Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor's Thesis Nr. 52b

Systems Group, Department of Computer Science, ETH Zurich

Multicore ARMv7-a support for Barrelfish

by

Samuel Hitz

Supervised by

Prof. Timothy Roscoe, Adrian Schuepbach

March 2012 – August 2012

Informatik
Computer Science

Abstract

Commodity computer systems contain more and more specialized hardware tailored to perform certain tasks with optimal efficiency and power consumption. Due to its relatively simple RISC (Reduced Instruction Set Computer) based instruction set, which allows for cheap and highly optimised hardware with low power consumption, ARM based chips have become increasingly popular.

This thesis describes the port of Barrelfish, a research operating system developed at ETH Zurich, to a multi-core ARMv7-a architecture simulated by Gem5. Barrelfish's multikernel architecture inherently supports heterogeneous processor systems and with Gem5 we target a very interesting hardware simulator. It allows us to simulate a wide range of system configurations, from simple single-core one-cycle-per-instruction CPU system without caches, to a possibly heterogeneous, multi-core, pipelined, out-of-order CPU system with arbitrarily deep levels of caches.

We managed to bring up Barrelfish on Gem5 supporting up to four cores interacting with each other. The evaluation of our port shows, at least qualitatively, that it performs comparably to the x86-64 port of Barrelfish running on Gem5.

Contents

1 Introduction and Motivation

With today's trend towards highly specialized, low power consuming systems, ARM based chips have become increasingly popular. Its relatively simple RISC (Reduced Instruction Set Computer) based instruction set allows for cheap and highly optimised hardware with low power consumption [15]. The extensive use of ARM-based platforms in the embedded space has led to a wide variety of different configurations. The use of heterogeneous and performance asymmetric processor cores, such as the recently announced ARM big.LITTLE architecture [10], makes it an attractive target for Barrelfish. In particular, it provides a good environment to evaluate the multikernel's inherent support for heterogeneous processor systems.

With Gem5 we target a very interesting hardware simulator. It allows us to simulate a wide range of system configurations, from simple single-core one-cycle-per-instruction CPU system without caches, to a, possibly heterogeneous multi-core, pipelined, out-of-order CPU system with arbitrarily deep levels of caches.

This thesis describes the port of Barrelfish to a multi-core ARMv7-a architecture and will serve as a basis for further research into low-power OS design and support for heterogeneous many-core systems. Our approach is to first port the existing single-core ARMv5 Barrelfish port to the ARMv7-a architecture and in a further step, add support for multiple cores to it. In our attempt of doing so we try to answer, among others, the following questions:

- What needs to be changed to bring up the current Barrelfish ARMv5 port on a ARMv7-a platform?
- How can we extend this basis to support multiple cores?
- What changes do we need to make to inter-core message passing drivers?
- How does our system perform compared to the existing x86-64 port on Gem5?

Section 2 gives an overview of related work in this area. Section 3 gives some background information about Barrelfish, the ARMv7-a architecture and Gem5, the architecture simulation environment used in this thesis. In section 4 we go into the design decisions and implementation of the port. Evaluation results are presented in section 5 and the last two sections include conclusions and an outlook on possible future work.

2 Related Work

It is hard to find related work, given the nature of the topic. Nevertheless we will list some related papers and ARM ports here.

Bram Scheidegger describes in *Barrelfish on Netronome* [21] the port of Barrelfish to the ARMv5 Intel XScale based Netronome network card. He got the CPU driver working and describes the port from a little endian to a big endian system as being tricky.

In *Booting ARM Linux SMP on MPCore* [5], Charly Bechara describes the boot up process of Linux on multiple ARM cores, which helped us in understanding how to boot an additional core on an ARM platform.

With the increasing importance of the ARM architecture in consumer electronics, mainly phones, tablets and embedded devices, there have been many ports of existing operating systems to ARM. These include Linux (e.g. Arch Linux [2]), FreeBSD [19], the L4 microkernel [13] and with the recent announcement of Windows RT, even Microsoft Windows [23].

3 Background

3.1 Barrelfish

Barrelfish is a research operating system developed at ETH Zurich in collaboration with Microsoft Research [3]. It distinguishes itself from 'traditional' operating systems like Linux or Windows through various design principals which we describe in this section. Most of the following information about Barrelfish is from *The multikernel: a new OS architecture for scalable multicore systems* [3].

3.1.1 The multikernel model

The multikernel model is an OS architecture for heterogeneous multicore machines [3]. It is guided by three design principles:

1. Make all inter-core communication explicit.
2. Make OS structure hardware neutral.
3. View state as replicated instead of shared.

To achieve these principles, Barrelfish employs multiple independent OS instances, which communicate via explicit message passing [4]. Each instance consists of a privileged-mode *CPU driver* and a user-mode *monitor* process. The CPU driver is architecture specific and handles interrupts, enforces memory protection via MMU and timeslices processes. Monitors handle inter-core communication, coordinate system-wide state and encapsulate much of the functionality found in the kernel of a traditional OS. Since the monitor runs in user-space and all the hardware specific functionality are handled by CPU- and device drivers, it is (mostly) hardware independent.

3.1.2 Message Passing Interface

Barrelfish does not rely on shared memory and cache coherence mechanisms, but uses message passing for inter-core communication. Depending on the hardware platform, this can be implemented using shared memory and cache coherence, but could also use other mechanisms available. Barrelfish distinguishes between two types of messages, intra-core messages, which are handled by the CPU driver's lightweight inter-process communication interface and inter-core messages, which take place in the user-space and are handled by the monitor using a variant of user-level RPC (URPC) [6].

3.1.3 Process structure

Processes in Barrelfish have a different structure than a typical monolithic OS, due to its multikernel model. A process is represented by a collection of *dispatcher* objects. Each core on which the process might execute has one dispatcher object for a given process. Dispatchers on a core are scheduled by the corresponding CPU driver [18]. Each dispatcher has a core-local user-level thread scheduler, so threads are not supported by kernel threads.

3.1.4 Memory management

Barrelfish uses a capability system for memory management [12]. All memory management is performed by manipulating capabilities through system calls. Capabilities itself are user-level references to kernel objects and regions of physical memory. Virtual memory management is completely done in user-level code except the things that need privileged rights, such as actual page table manipulation, which are done by the CPU driver. Since physical memory is a global resource, capabilities management has to be coordinated between cores.

3.1.5 System knowledge base

As a multikernel OS, and as such predestined for heterogeneous hardware, Barrelfish needs to choose appropriate system mechanisms. The *system knowledge base* [22] maintains knowledge of the underlying hardware. It is populated with information gathered through hardware discovery, latency measurements from IPC and some hard-coded facts that can not be discovered or measured. This information can be used e.g. for managing core diversity, by scheduling tasks to cores with specific features which benefits the execution of this application.

3.2 Gem5

The Gem5 [7] simulator combines the best aspects of the M5 [8] and GEMS [16] simulators. With its flexible and highly modular design, Gem5 allows the simulation of a wide range of systems. Gem5 supports a wide range of ISAs like x86, SPARC, Alpha and, in our case most importantly, ARM. In the following we will list some features of Gem5.

3.2.1 CPU models

Gem5 supports four different CPU models: AtomicSimple, TimingSimple, InOrder and O3.

The first two are simple one-cycle-per-instruction CPU models. The difference between the two lies in the way they handle memory accesses. The AtomicSimple model completes all memory accesses immediately, whereas the TimingSimple CPU models the timing of memory accesses. Due to their simplicity, the simulation speed is far above the other two models.

The InOrder CPU models an in-order pipeline and focuses on timing and simulation accuracy. The pipeline can be configured to model different numbers of stages and hardware threads.

The O3 CPU models a pipelined, out-of-order and possibly superscalar CPU model. It simulates dependencies between instructions, memory accesses, pipeline stages and functional units. With a load/store queue and reorder buffer its possible to simulate superscalar architectures as well as multiple hardware threads.

3.2.2 Execution modes

Gem5 provides two different execution modes: system-call emulation (SE) and full-system (FS).

In system-call emulation Gem5 emulates most system calls by just passing it to the host operating system. It is not possible to run privileged code in SE mode.

In contrast the full-system emulation mode simulates a complete system suitable for running an OS. Therefore interrupts, exceptions, privileged levels and some devices like UART, interrupt controllers, timers and network interfaces are supported.

3.2.3 Python integration

The Gem5 simulator provides a tight integration of Python into the simulator. Python is mainly used for system configuration. Every simulated building block of a system is implemented in C++ but are also reflected as a Python class and derive from a single superclass *SimObject*. This provides a very flexible way of system construction and allows to tailor nearly every aspect of the system to our needs.

Python is also used to control the simulation, taking and restoring snapshots as well as all the command line processing.

3.3 ARM

ARM is a 32-bit reduced instruction set computer (RISC) instruction set architecture (ISA). The following are the key characteristics of the architecture [15]:

1. a large uniform register file
2. a load/store architecture, meaning data-processing is only possible on register contents not memory content
3. simple addressing modes
4. instructions that combine a shift with an arithmetic or logical operation (barrel shifter)
5. auto-increment/-decrement addressing modes
6. load/store multiple data instructions to maximize data throughput
7. conditional execution of instructions to maximize execution throughput

Its important to note that ARM is not a processor but only a CPU core design and instruction set architecture, which can be licensed by manufacturer to create their own ARM based chips. The current version of the architecture is ARMv7 and it distinguishes three profiles:

- Application profile: ARMv7-a and Cortex-A series

- Real-time profile: ARMv7-r and Cortex-R series
- Microcontroller profile: ARMv7-m and Cortex-M series

Since we are porting a general purpose operating system, we will focus on the application profile ARMv7-a, although many concepts are found in any of the three profiles.

3.3.1 Registers and register banking

The ARM architecture provides 16 core registers, r0-r12, the stack pointer (SP), the link register (LR) and the program counter (PC). These registers are selected from a larger set of registers, that includes banked copies of some registers, with the current register selected by the execution mode. Additionally there is a current program status register (CPSR) and in some modes a saved program status register (SPSR). These hold information about the current CPU mode, flags set by the ALU, masked interrupts etc. [15]

3.3.2 CPU modes

The ARM architecture defines several CPU modes. The processor can only be in one mode at a time. [15]

User mode User mode is the only unprivileged mode. An operating system typically runs application in user mode to protect system resources from unprivileged access.

System mode Software executing in System mode is privileged. System mode has the same registers available as User mode and is not entered by any exception.

Supervisor mode The supervisor mode is entered after a software interrupt occurred. SP and LR registers are banked.

Abort mode Abort mode is the default mode to which a Data Abort exception or Prefetch Abort exception is taken. SP and LR registers are banked.

Undefined mode Undefined mode is the default mode to which an instruction-related exception, including any attempt to execute an undefined instruction, is taken. SP and LR registers are banked.

IRQ mode IRQ mode is the default mode to which an IRQ interrupt is taken. IRQ interrupts are typically interrupts from 'slow' devices. SP and LR registers are banked.

FIQ mode FIQ mode is the default mode to which an FIQ interrupt is taken. FIQ interrupts are typically interrupts from fast, low latency devices, therefore r8-r12 are additionally banked, used for passing arguments to the interrupt handler without the need of stack manipulations.

3.3.3 MMU and page table layout

The ARMv7-a architecture includes a memory management unit (MMU) which translates virtual to physical addresses. There is also an associated translation lookaside buffer (TLB), which caches frequently used page table entries to speed up address translation. ARM uses 32-bit addresses, so the address space consists of 2^{32} bytes (4GB). The whole address space is divided into sections and pages and a two level page table is used for address translation.

L1 page table The L1 page table divides the 4GB address space into sections of 1MB each. There are two types of L1 page table entries, either it maps the whole section directly or it is a pointer to a L2 page table, which allows a finer grained control over memory. A L1 page table entry has a size of 4 bytes and since there are 4096 sections the L1 page table consumes 16KB of memory.

L2 page table A L2 page table divides the 1MB section further into either large pages of 64KB or small pages of 4KB size each. There are also tiny pages of 1KB size, but those are deprecated in ARMv7. Each L2 page table entry is 4 bytes and contains beside the page base address also memory access bits and other information, such as whether data of this page should be cached or executed. There are 256 L2 page table entries per table, thus a L2 page table consumes 1KB of memory.

3.3.4 Exception handling

Like the x86 architecture, ARM uses an exception vector at a fixed location as an entry point for exception handling. Unlike the exception vector in x86, which contains the address to the appropriate exception handler, the ARM exception vector contains the actual jump instruction. Each possible exception has a fixed offset in the exception vector and when the CPU enters an exceptional state it executes the instruction at EX_VECTOR_BASE + EX_OFFSET. The exception vector is located at address 0x00000000 by default, but can be relocated to address 0xffff0000.

3.3.5 Generic Interrupt Controller

The ARM GIC architecture splits logically in two parts: the distributor and one or more CPU interfaces. The distributor performs interrupt prioritization and distribution to the CPU interfaces that connect to the processors in the system [14]. It provides a programming interface for:

- globally enabling forwarding of interrupts to the CPU interfaces
- enabling/disabling each interrupt
- setting priority and target processors of each interrupt
- sending a software generated interrupt to one or more processors
- getting the state of each interrupt
- set/clear the pending state for a peripheral interrupt

A CPU interface provides the interface for a processor that operates with the GIC. Each CPU interface provides a programming interface for:

- enabling the signalling of interrupt requests by the CPU interface
- acknowledging an interrupt
- indicating completion of the processing of an interrupt
- setting priority mask for the processor
- defining preemption policy for the processor
- determining pending interrupt with the highest priority

When an interrupt gets forwarded to the CPU interface by the distributor, it determines whether the interrupt has sufficient priority to be signaled to the processor, by using the interrupt priority mask and preemption settings of the CPU interface.

Interrupts from sources are identified by ID numbers. Each CPU interface can see up to 1020 interrupts. Interrupt numbers ID0 - ID31 are private to a CPU interface and therefore banked at the distributor. ID numbers ID32 - ID1019 can be assigned to peripheral interrupts and are global.

3.3.6 Coprocessors

The ARM architecture supports sixteen coprocessors, usually referred as CP0 to CP15. Coprocessors 8 to 15 are currently used or reserverd for future use by the architecture. Coprocessors 0 to 7 can be used by vendors to provide vendor-specific features. CP15 is the most important and provides system control functionality.

4 Design and Implementation

In this section we will describe the design and implementation of the Barrelfish ARMv7-a port. First we will show what it takes to bring Barrelfish up on a single-core ARMv7-a architecture, simulated by Gem5. In a second step, we will enhance our design to boot multiple CPUs and focus on the changes which had to be made to the single-core implementation.

Since there was already an existing ARMv5 port of Barrelfish for QEMU and the Intel XScale architecture, we could build our work upon those achievements. However, since the ARMv7-a architecture differs significantly from the ARMv5 and Gem5 was a new simulating environment with its own quirks and flaws, lots of changes had to be made. We could still profit from the overall boot up structure given by those ports.

4.1 Single-core implementation

Our first goal was to rewrite the existing ARMv5 port of Barrelfish for QEMU to support the ARMv7-a architecture. By rewriting this existing port we could focus on the architecture specific parts, since the overall boot up structure was already given. Another reason to first target just a single core was that several things are greatly simplified compared to a system with multiple processors. These include, among others:

- No explicit inter-core communication
- No other core interfering with physical memory
- The boot up protocol is greatly simplified
- Critical system resources can be protected by disabling interrupts

4.1.1 High-level boot up process overview

This section gives a high-level overview of the boot up process of the Barrelfish kernel on ARMv7-a. In subsequent sections we will go more into details involved in the single steps.

1. Setup kernel stack and ensure privileged mode
2. Allocate L1 page table for kernel
3. Create necessary mappings for address translation
4. Set translation table base register (TTBR) and domain permissions
5. Activate MMU, relocate program counter and stack pointer
6. Invalidate TLB, setup arguments for first C-function arch_init
7. Setup exception handling
8. Map the available physical memory in the kernel L1 page table
9. Parse command line and set corresponding variables
10. Initialize devices
11. Create a physical memory map for the available memory
12. Check ramdisk for errors
13. Initialize and switch to init's address space
14. Load init image from ramdisk into memory
15. Load and create capabilities for modules defined by menu.lst
16. Start timer for scheduling
17. Schedule init and switch to user space
18. init brings up the monitor and mem_serv
19. monitor spawns ramfsd, skb and all the other modules

4.1.2 Memory Layout

Like many other popular operating systems, Barrelfish employs a memory split. The idea behind a memory split is to separate kernel code from user space code in the virtual address space. This allows the kernel to be mapped in every virtual address space of each user space program, which is necessary to allow user space code to access kernel features through the system call interface. If the kernel was not mapped into the virtual address space of each program, it would be impossible to jump to kernel code without switching the virtual address space. Additionally ARMv7-a provides two translation table base registers, TTBR0 and TTBR1. We can configure the system to use TTBR0 for address translations of virtual addresses below 2GB and TTBR1 for virtual address above 2GB. This saves us the explicit mapping of the kernel pages into every L1 page table of each process.

Even though the kernel is mapped to each virtual address space, it is invisible for the user space program. Accessing memory, which belongs to the kernel, leads to a pagefault. Since many mappings can point to the same physical memory, memory usage is not increased by this technique.

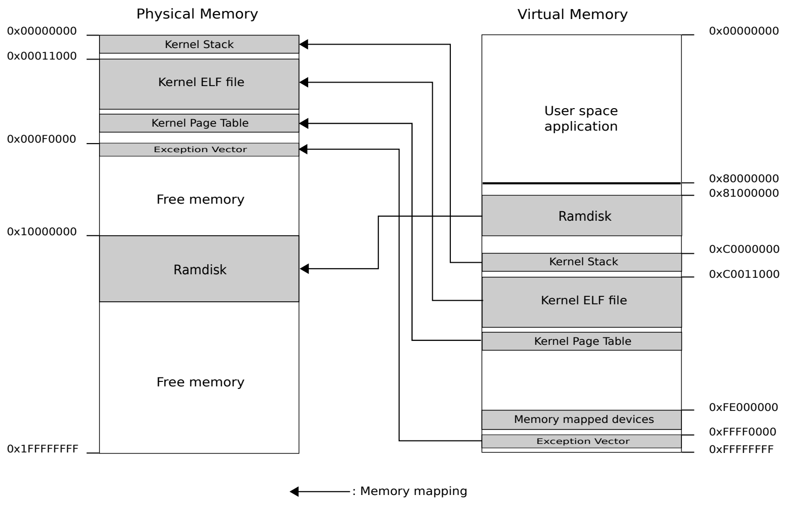

Figure 1: Barrelfish memory layout

Figure 1 shows the memory layout of the single-core ARMv7-a port of Barrelfish. We have a memory split at 2GB, where everything upwards is only accessible in privileged mode and the lower 2GB of memory is accessible for user space programs. The position of the kernel in the virtual space is hardcoded. The L1 page table of the kernel address space is located right after the kernel and aligned to 16KB. We map the whole available physical memory into the kernel's virtual address space. The locations for the 64KB kernel stack, the ramdisk aswell as the section for memory mapped devices are hardcoded. Overall the memory layout is very static and turned out to be problematic with regard to multicore support, which will be explained in section 4.2.

4.1.3 MMU Setup

In order to turn on the MMU, we first need to set up the L1 page table and map the kernel section to the corresponding section in physical memory. We also map the section containing the high memory relocated exception vector to the kernel section. We use that when we later set up exception handling. After the MMU is turned on, we need to relocate the instruction pointer to point to the next instruction in virtual memory. We can easily let the linker do the job for us, by creating a label pointing to the next instruction after turning on the MMU. The stack pointer needs also to be relocated in order to be able to access the stack.

Controlling the MMU is done via the system control register accessible through coprocessor 15. We first read out the current configuration, set the MMU-enable bit as well as the bits for enabling the instruction and data cache and alignment checking and write the resulting value back to the system control register.

Due to the pipeline in ARM processors the CPU will already have fetched two instructions by the time the MMU gets active. We solve this problem by inserting a NOP operation after relocating the instruction pointer. Listing 1 shows the MMU setup procedure.

```
start_mmu_config:
  mrc    p15, 0, r0, c2, c0, 2       // read out TTBCR
  orr    r0, r0, #1                  // VA >= 2GB are now translated
  mcr    p15, 0, r0, c2, c0, 2       // with L1 table saved in TTBR1.

  mcr    p15, 0, r8, c2, c0, 0       // store L1 address temporarily
                                     // in TTBR0 (for 1:1 mapping)
  mcr    p15, 0, r8, c2, c0, 1       // store L1 address in TTBR1
  ldr    r0, =0x55555555             // Initial domain permissions
  mcr    p15, 0, r0, c3, c0, 0

  ldr    lr, =$start_with_mmu_enabled     // Address to continue at
  ldr    r0, =KERNEL_OFFSET               // when paging is enabled
  add    sp, sp, r0                       // relocate stack
  sub    sp, sp, r9

  ldr    r1, =0x1007                 // Enable: D-Cache, I-Cache
                                     // Alignment, MMU
  mrc    p15, 0, r0, c1, c0, 0
  orr    r0, r0, r1
  mcr    p15, 0, r0, c1, c0, 0       // MMU is enabled
  mov    pc, lr                      // relocate program counter
  mov    r0, r0
```

Listing 1: MMU set up procedure

4.1.4 Context Switch

Due to the memory split, a context switch is easily implemented. Process A makes a system call or gets preempted and is thus executing kernel code in its virtual address space. We save the process state (register values) in the corresponding dispatcher control block (DCB) and switch into the virtual address space of process B. Since we implemented the memory split with the use of TTBR0 and TTBR1, we effectively just write the address of B's L1 page table into TTBR0, but we are still in high memory, which gets translated by TTBR1. We have to flush the TLB, to get rid of cached values of the old mapping. Now we can restore the state of process B and resume it by updating the program counter accordingly.

4.1.5 Exception Handling

The exception vector clearly belongs to the kernel space. As it is by default located at address 0x0000000, we first relocate it to its high memory address (0xffff0000) to be consistent with the memory split. Since the CPU mode, in which we handle the exception, is dependant on the type of the exception and each CPU mode has its own copy of the stack pointer, we have to setup a small stack for each mode.

Some difficulties arise by the fact that on ARM the exception vector contains actual instructions, instead of jump addresses like on x86. Normal ARM instructions are encoded in 32 bit, but this also includes the op code. Since the whole virtual address space is 4GB (32 bit) we have to employ a little trick to be able to jump to any address. At a constant offset to the exception vector we create a jump table containing the addresses of the corresponding exception handlers. In the exception vector itself we use the LDR instruction, which allows us to load a value from a pc-relative address into a register. We use this instruction to update the program counter with the address of the exception handler. Listing 2 shows the code for filling the exception vector with the right instructions and listing 3 shows a memory dump of the exception vector and the jump table (with constant offset 0x100).

```
exceptions_install_handler:  //(r0 = exoffset, r1 = addr of handler)
  ldr   r2, =ETABLE_ADDR                  // store handler address at
  add   r2, r2, #JUMP_TABLE_OFFSET        // ETABLE_ADDR + JT_OFF
  add   r2, r2, r0                        // + exception_offset
  str   r1, [r2]
  ldr   r2, =ETABLE_ADDR                  // load constants
  ldr   r3, =LDR
  mov   r4, #JUMP_TABLE_OFFSET
  add   r2, r2, r0                        // add offset to base
  sub   r4, r4, #0x8                      // subtract 8 bc of pipeline
  orr   r3, r3, r4                        // =LDR pc,<addr of handler>
  str   r3, [r2]                          // store instt in ex vec
```

Listing 2: Exception vector setup

```
   0xffff0000:  b      0xffff0000
   0xffff0004:  ldr    pc, [pc, #248]   // pc = value at 0xffff0104
   0xffff0008:  ldr    pc, [pc, #248]   // pc = value at 0xffff0108
   0xffff000c:  ldr    pc, [pc, #248]   // pc = value at 0xffff010c
   0xffff0010:  ldr    pc, [pc, #248]   // pc = value at 0xffff0110
   0xffff0014:  b      0xffff0014
   0xffff0018:  ldr    pc, [pc, #248]   // pc = value at 0xffff0118
   0xffff001c:  ldr    pc, [pc, #248]   // pc = value at 0xffff011c

   0xffff0104: <addr of undef handler>
   0xffff0108: <addr of swi handler>
   0xffff010c: <addr of pabt handler>
   0xffff0110: <addr of dabt handler>
   0xffff0118: <addr of irq handler>
   0xffff011c: <addr of fiq handler>
```

Listing 3: Exception vector memory dump

The various assembly level exception handlers itself basically save the current context (eg. the faulting instruction in case of a page fault) and pass the arguments they received along to the C level exception handlers, which do the actual exception handling.

4.1.6 Devices

In order to provide basic functionality like basic i/o, generating and handling interrupts and timer based scheduling, we had to implement several device drivers.

For a large part we used Mackerel [20], a domain-specific language for describing hardware devices. The Mackerel language is designed to simplify the transcription from a hardware data sheet into code. Once this transcription process is done for a particular device, Mackerel generates code for accessing and manipulating the specified registers, typically in form of a large C header file consisting of a large number of inline functions. Mackerel uses shift operations to create the appropriate values and is therefore endian independent.

UART Gem5 simulates the same UART device for the ARM architecture as QEMU and therefore we did not have to write a separate device driver, but could use the existing of the Barrelfish ARMv5 port with some minor modifications. The serial driver is very simple. It has the ability to send/receive a single character to/from the serial port. To print a string we have to loop over the characters and print each character separately.

In order to set up the serial subsystem, we map the device base address into the kernel virtual address space and initialize the device with appropriate configuration values. We actually initialize two UART devices in order to provide a normal serial console and a debug console.

Interrupt Controller The simulated interrupt controller complies to the ARM GIC architecture [14]. We first map it into the kernel's virtual address space and extract some implementation specific information about the interrupt controller, like the number of interrupts supported and the number of active CPU interfaces. We then set up the CPU interface by setting the priority mask to the lowest priority, since we want every interrupt forwarded to our single CPU, and enabling the forwarding from the CPU interface to the processor. Finally we globally enable the forwarding of interrupts from the distributor to the CPU interface.

To enable an interrupt we provide the pic_enable_interrupt-function. Listing 4 shows the prototype of this function.

```
void pic_enable_interrupt(uint32_t int_id,
                          uint8_t cpu_targets,
                          uint16_t prio,
                          bool edge_triggered,
                          bool one_to_n)
```

Listing 4: Interrupt enable prototype

int_id is the ID number of the interrupt. cpu_targets is a one byte bit-field where each one means the interrupt should be forwarded to the corresponding CPU interface. In our single-core implementation the value of this parameter will always be one for obvious reasons. prio denotes the interrupt priority. egde_triggered selects whether the interrupt should be edge-triggered or level-sensitive. An edge-triggered interrupt is asserted on detection of a rising edge of an interrupt signal and remains asserted until it is cleared. An interrupt is level-sensitive if it is asserted whenever the interrupt signal level is *high* and deasserted whenever the level is *low*. The last parameter has no importance in a single-core system, but we will come back to it, when we describe the multi-core implementation.

When an interrupt gets forwarded to the processor, we first acknowledge it by reading the *Interrupt Acknowledge Register* in the CPU interface, which returns us the interrupt ID. When the processor acknowledges the interrupt at the CPU interface, the Distributor changes the status of the interrupt from pending to active. At this point the CPU interface can signal another interrupt to the processor, to preempt interrupts that are active on the processor. If there is no pending interrupt with sufficient priority for signalling to the processor, the interface deasserts the interrupt request signal to the processor.

When the interrupt handler on the processor has completed the processing of an interrupt, it writes to the *End of Interrupt Register* to indicate interrupt completion. When this happens, the distributor changes the status of the interrupt to inactive.

Timers In our implementation we have two kinds of timers, a peripheral timer, which we use for preemptive scheduling and a CPU local timer, which we use as a time stamp counter for debugging and benchmarking purposes. Since these are different devices we had to implement a device driver for each one of them.

The scheduling timer is a periodic timer, meaning it gets automatically reloaded with the initial load value after counting down to zero, which is what we want for scheduling. The load value can be configured by the user with a kernel command line parameter. The timeslice kernel command line parameter tells the kernel in which interval (in ms) a new process should get scheduled. The Timer is connected to the Amba Peripheral Bus (APB) clock and with each APB clock cycle the timer value gets decremented by one. We can therefore calculate the load value with

$$load\_val = timeslice \cdot APB\_CLOCK\_FREQ \ / \ 1000$$

At last we enable the timer interrupt with the function described in the previous section. Note that the timer has not started yet and we do that at a later stage, since we do not need scheduling at this stage.

We initialize the time stamp counter to be auto reloading, load it with the maximal possible value and start it immediately. When we read out the timer value we negate it since the timer is counting down but we want increasing timestamps.

4.1.7 Ramdisk

All the user space programs are stored in a ramdisk, because we do not support a hard disk yet. The ramdisk gets mapped into memory by Gem5 at a fixed location. Normally the boot loader would pass the location and the size of the ramdisk to the kernel via ATAG headers [24].

Unfortunately Gem5 does not set the corresponding ATAG header so we hardcoded the location into the code. Since the size of the ramdisk is constantly changing whenever there are changes to some of the modules, we use a script which hooks into the compilation system. It first compiles every module, creates the ramdisk and writes the size into a header file. Finally it compiles the kernel again to include the updated size information. Note this script was already available from previous ARM ports which ran into the same problems.

4.1.8 Setting up user space

In order to launch the first user space application *init*, we need to prepare its environment first:

1. Allocate and populate pagetables and switch to inits address space
2. Set up important capabilities for init
3. Initialize and map the bootinfo struct, which holds essential information for the init process to allocate and manage its address space
4. load the init image from the ramdisk into memory
5. Prepare command line arguments for init
6. Creating and initializing a DCB
7. Allocating memory and creating memory capabilities for all modules in the ramdisk
8. Schedule the DCB

init will further initialize the user space and eventually start *mem_serv* and the monitor process. The monitor takes over from then on, since it is the unprivileged user space counterpart of the CPU driver and will bring up all the other modules.

For the single-core port we were done now. Most of the user space modules are not architecture specific or did not need any changes to work.

4.1.9 Problems encountered with Gem5 and the ARM GCC

This section is dedicated to document some of the many difficulties we encountered using the Gem5 simulator and the ARM GCC. They are not directly related to porting Barrelfish to ARMv7-a, but we feel we should also document them because other people building up on our work and working with those tools could learn from our solutions to those problems and because we had to invest a large amount of time to find and fix them.

Gem5 is a research simulator and therefore under heavy development, meaning lots of changes in a relative short amount of time. We had to constantly rewrite our Gem5 system configuration script, since a lot of interfaces changed and new features were added during our time working with Gem5. At one point we decided to fix a specific release of Gem5 to avoid constantly updating our system script and deal with new breakages in the Gem5 code.

The ARM simulation on Gem5 is very Linux specific, it does some things which we had to patch out, since they did not work with Barrelfish. Another problem was that Gem5 did not handle interrupts on ARM correctly. There was an error with bounds checking on interrupt numbers, which caused some interrupts to never fire. We ended up with a fixed release and patches for those problems which everyone has to apply to make Barrelfish run on Gem5.

To spot these problems we could debug Gem5 with an ordinary debugger and step the code to see, what exactly is going on inside Gem5. Usually we had some clues from the errors where we had to search, otherwise this approach would barely be possible, since Gem5 is a fairly complex software system.

The second tool which caused problems was the ARM compiler we used. On ARMv7-a every access to memory has to be aligned to a four byte boundary. This caused troubles when accessing data in a struct. Because of a bug in the compiler not every element of a struct was aligned to four bytes, leading to a data abort when accessing them. To fix the problem we had to manually force the alignment to four bytes of members of certain structs. Fortunately, these kinds of problems were easily isolated, since Barrelfish dumps the register contents and the faulting instruction. The difficulty here was to know about the alignment constraint of ARMv7-a in the first place.

In order to debug Barrelfish we compiled the code with optimizations turned off. After getting everything running we turned them on again, just to encounter a series of very strange errors. After countless hours of debugging at assembly level (since debugging the C code with optimizations turned on is fruitless with these kinds of problems) and analysing the problems by using the tracing ability of Gem5, where we could see each operation that got executed on the CPU, we saw that the compiler had problems with some inline functions. The easy fix was just to declare them as non-inline at the cost of losing performance, but that was not an issue for us.

All these problems were extremely hard to spot, since we had to rule out any possibility of the Barrelfish code failing, before turning to our tools and search for the error there. We hope we can give other people working with Barrelfish and those tools a starting point of where to look for possible sources of error.

4.2 From single-core to multi-core

After reaching our first goal, bringing up Barrelfish on Gem5 using a single core, we focussed on implementing multi-core support. Right at the beginning we had a fundamental design choice to make. The current ARM port was very static, meaning it relied on being loaded at a specific location in memory, had a lot of hardcoded addresses and was just not flexible. One approach would have been to build upon these prerequisites, which would have implied fixed hardcoded memory locations for the different kernel images, a lot of mapping hacks to support global shared data between kernels, since the kernel image could not be relocated and a very inflexible framework for future extensions.

Another approach was to rewrite the single-core port to support dynamic kernel relocations to be able to dynamically allocate the space for the kernel image and put it in memory where ever we want to. This simplifies a lot since the kernel images can be at different locations in the virtual address space. Despite the higher initial effort needed, we decided to go with the second approach, since we think it will also be good for future expansions. Another bonus with this approach was to bring the ARM port much more in line with the x86 port, which makes it easier for people working with the x86 port understanding the ARM port and simplifies the process of adding new functionality to both ports in parallel.

4.2.1 Writing a second stage boot loader

As a consequence of our decision to support dynamic kernel relocations, we also had to let GCC generate a relocatable kernel image. Even though Gem5 supports the loading of ELF binaries it does not support relocations, which needs to be done by the loader in order to resolve the missing, non position independent references. It also does not load important sections and the section header table needed to perform dynamic relocations. To overcome these shortcomings we decided to write a second stage boot loader. This also allowed us to mimic a multiboot [9] compliant boot loader and add multiboot support to the ARMv7-a port of Barrelfish. Such a boot loader is able to load several modules into memory and make this information available to the OS through a multiboot information struct. We can simulate this behavior with one big ELF file.

The way it works is the following: We first compile the kernel and every module we want to use into relocatable ELF images. Then we use *objcopy* [11] and generate for each image a new ELF file containing the compiled module as a binary image in its *.rodata* section. *objcopy* also sets some variables where the copied image begins and how large it is. Then we generate a C-file containing a function, which returns the multiboot information for all the modules by referencing the variables set by *objcopy*. At last we compile the loader code, including the generated multiboot C-file, and link everything together in one big ELF file. Figure 2 shows the layout of the final ELF file, which gets loaded by Gem5.

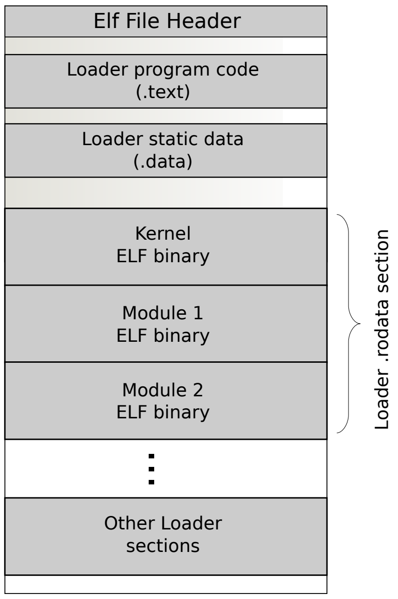

Figure 2: Final ELF file loaded by Gem5

The loader basically gets the generated multiboot information, loads the kernel image into memory and performs the relocation of the non position independent references. It then updates the multiboot information with the relocated kernel image and performs the transition to the kernel entry. The multiboot information gets passed to the kernel as a parameter completing the whole loading process.

4.2.2 Multi-core boot up overview

The boot up protocol for the multi-core port differs in various ways from the boot up procedure of our previous single-core port. We therefore include this revised overview here. The first core is called the *bootstrap processor* and every subsequent core is called an *application processor*

On bootstrap processor:

1. Pass argument from bootloader to first C-function arch_init

2. Make multiboot information passed by bootloader globally available
3. Create 1:1 mapping of address space and alias the same region at high memory
4. Configure and activate MMU
5. Relocate kernel image to high memory
6. Reset mapping, only map in the physical memory aliased at high memory
7. Parse command line and set corresponding variables
8. Initialize devices
9. Initialize and switch to init's address space
10. Load init image into memory
11. Create capabilities for modules defined by the multiboot info
12. Schedule init and switch to user space
13. init brings up the monitor and mem_serv
14. monitor spawns ramfsd, skb and all the other modules
15. spawnd parses its cmd line and tells the monitor to bring up a new core
16. monitor setups inter-monitor communication channel
17. monitor allocates memory for new kernel and remote monitor
18. monitor loads kernel image and relocates it to destination address
19. monitor setups boot information for new kernel
20. spawnd issues syscall to start new core
21. Kernel writes entry address for new core into SYSFLAG registers
22. Kernel raises software interrupt to start new core
23. Kernel spins on pseudo-lock until other kernel releases it
24. repeat steps 15 to 23 for each application processor

On application processor:

25. Gem5 bootloader reads address in SYSFLAG and jumps there
26. Get boot information from well known address
27. Do early boot up like on bootstrap processor (steps 3 to 8)
28. Release pseudo-lock
29. Initialize and switch to monitor's address space
30. Load monitor image into memory
31. Schedule monitor and switch to user space
32. monitor initializes its end of UMP
33. monitor spawns spawnd and signals other monitors about its progress

Please note that we could theoretically start the new core dynamically at any point after the monitor has booted, i.e. the boot process of an additional core is not tied to a particular stage in the overall boot up procedure.

4.2.3 Memory Layout

We still implement a memory split the same way we did in the single-core port. The only real difference is that the kernel image can now be loaded anywhere in memory instead of only at a fixed location. The kernel stack and page tables are statically allocated and part of the kernel image. Since the kernel is not statically linked to a fixed address in high memory anymore, we have to relocate the kernel image during boot up. We do this by adding our memory offset constant (2GB) to the current kernel location to make sure the kernel is in high memory after the relocation. The program counter and stack pointer need to be relocated as well and we do this again by just adding the memory offset constant to their current values. After these relocations we can reset the 1:1 mapping of the physical memory and only map in the high memory aliased regions to finally get our desired memory layout.

To simplify global shared data between kernels at boot up, i.e. before the message passing has been set up, we reserved the first MB of physical memory for this purpose.

4.2.4 Exception Handling

There was nothing to be changed in the way we handle exceptions. We only had to change the way we map in the exception vector in the kernels address space. In the single-core port we just mapped the exception vector to the kernel section. This worked because relative to a section boundary the exception vector is at an offset of 0xf0000 and our kernel is small enough to fit in the first 0xf0000 bytes (approx. 980 KB) of a section. If the kernel can be placed at an arbitrary location in the virtual address space, we can not assume anymore that we will not cross this boundary, i.e. the exception vector would overwrite some kernel code.

To solve this problem we map each exception vector (for each core one) to a well known address in the first section of physical memory. This mapping looks as follows:

$$\begin{array}{lccc}
\text{core 0:} & 0xffff0000 & \rightarrow & 0x80000 \\
\text{core 1:} & 0xffff0000 & \rightarrow & 0x81000 \\
\text{core 2:} & 0xffff0000 & \rightarrow & 0x82000 \\
 & & \vdots &
\end{array}$$

4.2.5 Interrupt Controller

As already mentioned the interrupt controller consists of a centralized distributor and a local CPU interface for each core. We had to change the initialization of the interrupt controller in that we initialize the distributor and the CPU interface if we are on core 0, the bootstrap processor, and only initialize the CPU interface on all other cores. Otherwise the things described in the previous section also apply here.

We want to focus here on the last parameter of the pic_enable_interrupt-function, which selects how an interrupt should be handled in a multi-core system. There are two models for handling interrupts:

1-N model In this model only one processor handles this interrupt. The processor acknowledging this interrupt first, gets the corresponding interrupt number. All other processors acknowledging this interrupt only receive a *spurious* interrupt number (ID 1023), meaning some other processor has already acknowledged this interrupt before them. It can also be that two processors receive the correct interrupt number, if they acknowledge the interrupt at nearly the same time. If the system relies on executing a particular interrupt handler only once, it has to be guarded with some kind of guarding mechanism, e.g. a semaphore.

N-N model In this model all processors receive the interrupt independently. When a processor acknowledges the interrupt the interrupt remains pending for the other processors.

At the moment we only really use the interrupt of the timer for scheduling. Clearly all processors should get this interrupt, so we use the N-N model there. However as more interrupts are added, one has to decided for each interrupt how it should be handled and configure the interrupt controller accordingly. Special care has to be taken for interrupts which must not be taken more than once.

Since we have multiple cores in our system, we can use software generated interrupts (SGI) as a way of communication between cores. The ARM GIC specifies a register for this purpose, the *Software Generated Interrupt Register*. We can write this register with a CPU target list and the interrupt ID. The ID field is only 4 bits, because only interrupt IDs 0-15 are valid software generated interrupts. In our driver for the GIC, we provide the function `pic_raise_softirq` to interrupt the cores in `cpumask` with with `irq`.

```c
void pic_raise_softirq(uint8_t cpumask, uint8_t irq)
{
  uint32_t regval = (cpumask << 16) | irq;
  pl130_gic_ICDSGIR_rawwr(&pic, regval);
}
```

Listing 5: Raise SGI function

4.2.6 Snoop Control Unit

The Snoop Control Unit (SCU) connects up to four processors to the memory system. It maintains data cache coherency between the processors, initiates L2 memory accesses, arbitrate between processors requesting L2 accesses and manage accesses to the Accelerator Coherency Port (ACP) [1]. Through the ACP it is possible to connect other devices, e.g. a DMA-controller, such that the SCU can maintain cache coherency. The SCU can also be used to discover the number of cores present in a system.

We had to implement a device driver for the SCU, but since this device works mostly on its own without further interaction needed, there was not much to be done. The SCU gets enabled by the bootstrap processor, if there is more than one core present. To get the number of cores we can just read out the configuration register of the SCU. Note the SCU simulated by Gem5 has no cache maintenance functionality. It can only be used to discover the number of cores present in the system. Gem5 has a special way of handling caches and cache coherency. We will explain the implications of this in section 4.2.11.

4.2.7 Setting up user space on bootstrap processor

The way the user space is set up on the bootstrap processor did not change much compared to our single core port. We still have to set up the environment for *init*, load it into memory, create a DCB and finally schedule it. The biggest change is that we do not use a ramdisk for loading the modules anymore. As already described in section 4.2.1 we can rely on the multiboot information passed to the kernel to find and load the modules.

After dispatching *init*, it will further initialize the user space and eventually start *mem_serv* and the monitor process. The monitor takes over from then on, since it is the unprivileged user space counterpart of the CPU driver and will bring up all the other modules.

4.2.8 Starting an additional core

In this section we will describe the process of setting up the environment for bringing up an additional kernel and how to signal the other core when and where to start.

The entry point for starting up additional cores is in the *spawnd* process. The spawn daemon decides during boot up which cores to spawn. After boot, it offers a service on each core to spawn programs from the file system. There are two ways for *spawnd* to decide which cores to boot. It queries either the SKB for a list of available cores or parses its command line arguments. Since the SKB on ARM is a simplified version of the SKB on x86, it did not support core discovery, so we had to specify the bootable cores on the command line. *spawnd* sends for each application processor a boot core request to the monitor, waits until it gets a notification that each core is booted and then enters its spawn service routine.

In the monitor we set up the environment in which the new kernel gets booted. First we create and initialize a user message passing (UMP) binding, which is later used by the other monitor for inter-monitor communication. Next we lookup the location of the CPU driver and monitor binary, using the multiboot information and map them in our virtual space. We create two frame capabilities, which points to newly allocated RAM where the CPU driver and the monitor are going to be loaded. These capabilities have to be marked as remote, since they belong to the capability space of the new core. The CPU driver gets loaded right away, whereas the monitor is loaded during boot up of the application processor. After that we have to relocate the CPU driver ELF binary to the physical address, where it had been loaded into. Note that relocation here means resolving all relocation entries in the ELF file. There is no copying going on. Finally the monitor sets up the *arm_core_data* struct. This struct holds important information, needed by the new kernel to boot, such as the memory region, where it can load the monitor binary into, the location of the monitor, the frame for its UMP binding, the core id of the bootstrap processor etc. We pass that information to the new kernel by placing it at a well known location, i.e. one page before the start of the new kernel in memory. We then invoke the 'spawn-new-core' syscall and let the kernel take over.

The way the kernel starts the new core is very platform specific. Gem5's 'firmware' inspects the core ID of the core it is being run on. If this is ID 0 the program counter jumps to the entry point of the kernel and starts executing.

Otherwise the core executes the wfi (wait for interrupt) instruction and does exactly what this instruction says. After getting woken up by an interrupt, it reads the entry address from *SYSFLAGS*, a system-wide general purpose register, and jumps to that address. Together with the fact that we have a separate CPU driver for each core, this also implies that we have to boot up the cores sequentially.

We get the entry address and the core ID of the application processor in the kernel from the monitor (via the syscall). First we set a global pseudo-lock to AP_STARTING_UP, which is later used to detect the boot up of the new core. Then we write the entry address to the *SYSFLAGS* register and signal the other core by raising a software interrupt using the pic_raise_softirq function. Before we return, the kernel spins on the pseudo-lock until the application processor writes AP_STARTED to it.

The application processor has the same entry point as the bootstrap processor. Right at the beginning we get the information passed to the application processor from the well known location, which contains everything, the new core needs to know to boot. The early boot up is basically the same as on the bootstrap processor. After we have initialized all devices, we set the value of the pseudo-lock to AP_STARTED, to signal the bootstrap processor our progress.

4.2.9 Setting up user space on application processor

Unlike on the bootstrap processor, *init* is not the first process started, in fact it is not started at all. We start the monitor as the first process, but before we can do that we need to set up its environment, which was already partly done by the bootstrap processor. The basic setup is the same as on the bootstrap processor. The memory allocator will allocate memory in the frame, which the bootstrap processor has allocated for us. Additionally we have to create a frame for the UMP binding, which we got from the kernel on core 0 and map it in the monitors virtual space. Before we dispatch the monitor DCB and enable interrupt forwarding to the CPU, we acknowledge the SGI we got from the bootstrap processor. Otherwise we get interrupted unnecessarily, as soon as we enable interrupt forwarding.

The monitor will bring up the rest of the modules and signal all the other monitors, that it has started up and is ready for communication.

4.2.10 Inter-core communication

Barrelfish uses explicit message passing for communication between cores. ARM is a shared memory architecture with cache coherency and therefore the message passing interface is implemented on top of this. Fortunately x86 architectures are also shared memory based and we could use the existing implementation of the message passing. We encountered some strange errors when remotely allocating memory. By tracing the created capability through the system, we found that something went wrong, when transferring the remotely allocated capability to the requesting core. The reason was the configured payload size of an UMP packet of 56 bytes. ARM caches have a line size of 32 bytes and therefore the payload could not fit into a cache line, which is essential for sending messages via cache coherency protocol messages. We solved the issue by setting the payload size to 24 bytes.

4.2.11 About caches

When doing multi-core OS development, cache handling and maintenance plays an important role. One has to carefully consider when to flush caches, what parts of it and how cache coherency is maintained by the system, but not so on Gem5. Since Gem5 is a research simulator, it allows researchers to create completely new memory hierarchies. There can be arbitrarily deep levels of caches and therefore normal cache maintenance operations of an architecture will not suffice. Gem5 handles the caches by itself and there has nothing to be done by the system programmer.

This approach has one major flaw. If one core writes to a memory location with caches enabled while another core had its caches disabled at the time the write occurred, Gem5 will not maintain cache coherence. When the second core now reads from this location, even if it turned on its caches in the meantime, there is a chance that this core will not read the value written by the other core, and instead reads whatever was there before, since the actual value is still in the cache of the first core.

This is exactly the situation when we boot up an additional core. Its environment was set up by the bootstrap processor, which has caches turned on while the application processor has not been started yet and therefore not activated its caches. All kinds of strange behavior occurred, because some values, which are parts of the kernel code in this case, were still in the cache of the bootstrap processor and not written to memory yet.

To solve this problem we had to map the frame, where the new kernel is loaded into, uncacheable. This poses a major performance penalty, since all the accesses to kernel memory will not be cached for application processors. However on real hardware this will not be a problem, since we could just flush the cache of the bootstrap processor to make sure, everything gets written to memory.

4.2.12 Mutual Exclusion

In a multi-core system we need to be able to protect critical resources and code segments from simultaneous accesses and modifications from different cores. Despite using message passing and avoiding the use of shared memory (other than to implement message passing), Barrelfish still needs mutual exclusion in for example thread synchronization and printing to the console. The job of an operating system is to provide a low level primitive, which achieves mutual exclusion, upon which more sophisticated mechanisms can be built. One such primitive is a spin-lock.

Most computer architectures provide a mechanism for reading and modifying a location in one atomic step, like compare-and-swap (CAS), or something equivalent, e.g load-link/store-conditional (LL/SC). ARM provides LL/SC primitives `ldrex` and `strex`. Listing 6 shows our implementation of a spin-lock on ARM

```
 1 acquire_spinlock: //r0 holds pointer to lock
 2        1: ldrex  r1, [r0]
 3        teq r1, #0
 4        wfene
 5        strexeq r1, #1, [r0]
 6        teqeq r1, #0
 7        bne 1b
 8
 9 release_spinlock:
10        str 0, [r0]
11        sev
```

Listing 6: Spin-lock implementation

On line 2 we use `ldrex` to load the lock's current value into `r1`. The value 0 represents a free lock and 1 an acquired one. We test if the value of the lock is 0 (line 3). If this is not the case we execute `wfe` (wait-for-event) and wait for an event from another core. Otherwise we try to write 1 into the lock (line 5). If this succeeds we have successfully acquired the lock, otherwise we try again from the beginning (line 6 and 7).

The `release_spinlock` function unconditionally writes 0 to the lock (line 10) and then executes `sev` (send-event) to wake up other cores waiting for the lock. Note that `sev` is not a SGI, but some other implementation defined mechanism. It is required by the ARMv7-a architecture that, if `sev` is implemented by the system, `wfe` is implemented as well.

This concludes the description of our multi-core port. We are able to boot up to four cores on Gem5 with inter-core communication working. We can also run applications and benchmarks on top of our system, which we will describe in the next section.

5 Evaluation

In this section we present the results of our system evaluation. This includes simple tests, in which certain functionalities are tested, and microbenchmarks, which measure certain performance characteristics.

5.1 Test systems

We have configured several test systems on which we performed our evaluation.

arm_X This is our basic simulated ARMv7-a system with X cores. Each core is clocked at 1 GHz and has a 64 KB L1 data and a 32 KB L1 instruction 2-way associative cache. Additionally we configured a 2 MB L2 8-way associative L2 cache, shared between all cores. The size of physical memory is 512 MB. We use the AtomicSimple CPU model (see Section 3.2.1) for all cores. The values for the memory system are chosen to represent a realistic system configuration.

arm_detailed_X This system tries to simulate a 'real' ARMv7-a system as closely as possible. We use the O3 CPU model for each of the X cores. L1 data and instruction caches are 32 KB 2-way associative and we have a 1 MB shared 16-way associative L2 cache. We also configure different cycle counts for the typical operations. Unfortunately, there was an issue with the speculative execution of the O3 model, which prevented the boot process to proceed after bringing up the monitors, if more than one core was enabled, which we could not solve due to time limitations.

x86_64_X We use this system to compare the results of our benchmarks with the arm_X systems. It simulates X Intel x86-64 cores and is configured as closely as possible to the arm_X system. Each of the X cores is clocked at 1 GHz and has a 64 KB L1 data and a 32 KB L1 instruction 2-way associative cache. Additionally we configured a 2 MB L2 8-way associative L2 cache, shared between all cores. The size of physical memory is 512 MB. We use the AtomicSimple CPU model for all cores.

5.2 Tests

We used various tests to test certain functionality of our system and to check, whether we are able to run applications on top of our base system. Most of them are very simple tests, but needed to be passed to run more complex applications and benchmarks. Only the *Multihop test* is a bit more complex and tests the functionality of the multi-hop interconnect driver.

Monitorboot test This very simple test just tests, if the system is able to boot the monitor. Even the detailed multi-core ARM system could pass this test.

Machine	Result
arm_1	pass
arm_2	pass
arm_4	pass
arm_detailed_1	pass
arm_detailed_2	pass

Table 1: Monitorboot test results

Hellotest The mandatory 'Hello World' test can not be missing. We used this test to see if we can run a very simple application, which basically just prints 'Hello World' to the serial console, on top of our base system. Table 2 shows the results for each tested system. Note that the detailed system fails the test, because it can not completely boot and therefore is not able to run the test. This holds for each subsequent test we did.

Machine	Result
arm_1	pass
arm_2	pass
arm_4	pass
arm_detailed_1	pass
arm_detailed_2	fail

Table 2: Hellotest results

Memtest Memtest is a simple test for testing the memory integrity and memory allocation on a single core. Memtest_multicore does the same for every core in the system. Table 3 shows the results

Machine	Result
arm_1	pass
arm_2	pass
arm_4	pass
arm_detailed_1	pass
arm_detailed_2	fail

(a) Memtest

Machine	Result
arm_1	n.a.
arm_2	pass
arm_4	pass
arm_detailed_1	n.a.
arm_detailed_2	fail

(b) Memtest_multicore

Table 3: Memtest results

Multi-hop test In this test we send messages from the client to the server and the opposite way in order to make sure that the multi-hop interconnect driver works for bidirectional message passing. The multi-hop driver allows communication between cores, which are not connected directly, but need to communicate via other cores. It is essentially a routing layer on top of the underlying point to point interconnect driver. The multi-hop driver worked for our ARM port out of the box, without any modification needed as Table 4 shows.

Machine	Result
arm_1	n.a
arm_2	pass
arm_4	pass
arm_detailed_1	n.a
arm_detailed_2	fail

Table 4: Multi-hop test results

5.3 Benchmarks

Besides tests we also ran various micro-benchmarks in order to compare our ARM port to the existing x86-64 port of Barrelfish. The x86-64 port of Barrelfish is not maintained for Gem5 and we had to use both, an older version of Gem5 (revision eb82084f1f4f) and an older release of Barrelfish (rev 3274c00b02e5), both from July 2011, to get it running. Despite these shortcomings, we can still, at least qualitatively, compare the performance of the ARMv7-a and the x86-64 port of Barrelfish running on Gem5.

5.3.1 Memory usage

The first quantity we want to measure is the memory usage of our system. To measure this, we wrote a small program, which queries the memory server about the available and total memory. It is also interesting to look at the memory usage of the CPU driver separately. The CPU driver is static, meaning it does not allocate memory dynamically and we can therefore measure its memory usage by looking how much memory the loaded image needs.

Machine	**Memory usage**
arm_1	40 MB
arm_2	51 MB
arm_4	74 MB

(a) ARM system

Machine	**Memory usage**
x86_64_1	57 MB
x86_64_2	72 MB
x86_64_4	102 MB

(b) x86-64 system

Table 5: Overall system memory usage

CPU driver	**on disk**	**in memory**
ARM	112 KB	304 KB
x86-64 (2011)	156 KB	4842 KB
x86-64 (2012)	168 KB	4854 KB

Table 6: CPU driver memory usage

Let us first analyse the memory usage of the whole system, which is shown in Table 5. The base ARM system with one core uses 17 MB less memory than the corresponding x86-64 system. Both systems are comparable in functionality and modules which get loaded, so this discrepancy is probably due to the 64-bit architecture of the x86-64 port. 64-bit applications use in general more memory than their 32-bit counterparts, because pointers are double the size (8 bytes vs. 4 bytes) and alignment constraints.

One can see that the addition of a core uses 11 MB on ARM and 15 MB on x86-64. This includes space for the additional CPU driver and monitor, as well as all the modules which are spawned on an application processor. Interesting is the fact that the difference of memory cost of an additional core between the ARM and the x86-64 is roughly the difference in the memory usage of the CPU driver, as can be seen from the data in Table 6.

When we look at the memory usage of the CPU driver, we can see several things. First they are all of comparable size when we look at the size of the image on disk. However there is big discrepancy in memory usage when loaded into memory. The difference lies in the .bss section. The x86-64 CPU driver seems to have a much larger amount of uninitialized data, than its ARM counterpart. The second thing we can see, is that the size of the x86-64 CPU driver has not changed much from the 2011 release to the 2012 release, despite a huge increase in functionality. This shows nicely the micro kernel approach of Barrelfish. In other operating systems, like Linux or Windows, a lot of added functionality would have gone into the kernel (e.g. drivers, network stack), whereas in Barrelfish those are all separate user space modules.

5.3.2 UMP benchmarks

We used a set of benchmarks to compare the user message passing performance between our ARM port and the x86-64 port. On both systems UMP is implemented on top of shared memory and therefore we expected comparable performance on both systems.

We always measure the number of cycles a certain benchmark needs to complete. Since we are using the AtomicSimple CPU model, where each operation takes one cycle, for our measurements, we roughly measure the amounts of machine instructions needed to accomplish a certain goal. Please note that the measurements taken in this model are far from measurements on real hardware or even QEMU. We therefore just compare the ARM port to the x86-64 port on Gem5, since everything else would not allow us to make any serious comparisons.

Since we are measuring CPU cycles we need a cycle counter, which is available to a user space application on x86 through the rdtsc instruction. The ARM architecture defines a *Performance Monitor Unit*, which would provide such a counter as well, but at the time of this writing Gem5 does not implement this. Instead we use the CPU local clock as a time stamp counter. This also means that we need to access this counter through the system call interface. Obviously much more overhead is involved to get a cycle count on our ARM port than on the x86-64 port. In fact, the overhead was so large that our measurements on the ARM port showed roughly the same results for all benchmarks, whereas they differed significantly on the x86-64 port. We did overcome this shortcoming by first measuring the overhead involved in taking a time stamp, which is roughly 119 cycles (vs. ˜20 cycles on x86-64) and subtract that value from each measurement.

We always measure a certain operation between core 0 and all the other cores in the system. In the tables below 'dest' is the core ID of the target core.

UMP send/receive This benchmark measures the cycles needed to send/receive a message. On a shared memory system sending involves writing something to memory and adjust a pointer to point to the next position in the message buffer. Receiving is done via polling. If there is a message in the receive buffer available return it and advance the position pointer to point to the next slot. The results are shown in tables 7 and 8.

As one can see in case of the UMP send benchmark the number of cycles required to send a message differs only by one cycle. Note we do not have big fluctuations and the results are the same for each core on a system. This is because Gem5 works deterministically. It will always simulate the code in the same way for the same system and code and if there is no random component involved the outcome is always the same. Receiving a message takes 4 cycles less on the x86-64 system, but this is still in a comparable range. We traced the execution of one receive on the x86-64 and the ARM system and saw, that

Machine	**dest**	**samples**	**median**	**mean**	**sigma**	**min**	**max**
arm_2	1	90	20.0	20.0	0.0	20	20
arm_4	1	90	20.0	20.0	0.0	20	20
	2	90	20.0	20.0	0.0	20	20
	3	90	20.0	20.0	0.0	20	20

(a) UMP send ARM

Machine	**dest**	**samples**	**median**	**mean**	**sigma**	**min**	**max**
x86_64_2	1	90	21.0	21.3	1.616	21	30
x86_64_4	1	90	21.0	21.3	1.616	21	30
	2	90	21.0	21.3	1.616	21	30
	3	90	21.0	21.3	1.616	21	30

(b) UMP send x86-64

Table 7: UMP send benchmark results

Machine	**dest**	**samples**	**median**	**mean**	**sigma**	**min**	**max**
arm_2	1	90	20.0	20.056	0.524	20	25
arm_4	1	90	20.0	20.056	0.524	20	25
	2	90	20.0	20.056	0.524	20	25
	3	90	20.0	20.056	0.524	20	25

(a) UMP receive ARM

Machine	**dest**	**samples**	**median**	**mean**	**sigma**	**min**	**max**
x86_64_2	1	90	16.0	16.2	1.077	16	22
x86_64_4	1	90	16.0	16.2	1.077	16	22
	2	90	16.0	16.2	1.077	16	22
	3	90	16.0	16.2	1.077	16	22

(b) UMP receive x86-64

Table 8: UMP receive benchmark results

the x86-64 system executes less instructions, than the ARM system. For sending both systems execute roughly the same amount of instructions. Since we make these measurements on the AtomicSimple CPU model, this explains the difference of 4 cycles in receiving a message.

UMP latency This benchmark measures the latency, i.e. the time from sending a message until the arrival of the answer. The destination core is replying as soon as it receives the message from core 0. Table 9 shows the results of the ARM respectively x86-64 port.

We expected the latency to be roughly the sum of the send and the receive benchmark plus some overhead to propagate the values to the cache of the other

Machine	dest	samples	median	mean	sigma	min	max
arm_2	1	900	60.0	60.233	1.462	60	73
arm_4	1	900	60.0	60.233	1.462	60	73
	2	900	60.0	60.233	1.462	60	73
	3	900	60.0	60.233	1.462	60	73

(a) UMP latency ARM

Machine	dest	samples	median	mean	sigma	min	max
x86_64_2	1	900	85.0	86.12	4.523	85	108
x86_64_4	1	900	85.0	86.12	4.523	85	108
	2	900	85.0	86.12	4.523	85	108
	3	900	85.0	86.12	4.523	85	108

(b) UMP latency x86-64

Table 9: UMP latency benchmark results

core and response time of the destination core. On the ARM system this seems to be the case, whereas on the x86-64 system this overhead seems to be rather large. We examined the amount of polling necessary to receive a message and saw, that the x86-64 system had to poll on average once more than the ARM system, which explains the bigger overhead on the x86-64 system.

UMP throughput This benchmark measures the throughput of the message passing system. It first fills the message buffer and then continues to send, but can obviously only send something if the receiver has read and processed a message. We measure the time between consecutive sends which is the throughput of the system (every X cycles we can send one message).

Machine	dest	samples	median	mean	sigma	min	max
arm_2	1	100	42.0	41.92	1.932	24	47
arm_4	1	100	42.0	41.92	1.932	24	47
	2	100	42.0	41.92	1.932	24	47
	3	100	42.0	41.92	1.932	24	47

(a) UMP throughput ARM

Machine	dest	samples	median	mean	sigma	min	max
x86_64_2	1	100	56.0	56.23	3.205	31	66
x86_64_4	1	100	56.0	56.23	3.205	31	66
	2	100	56.0	56.23	3.205	31	66
	3	100	56.0	56.23	3.205	31	66

(b) UMP throughput x86-64

Table 10: UMP throughput benchmark results

As one can see in the results of Table 10, the throughput on both systems seem to be roughly the same. This is expected, because of the same memory model and hierarchy.

With these benchmarks we wanted to compare the ARM port to the x86-64 port in a important subsystem of Barrelfish. We know this evaluation does not allow us to quantitatively say something about the performance on a real system, but we think it shows, that the ARM port and the x86-64 port are behaving similarly and therefore should not be too far from each other on real hardware (or in another CPU model).

6 Conclusions

We reached our initial goal to boot Barrelfish on a multi-core ARMv7-a system simulated by Gem5, despite the time limitation and the serious obstacles encountered with the simulator and the compiler. We can fully boot up to four cores with inter-core communication completely working. We added support for dynamic kernel relocations and multiboot compliant boot loaders and in general brought the ARM port of Barrelfish much more in line with the x86 ports, which helps researchers develop new applications or expansions for both platforms, since it is much easier for them to understand both systems.

Our evaluation has shown, that the overall system stability is quite good and we can run different applications and benchmarks on top of the base system. Together with the memory usage and message-passing benchmarks, which, at least qualitatively, show that our port performs similar to the x86-64 port, this work sets a solid foundation for further research into low-power OS design and support for heterogeneous many-core systems.

During the course of this thesis we also learned a lot. In the end it would have probably been better to rewrite the single-core port with dynamic kernel relocation support right from the beginning, instead of porting a somewhat messy foundation to ARMv7-a and Gem5 and later discarding much of the code written during this process. However, one can argue that we had to encounter all the difficulties and implications this approach brought along first, before we could really make an educated decision in which direction to proceed.

Gem5 turned out to be a blessing and a curse at the same time. It is really easy to configure to resemble a real system or design a completely new one. It also has nice tracing and remote debugging support, which comes in very handy, when debugging system software. On the other hand it is a research hardware simulator and therefore under heavy development. New features are added and old ones broken on a weekly basis and since there is a relatively small user base (compared to QEMU for example) it is not that well tested. This leads to situations, where one simply does not know if the simulator or the simulated code is failing, which can be really cumbersome. It will also take a considerable amount of work, if we want to support Gem5 with future releases of Barrelfish. All in all we believe it was worth targeting Gem5, because of the ability to test Barrelfish on a wide range of different system and as an intermediate platform, when targeting a new hardware platform, since Gem5 can simulate real hardware much more accurately than for example QEMU.

7 Future Work

There are several areas where we see potential for future work. Two main issues are remaining, which should have been covered by this thesis, but could not be done because of time limitations. One issue is the lack of multi-core support for the O3 CPU model. This could be an issue with the Barrelfish code, but it could also be that Gem5 does not handle speculative execution correctly in all cases. We just did not have time left to look further into this issue. The second issue is the very minimalistic evaluation. There are many more areas, where a benchmark would be interesting, like context switch time, FPU performance, interactive workloads etc. We just took the readily available micro-benchmarks, which we could run on our system with minimal modifications, due to lack of time porting more sophisticated benchmarks. It would also be interesting to compare our current ARM port of Barrelfish to an ARM port of Linux running on Gem5.

In a next step one could further enhance this port by implementing support for PCI and the shell *fish*. This would somewhat complete the user space, since an interactive shell is something a user expects of a desktop operating system. Implementing a network card driver for the simulated network card would open up a new set of possibilities to investigate how the network stack performs and compare it to the performance of the Linux network stack on Gem5.

Probably the most interesting thing to do, research-wise, is to take this code base and port it to real hardware, since properties like power consumption can only really be measured on real hardware. Some researchers of the Systems Group at ETH Zurich are working on bringing up Barrelfish on the PandaBoard [17] using this work as the foundation. The ultimate goal would be getting hands on a board implementing the ARM big.LITTLE architecture [10] and enhance Barrelfish to support this heterogeneous multi-core system. Let us suppose the system has two high-performance Cortex-A15 and two Cortex-A7 low-power cores, which we want to use optimally, considering that we want to get the best user experience together with the lowest possible power consumption. This would bring up some very interesting questions for the scheduler, some of them are:

- How does the scheduler decide, which task gets scheduled on which core?
- When and how should the system migrate task from the low-power to the high-performance core and vice versa?
- Should the scheduler auto-tune its behavior and if yes, how can this be achieved?

These and other questions require lots of research and measurements, but are very interesting and fundamental for future low-power OS design on heterogeneous many-core systems.

References

[1] ARM. *Cortex-A9 MPCore Technical Reference Manual.* ARM, 110 Fulbourn Road, Cambridge, England CB1 9NJ, r3p0 edition, July 2011.

[2] Arch Linux ARM. Arch linux on arm. `http://archlinuxarm.org/about`, 2011-2012.

[3] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[4] Andrew Baumann, Simon Peter, Adrian Schüpbach, 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, pages 12–12, Berkeley, CA, USA, 2009. USENIX Association.

[5] Charly Bechara. Booting arm linux smp on mpcore. `http://linux-arm.org/LinuxBootLoader/SMPBoot`, March 2012.

[6] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. User-level interprocess communication for shared memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(2):175–198, May 1991.

[7] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[8] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, July 2006.

[9] Inc. Free Software Foundation. Multiboot specification. `http://www.gnu.org/software/grub/manual/multiboot/multiboot.html`, 2009.

[10] Peter Greenhalgh. Big.little processing with arm cortex-a15 & cortex-a7. Technical report, ARM, September 2011.

[11] Free Software Foundation Inc. objcopy. `http://sourceware.org/binutils/docs/binutils/objcopy.html`.

[12] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[13] Ben Leslie. L4 microkernel on arm. `http://l4hq.org/arch/arm/`, 2002-2007.

[14] ARM Limited. *ARM Generic Interrupt Controller Architecture Specification*. ARM Limited, 110 Fulbourn Road, Cambridge, England CB1 9NJ, architecture version 1.0 edition, September 2008.

[15] ARM Limited. *ARM Architecture Reference Manual*. ARM Limited, 110 Fulbourn Road, Cambridge, England CB1 9NJ, armv7-a and armv7-r edition, November 2011.

[16] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, November 2005.

[17] Pandaboard.org. Pandaboard. `http://pandaboard.org/content/platform`, 2012.

[18] Simon Peter, Adrian Schüpbach, Paul Barham, Andrew Baumann, Rebecca Isaacs, Tim Harris, and Timothy Roscoe. Design principles for end-to-end multicore schedulers. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, HotPar'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[19] The FreeBSD Project. Freebsd on arm. `http://www.freebsd.org/platforms/arm.html`, 1995-2012.

[20] Timothy Roscoe. *Mackerel 1.4 User Guide*. Systems Group, ETH Zurich, 2011.

[21] Bram Scheidegger. Barrelfish on netronome. Master's thesis, ETH Zurich, July 2011.

[22] Adrian Schüpbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems (MMCS)*, Boston, MA, USA, June 2008.

[23] Microsoft Inc. Steven Sinosky. Building windows for the arm processor architecture. `http://blogs.msdn.com/b/b8/archive/2012/02/09/building-windows-for-the-arm-processor-architecture.aspx`, February 2012.

[24] Inc. Vincent Sanders. Atag header reference. `http://www.simtec.co.uk/products/SWLINUX/files/booting_article.html#appendix_tag_reference`, June 2004.