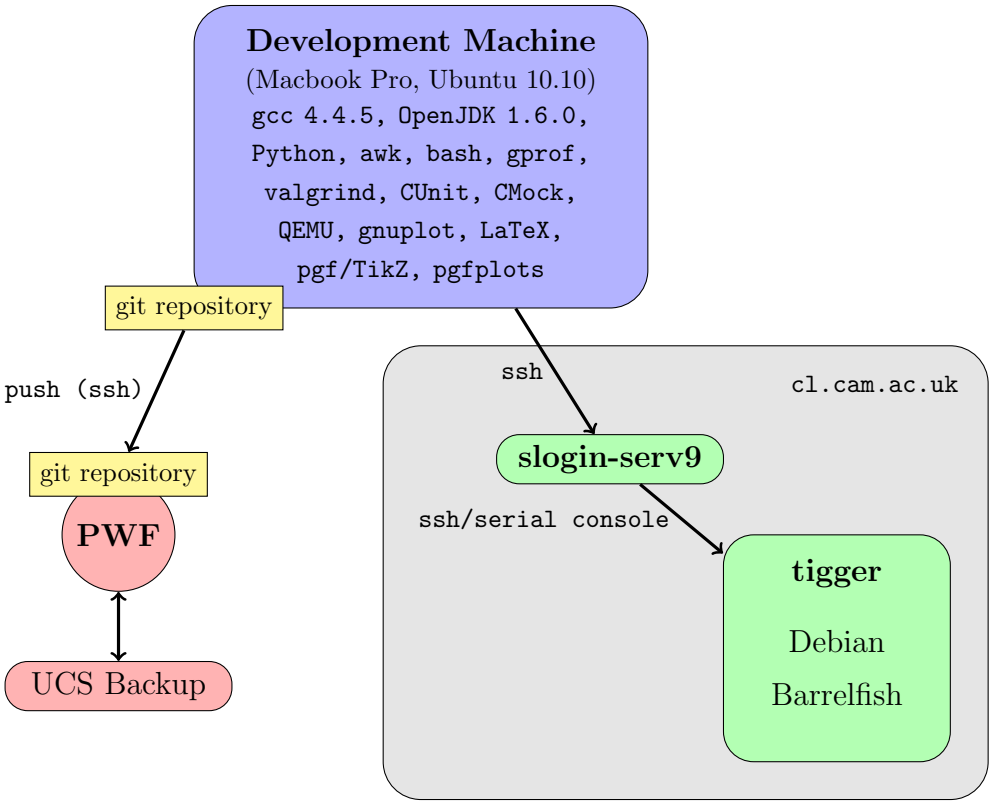

Figure 2.8: Development environment

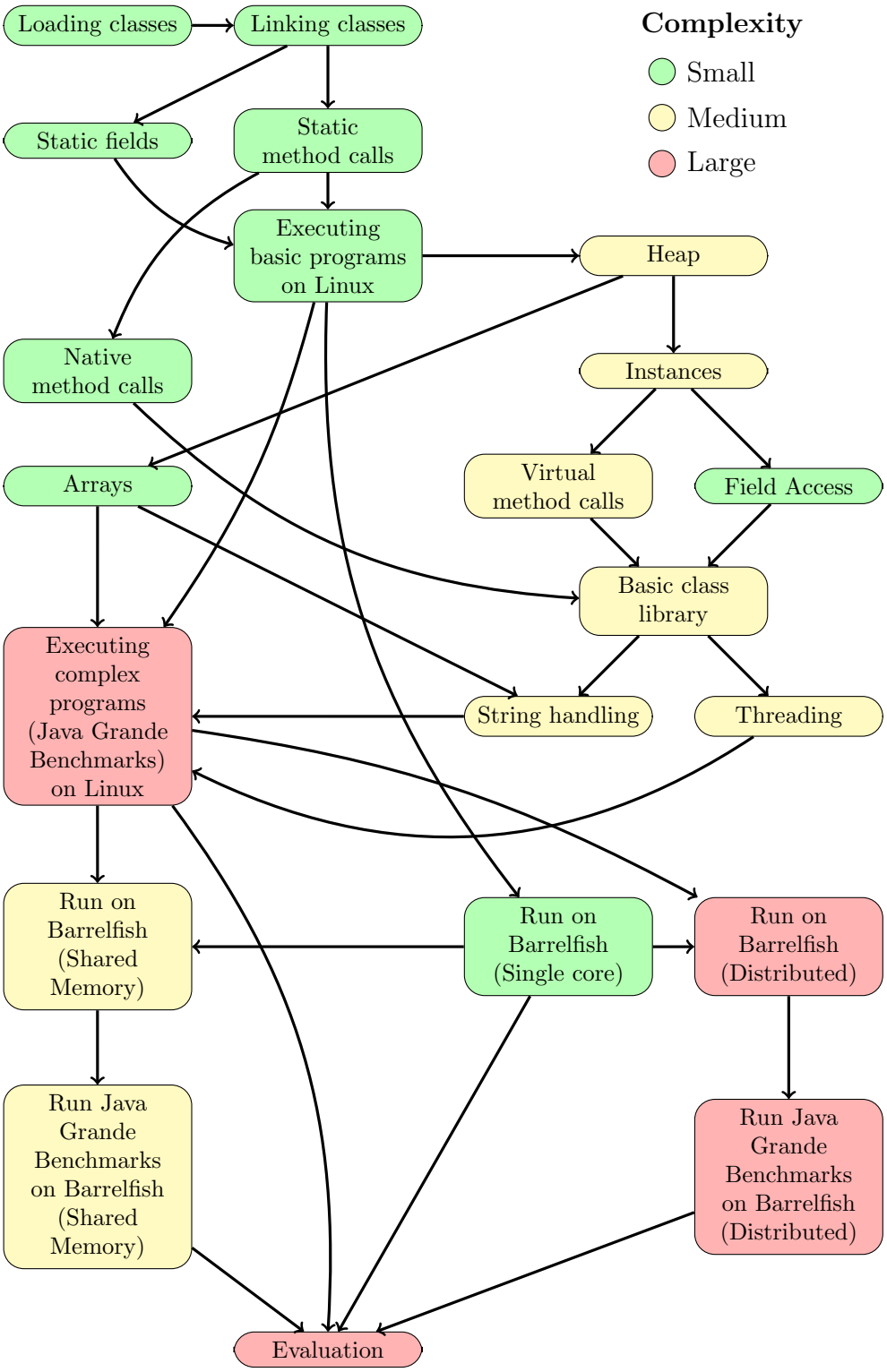

Figure 2.9: Dependency Graph

3 Implementation

This chapter describes the implementation details of the Barrelfish JVM and explains major design decisions, performance trade-offs and optimisations.

3.1 Overview

The requirements introduced in the previous chapter represent a minimum set of Java features that had to be implemented. The Barrelfish JVM supports more than this, including the main extensions. I have also filled some additional gaps to allow the execution of a wider range of programs. The result is a Java Bytecode interpreter that supports 198 of the 201 Java Bytecode instructions (the missing instructions are `wide`, `goto_w` and `jsr_w`).

In addition to the features from the project proposal, the JVM supports inheritance, strings, all basic types (including `long`, `float` and `double`) and arrays (both one-dimensional and multi-dimensional). The features that are unsupported include exception handling (`athrow` will only produce a stack trace and exit), garbage collection, most of the class library and validation (which includes type checking at run-time). In accordance with the project proposal, linking and class loading are static rather than dynamic.

The JVM runs on Linux and Barrelfish, both on single and multiple cores.

For Barrelfish, both the shared memory approach and the distributed approach have been implemented. The Barrelfish JVM runs successfully within QEMU [15] and on real hardware.

3.2 Fundamental Decisions

A fundamental decision for the project was whether to implement a new JVM or bring up an existing system on Barrelfish (e.g. the Jikes RVM [16]). After discussing these options with my supervisors, we decided on the first option, since Barrelfish's support for OS features required by Jikes (such as sophisticated memory management) is incomplete or experimental. The risk associated with this was deemed too high for the nature of this project.

A second major decision was whether to build a Java Bytecode interpreter or a JIT compiler. As the focus of the project was on design rather than performance, the benefits of a JIT compiler would have been limited, while increasing the complexity of the project and distracting from the task of making the JVM distributed on Barrelfish. Furthermore, JIT would be more difficult to port to different architectures, which is a disadvantage given that one of the core ideas of Barrelfish and the JVM is to run on heterogeneous systems. I therefore decided to implement an interpreter instead.

3.3 Structure

Figure 3.1 gives an informal, high-level overview of the JVM on Linux. This design does not include extensions to run the JVM on Barrelfish and only shows the main components. In the following sections, I will present the implementation of each part and make the design more precise.

The core component of the JVM is the *interpreter*, which executes the main loop and manages a per-thread JVM stack. It accesses the global *heap* and uses separate modules for *class* and *array* handling. The interpreter requests classes and class data from the *linker*, which populates and manages the *method area* (the part of the JVM that stores class information and code). The linker uses a *class loader* to read class data from disk or memory. A module for *native method invocations* allows the interpreter to call methods defined in C (using Java's `native` keyword). These methods provide access to System functionality such as the system clock or the console. Features related

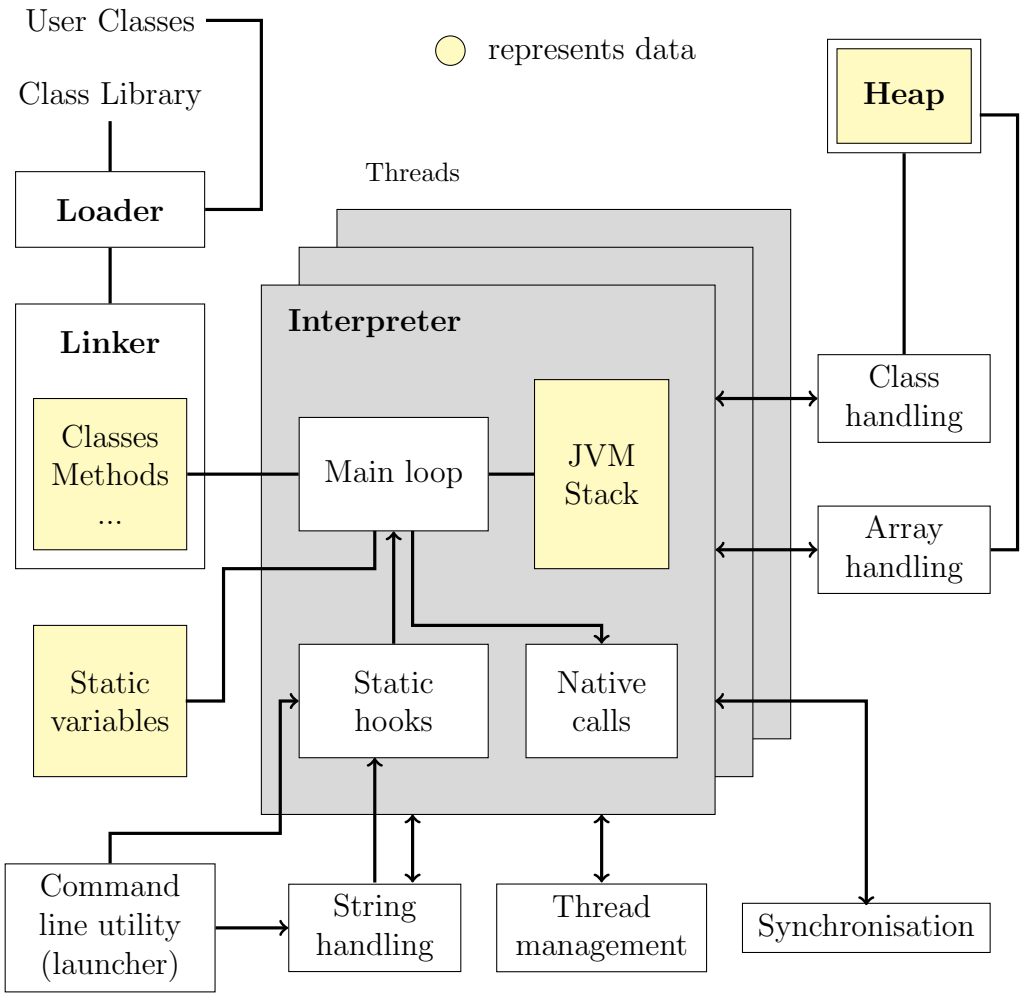

Figure 3.1: High-level design of the JVM (compare to Figure 2.1)

to *threading* and *synchronisation* are implemented in a separate module, which differs between platforms.

Most of the project was written in C, since this was the only language supported by Barrelfish when the project was started. As this prevented an object-oriented software design, a module-based approach was chosen to decouple the different components, provide a clear structure and facilitate testing. Some components have different implementations (e.g. for different platforms), which can be selected using pre-processor macros.

3.4 Implementation Details

This section outlines the implementation of the system's main components. The testing strategy from Section 2.4.1 was kept in mind throughout the project and all modules were designed with cohesion in mind, in order to facilitate unit testing and mocking. A set of regression tests (Java programs) was used to test each new feature. Testing is jointly covered in Section 4.1.

3.4.1 Class Loader

A difference between the Barrelfish JVM and real-world JVMs is the fact that class loading is static rather than dynamic. Most systems load classes dynamically from a range of different sources (e.g. the file system or network) using a hierarchy of class loaders implemented in Java itself. A dynamic linker will then request unknown classes from these class loaders. In the Barrelfish JVM, all classes are available on start-up (due to the lack of a file system). The class loader approach would therefore be unnecessarily complex and hence, a static approach has been used, where all classes are loaded during the initialisation phase of the JVM. However, this could easily be extended to the approach described above.

The loader reads class data from memory and translates it into an internal representation shown in Figure 3.2. It uses an approach similar to recursive decent parsing, using a set of mutually recursive functions to read data of a certain type and emit the appropriate structure. The loader skips most attributes, except for a set of special annotations supported by the JVM and the `Code` attribute, which contains the bytecode associated with a method.

While the class loader does not modify any data, it provides helper functions to extract data from the constant pool, such as methods or fields of a certain name and type. Appendix C presents a running example for the class loader.

3.4.2 Linker

The linker transforms the output of the class loader into the form that is used at run-time. It replaces references to classes, fields and methods by pointers to their actual representations, thereby translating the constant pool into a *run-time constant pool* (Figure 3.3). The linker also determines the memory layout of class instances. An example is given in Appendix C.

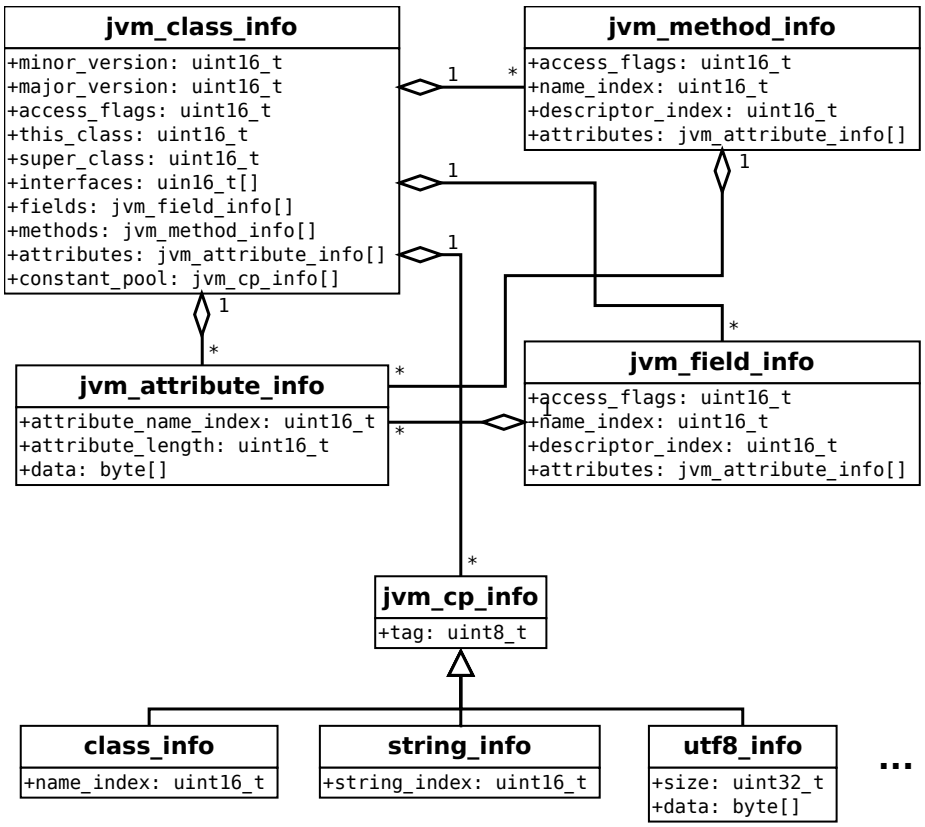

Figure 3.2: Class loader representation of the data (UML class diagram)

Classes are processed lazily. When a class is requested by the interpreter, the linker checks whether this class has already been linked and otherwise performs the following steps:

- **Transform data**: Information such as names, access flags or type descriptors are copied from the class loader representation. All textual descriptors are extracted from the constant pool and directly stored as strings. The linker also performs basic processing such as determining a method's number of arguments by parsing its type descriptor.

- **Resolve references**: The linker iterates over the constant pool and replaces reference entries such as `Class`, `MethodRef` and `FieldRef` by pointers to the referred objects. If a required class has not been linked before, this is done recursively. The linker also has to consider inheritance, recursively looking up fields and methods in the parent if they cannot be found in the class itself.

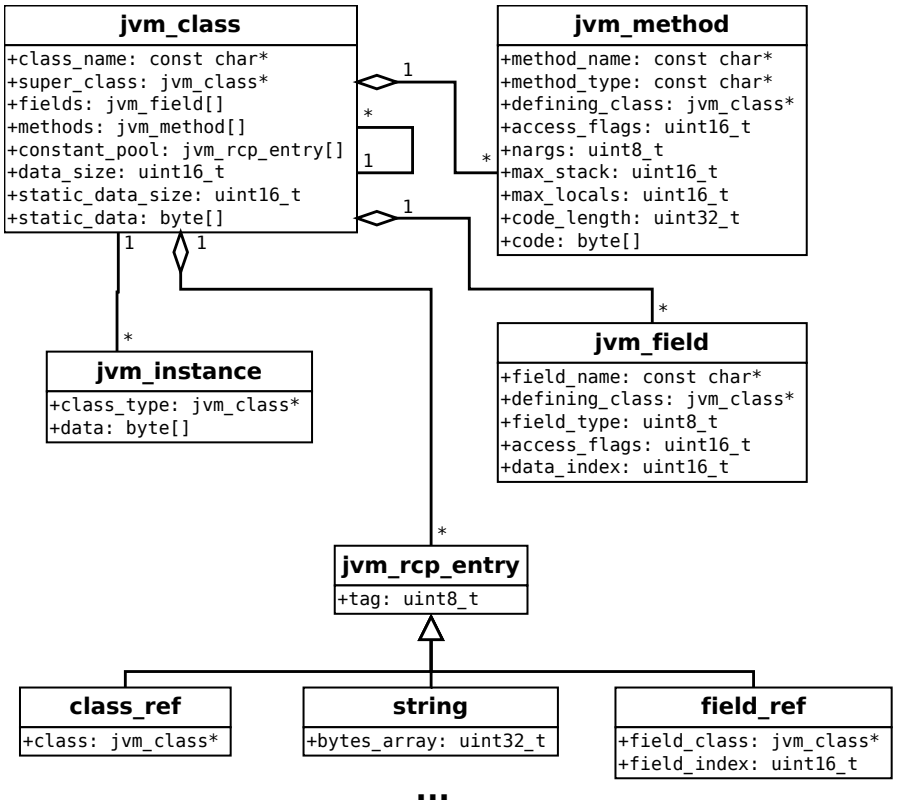

Figure 3.3: The method area (UML class diagram)

- **Calculate field offsets**: Once all field references have been resolved, the memory layout of the class instances is computed. The parent's fields are laid out first in memory, followed by the fields of the class itself (Figure 3.4). This allows a subclass to be treated as any of its ancestors. The calculation of static fields is simpler, since no inheritance has to be taken into account.

- **Run static initialiser**: Java classes may provide a static initializer[1] that is executed when the class is first loaded. These initializers are executed by invoking the interpreter on a method called "`<clinit>`".

Since multiple threads may access the linker concurrently, it is protected by a coarse-grained lock. For performance reasons, the evaluated version of the Barrelfish JVM links all classes on start-up, to eliminate linking time from the measurements.

[1] These blocks are defined by `static {...}`.

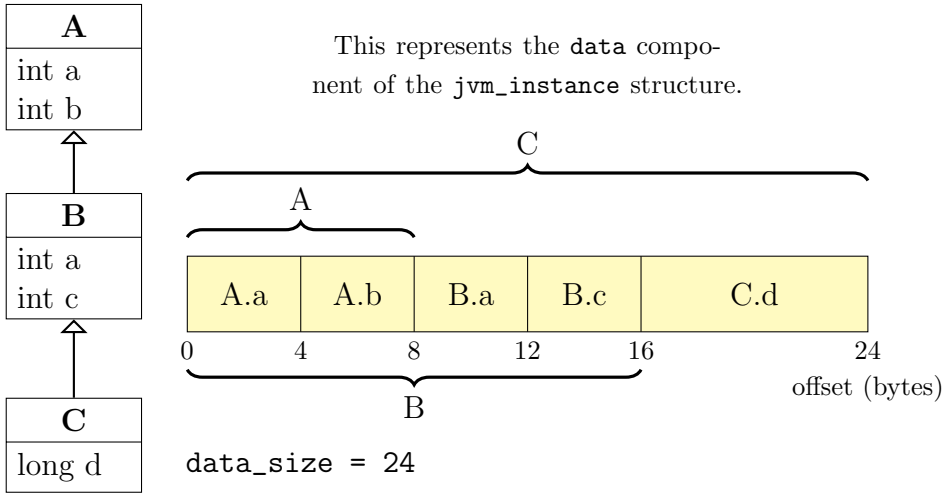

Figure 3.4: Memory Layout of a class instance

```
calculate_field_offsets(class):
  offset := 0;
  if (class->parent)
    offset := calculate_field_offsets(class->parent);

  for each non-static field f in class->fields
    f->offset := offset;
    offset += sizeof(f); // 4 for int, 8 for long, etc.

  return offset; // = data_size
```

Listing 3.1: Algorithm for calculating field offsets

3.4.3 Interpreter

The interpreter is the core of the JVM and performs most of the execution. It is based around a main loop performing the following steps:

- fetching the opcode at the program counter (PC)
- fetching any operands (instructions are variable-length)
- executing the opcode and updating the state

This is implemented as a large `switch` statement (almost 2,000 loc). Since much of it is uninteresting, this section will focus on some specific aspects.

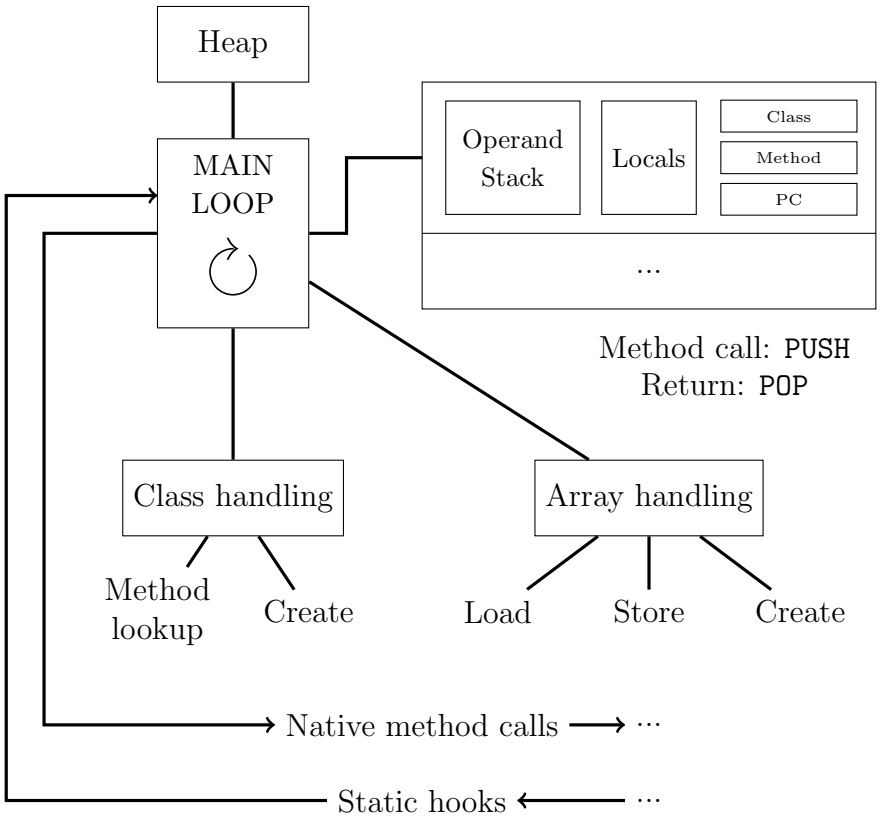

Figure 3.5: Structure of the interpreter

State

The interpreter manages a per-thread JVM stack where each stack frame contains an operand stack, local variables, the program counter and pointers to the current class and method (Section 2.1). All `long` and `double` values take up two stack entries or local variables and have to be treated accordingly.

Simple instructions

Many instructions (e.g. `iadd`, `ifeq`, `i2f`) only manipulate the operand stack and local variables. In this case, the interpreter simply executes the operation, e.g. takes two integers off the stack, adds them and puts the result back onto the stack. Some instructions also require updating the program counter or manipulating the JVM stack.

Method calls

At a method call, the interpreter looks up the method in the run-time constant pool and generates a new stack frame. It then copies the arguments from the caller's operand stack to the callee's local variables, sets the callee's PC to the start of the method's bytecode and pushes the frame onto the JVM stack.

If the callee is an instance method, the method's first parameter is a reference to the corresponding object. For virtual method calls (`invokevirtual`, `invokeinterface`), the interpeter has to look up the correct method using the type of the instance. The algorithm is given in the JVM specification [32] and first considers the class itself before recursively searching its ancestors. This step is skipped for static methods (`invokestatic`) and constructors or private methods (`invokespecial`).

When the callee returns (i.e. reaches an instruction such as `ireturn`), the top-most value from the stack is copied onto the caller's stack, the callee's stack frame is removed and execution continues at the PC of the caller.

Native method calls

Java supports native method calls via the `native` keyword. The JVM handles them by using a static dispatcher that looks up the methods class, name and signature, takes the arguments from the stack and calls the corresponding native implementation. This is used to implement many features of the JVM, such as threading (`java.lang.Thread`) or fast copying of arrays (`java.lang.System.arraycopy()`).

Static hooks

While native methods allow Java code to call into C code, the Barrelfish JVM also allows C code to run static Java methods, which I call a static hook. Amongst other uses, this is important to generate Java objects such as strings: While the JVM provides facilities to generate Java arrays within C, objects can only be generated in Java, since it has to be ensured that the constructor is executed correctly and that the memory layout remains correct in the presence of changes.

3.4.4 Heap

Java's heap manages a global memory store that maps 32-bit reference values to run-time representations of class instances and arrays. The heap also has to perform bookkeeping in order to allow for tasks such as garbage collection or object relocation in a distributed JVM. Abstractly, it exposes the interface presented in Listing 3.2.

```
ref heap_put_instance(instance)
instance heap_get_instance(reference)

reference heap_put_array(array)
array heap_get_array(reference)

// Distributed JVM only:
heap_set(reference, pointer)
heap_unset(reference)

// Managed heap only:
pointer heap_alloc_instance()
pointer heap_alloc_array()
```

Listing 3.2: Heap interface

The heap is a central data structure that is accessed by many instructions and multiple threads may access it concurrently. As shown in the evaluation, this can account for a large proportion of the JVM's run-time and should therefore, by Amdahl's Law, be subject to optimisation. In fact, many JVMs use sophisticated heap implementations to improve performance [5].

In order to investigate how this influences my results, I compared different implementations:

- **Naive Hashmap**: This is a simple hash-map implementation that uses open-addressing with linear probing [25, p. 239]. It maps references to pointers and doubles its size once it is 75% full. All operations are protected by a table-wide lock. While this approach is very inefficient, it is simple and makes good use of memory.
- **Array**: This implementation uses an array of pointers. Reference values are indices into the array and the size is doubled as soon as it is full. All operations are non-blocking, including resizing. While this is not a realistic implementation (no memory is ever freed), it is used to emulate an efficient heap implementations that maps references to pointers using a non-blocking data structure.

- **Managed heap**: This implementation emulates a heap where references are pointers into memory. The approach consists of pre-allocating a large block of memory and storing arrays and objects within this memory, rather than pointers to them. As with the previous approach, my implementation is not realistic, since it does not re-use memory.

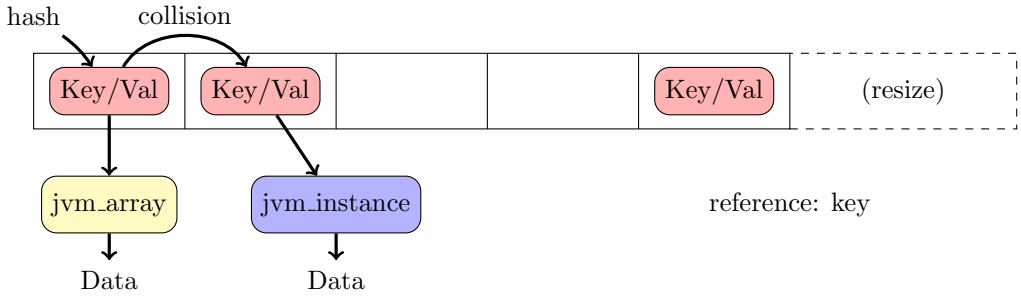

Figure 3.6: Naive hashmap implementation

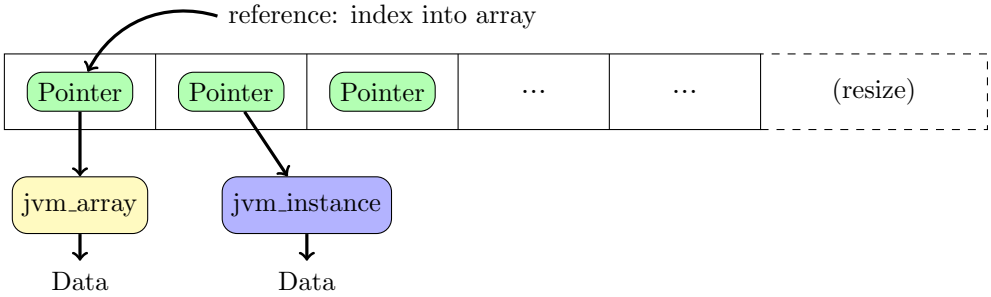

Figure 3.7: Array implementation

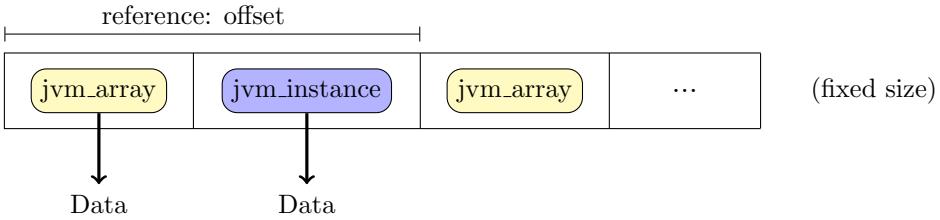

Figure 3.8: Managed heap implementation

3.4.5 Class library

The Barrelfish JVM implements a subset of classes and methods from the Java Class Library, such as `java.lang.System`, `java.util.Hashmap` and

`java.util.Vector`. The full list can be found in Appendix G. The implementations adhere to the specifications given by the official Java documentation [4] and often call native methods to implement functionality. Barrelfish also implements a set of additional classes in the `org.barrelfish.jvm` namespace.

3.4.6 String handling

Java represents strings by the `java.lang.String` class. My implementation of this class uses an internal `byte[]` array that holds the string's data, one byte per character (UTF8 characters over `0xFF` are not supported). Multiple strings may refer to the same byte array, so that calls to e.g. `substring()` do not copy any character data but create a new String that refers to the same array with a different offset and length.

String constants (which can be loaded from the constant pool using the `ldc` instruction) are handled by passing the constant pool entry's byte array to `org.barrelfish.jvm.StringManager.fromStringConstant()` using a static hook (Section 3.4.3). The method then creates a new `String` object and returns the reference to the interpreter.

I also support the `java.lang.StringBuilder` class, which is used to execute commands such as `System.println("Integer i (" + i + ")")`. It makes use of `java.lang.System.arraycopy()` to quickly copy data between strings. Conversion of numbers is handled by native calls to `sprintf` and `scanf` in `org.barrelfish.jvm.ValueConverter`.

Output is handled by `org.barrelfish.jvm.ConsoleOutputStream`, which is used by `System.out` (an instance of `java.io.PrintStream`) to write characters to the console.

3.4.7 Command line arguments

Command line arguments are handled similarly to string constants: A static hook is used to convert each argument to a `String` object and the references are stored in an array that is then passed to the `main(String[] args)` method.

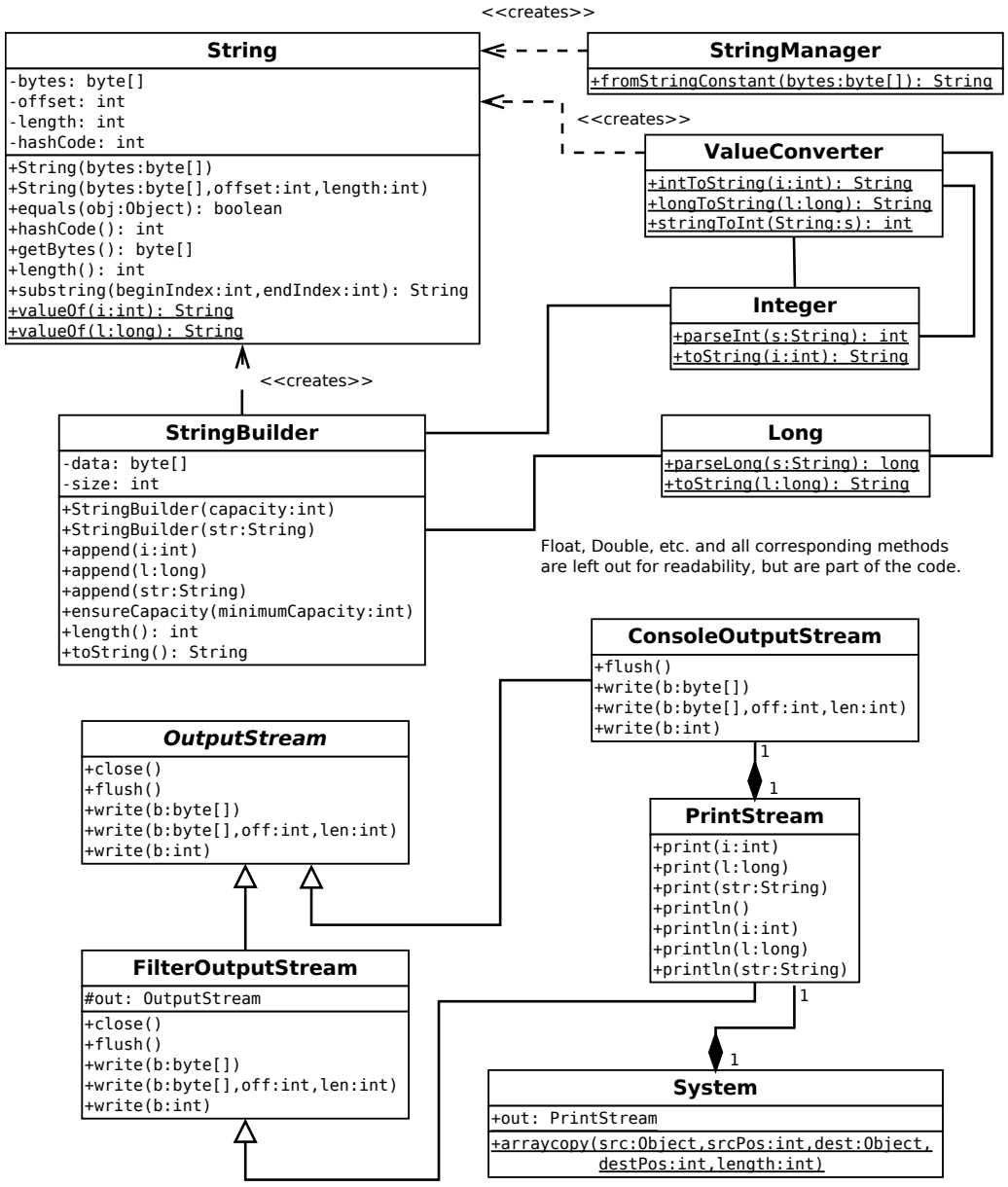

Figure 3.9: String handling in the Barrelfish JVM

3.4.8 Threads and synchronization

On Linux, the JVM implements `java.lang.Thread` and maps all functions to the corresponding pthread primitives (Table 3.1). Each thread invokes the interpreter, which creates a new JVM stack and executes the `run()` method

of the `Thread` object that called it.

Barrelfish JVM	pthreads
`java.lang.Thread.start()`	`pthread_create()`
`java.lang.Thread.join()`	`pthread_join()`
`monitorenter`	`pthread_mutex_lock`
`monitorexit`	`pthread_mutex_unlock()`

Table 3.1: Mappings to pthread functions

For the synchronization instructions `monitorenter` and `monitorexit` (which correspond to `synchronized(obj) {...}` blocks in Java), the JVM maintains a map from heap references to pthread mutexes which are locked on `monitorenter` and released on `monitorexit`.

3.4.9 Configuration

The Barrelfish JVM includes a configuration file which contains information about the types of threads to use, the available number of processor cores and the amount of heap/stack memory to use. The configuration is accessed via native methods in `org.barrelfish.jvm.Configuration`.

3.5 Running the JVM on Barrelfish

Modifying the JVM to support execution on Barrelfish was facilitated by the fact that Barrelfish supports both `libc` and a POSIX-like threading model. This meant that only minor modifications were required:

Modifying the code: The libraries used by Barelfish are slightly different from the ones used on Linux. This caused minor changes in order to compile the code for Barrelfish, such as using a different threading API and including additional header files. See Appendix E for a code sample.

Providing a service: To run the JVM on Barrelfish, it had to be turned into a service. The JVM executable was extended to set up a JVM service and register it with the nameserver, implementing the basic flounder interface shown in Listing 3.3. The server is launched on startup by an entry in the menu.lst file. When called from the Barrelfish shell (fish), the JVM creates a connection to a specified server, requesting the execution of a certain class with certain command line arguments (Figure 3.10).

```
interface jvm "Barrelfish JVM" {
  call execute(string class_name, string args);
}
```

Listing 3.3: Basic flounder interface for the single-core JVM

Loading classes into the JVM: The Barrelfish version I have been working with for most of the project did not support a file system. This made it necessary to find a different way to load class files in Barrelfish. My solution consists of a Python script (`packager.py`) that takes a set of class files and writes them as byte arrays into a C source file. This file is then compiled and linked into the JVM executable, allowing the JVM to access the class data.

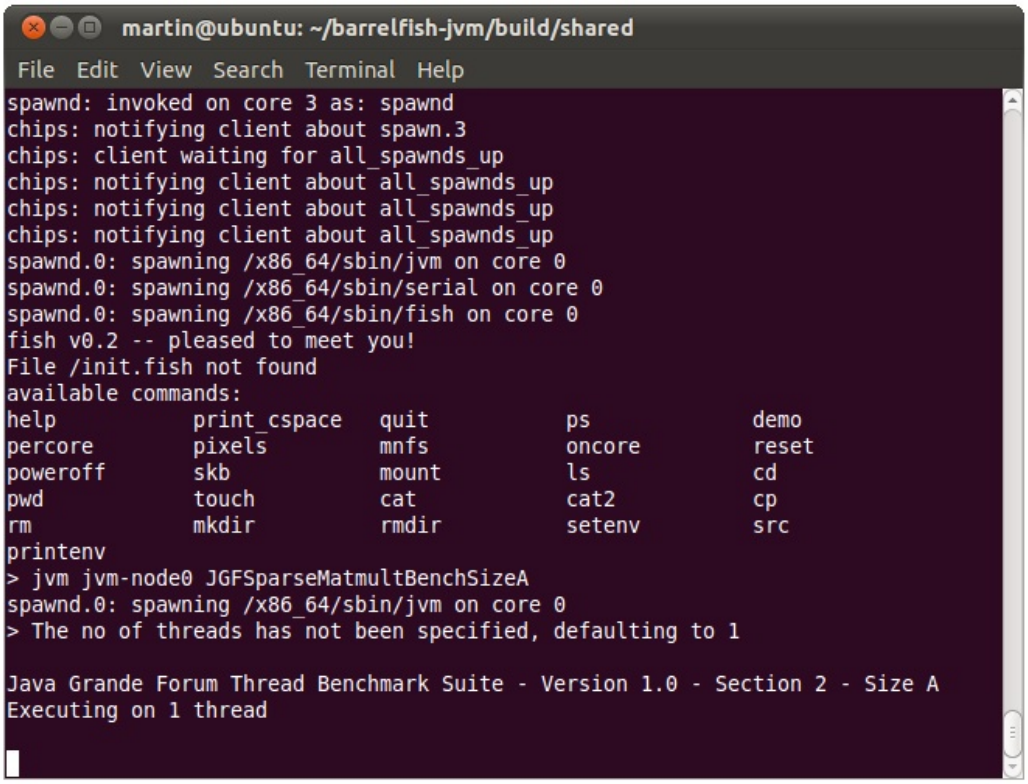

Figure 3.10: The JVM running on Barrelfish

After completing these tasks, it was possible to run the Barrelfish JVM on a single core on Barrelfish and execute Java programs. This provided the foundation for investigating design decisions in making the JVM run across multiple cores, the main extension of the project.

3.6 The Shared-Memory JVM

The first approach to support execution across multiple cores on Barrelfish uses shared memory, similar to the approach on a traditional operating system. As explained in Section 2.2.3, Barrelfish supports this by spanning a domain across multiple dispatchers.

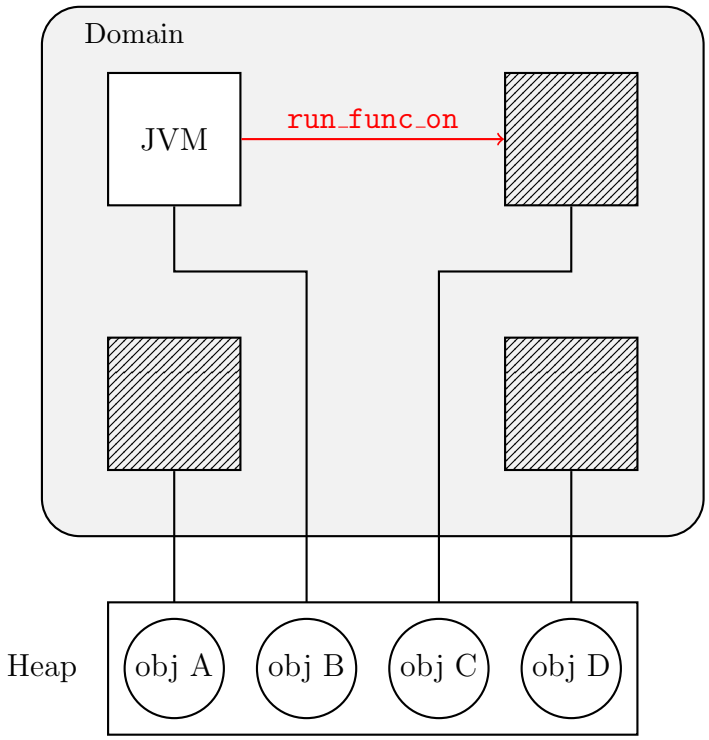

Figure 3.11: Overview of the shared approach

The JVM is first launched on a single core. It then spans a domain to different cores, using a Barrelfish API that enables it to spawn a new dispatcher on a different core and set up a connection between the two. Once this has been done, the JVM can create threads on the remote core, similar to creating local pthreads (Figure 3.11).

One of the main differences to Linux is that this solution requires the JVM to explicitly choose which core to run a thread on. I therefore implemented a `DomainThread` class which implements `java.lang.Runnable` and takes the core to run on as a parameter. The `java.lang.Thread` class then creates instances of `DomainThread` and assigns them, on creation, to different cores in a round-robin manner.

Another difference to Linux is the use of spinlocks as synchronisation primitives, since mutexes did not work across multiple cores in the earlier version of Barrelfish. While I provided my own, experimental implementation of this feature, I eventually decided to use spinlocks as they gave significantly better performance.

3.7 The Distributed JVM

The second approach for running the Barrelfish JVM on multiple cores is significantly more complex. It avoids the need for shared memory by running an instance of the JVM on every core and communicating solely via message passing (Figure 3.12).

I use an approach inspired by the dJVM [51] project. Each object has an associated *home node* where it resides. When a core performs an operation on an object, it sends a message to the object's home node, which executes the operation and returns an acknowledgement once it is finished.

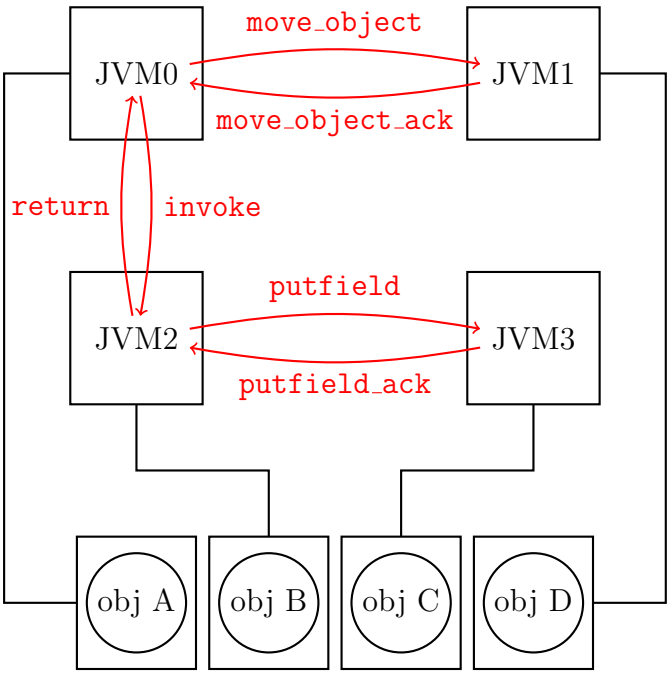

Figure 3.12: Overview of the distributed approach

To implement this approach, the JVM server was extended to support additional messages and functionality. Multiple server instances are launched on startup, each on a different core and with a unique identifier (e.g. `jvm-node0`). Each of these servers provides and manages its own set of components, including a loader, linker and heap. The policy for choosing home nodes is simple: An object's home node is the node that executed the `new` instruction that created it.

3.7.1 Inter-core communication

All JVM nodes are completely independent. In order to communicate with each other, they have to set up point-to-point connections between cores. These connections (or *bindings*, as they are called in Barrelfish) are created on demand and have an associated shared buffer to transfer bulk data (in a message-passing system, this would correspond to a message buffer). Since bindings are not thread-safe, they have to be protected by locks.

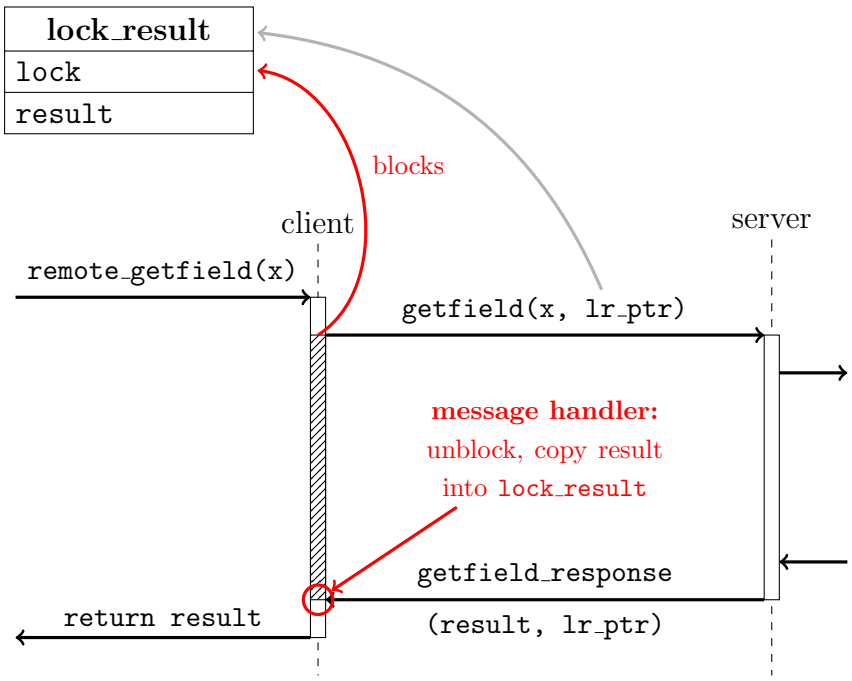

Figure 3.13: Typical communication between two nodes

The bindings are managed by a client module which keeps track of existing connections and sends asynchronous messages to other JVM nodes. A server module implements handlers for these messages, which may send replies back

to the client. This means that each JVM node acts as both server and client. Each binding only covers one direction, i.e. bi-directional communication requires two connections.

Each node runs a message-handler thread that never blocks. This thread handles incoming messages and spawns new threads as necessary (e.g. at a method call). All other threads may block at any time. For this purpose, they have a semaphore associated with them that blocks while e.g. waiting for a reply from another node. The reply handler then unblocks the thread and potentially stores a return value in a structure called a *lock-result* (consisting of the mentioned semaphore and a 64-bit value). To identify the thread that needs to be unblocked, a pointer to the lock-result is sent with each message and returned with the reply (Figure 3.13).

3.7.2 Object relocation

Moving objects between cores is essential to the distributed approach. While my JVM only relocates objects during the creation of a new thread, it could be extended to relocate objects during the program's execution, similar to O^2 scheduling [22].

Relocation is initiated by an object's home node. By default, only the object itself is moved. However, it is possible to annotate fields with a custom `@Sticky` annotation which instructs the JVM to recursively relocate any objects and arrays stored at that field as well.

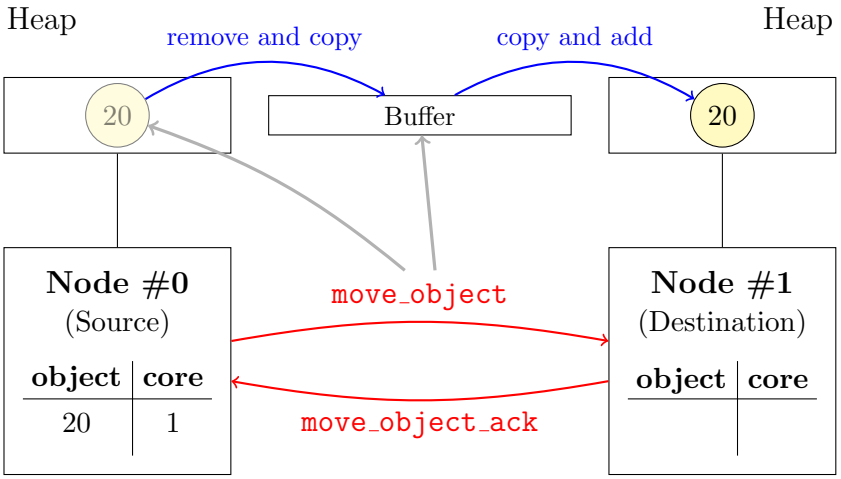

Figure 3.14: Object relocation

For each relocated object or array, the JVM sends a `move_object` message to the destination node, storing the data in the connections buffer. The destination node extracts the data, adds it to its local heap and replies with an acknowledgement (Figure 3.14).

It is important to note that heap references do not change when their entry is moved to a different node. This gives rise to the problem of keeping references unique across multiple nodes. I solved this problem by setting the first 8 bit of each reference to the id of the node that issued it. This ensures global uniqueness, as each core can provide uniqueness for its own references.

Once the transfer has finished, the source node removes the objects from its local heap and adds an entry to a *core table* containing a mapping from references to core ids. This table represents the core's knowledge about the locations of different heap entries.

3.7.3 Object lookup

When a node tries to access an object whose location is unknown to it, the node needs to perform a lookup to determine the object's home node. There are two basic approaches to this problem: a central directory of all object-location mappings or a broadcasting approach similar to the ARP protocol in computer networking [31, pp. 497ff.]. I decided on the latter, to keep the JVM decentralised and avoid bottlenecks.

Whenever the interpreter tries to access an object, it first checks the local heap. If the object cannot be found on the heap, it tries to look it up in the core table. If the core table does not contain the object either, the node broadcasts an `obj_request_bcast` message to all other nodes and blocks the current thread. When a core receives such a request, it checks its local heap for the object in question and, if the object can be found, replies to the sender. Upon receipt of the reply, the core unblocks the thread and enters the new object-location mapping into the core table (Figure 3.15).

3.7.4 Remote operations and method invocations

Remote operations are `getfield`, `putfield`, `aload` and `astore` instructions on an object or array located at a different node. They follow the scheme outlined in Section 3.7.1. Once the object's location has been determined (using the core-table and the lookup mechanism from Section 3.7.3), the JVM

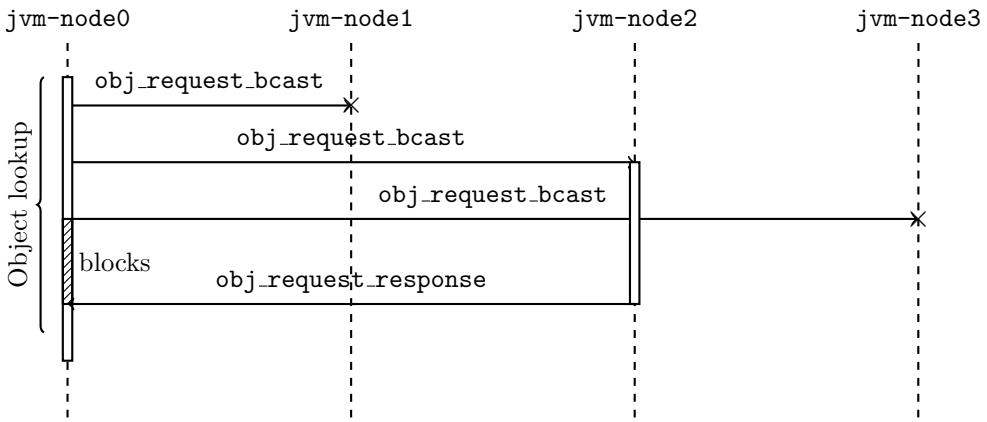

Figure 3.15: Object lookup

sends a message to the remote core (Table 3.2) and blocks the current thread. When the remote core receives the message, it performs the operation and sends a reply back to the client, which then unblocks the thread and passes on the result (Figure 3.16).

Method invocations to remote objects are similar to this. However, the remote core has to spawn a new thread to execute the method, since method invocations may block and therefore cannot be executed in the message-handler thread. Method parameters are transmitted using the connection buffer and return values are sent in an appropriate reply message, depending on the type of the return value.

Operation	Message	Reply
method call	`invoke_noparams`	`invoke_return`
	`invoke_allparams`	`invoke_return32`
		`invoke_return64`
getfield	`getfield`	`getfield_response`
putfield	`putfield`	`putfield_ack`
aload	`aload`	`aload_response`
astore	`astore`	`astore_ack`

Table 3.2: Messages for remote operations

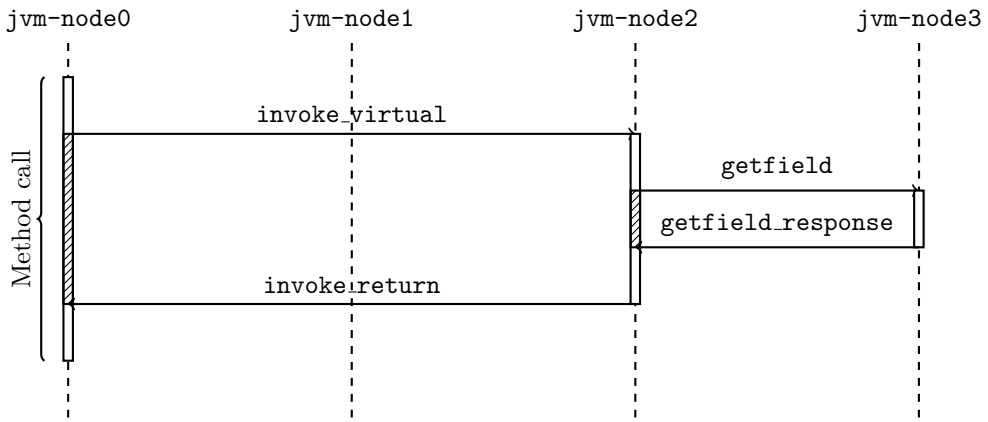

Figure 3.16: Remote operations

3.7.5 Static method calls and fields

Static methods and fields are special cases since they are not associated with any particular instance. While it would be possible to handle them like objects, assigning a home node to each class, I chose a simpler approach and stored all static data on core 0. Static method calls and `getstatic`/`putstatic` requests are handled by the same messages as their non-static counter-parts, with the object parameter set to 0.

However, it is not always desirable to execute static methods on a remote core, since it results in a significant overhead for stateless classes such as `java.lang.Math`. I therefore introduced a `@CoreLocal` annotation which instructs the JVM to treat the corresponding class as local. The downside of this approach is that the resulting behaviour may deviate form the JVM specification, so special care is required when using the annotation.

3.7.6 Running threads on different cores

Like the shared memory approach, the distributed JVM represents threads by a `DistributedThread` class that takes the identifier of the JVM node it is running on. The distributed threads are instantiated from `java.lang.Thread` using the same round-robin approach used in the shared memory JVM. When calling `DistributedThread.start()`, the object uses a native call to relocate itself to the desired JVM node. It is then launched via a remote method call to the new home node, which sets up a local thread, calls `start()` on it and returns (Listing 3.4).

```
package org.barrelfish.jvm;

@CoreLocal
public class DistributedThread implements Runnable {
  private native static int connectToService(byte[] service);
  private native static void moveThread(Object obj,
                                        int connectionId);

  @Sticky private String jvmService;
  private int connectionId = -1;
  @Sticky private LocalThread thread;

  public DistributedThread(String jvmService) {
    this.jvmService = jvmService;
  }

  protected void runThread() {
    thread = new LocalThread() {
      public void run() {
        DistributedThread.this.run();
      }
    };

    thread.start();
  }

  public void start() {
    if (connectionId < 0)
      connectionId = connectToService(jvmService.getBytes());

    moveThread(this, connectionId);
    runThread();
  }

  public void join() {
    thread.join();
  }
}
```

Listing 3.4: Threads in the distributed JVM (simplified)

3.7.7 Synchronization

The instructions `monitorenter` and `monitorexit` are implemented as messages to the home node of the object they apply to. For `monitorenter`, the node delays its reply until the thread has been admitted, while `monitorexit` returns immediately.

The implemented approach is similar to the one presented in Section 3.4.8. However, instead of blocking on a mutex, the node stores a queue of nodes that are blocking on a particular object. Incoming `monitorexit` messages trigger a reply to the front element of the queue while `monitorenter` mes-

sages are added to the end of it. This approach is required as the node cannot block threads directly, since they are potentially running on different cores. Local threads are added to the queue as well, but no messages are sent in this case. This is similar to queue-based locks, such as MCS locks [39].

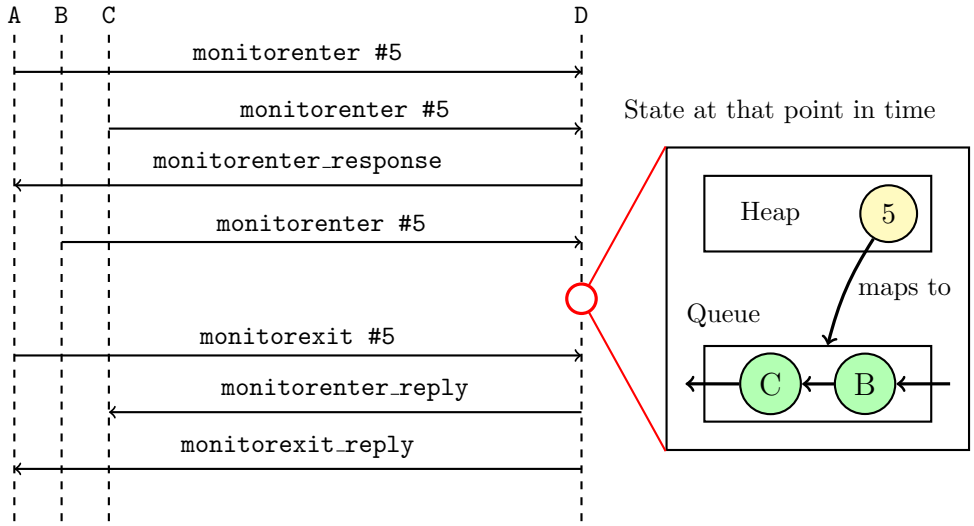

Figure 3.17: Remote synchronization

3.8 Additional implementation strategies

In addition to testing (Section 4.1), I used assertions to confirm the correctness of intermediate states and reduce the probability for errors. To identify bottlenecks and potential errors in the JVM, I performed a profiling analysis using `gprof` [27] (Appendix F). The results gave evidence that the JVM spends most of its time in the interpreter loop and that no other component unexpectedly dominates the run-time. I also used `valgrind` [42] to confirm that the JVM does not exhibit memory leaks or undefined value errors when running a representative set of programs (Java Grande).

3.9 Summary

This chapter presented the implementation of the JVM and showed that it includes all features captured by requirements J-1 to J-9.

4 Evaluation

This chapter presents the evaluation that was performed to analyse the performance trade-offs of the JVM. It summarises the results of testing the JVM, followed by an evaluation of its performance on Linux and Barrelfish, both on a single core and multiple cores.

4.1 Testing for correctness

In order to confirm that the implemented JVM works correctly and fulfils the requirements, I employed the testing methodology outlined in Section 2.4.1. This ensured that any performance evaluation was conducted using a correct JVM implementation. While testing is deeply integrated into the development process, I summarise it in this section for clarity. The full list of tests is given in Appendix B.

4.1.1 Conformance and regression testing

Before implementing each feature, a regression test was written. These are simple Java programs that make use of the feature in question. After completing the implementation, the program was run in the JVM to verify that its

output is correct and execution traces match the JVM specification (conformance). The tests were repeated between iterations, to ensure that features did not break during subsequent changes (regression).

4.1.2 Unit Testing

Unit tests were written to perform blackbox and whitebox testing on individual components of the JVM, such as the loader, linker and heap (the interpreter is covered by regression, conformance and integration tests). The tests were implemented using CUnit [30], a unit testing framework for C. Unused components were replaced by mocks using the CMock [47] mocking framework, which I adapted to interact correctly with CUnit.

4.1.3 Integration Testing

These tests confirm that the JVM as a whole works correctly, including all of its components and their interactions. I used two different kinds of integration tests.

Java Grande Benchmark Suite

I used the benchmarks from the JGF Benchmark suite to test the JVM on a set of real, complex programs. Each of the benchmarks validates the results of its computation, making them good candidates for integration tests.

The JVM correctly executes most of the sequential `section1` and `section2` benchmarks, including heap sort (`heapsort`), the IDEA encryption algorithm (`crypt`), calculating Fourier coefficients (`series`) and sparse matrix multiplication (`sparsematmult`). This covers `JGFHeapSortBenchSizeA` from the requirements analysis (Section 2.3).

The JVM also runs the `raytracer` benchmark from `section3` and produces a pixel-accurate rendering of the expected result (Figure 4.1). This is a good indicator for correctness, since this represents a long and complex execution that stresses most features of the JVM and class library.

Furthermore, the parallel JGF benchmark suite was used to confirm the correctness of the JVM for parallel execution of programs. Benchmarks were primarily executed on Linux, but most of them were checked on Barrelfish as well. In particular, `JGFSparseMatmultBenchSizeA` was correctly executed

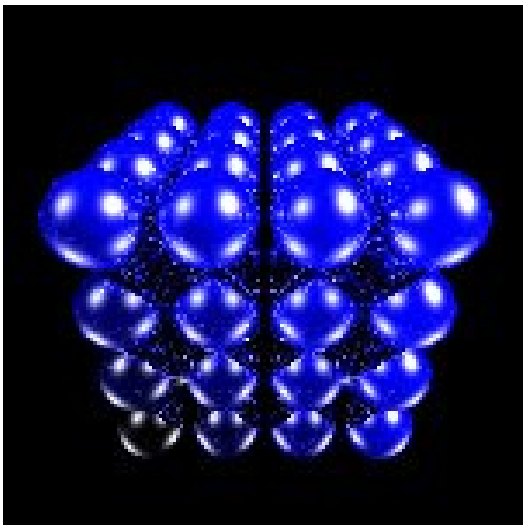

Figure 4.1: Output of the Ray Tracer on the Barrelfish JVM

on all variants of the JVM, adding to the fulfilment of requirements C-1, C-2, E-1 and E-2 from the requirements analysis.

Java tests

In addition to the JGF benchmarks, I implemented a second set of integration tests directly in Java. Since the Barrelfish JVM does not support reflection, traditional unit testing frameworks such as JUnit [14] could not be used. I chose *j2meunit* [3] instead, a variant of JUnit that avoids reflection but otherwise supports most of its features. The tests implemented under this framework cover various aspects of the JVM and the interpreter, as well as my implementation of the class library. Furthermore, the correct execution of the framework adds to the fulfilment of requirement C-1.

4.1.4 Stress Testing

Due to their large size and complexity, the JGF benchmarks could be reused as stress tests. For example, `JGFRayTracerBenchSizeA` allocated more than 100MB of heap memory and ran for more than 200s. Through these tests, it was determined that the JVM fails for large workloads when it runs out of heap memory, stack memory or pre-allocated items such as threads and locks. By choosing these sizes appropriately, the JVM could theoretically be

adapted to support arbitrary workloads. However, a practical limit appears to be reached for `JGFRayTracerBenchSizeB`, since the JVM could not allocate sufficient continuous memory (more than 200MB) on my development machine.

4.1.5 Summary

The tests demonstrate that the JVM can execute a wide range of complex real-world programs and fulfils requirements C-1, C-2, E-1 and E-2. The full *Test Report* can be found in Appendix B.

4.2 Performance evaluation

The purpose of the performance evaluation was to determine the impact of the implemented design decisions and compare the different approaches.

4.2.1 Test environment

The project proposal mentioned QEMU [15] as primary environment for benchmarks and testing. During the project, it became clear that this was not feasible. The run-time and variance exhibited by running software on QEMU is too large to give conclusive results, particularly when simulating a many-core environment (e.g. `JGFHeapSortBenchSizeA` exhibited a single-core run-time of more than 10 minutes, compared to less than 15 seconds on the host's real hardware).

It was therefore necessary to run the benchmarks on real hardware. The test machines of the Computer Labs Systems Research Group (SRG) were used for this purpose. While the 2009 snapshot of Barrelfish did not work on any of the machines, I was able to run the March 2011 version on the "`tigger`" test machine, after a minor modification to Barrelfish's hardware configuration component. This machine was used throughout the entire evaluation.

Tigger is a 48-core AMD Magny-Cours [24] (Opteron 6168) system. It contains four 2×6-core processors running at 1.9GHz with each processor accessing 2×8GB of RAM. Each core has a 512KB L2-cache and groups of six cores share a 6MB L3-cache. Tigger is a NUMA system and has 8 NUMA nodes, each with 8GB of memory (Figures 4.2, 4.3).

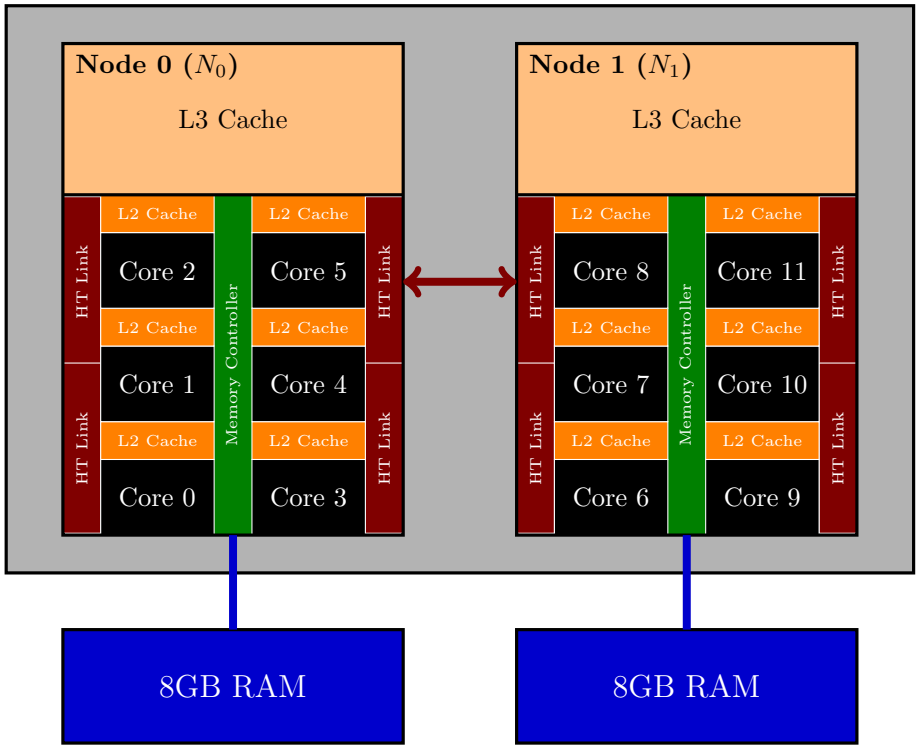

Figure 4.2: A single Magny-Cours chip (based on [24])

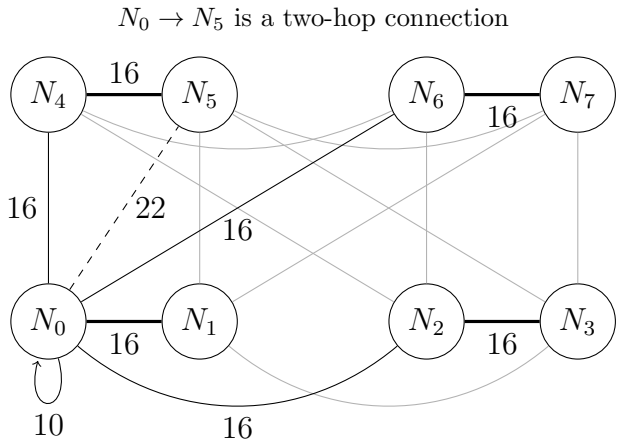

Figure 4.3: Topology of `tigger` (distances based on `numactl`)

4.2.2 Challenges

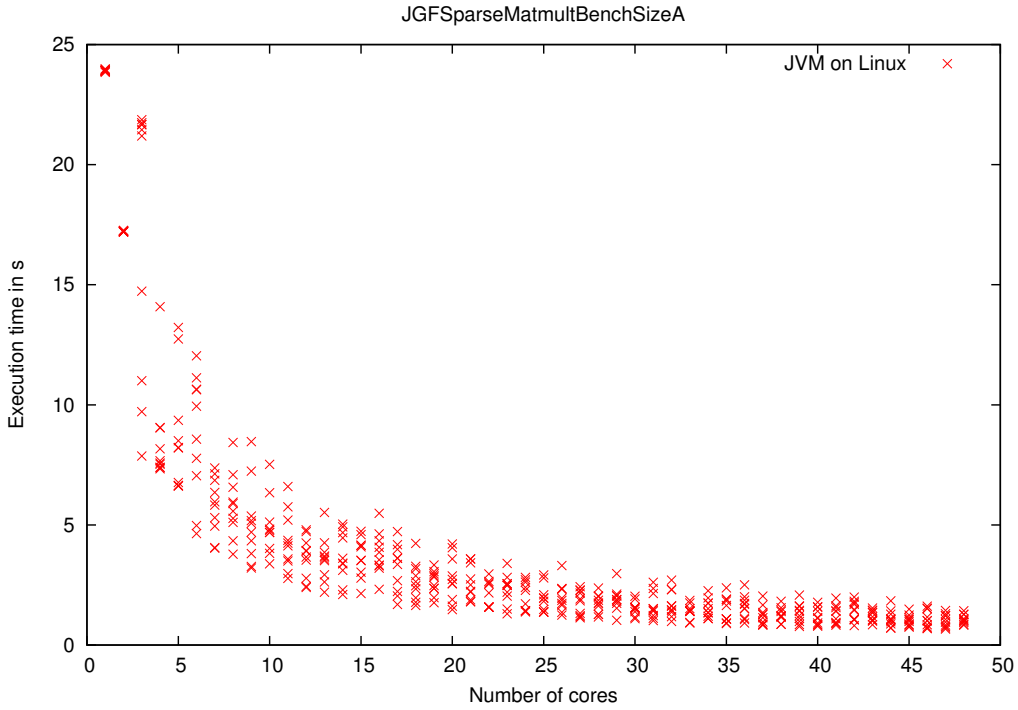

Figure 4.4: High variance in preliminary results

Preliminary test-runs of the JVM on Linux gave a very high variance (Figure 4.4), making it impossible to extract meaningful results. I took the following steps to address these problems:

- **Pinning threads to cores**: It had to be ensured that each thread is executed at the same core during each run of the benchmark. This was achieved using the Linux pthreads API[1].
- **Memory affinity**: Similarly, it had to be ensured that memory is always allocated on the same NUMA node. This was achieved using the `numactl` [29] command line tool.
- **Stack allocation**: It turned out that the high variance was partly related to how the JVM stacks were managed: New stack frames were allocated from the heap and Linux always assigned consecutive regions in memory on the same NUMA node. A possible explanation is that

[1] Barrelfish pins threads by default.

this led to stack frames of different threads (on different cores) overlapping in the same cache line, as the size of the allocated regions is often smaller than the size of a cache line (128 bytes). This would imply that the variance was caused by the cache coherence protocol. Storing the JVM stacks in a consecutive block of memory on the core-local call stacks solved the problem.

These issues were detected through extensive micro-benchmarking and experimenting with different ways of managing memory in the JVM. After performing the changes, the variance was significantly reduced (Figure 4.5).

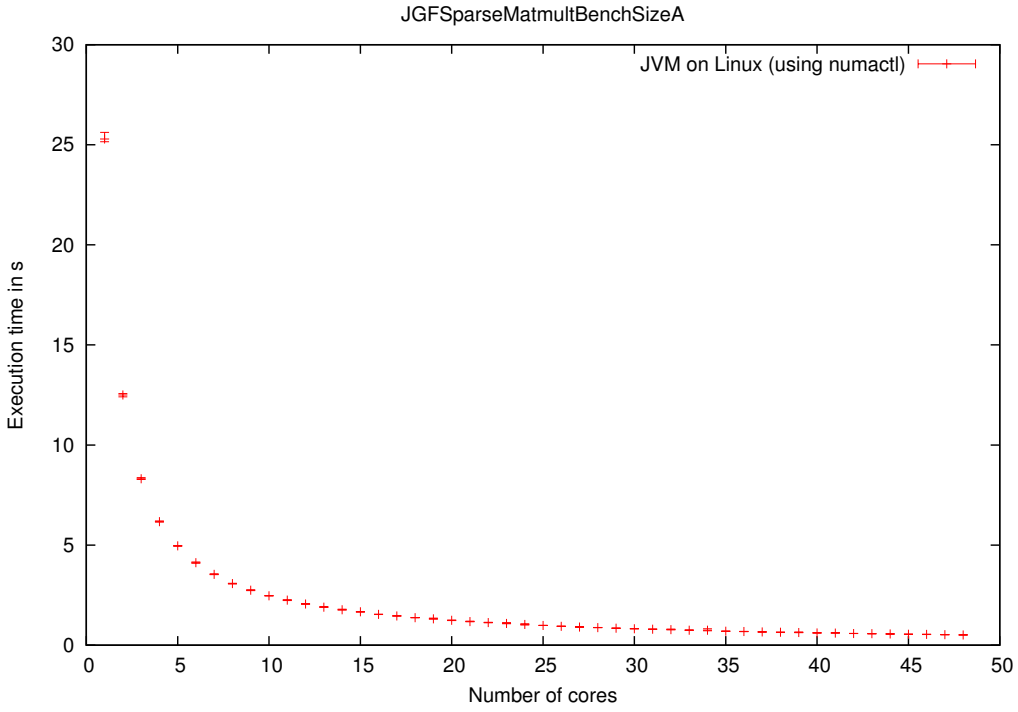

Figure 4.5: Preliminary results after the changes

4.2.3 Evaluation Principles

Measurements were repeated three times and the arithmetic mean was taken. This number was chosen to filter out minor distortions, while the variance was found to be sufficiently small to justify this approach (Figure 4.5). Error bars are shown where appropriate (Figures 4.10 and 4.12 omit them since they

were smaller than the markers[2]). On Barrelfish, the system was rebooted after every benchmark, since Barrelfish's resource management is not very reliable and there was a risk that this could distort the results.

On Linux, `bash` scripts were used to automate the test runs. For Barrelfish, an `expect` script was used to communicate with the test machine over a serial console. The Linux kernel used was `2.6.32-5`.

All benchmarks used the *managed* heap approach (this choice is discussed in Section 4.3). The JVM was compiled on `gcc 4.4.5` using the `-O2` and `-DNDEBUG` compiler flags (disabling assertions). The JVM was configured to use 20MB of heap memory per core and 100KB of stack memory per thread.

4.3 Heap performance

Before starting the main evaluation, I compared the performance of the different heap implementations introduced in Section 3.4.4.

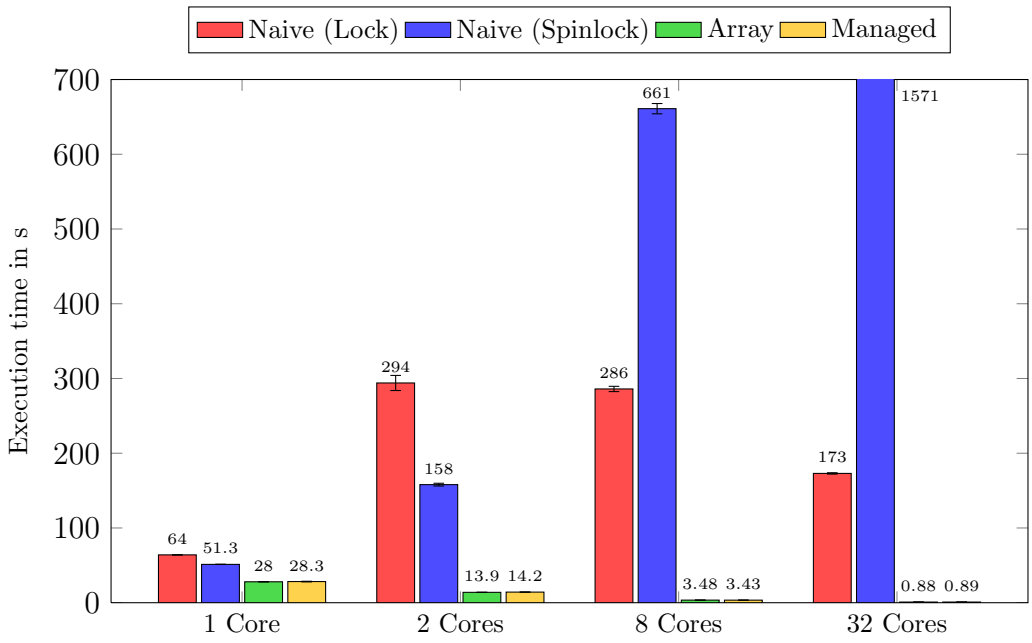

Figure 4.6: Heap benchmarking results (`JGFSparseMatmultBenchSizeA`)

All tests were performed on Linux, since the relative results are platform-independent, up to the cost of different types of locks. To evaluate the heap

[2]All ranges were $\leq$ 0.981s and only 4 of 171 measurements had a range $>$ 0.17s.

performance, a heap-intense benchmark was used. The sparse matrix multiplication benchmark was well-suited for this, since its computation stresses reading and writing to arrays stored on the heap (Appendix F).

The results, presented in Figure 4.6, indicate that a blocking heap implementation is unsuitable, since the overhead of locking (single-core performance) and contention (multi-core performance) dominates the run-time. Based on these results, the managed heap approach was chosen for all further benchmarks.

An interesting result is the fact that the additional level of indirection that the array approach adds over the managed heap does not lead to a change in run-times. This can be seen as an indicator that the heap overhead is negligible to the rest of the run-time, as long as there is no contention.

4.4 Single-core performance

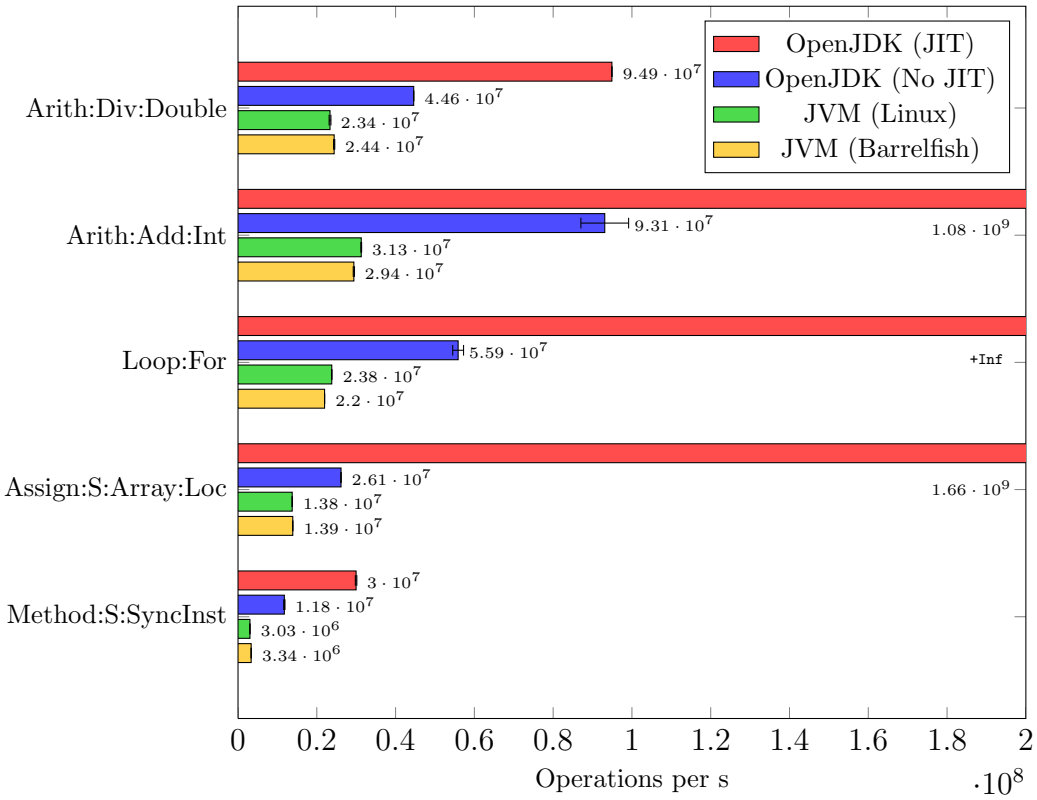

Figure 4.7: Selected micro-benchmarks on a single core

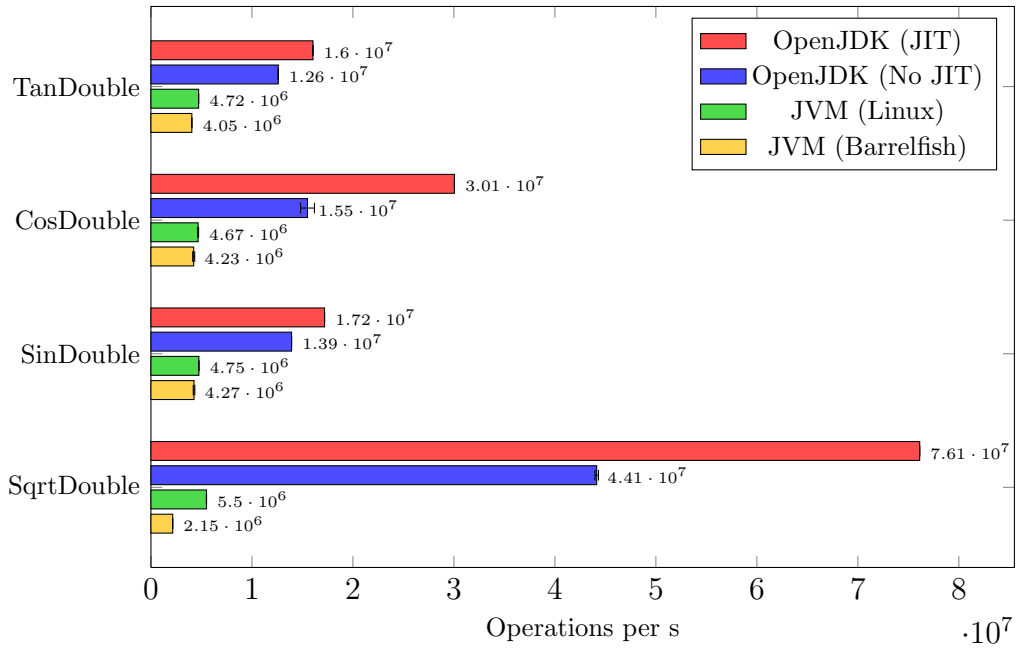

Figure 4.8: `java.lang.Math` operations on a single core

To evaluate the performance on a single core, I used a set of benchmarks from the Java Grande benchmark suite. I compared single-threaded run-times of the Barrelfish JVM on Linux and Barrelfish to the OpenJDK runtime environment (version 1.6.0) with and without JIT compilation (using the `-Djava.compiler=NONE` switch)

These results allow for the following conclusions:

- The Barrelfish JVM is slower than OpenJDK without JIT by a factor of 2-3 (except for some micro-benchmarks). I consider this a success, since the Barrelfish JVM is not optimised and was not designed with performance in mind. The fact that JIT compilation leads to results that are faster by an order of magnitude is unsurprising.
- The performance of the Barrelfish JVM on Linux and on Barrelfish is similar (except for FFT, which uses a large amount of floating point math[3]). This is an important result since it allows a connection to be drawn between results on the two platforms. It also shows that running the system on Barrelfish does not give a significant penalty for single-core performance, which is important when arguing whether Barrelfish should be used as a platform.

[3]This discrepancy may be due to Barrelfish using a different native math library.

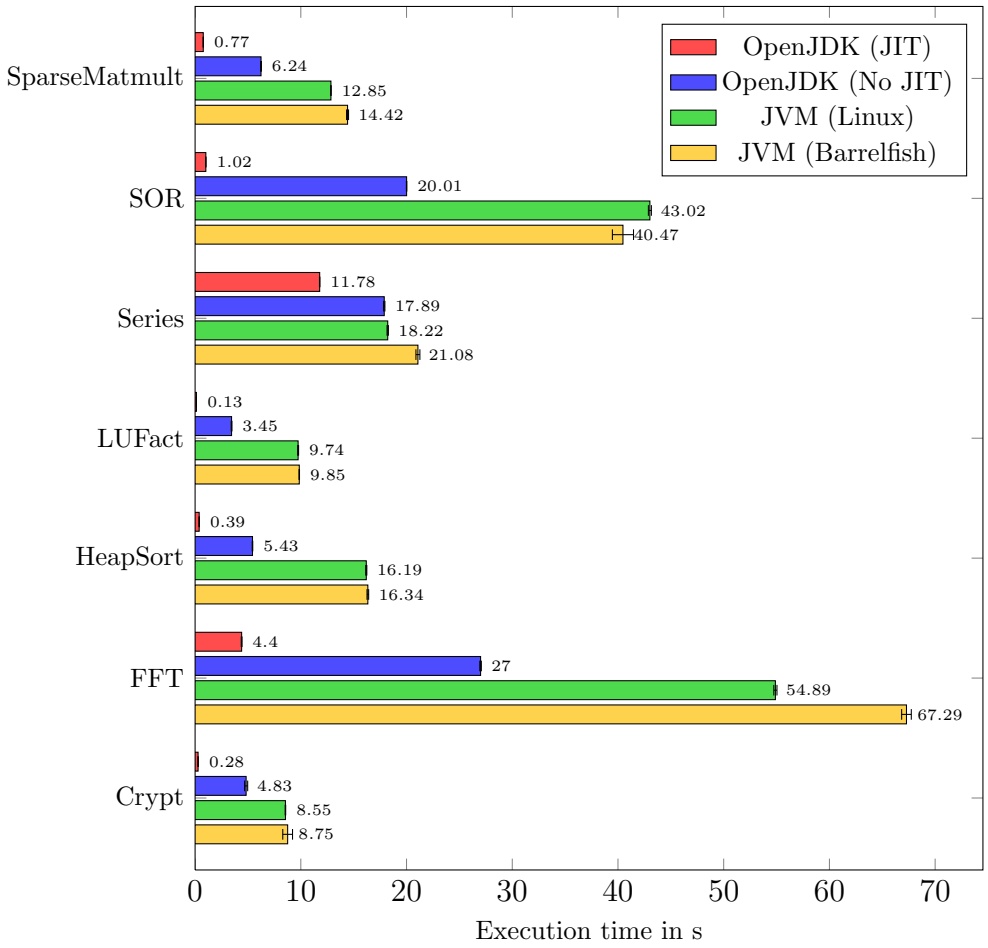

Figure 4.9: Application-benchmarks on a single core (`SizeA`)

While these results do not represent new findings, they are important for evaluating the success of the project. They show that the developed JVM provides the performance necessary to run real-world software, both on Linux and Barrelfish.

4.5 Multi-core performance

Performance on multiple cores was evaluated using the parallel `SparseMatmult` benchmark from the JGF benchmark suite. The benchmark was chosen since it stresses inter-core communication and does not use `Math.sqrt()`, which exhibits different performance on Barrelfish and Linux (Figure 4.8).

The sparse matrix multiplication benchmark measures the execution time of a program that multiplies the compressed representation of a sparse matrix with a vector. It reads from four arrays and all threads write to different ranges of the output array, avoiding contention (Listing 4.1).

```
public void run() {
  for (int reps=0; reps<NUM_ITERATIONS; reps++) {
    for (int i=lowsum[id]; i<highsum[id]; i++) {
      SparseMatmult.yt[ row[i] ] += x[ col[i] ] * val[i];
    }
  }
}
```

Listing 4.1: Kernel of the `JGFSparseMatmultBenchSizeA` Benchmark

4.5.1 Linux

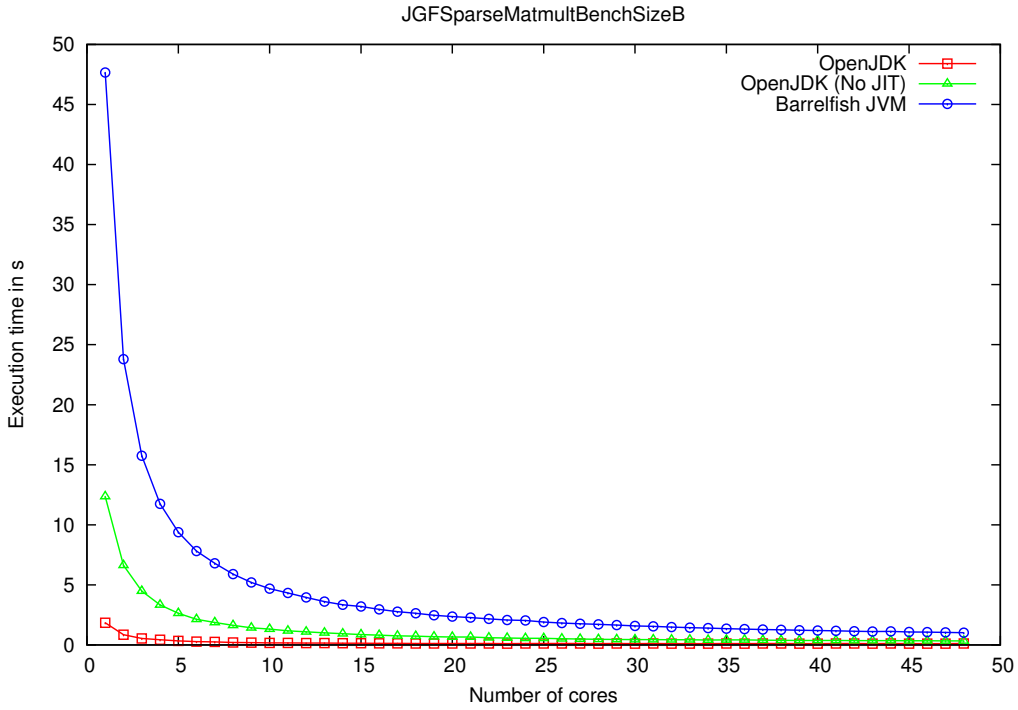

Figure 4.10: Performance on Linux

I first compared the performance of the Barrelfish JVM and OpenJDK on Linux. The results are similar to the results for the single-threaded benchmark in that the performance is within a factor of 2-4. The benchmarks also

demonstrate that the Barrelfish JVM scales very well to multiple cores on Linux, maintaining almost linear speed-up. Here, speed-up is defined as:

$$speed\text{-}up(n) = \frac{t_0}{t_n}$$

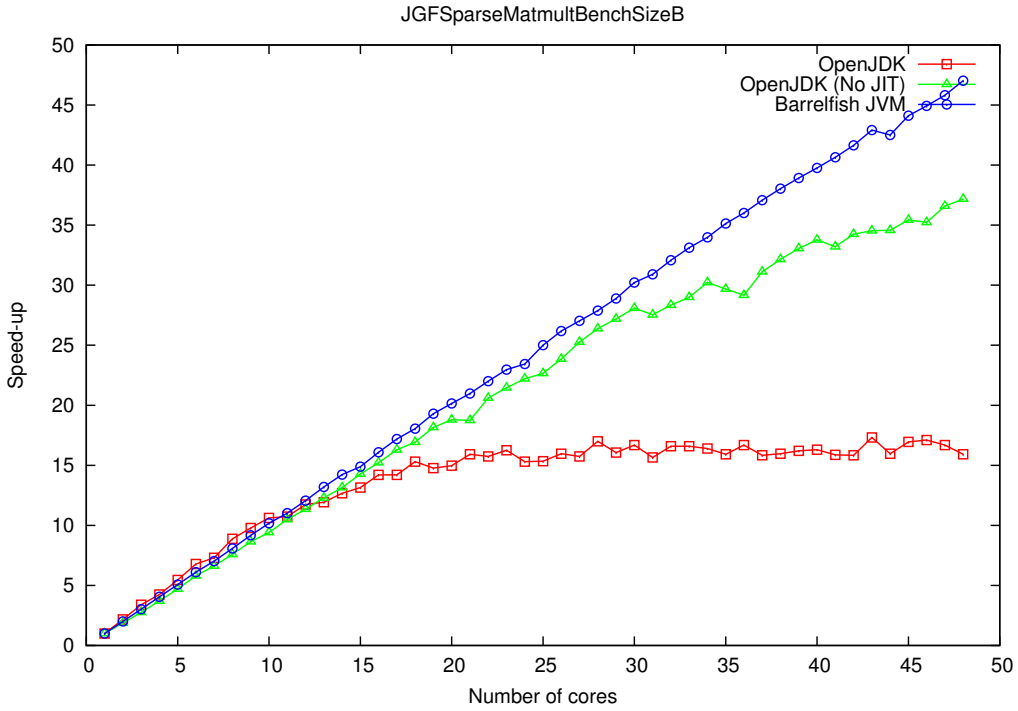

Figure 4.11: Average speed-up on Linux

While this implies that the Barrelfish JVM scales almost perfectly, these numbers have to be taken with a grain of salt: t_0 is 26 times higher for the Barrelfish JVM than it is for OpenJDK with JIT and 4 times higher than for OpenJDK without JIT. The overhead of the JVM is arguably caused by the interpreter loop (Section 3.4.3, Appendix F), which is independent between cores and therefore scales linearly. It is conceivable that the runtime of OpenJDK is dominated by other components that do not scale so well. Nonetheless, the fact that the JVM scales perfectly demonstrates that the multi-core implementation works as expected.

With the Linux results as baseline, I investigated the JVM's performance on Barrelfish. I ran experiments for each of the two approaches and compared the results to the JVM on Linux.

4.5.2 Barrelfish (Shared memory)

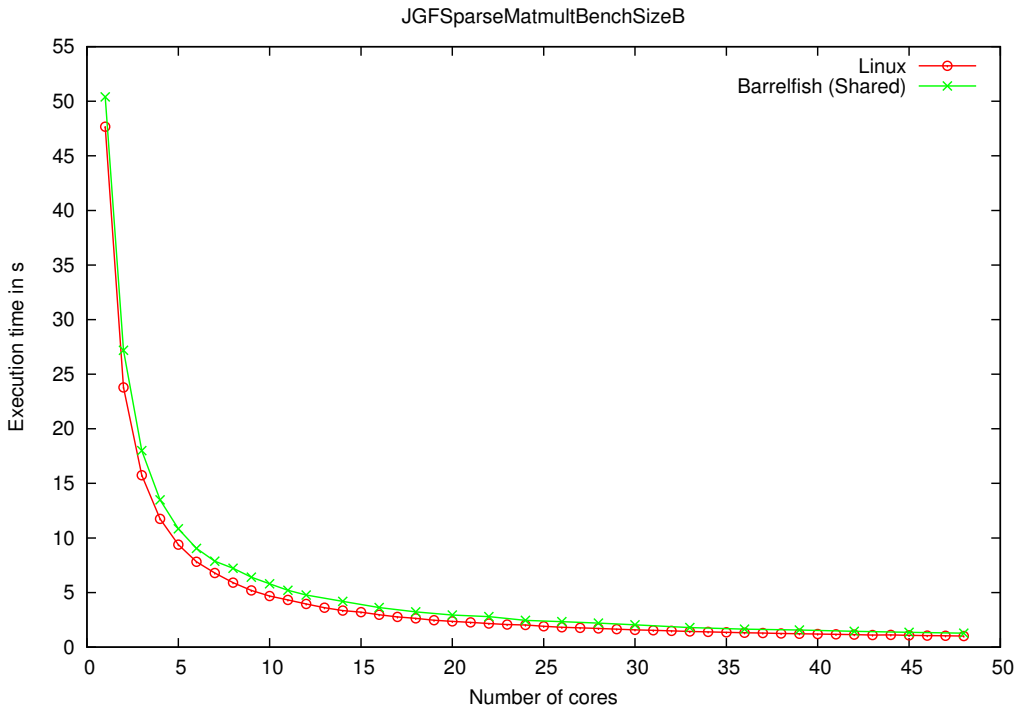

Figure 4.12: Performance of the Shared-memory approach

Figure 4.12 shows that the performance of the shared-memory approach on Barrelfish is similar to the performance on Linux. This demonstrates that the overhead caused by Barrelfishs inter-dispatcher communication does not prevent the JVM from scaling to 48 cores. However, Figure 4.13 reveals that an overhead exists, which could become significant in a more efficient JVM implementation.

4.5.3 Barrelfish (Distributed JVM)

While it was expected that the run-times for the distributed JVM would be longer than for the shared-memory approach, the measurements reveal that the overhead factor is much larger than expected. On two cores, the distributed JVM was about 300 times slower than the shared-memory JVM.

For these experiments, I modified `JGFSparseMatmultBenchSizeA` to reduce the total number of iterations by a factor of 10. This was done to reduce the

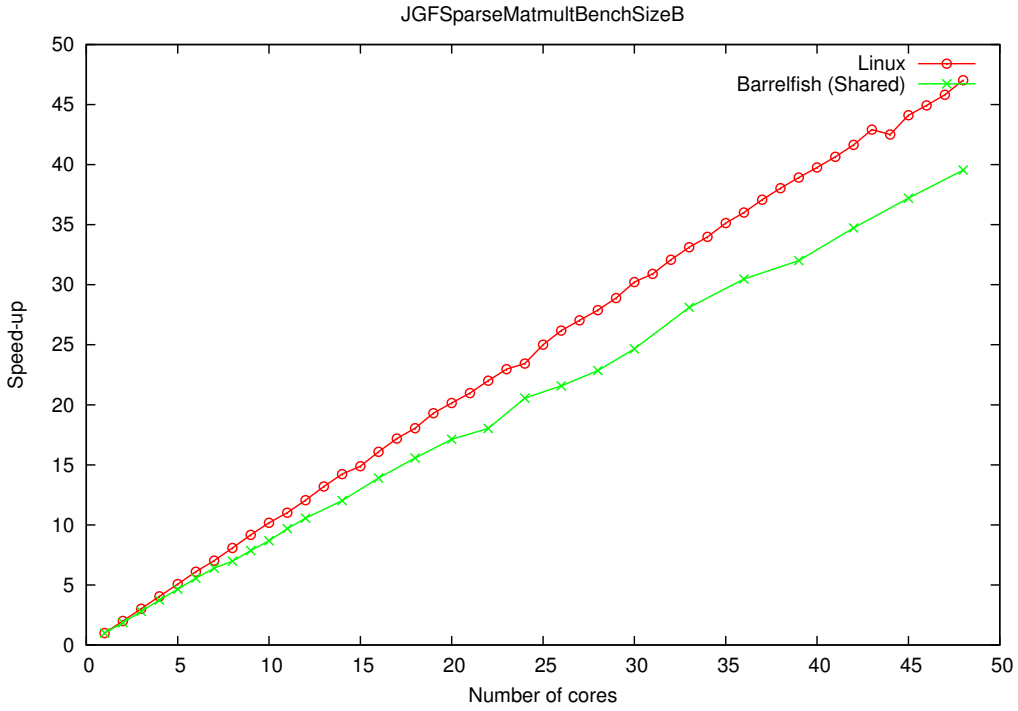

Figure 4.13: Average speed-up of the shared-memory approach

run-time of the experiments (which would have been up to 25h otherwise). Since the benchmark only measures the execution of the kernel, this gives a very close approximation for the run-time of `JGFSparseMatmultBenchSizeA`, divided by 10. I call this benchmark `JGFSparseMatmultBenchSizeA*`.

Cores	Run-time in s	σ (Standard deviation)[4]
1	2.701	0.0017
2	457.893	7.8906
3	395.681	3.5453
4	402.342	7.6161
5	444.382	2.1277
6	514.251	36.769
7	1764.32	247.74
8	2631.27	335.90
16	9333.87	(only executed once)

Table 4.1: Results of `JGFSparseMatmultBenchSizeA*`

[4]Since some executions exhibited a high variance, σ is given for this experiment.

The results show that without optimisation, the distributed approach is too slow to be feasible, at least for this benchmark. Measuring the run-time of each individual thread gives evidence that this is caused by the overhead of message passing: While a thread running on the home node of the working set (`jvm-node0`) completes very quickly, threads on other cores take orders of magnitude longer (Figure 4.14). The diagram also confirms that communication with cores on other chips (#6 and #7) is significantly more expensive than on-chip communication (Figure 4.3).

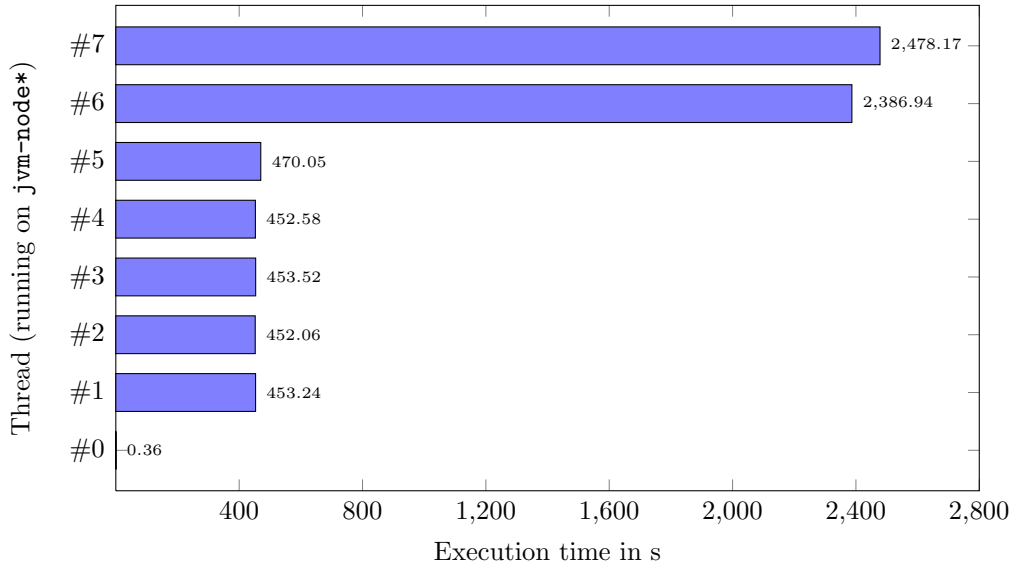

Figure 4.14: Run-times of individual threads

For this particular benchmark, the distributed JVM has to exchange 7 pairs of messages for each iteration of the loop in Listing 4.1 (1 `getfield`, 1 `astore`, 5 `aload`), while the shared-memory approach requires almost no inter-core communication (all arrays reside in the local cache most of the time and there is little contention, since different threads write to different parts of the output array). There are two basic aspects that add to the overhead of the message passing:

- **Inter-core communication**: Each message transfer has to invoke the cache coherence protocol, causing a delay of up to 150-600 cycles, depending on the architecture and the number of hops [12].
- **Message handling**: The client has to yield the interpreter thread, poll for messages, execute the message handler code and unblock the interpreter thread. This involves two context switches and a time in-

terval during which the core is polling for messages. Running multiple threads on the same core exacerbates this problem.

The fact that the JVM has a lower run-time on three cores than on two and four cores may indicate that the run-time on two cores is limited by the latency of inter-core communication and message handling at the client, while the performance on four cores seems to be limited by the message handling on node 0. The high variance for six and more cores is most likely introduced by a saturation of the bus through the cache coherence protocol (similar to what was seen in Section 4.2.2).

Summary

The findings of these experiments are meaningful results, since every JVM that uses the naive distributed approach presented in this dissertation will suffer from this problem. However, this does not imply that the distributed approach in general is unsuitable. If it is possible to reduce the latency of message passing or reduce the number of messages sent between two cores, it may well be possible to improve the performance. The strongest conclusion that can be drawn at this point is therefore that the implemented, unoptimised distributed approach performs poorly on the given cache-coherent NUMA system.

5 Conclusions

This chapter draws conclusions from the results, summarises the outcome of the project and presents future work.

5.1 Results

All core requirements were achieved: The JVM supports features J-1 to J-9 and runs all real-world programs specified in the requirements (C-1, C-2), both on Linux and Barrelfish. It was successfully extended to run across multiple cores on Barrelfish, contrasting a shared memory and a distributed approach (E-1, E-2). In addition to the goals and extensions from the project proposal, the JVM supports additional Java features and programs, hence exceeding the requirements.

It was shown that the developed JVM is sufficiently performant and correct to execute significant real-world Java programs, and scales well to large numbers of cores using shared memory on both Linux and Barrelfish. At the same time, it was determined that the distributed approach performs too poorly to be considered feasible on the given hardware. The overhead introduced by message-passing has been identified as the limiting factor.

It is conceivable that these problems can be alleviated by introducing additional optimisations to reduce this overhead. Broadly, these optimisations fall into two categories:

- **Reducing the number of messages**: This could be achieved by caching classes and arrays. For example, the number of messages in the `JGFSparseMatmultBenchSizeA` benchmark could be reduced to almost zero by this, as most of the arrays are read-only and threads access different ranges of the output array.
- **Reducing the latency of messages**: Different approaches could be used to reduce the latency. The Barrelfish team discusses this topic in the documentation of the March 2011 release and proposes the use of inter-processor interrupts (IPI). Another aspect that could reduce the latency of message passing is the use of platforms that support message passing in hardware.

I addressed the first point by starting the implementation of an array-caching mechanism similar to a directory-based MSI cache coherence protocol [28, pp. 232]. At the time of writing, this feature is still incomplete.

Each array is split into chunks of equal size and the array's home node stores a set of sharers for each chunk. Cores can request read access (in which case they will be added to the set of sharers) or write access (in which case the home node sends an invalidate message to all sharers before marking the chunk as modified and returning it to the requester). When a node tries to access a modified chunk, the chunk's home node prompts the holder of the chunk to write it back.

5.1.1 Future Work

Future work will focus on making the distributed approach more efficient by exploring some of the optimisations introduced in the previous section.

- **Array caching**: Finishing my implementation of array caching and determining the speed-up that can be achieved by it. According to theoretical considerations confirmed by preliminary test runs, it should be possible to eliminate a large proportion of the messages.
- **Notifications and IPI**: Using Barrelfish's notification features to reduce the latency of messages. According to the Barrelfish documentation, latencies of 4,000 cycles per message could be achievable (instead of 25,000 cycles in the current version).

- **Object relocation** The current JVM only relocates objects when a new thread is created. This could be changed to make placement decisions at run-time and move objects between cores. In such a system, a core would have to send a special message if another core tries to access an object that has been relocated.
- **Thread pools**: A simple optimisation would be to use thread pools instead of spawning new threads at each remote method call. None of the benchmarks I used stresses remote method calls, but it is an important feature for many applications.
- **Running on the SCC**: Running Barrelfish on the Computer Lab's Intel SCC [34] would be an opportunity to evaluate the performance on a system that supports message passing in hardware.

Future work could also re-visit the idea of bringing up the Jikes RVM [16] on Barrelfish. With the new features introduced in the March 2011 release, this task could have become easier and many of the lessons from this project still apply.

5.2 Lessons learned

During the project, I learned about many aspects of Barrelfish and JVMs, much of which could not be covered in this dissertation. By studying Barrelfish's source code, I gained a deeper understanding of the project and subtleties in its implementation. I also encountered many obscure low-level problems and bugs, both in my own code and within Barrelfish. This made the project very educational and taught me many skills that will be useful for future research in Operating Systems.

At the same time, the nature of the project led to a large degree of uncertainty, especially with respect to Barrelfish. There are numerous examples for this, such as a bug in the 2009 version of Barrelfish that caused URPC messages to be lost, or the variance problem from Section 4.2.2. Furthermore, I was unable to run Barrelfish on real hardware until March 2011.

Solving these problems took a large amount of time and introduced a certain risk for a time-limited project like this, an aspect I was not completely aware of in the beginning. While it did not prevent the project from being completed successfully, it will still serve as a lesson learned for future projects.

5.3 Success of the project

The implemented JVM satisfies all requirements and core extensions (Section 2.3) and supports additional features that allow it to run a larger set of real-world programs than specified, including a ray tracer (Section 4.1.3).

The JVM runs on Barrelfish, both using a shared memory approach (Section 3.6) and a distributed approach (Section 3.7). Performance measurements have been taken for both approaches and allow to draw conclusions about design-decisions for JVMs on Barrelfish (Section 4.2).

The project therefore satisfied or exceeded all its goals and can hence be considered a success.

Bibliography

[1] Cyclone Microsystems - Programmable Network Adapters: PCI Express and PCI-X. http://www.cyclone.com/products/network_adapters/index.php.

[2] "Distributed JVM" - Google Scholar. http://scholar.google.co.uk/scholar?hl=en\&q=\%22Distributed+JVM\%22.

[3] J2MEUnit - A J2ME Unit Testing Framework. http://j2meunit.sourceforge.net/.

[4] Java Platform, SE 6 - API Specification. http://download.oracle.com/javase/6/docs/api/index.html.

[5] Memory management in the Java HotSpot virtual machine. http://java.sun.com/j2se/reference/whitepapers/memorymanagement_whitepaper.pdf.

[6] OpenJDK. http://openjdk.java.net/.

[7] The Java HotSpot Performance Engine Architecture. http://java.sun.com/products/hotspot/whitepaper.html.

[8] J. Andersson, S. Weber, E. Cecchet, C. Jensen, and V. Cahill. Kaffemik-a distributed JVM on a single address space architecture. In *Java Virtual Machine Research and Technology Symposium*, 2001.

[9] Y. Aridor, M. Factor, and A. Teperman. cJVM: a single system image of a JVM on a cluster. In *icpp*, page 4, 1999.

[10] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '00, page 4765, New York, NY, USA, 2000. ACM.

[11] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52:5667, October 2009.

[12] A. Baumann, P. Barham, P. E Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schpbach, and A. Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *SOSP*, volume 9, page 2944, 2009.

[13] Andrew Baumann, Simon Peter, Adrian Schpbach, Akhilesh Singhania, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. Your computer is already a distributed system. Why isn't your OS? In *Proceedings of the 12th conference on Hot topics in operating systems*, HotOS'09, page 1212, Berkeley, CA, USA, 2009. USENIX Association. ACM ID: 1855580.

[14] Kent Beck and Erich Gamma. JUnit. `http://www.junit.org/`.

[15] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX 2005 Annual Technical Conference, FREENIX Track*, page 4146, 2007.

[16] Alpern Augart Blackburn, S. Augart, S. M Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S Mckinley, M. Mergen, J. E. B Moss, T. Ngo, and V. Sarkar. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Syst. J.*, 44:399–417, January 2005.

[17] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. *JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING*, 37:207—216, 1995.

[18] B. W Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, May 1988.

[19] Erich Boleyn. GRUB - GRand Unified Bootloader. `http://www.gnu.org/software/grub/`, 1996.

[20] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An Operating System for Many Cores. In *Proceedings of the 8th USENIX Symposium on Op-*

erating Systems Design and Implementation (OSDI '08), page 4357, San Diego, California, December 2008.

[21] Silas Boyd-Wickizer, Austin Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, October 2010.

[22] Silas Boyd-Wickizer, Robert Morris, and M. Frans Kaashoek. Reinventing scheduling for multicore systems. In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS-XII)*, Monte Verit, Switzerland, May 2009.

[23] J. M Bull, L. A Smith, M. D Westhead, D. S Henty, and R. A Davey. A Benchmark Suite for High Performance Java. *Proceedings of the ACM Java Grande Conference*, pages 81—88, 1999.

[24] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor. *IEEE Micro*, 30(2):16–29, March 2010.

[25] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2nd edition, September 2001.

[26] T. B Downing. *Java RMI: remote method invocation*, volume 9. IDG Books Worldwide, 1998.

[27] J. Fenlason and R. Stallman. GNU gprof. *GNU Free Software Foundation*, 1998.

[28] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[29] A. Kleen. An NUMA API for Linux. 2004.

[30] Anil Kumar and Jerry St. Clair. CUnit - A Unit Testing Framework for C. `http://cunit.sourceforge.net/`.

[31] James F Kurose and Keith W Ross. *Computer Networking: A Top-Down Approach (5th Edition)*. Addison Wesley, 5 edition, March 2009. Published: Hardcover.

[32] Tim Lindholm and Frank Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, April 1999.

[33] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanovi, and J. Kubiatowicz. Tessellation: Space-time partitioning in a manycore client OS. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, page 1010, 2009.

[34] T. G Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, et al. The 48-core SCC processor: the programmer's view. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, page 111, 2010.

[35] Timothy G Mattson, Rob Van der Wijngaart, and Michael Frumkin. Programming the Intel 80-core network-on-a-chip terascale processor. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, page 38:138:11, Piscataway, NJ, USA, 2008. IEEE Press.

[36] Ross McIlroy and Joe Sventek. Hera-JVM: abstracting processor heterogeneity behind a virtual machine. In *Proceedings of the 12th conference on Hot topics in operating systems*, HotOS'09, page 1515, Berkeley, CA, USA, 2009. USENIX Association.

[37] Paul E. McKenney and Jonathan Walpole. What is RCU, fundamentally? Available: `http://lwn.net/Articles/262464/`, December 2007.

[38] Erik Meijer, Redmond Wa, and John Gough. Technical Overview of the Common Language Runtime. Technical report, Microsoft, 2000.

[39] John M. Mellor-Crummey and Michael L. Scott. Synchronization without contention. *SIGARCH Comput. Archit. News*, 19:269–278, April 1991.

[40] Kevin Modzelewski, Jason Miller, Adam Belay, Nathan Beckmann, Charles Gruenwald, David Wentzlaff, Lamia Youseff, and Anant Agarwal. An Operating System for Multicore and Clouds: Mechanisms and Implementation. `http://dspace.mit.edu/handle/1721.1/51381`.

[41] D. G Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. CIEL: a universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'11, Berkeley, CA, USA, 2011. USENIX Association.

[42] N. Nethercote and J. Seward. Valgrind:: A Program Supervision Framework. *Electronic notes in theoretical computer science*, 89(2):4466, 2003.

[43] E. B Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[44] D. Pham, S. Asano, M. Bolliger, M. N Day, H. P Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation CELL processor. *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, pages 184–592 Vol. 1, February 2005.

[45] T. L Rodeheffer. Code Generation for the Beehive ISA. Technical Report MSR-TR-2010-113, Microsoft Research, 2010.

[46] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for Java. *ECOOP 2008 Object-Oriented Programming*, page 104128, 2008.

[47] Mark VanderVoord. CMock - Mock Module Generation Framework for C. `http://sourceforge.net/apps/trac/cmock/wiki`.

[48] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2009.

[49] Y. Yan, M. Grossman, and V. Sarkar. JCUDA: a programmer-friendly interface for accelerating Java programs with CUDA. *Euro-Par 2009 Parallel Processing*, page 887899, 2009.

[50] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: a high-performance Java dialect. *Concurrency: Practice and Experience*, 10(1113):825–836, September 1998.

[51] J. N Zigman and R. Sankaranarayana. dJVM - A distributed JVM on a cluster. Technical report, Australian National University, 2002.

A

Requirements Analysis

This appendix contains the full version of the first part of the requirements analysis that was presented in Section 2.3.1. It lists the entire set of Java features that had to be supported by the Barrelfish JVM, together with an account of why they are required (where appropriate).

The following notation is used:

- (∗) - required by `JGFHeapSortBenchSizeA`
- (†) - required by `JGFSparseMatmultBenchSizeA`
- (⋆) - required by `j2meunit`

Unmarked entries are generic requirements that are necessary to execute any sufficiently complex Java program.

#	Description
J-1	*Loading and linking classes (supporting inheritance)*
	Parsing class files to internal representation
	Handling and extracting constant pool entries
	Linking references to their actual representations
	Calculating a memory layout for class instances
	Looking up methods in ancestor classes
	Supporting numeric constants (`int`, `double`) ∗ †

#	Description
J-2	*Executing basic programs (supporting arithmetic, control transfer)*
	Integer arithmetic (`int`, `long`) ∗ †
	Floating-point arithmetic (`double`, `float`) ∗ †
	Control-transfer instructions (`ifeq`, `goto`, etc.)
J-3	*Static method calls and static field access*
	Managing the JVM-stack (creating, deleting frames)
	Looking up static methods and fields
J-4	*Creating instances (heap), field access and virtual method calls*
	A working heap implementation
	Running constructors and static initialisers
	Creating class instances
	Looking up virtual methods and fields from objects
J-5	*Creating and manipulating arrays on the heap*
J-6	*Supporting native method calls and system features*
	Writing characters to the console ∗ † ⋆
	Fast copying of array contents (`System.arraycopy()`)
	System time (`System.getCurrentTimeMillis()`) ∗ †
	Math functions (`java.lang.Math`) ∗ †
J-7	*String handling (including command line arguments)*
	Supporting the `java.lang.String` class ∗ † ⋆
	Loading string constants from the constant pool ∗ † ⋆
	Using strings as keys (`equals()`, `hashCode()`) ∗ † ⋆
	Conversion between numbers and strings ∗ † ⋆
	Support for composing strings (`StringBuilder`) ∗ † ⋆
	Support for `java.lang.StringBuffer` (thread-safe) ⋆
	Support for command line arguments ∗ † ⋆
J-8	*Spawning and joining threads, synchronisation primitives*
	Creation of threads and assigning them to cores †
	Joining threads (potentially on a different core) †
	Support for `monitorenter` and `monitorexit` † ⋆
J-9	*Basic classes from the class library (e.g.* `java.util.HashMap`*)*
	Very basic introspection (`java.lang.Class`) ⋆
	Command line output (`java.io.PrintStream`, etc.) ∗ † ⋆
	Java-compliant random numbers (`java.util.Random`) ∗ †
	Collections (`java.util.Hashtable`, `java.util.Vector`) ∗ † ⋆

B

Test Report

This appendix lists the test that were performed to confirm the correctness of the JVM. Information about the underlying testing methodology is given in Section 2.4.1, while Section 4.1 summarises the results of the testing.

B.1 Regression and conformance tests

These are simple Java programs that were specified during the development and capture a particular JVM feature. Some of them (marked by ∗) are also used as conformance tests and their execution trace is checked.

args, arrays*, concat, conditionals, constants*, constructor, distributed1, distributed2, fibonacci*, hashtable, helloworld, inheritance, instantiation, multicore1, multicore2, multiple classes, native calls, random, simple, sleep, static fields*, strings, synchronized, threads

```
Barrelfish JVM - Conformance Testing

Test: Arrays... PASS
Test: Constants... PASS
Test: Fibonacci... PASS
Test: StaticFields... PASS

ALL TESTS OK
```

B.2 Unit tests

Unit tests for some of the JVM's components. They have been implemented using the CUnit [30] unit testing framework and CMock [47] for mocking.

```
Suite: loader
  Test: 001 - loading an empty class ... passed
  Test: 002 - loading a more complex class ... passed
  Test: 003 - loading multiple classes ... passed
Suite: hashmap-naive
  Test: 001 - get/set without concurrency ... passed
  Test: 002 - unset without concurrency ... passed
  Test: 003 - check for unset corner-case ... passed
  Test: 004 - concurrent readers ... passed
  Test: 005 - concurrent readers and writers ... passed
Suite: hashmap-array
  Test: 001 - get/set without concurrency ... passed
  Test: 002 - unset without concurrency ... passed
  Test: 003 - concurrent readers ... passed
  Test: 004 - concurrent readers and writers ... passed
Suite: linker
  Test: 001 - linking an empty class ... passed
  Test: 002 - linking a single, more complex class ... passed
  Test: 003 - linking a hierarchy of classes ... passed
  Test: 003 - resolution of references ... passed
  Test: 004 - execution of static initializers ... passed
  Test: 005 - reading special annotations ... passed
```

B.3 Integration tests

Java-based unit tests (j2meunit)

These tests contain more thorough testing of Java features and are written in Java itself, executed by the unit testing framework j2meunit [3].

```
J2ME Unit 1.1.1 by RoleModel Software, Inc.
Original JUnit by Kent Beck and Erich Gamma

Suite: ArithmeticTest
..testIntegerArithmetic
..testLongArithmetic
..testFloatArithmetic
..testDoubleArithmetic
..testIntegerBitwiseOperators
..testLongBitwiseOperators

Suite: CastTest
..testByteCasts
..testIntegerCasts
..testFloatingPointCasts
..testObjectCasts

Suite: MethodCallTest
..testStaticMethods
..testInterfaceMethods
..testVirtualMethods
..testPrivateMethods
..testMethodOverloading

Suite: InheritanceTest
..testInheritance
..testCircularDependency

Suite: ArrayTest
..testIntArray
..testLongArray
..testObjectArray
..testMultidimensionalArray

Suite: ConcurrencyTest
..testThreads
..testSyncMethods
..testSyncOnObject

Suite: ClassLibraryTest
..testHashtable
..testIntrospection
..testMath
..testRandom
..testString
..testStringBuilder
..testVector
```

Java Grande Benchmark Suite

The following benchmarks were successfully tested on the Barrelfish JVM (all benchmarks were tested on Linux and the shared-memory JVM on Barrelfish).

Section 1 (sequential): JGFArithmeticBench, JGFAssignBench, JGFLoopBench, JGFMethodBench; JGFMathBench (partial)

Section 2/3 (sequential): JGFCryptBench, JGFFFTBench, JGFHeapSortBench, JGFLUFactBench, JGFSeriesBench, JGFSORBench, JGFSparseMatmultBench (all sizes); JGFRayTracerBenchSizeA (Linux only)

Parallel: JGFSparseMatmultBenchSizeA/B (1-48 cores); JGFSyncBench, JGFCryptBenchSizeA, JGFSeriesBenchSizeA, JGFSORBenchSizeA (4 cores)

C

Running example

This appendix presents a running example to accompany the implementation chapter. It will show how a simple Java program is loaded by the class loader, processed by the linker and executed by the interpreter.

Consider the following example program:

```
public class Example {
  public static final long CONSTANT = System.currentTimeMillis();
  static class A { int a; public String getA() { return "Hello"; } }
  static class B extends A { long b; public String get() { return getA(); } }
  public static void main(String[] args) { System.out.println(new B().get()); }
}
```

Class loader

The Java compiler generates 3 class files: `Example.class`, `Example$A.class` and `Example$B.class`. Each of these class files contains a constant pool (Section 2.1.2). The loader parses all these files into appropriate data structures (Figure 3.2), including the following:

- A `jvm_class_info` for each class (with associated methods, constant pool and other data). For example, the `super_class` entry of `B` contains an index (`#4`) into the constant pool where a `Class_info` entry is stored, pointing to a string `"Example$A"`.

- `jvm_method_info` entries for `get()`, `getA()`, `main()` and three implicit constructors. Each contains an index to name and signature strings in the constant pool, e.g. `"get()"` and `"()Ljava/lang/String;;"`.
- A `jvm_attribute_info` entry for each method, with a name index pointing to a string `"Code"` (stored in the constant pool of the method's class) and containing the method's bytecode, stack size, etc.

Linker

The structures produced by the class loader are translated to equivalent structures in the linker. Each `jvm_class_info` is translated to a `jvm_class`, storing, for example, the class name and the parent class. The `jvm_class` for `Example$B` contains a `super_class` pointer to `Example$A`, the other classes point to `java.lang.Object`, which was loaded automatically.

Methods are translated to `jvm_method` structures, including their name, signature, number of arguments, stack size and other values such as a pointer to their byte code.

The constant pool is translated to a *run-time constant pool*, replacing indices by explicit pointers. For example, entry `#2` of `B`'s constant pool contains a `Methodref_info` entry for `getA()`. It is replaced by an explicit pointer to its `jvm_method` in the run-time constant pool. The reference is resolved by first looking up `getA()` with signature `"()Ljava/lang/String;;"` in `B` and when this fails, searching its super class `A`.

Furthermore, the linker calculates the memory layout for each of the classes. An instance of `A` consists a pointer to `A`'s `jvm_class` followed by a 32-bit value (`int a`), while instances of `B` consist of a pointer to `B`'s `jvm_class` followed by a 32-bit value and a 64-bit value.

The linker will also invoke the interpreter to execute static initialisers. While this program does not contain explicit initialisers, the compiler will automatically generate an initialiser to execute `System.currentTimeMillis()` and store its result in `CONSTANT`.

Interpreter

The interpreter looks up the `main` method in `Example`, sets up a JVM stack and starts execution. For an example execution trace, see Appendix D.

D

Sample output

This appendix presents the abridged output of the Barrelfish JVM for `fib(3)`*, the Fibonacci program given in Listing 2.1. All debug output is enabled.*

```
Loading classes...
Loaded class 'org/barrelfish/jvm/DistributedThread'.
...
Loaded class 'Fibonacci'.
Linking classes...
Linking class 'org/barrelfish/jvm/DistributedThread'
...
Initializing class java/lang/System...
> new 'java/io/PrintStream'
...
> putstatic #6 //Field java/lang/System.out
STACK: [ ]
> return
...
Linking class 'Fibonacci'
Initializing class Fibonacci...
Running main method...
> iconst_3
STACK: [ 3 ]
> invokestatic #2 //Method fib:(I)I
STACK: [ ]
> iload_0
STACK: [ 3 ]
> iconst_1
STACK: [ 3 1 ]
> if_icmp* 5
STACK: [ ]
> iload_0
STACK: [ 3 ]
> iconst_1
STACK: [ 3 1 ]
> isub
STACK: [ 2 ]
```

```
> invokestatic #2 //Method fib:(I)I
STACK: [ ]
> iload_0
STACK: [ 2 ]
> iconst_1
STACK: [ 2 1 ]
> if_icmp* 5
STACK: [ ]
> iload_0
STACK: [ 2 ]
> iconst_1
STACK: [ 2 1 ]
> isub
STACK: [ 1 ]
> invokestatic #2 //Method fib:(I)I
STACK: [ ]
> iload_0
STACK: [ 1 ]
> iconst_1
STACK: [ 1 1 ]
> if_icmp* 5
STACK: [ ]
> iconst_1
STACK: [ 1 ]
> ireturn
STACK: [ 1 ]
> iload_0
STACK: [ 1 2 ]
> iconst_2
STACK: [ 1 2 2 ]
> isub
STACK: [ 1 0 ]
> invokestatic #2 //Method fib:(I)I
STACK: [ ]
> iload_0
STACK: [ 0 ]
> iconst_1
STACK: [ 0 1 ]
> if_icmp* 5
STACK: [ ]
> iconst_1
STACK: [ 1 ]
> ireturn
STACK: [ 1 1 ]
> iadd
STACK: [ 2 ]
> ireturn
STACK: [ 2 ]
> iload_0
STACK: [ 2 3 ]
> iconst_2
STACK: [ 2 3 2 ]
> isub
STACK: [ 2 1 ]
> invokestatic #2 //Method fib:(I)I
STACK: [ ]
> iload_0
STACK: [ 1 ]
> iconst_1
STACK: [ 1 1 ]
> if_icmp* 5
STACK: [ ]
> iconst_1
STACK: [ 1 ]
> ireturn
STACK: [ 2 1 ]
> iadd
STACK: [ 3 ]
> ireturn
STACK: [ 3 ]
> pop
STACK: [ ]
> return
```

E

Sample code

This appendix shows a piece of example code to visualise the style of the Barrelfish JVM. The example is taken from the command line interface to launch the JVM on Barrelfish. It presents aspects of running the JVM on Barrelfish, handling messages and setting up the different JVM components.

```
/**
 * \file
 * \brief The command-line interface of the JVM on Barrelfish. It is used to
 * spawn a JVM service on a core and can be invoked from the shell to execute
 * a Java program on a particular core.
 */

// System includes
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Barrelfish includes
#include <barrelfish/barrelfish.h>
#include <barrelfish/dispatch.h>
#include <barrelfish/nameservice_client.h>

// Flounder definitions (this is automatically generated)
#include <if/jvm_defs.h>

// JVM includes (JVM, helpers, Barrelfish-specific)
#include "jvm/casts.h"
#include "jvm/class.h"
#include "jvm/debug.h"
#include "jvm/heap.h"
#include "jvm/interpreter.h"
#include "jvm/linker.h"
#include "jvm/loader.h"
#include "jvm/threads.h"
```

```
#include "util/hashmap.h"
#include "runner.h"

// Declare the message handlers on the distributed JVM
#ifdef DISTRIBUTED
...
#endif

// Interface to access the Java classes packaged with the executable
#include "package.h"

// Forward-declare JVM handler methods
static void execute(struct jvm_binding *b, char *class_name, char *args_str);
static void reset(struct jvm_binding *b);

/**
 * A vtable containing the message handlers on the server-side. This is used
 * by Flounder to dispatch incoming messages to their respective handler.
 */
static struct jvm_rx_vtbl server_vtbl = {
  .execute = execute,
  .reset = reset,

  // Messages for the distributed JVM (defined in server.c)
  #ifdef DISTRIBUTED
  ...
  #endif
};

/**
 * Indicates that the connection was successfully set up. No lock is required,
 * since this is only used during the setup phase, where only one thread can
 * access this variable at a time. This variable is flipped from false to true,
 * once the connection is ready, which is the only write access to it.
 */
static bool request_done = false;

/// A connection to a JVM node.
static struct jvm_binding *binding = NULL;

/// Name of the JVM node this client is connecting to.
static char *service_name = "jvm_default";

/**
 * Callback that is executed to configure the server's binding.
 *
 * \param st State passed to the callback (ignored).
 * \param b The binding that has been created.
 *
 * \return Any errors that ocurred.
 */
static errval_t connect_cb(void *st, struct jvm_binding *b) {
  assert(b != NULL);

  // Set the server's message handler vtable
  b->rx_vtbl = server_vtbl;

  return SYS_ERR_OK;
}


/**
 * Callback that is executed to register with the nameserver.
 *
 * \param st State passed to the callback (ignored).
 * \param err Any errors that occured while binding.
 * \param iref An interface reference that identifies the service.
 */
static void export_cb(void *st, errval_t err, iref_t iref) {
  assert(!err_is_fail(err));
  nameservice_register(service_name, iref);
}
```

```
/**
 * Callback that is executed once the client successfully connected to a
 * service.
 *
 * \param st State passed to the callback (ignored).
 * \param b The binding that has been created.
 *
 * \return Any errors that ocurred.
 */
static void bind_cb(void *st, errval_t err, struct jvm_binding *b) {
  assert(!err_is_fail(err));
  assert(b != NULL);

  binding = b;
  request_done = true;
}

/**
 * Handler for the execute message. This requests a server to execute a Java
 * class. The class has to be known to it and contain a static main method
 * with the following signature:
 *
 * public static void main(String[] args)
 *
 * The handler sets up a new thread and returns. It takes ownership of the
 * string parameters passed to it.
 *
 * \param b A binding to send reply messages (ignored).
 * \param class_name The name of the class to execute.
 * \param An optional command line argument, "" if no argument is given.
 */
static void execute(struct jvm_binding *b, char *class_name, char *args_str) {
  assert(class_name != NULL);
  assert(args_str != NULL);

  LOG("JVM node \"%s\": Executing class \"%s\".\n", service_name, class_name);

  // XXX: Support more arguments if necessary.
  uint32_t *args = (uint32_t*)malloc(sizeof(uint32_t));
  if (strlen(args_str) == 0) {
    // No argument given.
    *args = heap_put_array(create_array(T_REFERENCE, 0));
  } else {
    // One argument given.
    struct jvm_array *array = create_array(T_REFERENCE, 1);
    array_store(array, 0, load_string(args_str));
    *args = heap_put_array(array);
  }

  // Look up the class to execute.
  struct jvm_class *main_class = get_class(class_name);

  if (main_class == 0) {
    printf("[Error] Class \"%s\" not found.\n", class_name);
    return;
  }

  // Look up the main method.
  struct jvm_method *entry_method = get_method(
      main_class, "main", "([Ljava/lang/String;)V");

  if (entry_method == 0) {
    printf("[Error] No entry method found in \"%s\".\n", class_name);
    return;
  }

  // Run the interpreter
  LOG("Running entry method...");
  create_runner_thread(main_class, entry_method, args, 1);
}
```

```
/**
 * Handler for the reset message. This requests the JVM to reset its state to
 * the initial configuration. However, the internal state will not be exactly
 * the same (different memory allocations, etc.). This method is therefore not
 * suitable for benchmarks.
 *
 * \param b A binding to send reply messages (ignored).
 */
static void reset(struct jvm_binding *b) {
  LOG("JVM node \"%s\": Reset request.\n", service_name);
  heap_reset();
  jvm_linker_reset();
  jvm_init_threads();
  printf("JVM reset completed.\n");
}

/**
 * The entry point of the JVM. When called with one parameter, it launches a
 * JVM server on the current core (this core is given in GRUB's menu.lst, e.g.
 * a line from menu.lst would look like
 *
 * module /x86_64/sbin/jvm core=0 jvm-node0
 *
 * When called with more than one parameter (usually from the shell), the
 * application will connect to the jvm-server given as the first parameter
 * and launch the class given as the second parameter, e.g.
 *
 * jvm jvm-node0 ClassA
 */
int main(int argc, char** argv) {
  errval_t err;

  // At least one argument has to be given.
  if (argc <= 1) {
    printf("Usage: jvm <service-name> [class-name]\n");
    return 1;
  }

  // The first argument is the service name.
  service_name = argv[1];

  // If one argument is given, start a server.
  if (argc == 2) {
    // Initialize the JVM server.
    LOG("Initializing JVM node...");

    // Initialize thread and synchronisation management.
    jvm_init_threads();

    #ifdef DISTRIBUTED
    jvm_init_sync();
    #endif

    // Load all classes packaged with the executable.
    uint16_t class_count = get_package_classes_count();
    void **class_data = get_package_classes();

    struct jvm_class_info **loaded_classes = (struct jvm_class_info**)
        malloc(sizeof(struct jvm_class_info*) * class_count);
    assert(loaded_classes != NULL);

    LOG("Loading classes...");
    for (int i = 0; i < class_count; i++) {
      loaded_classes[i] = load_class(class_data[i], 0);
    }

    // Initialise the core JVM (e.g. Heap, lookup tables, etc.)
    #ifdef DISTRIBUTED
    jvm_init(disp_get_core_id());
    init_distributed_jvm();
    #else
    jvm_init();
    #endif
```

```
    // Link all classes.
    LOG("Linking classes...");
    jvm_init_linker(class_count, loaded_classes);

    // Set up the JVM service for this node.
    LOG("Setting up JVM service...");
    err = jvm_export(NULL, export_cb, connect_cb,
        get_default_waitset(), IDC_EXPORT_FLAGS_DEFAULT);

    if (err_is_fail(err)) {
      ERROR("Setting up the JVM service failed.\n");
      exit(-1);
    }

    LOG("JVM server at \"%s\" is ready.", service_name);

    // Main message handling loop. Barrelfish applications never exit.
    while (1) {
      messages_wait_and_handle_next();

      // Prevents the message-handling loop from hogging the system for an
      // entire time-slice (80ms). This is necessary since incoming messages
      // may unblock other threads that need to be executed to avoid delays.
      // This has no effect if there is no other thread running.
      thread_yield();
    }
  } else {
    // Create a client and execute a class on a different JVM node.
    LOG("Launching JVM client...");

    // Look up the JVM service from the name server.
    iref_t iref;
    err = nameservice_blocking_lookup(service_name, &iref);

    if (err_is_fail(err) && iref != 0) {
      printf("[Error] Failed to lookup JVM node.\n");
      exit(-1);
    }

    // Create a connection to the service.
    jvm_bind(iref, bind_cb, NULL, get_default_waitset(),
        IDC_BIND_FLAGS_DEFAULT);

    // Wait until setting up the connection is finished.
    while (!request_done) {
      messages_wait_and_handle_next();
    }

    // XXX: For now only support a single argument.
    char *args = "";
    if (argc > 3)
      args = argv[3];

    if (!strcmp(argv[2], "reset")) {
      // Send a reset request to the server.
      jvm_reset__tx(binding, NOP_CONT);
    } else {
      // Send an execution request to the server.
      jvm_execute__tx(binding, NOP_CONT, argv[2], args);
    }

    // Main message-handler loop (details above).
    while (1) {
      messages_wait_and_handle_next();
      thread_yield();
    }
  }

  // This should never happen.
  return 0;
}
```

F

Profiling results

This appendix presents the output of `gprof` *when applied to the Barrelfish JVM running the sequential* `JGFSparseMatmultBenchSizeA` *benchmark using the managed heap approach (Section 3.4.4) on Linux. It shows that even on a benchmark dominated by memory access, the JVM spends most of its time in the interpreter loop and that there are no unexpected bottlenecks.*

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 79.62     11.94    11.94       16     0.75     0.91  run
  7.27     13.03     1.09 150250001     0.00     0.00  array_load64
  4.40     13.69     0.66 603601432     0.00     0.00  heap_get_array
  3.33     14.19     0.50 101000638     0.00     0.00  array_load
  3.13     14.66     0.47      129     0.00     0.00  load_class
  1.20     14.84     0.18       73     0.00     0.00  array_length
  0.40     14.90     0.06 50300003     0.00     0.00  array_store64
  0.33     14.95     0.05  2400287     0.00     0.00  create_frame
  0.27     14.99     0.04   500020     0.00     0.00  array_store
  0.07     15.00     0.01  1900311     0.00     0.00  get_method
  0.00     15.00     0.00  7452574     0.00     0.00  heap_get_instance
  0.00     15.00     0.00  2400287     0.00     0.00  push_frame
  0.00     15.00     0.00  1900158     0.00     0.00  lookup_method
  0.00     15.00     0.00   135416     0.00     0.00  get_cp_string
  0.00     15.00     0.00   121263     0.00     0.00  get_class_name
  0.00     15.00     0.00     2197     0.00     0.00  get_class_index
  0.00     15.00     0.00     2124     0.00     0.00  get_class_by_index
  0.00     15.00     0.00     1191     0.00     0.00  extract_ref_info
  0.00     15.00     0.00      951     0.00     0.00  extract_special_flags
  0.00     15.00     0.00      944     0.00     0.00  get_method_index_by_name_and_type
  0.00     15.00     0.00      944     0.00     0.00  lookup_method_definition
  0.00     15.00     0.00      647     0.00     0.00  extract_class_ref
  0.00     15.00     0.00      499     0.00     0.00  extract_code_attribute
  0.00     15.00     0.00      316     0.00     0.00  get_field_index_by_name_and_type
```

```
  0.00     15.00     0.00      316     0.00     0.00  lookup_field_definition
  0.00     15.00     0.00      250     0.00     0.00  heap_alloc_array
  0.00     15.00     0.00      250     0.00     0.00  heap_put_array
  0.00     15.00     0.00      216     0.00     0.00  make_array
  0.00     15.00     0.00      146     0.00     0.00  get_class
  0.00     15.00     0.00      129     0.00     0.00  calculate_field_offsets
  0.00     15.00     0.00      129     0.00     0.00  create_class
  0.00     15.00     0.00      129     0.00     0.00  jvm_init_class
  0.00     15.00     0.00      129     0.00     0.00  link_class
  0.00     15.00     0.00       39     0.00     0.00  dispatch_native_call
  0.00     15.00     0.00       34     0.00     0.00  create_array
  0.00     15.00     0.00       31     0.00     0.00  create_instance
  0.00     15.00     0.00       31     0.00     0.00  heap_alloc_instance
  0.00     15.00     0.00       31     0.00     0.00  heap_put_instance
  0.00     15.00     0.00       26     0.00     0.00  System_arraycopy8
  0.00     15.00     0.00       15     0.00     0.00  jvm_run
  0.00     15.00     0.00        9     0.00     0.00  class_instanceof
  0.00     15.00     0.00        3     0.00     0.00  ConsoleOutputStream_write_AB
  0.00     15.00     0.00        3     0.00     0.00  ConsoleOutputStream_write_I
  0.00     15.00     0.00        2     0.00     0.00  Configuration_get_core_count
  0.00     15.00     0.00        2     0.00     0.00  System_currentTimeMillis
  0.00     15.00     0.00        1     0.00     0.00  Configuration_get_threading_mode
  0.00     15.00     0.00        1     0.00     0.00  ValueConverter_double2string
  0.00     15.00     0.00        1     0.00     0.00  ValueConverter_float2string
  0.00     15.00     0.00        1     0.00     0.00  get_package_classes
  0.00     15.00     0.00        1     0.00     0.00  get_package_classes_count
  0.00     15.00     0.00        1     0.00     0.00  hashmap_create
  0.00     15.00     0.00        1     0.00     0.00  jvm_heap_init
  0.00     15.00     0.00        1     0.00     0.00  jvm_init
  0.00     15.00     0.00        1     0.00    14.31  jvm_init_linker
  0.00     15.00     0.00        1     0.00     0.00  jvm_init_threads
  0.00     15.00     0.00        1     0.00     0.11  jvm_run_args
```

This appendix only shows the *flat profile* given by `gprof`, which shows how much time is spent in each of the JVM's functions. It can be seen that the JVM spends 79.62% of its time in the interpreter (function `run`), which is a good result given that the benchmark performs a significant amount of heap access and `aload`/`astore` operations.

For additional information about the meaning of the different columns, please refer to the `gprof` documentation [27].

G Class Library

This appendix lists the classes of the Java Class Library [4] that have been implemented for the Barrelfish JVM. Most implementations are partial and supported features are given as appropriate. In addition to the classes from the Class Library, this section also gives classes of the Barrelfish JVM.

G.1 Java Class Library

Class	Comments
`java.io.FilterOutputStream`	full implementation
`java.io.OutputStream`	full implementation
`java.io.PrintStream`	`println`, `print`, etc.
`java.io.Serializable`	empty marker interface
`java.lang.ArithmeticException`	no exception handling
`java.lang.Class`	`forName`, `newInstance`, etc.
`java.lang.Double`	`equals`, `toString`, `valueOf`
`java.lang.Error`	no exception handling
`java.lang.Exception`	no exception handling
`java.lang.Float`	`equals`, `toString`, `valueOf`
`java.lang.Integer`	`equals`, `toString`, `valueOf`

Class	Comments
java.lang.InterruptedException	no exception handling
java.lang.Long	equals, toString, valueOf
java.lang.Math	wraps parts of libc
java.lang.Object	toString, getClass and equals
java.lang.Runnable	full interface
java.lang.RuntimeException	no exception handling
java.lang.StringBuffer	core functionality (append)
java.lang.StringBuilder	core functionality (append)
java.lang.String	core functionality (+ substring)
java.lang.System	out, arraycopy, currentTime...
java.lang.Thread	implements threading for JVM
java.lang.Throwable	empty marker interface
java.util.Dictionary	core functionality
java.util.Enumeration	core functionality
java.util.Hashtable	core functionality
java.util.Random	from documentation
java.util.Vector	core functionality

G.2 Barrelfish JVM Classes

All these classes are in the org.barrelfish.jvm package.

Class	Comments
Configuration	access to configuration settings
ConsoleOutputStream	command line output
CoreLocal	custom annotation
DistributedThread	thread on a different core (distributed)
DomainThread	thread on a different core (shared memory)
LocalThread	thread on the same core
Sticky	custom annotation
StringManager	create strings from constant pool entries
ValueConverter	convert between numerical values and strings

H

Project Proposal

The next pages present the original project proposal. Some aspects of the project have been further extended and evidence for this is given in the main part of the dissertation.

A JVM for the Barrelfish operating system
Part II Project Proposal - Martin Maas

1 Introduction and Description of the Work

Barrelfish is a research operating system developed at ETH, Zurich and Microsoft Research, Cambridge. It is based on the multikernel model [1], an OS structure that treats multicore systems as networks of independent nodes communicating via message-passing. This model arguably provides a better match for modern hardware: As multicore systems contain rising numbers of cores and are increasingly heterogeneous, providing a shared memory model becomes more difficult and expensive in terms of performance. At the same time, hardware is becoming more diverse, so that optimising an OS for different architectures can be prohibitively complex. Barrelfish aims to avoid these problems by making the OS hardware-neutral and viewing state as replicated instead of shared.

This project is concerned with the implementation of a Java Virtual Machine (JVM) for Barrelfish. It has been demonstrated that a JVM can be seen as a suitable abstraction for a heterogeneous multi-core system since it hides the distributed model and core heterogenity from the programmer. A JVM implementation on Barrelfish could be used for research in areas such as design of distributed JVMs on multicore systems, garbage collection in distributed systems and thread-scheduling for multicore systems.

The core part of the project involves the implementation of a feature-reduced Java Bytecode interpreter and bringing it up on Barrelfish. Extensions will explore enabling this JVM to support execution of threads on multiple nodes of a system, through both a traditional shared memory approach and through a distributed system approach similar in style to Barrelfish's share-nothing model.

2 Starting Point

1. Basic knowledge of *Java Bytecode* and the *Java Virtual Machine* from the Part IB courses Compiler Construction, Computer Design and Further Java.
2. Rudimentary knowledge of the *multikernel model*, the *Barrelfish operating system* and *distributed Java Virtual Machines* (from preliminary reading).
3. Successful *compilation* and *execution* of Barrelfish (on qemu).
4. Some *experimental code* written during the first weeks of Michaelmas term.

3 Substance and Structure of the Project

The core version of the JVM will support the following features of Java Bytecode: *Object allocation*, *field access*, *static method calls*, *simple virtual method calls*, *integer arithmetic* and *simple threading*. These features are sufficient for executing meaningful programs while leaving out parts that are not significant for research applications.

The core part of the project contains the following elements and features. Dependencies are implied by the order, but parallelisation is possible for some items.

Implementing a feature-reduced JVM on a traditional operating system (Linux)

- A basic *class loader* supporting `class_info`, `field_info` and `method_info` entries, constants (except for `CONSTANT_Float` and `CONSTANT_Long`) as well as ConstantValue and Code `attribute_info` entries. Classes will not be loaded at runtime, i.e. all required classes must be provided at start-up. Interfaces will be ignored and the verification stage will be omitted.
- The *main part of the interpreter*, including the *main loop*, the *method area* and the *call stack* (where frames contain the *program counter*, *local variables*, the *operand stack*, the *current class* and the *current method*).
- Support for static method invocations and related instructions (`invokestatic`, `*return`). This involves managing frames on the call stack.
- Support for instructions related to integer arithmetic (`iadd`, `isub`,...) as well as control transfer and conditionals (`ifeq`, `goto`,...).
- Support for `new` instruction and object creation.
- Support field access using the `getfield`, `putfield`,... instructions.
- Simple virtual method invocations (i.e. without inheritance).
- Facilities for basic file input/output (providing native, partial implementations of `java.io.FileOutputStream` and `java.io.FileInputStream`) as well as simple terminal output for debug information.
- Threading support via a native, partial implementation of `java.lang.Thread` (only providing the run and join methods).
- A set of tests to verify the correctness of the JVM. This includes writing simple Java programs and comparing their results to the ones given by a reference JVM implementation (Sun/Oracle Java SE 6).
- Benchmarks to evaluate performance compared to the reference implementation.

Bringing up the JVM on a single Barrelfish node.

- Adapting the code to work with peculiariaties of Barrelfish such as file handling, I/O and threading.
- Providing facilities to load class files, either using basic file system support or bundling the class files with the JVM executable.
- Evaluating success using the tests written previously. Use the benchmarks to compare the performance of the JVM running on Barrelfish vs. the JVM running on Linux (within the same setup, i.e. QEMU).

In addition to this core part, there are a number of extensions to the project that will be attempted if the implementation of the core part leaves enough time:

- Making the JVM run distributedly on multiple nodes using shared memory.
  - Facilities to spawn threads on different nodes. Threads will be evenly distributed across the available cores without any advanced scheduling policy.
  - Synchronization and a shared heap between the threads using mechanisms that are already provided by Barrelfish.
- Implementing the shared heap based on a distributed model.
- Evaluating the correctness of the two implementations and compare their performance. This requires the implementation of additional tests and benchmarks.

Further features of Java Bytecode (e.g. arrays) will be implemented as necessary. If it turns out to be sensible, a simple mark-and-sweep Garbage Collector running on one node might be implemented as well.

4 Success Criterion

The core parts of the project will have been a success if the JVM:

1. runs on the Barrelfish operating system
2. can load a set of classes and run a specified entry-point method
3. gives the same result as the reference JVM on all of these calculations

This refers to classes and bytecode only using the specified features. Success will be evaluated

5 Timetable and Milestones

Note that this timetable allows sufficient time for revision during the vacations. Unassigned time slots might also be used to work on extensions.

Block 1: 7 October - 22 October
Discuss project with supervisors, write project proposal, preliminary reading, familiarise with Barrelfish and the JVM Specification, run small test applications on Barrelfish.

MILESTONES: Project proposal, test applications

Block 2: 23 October - 5 November
Load class files, execute basic bytecode (integer arithmetic, run static methods, read/write static fields, execute control transfer instructions), write simple Java programs and tests.

MILESTONES: Basic JVM that supports these features

Block 3: 6 November - 19 November
Implement object allocation, virtual method-calls, non-static fields, rudimentary input/output, basic threading, write Java programs and tests to verify these features.

MILESTONES: JVM supports these features.

Block 4: 20 November - 3 December
Write benchmarks and measure performance, Time to catch up, fix bugs, tidy up code.

MILESTONES: JVM supports dynamic linking and Linux-JVM code is finalised.

Block 5: 4 December - 17 December
Make this JVM run on a single Barrelfish node and verify correctness using the tests.

MILESTONES: JVM running on a single Barrelfish node.

Block 6: 1 January - 14 January
Finish migration to Barrelfish, perform evaluation, write dissertation outline.

MILESTONES: Barrelfish-JVM code is finalised, a suite of test cases and benchmarks for evaluation, notes for the structure and the content of the dissertation.

Block 7: 15 January - 28 January
Start writing introduction and preparation chapter of the dissertation, work on extensions.

MILESTONES: Introduction and preparation chapter are finished.

Block 8: 29 January - 11 February
Create progress report and presentation, work on extensions and dissertation.

MILESTONES: Introduction and preparation chapter are finished.

Block 9: 12 February - 25 February
Work on extensions and dissertation.

MILESTONES: Implementation and evaluation chapter are finished.

Block 10: 26 February - 11 March
Complete dissertation draft, incorporate extensions, finalise code.

MILESTONES: Dissertation draft and code are finished.

Block 11: 23 Apr - 6 May
Finalise the dissertation and proof-reading.

MILESTONES: Dissertation completed and ready to hand in.

Block 12: 7 May - 20 May
This block is intended as a buffer for any serious issues.

MILESTONES: Dissertation handed in by the deadline.

References

[1] A. Baumann, P. Barham, P. E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: A new os architecture for scalable multicore systems. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44, New York, NY, USA, 2009. ACM.