

Exploring heterogeneous OS architecture

Master Thesis

Author(s):

Buitendijk, Dennis

Publication date:

2023-09

Permanent link:

<https://doi.org/https://doi.org/10.3929/ethz-b-000635117>

Rights / license:

[In Copyright - Non-Commercial Use Permitted](#)



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich



Master's Thesis Nr. 462

Systems Group, Department of Computer Science, ETH Zurich

Exploring heterogeneous OS architecture

by

Dennis Buitendijk

Supervised by

Prof. Timothy Roscoe, Nora Hossle, Daniel Schwyn, Roman Meier

March 2023 – September 2023

DINFK

Acknowledgements

I would like to thank my immediate supervisors Daniel Schwyn, Roman Meier, Nora Hossle, and also Pengcheng Xu for their support, guidance, and expertise during this thesis.

I would also like to thank Prof. Timothy Roscoe for the opportunity to be part of the Barrelfish project.

Contents

Contents	2
1 Abstract	5
2 Introduction	7
2.1 Thesis outline	7
3 Background and related work	9
3.1 Classical Memory Model	9
3.2 Capabilities	10
3.3 Barrelfish	10
3.3.1 Capabilities in Barrelfish	10
3.3.2 Memory management	11
3.3.3 Kernel	11
3.4 i.MX 8X	11
3.4.1 Cortex-A35 Subsystem	11
3.4.2 Cortex-M4 Subsystem	11
3.4.3 System Controller Unit (SCU)	12
3.4.4 Messaging Unit (MU)	12
3.5 ARM Exception handling	13
3.5.1 ARMv8-A exception handling	13
3.5.2 ARMv7-M exception handling	13
4 Design	15
4.1 General design of the Barrelfish implementation	15
4.1.1 Paging	16
4.1.2 Threads	17
4.1.3 CPU driver	18
4.1.4 Message passing	18
4.1.5 Upcalls	18

4.2	Memory Layout on i.MX 8X	19
4.3	Capability transfer	20
4.3.1	Too high to be read by the M4	20
4.3.2	Memory local to the M4 at different address on the A35	21
4.3.3	Complete transformation	21
4.3.4	General design	22
4.4	Booting the M4	23
4.4.1	Memory allocation	23
4.4.2	Start command	24
5	Implementation	25
5.1	Barrelfish implementation	25
5.1.1	Paging	25
5.1.2	Applications	26
5.1.3	CPU driver	27
5.1.4	Monitor	29
5.1.5	Message passing	29
5.1.6	Upcalls	30
5.2	Atomic operations in Barrelfish	30
5.3	Capability transfer	33
5.3.1	Issues with current implementation	34
5.4	Booting the M4	34
5.4.1	A35 side	34
5.4.2	M4 side	36
5.5	Spawning on the M4	38
5.5.1	Issues with current implementation	39
5.6	SCU support in Barrelfish	40
5.6.1	Inter-Processor Communication (IPC)	40
5.6.2	Remote Procedure Call (RPC)	40
5.6.3	Integration into Barrelfish	41
6	Evaluation	43
6.1	Communication performance across cores	43
6.1.1	Latency	43
6.1.2	Throughput	44
6.1.3	Conclusion	45
6.2	Time to boot	46
6.3	User code performance	48
7	Conclusion	51
7.1	Summary	51
7.2	Future work	51
	Bibliography	55

Chapter 1

Abstract

In recent years the computer systems have been getting increasingly complex. In particular, these systems have increasingly heterogeneous cores. These heterogeneous cores often have a different view of shared memory, where they might see the same memory at different addresses. We show that software can run across cores with such a view of memory. We show a design of how we transform the addresses in a capability based system. This design is implemented in Barrelfish OS on an i.MX 8X SoC. We demonstrate that the Barrelfish OS can run on a system where two cores with not only a different ISA, but also a different word size is used. This implementation can be easily extended to other platforms to enable Barrelfish OS to handle different heterogeneous systems.

Chapter 2

Introduction

Modern computer systems are getting increasingly complex and heterogeneous. Most systems include multiple cores that are getting increasingly heterogeneous. These cores can be often divided into high-performance cores and low-power cores where the workload can be allocated to the most suitable core.

This ever-increasing complexity results in the software running on the systems becoming increasingly hard to write. The system has to be able to handle all the differences of the cores it runs on. This can range from just a slower clock speed, to private memory, or even totally different ISAs.

This results in each core having different views of the memory layout on the system. As such the system needs to be very careful in how we manage the resources of the system. In this thesis we will be exploring some possible ways to handle this complexity by means of a concrete example.

The goal of this thesis is to extend the Barrelfish implementation used in an OS-course at ETH to run on the Cortex-A35s and the Cortex-M4 simultaneously. As these cores have not only a different ISA, they also have a different word size, and as such the memory view is vastly different. Therefore, this thesis will explore ways to handle this complexity.

2.1 Thesis outline

Chapter 3 provides the needed background for this thesis including Barrelfish, and the used platform. Next in chapter 4 introduces the design for including the M4 into the current Barrelfish implementation and provide detail on the memory transformation between the cores. Afterwards, chapter 5 provides deeper insights into the implementation of Barrelfish on the M4 and a couple of encountered hardware quirks. In chapter 6 we provide some performance measures on the implemented features. And finally, in chapter 7 we conclude

2. INTRODUCTION

the results of the thesis and provide an outline of work that still needs to be done.

Chapter 3

Background and related work

In this chapter we provide short overview of the background information needed for this thesis. In this chapter we provide a short overview of the traditional and current memory models. This is followed by a brief introduction to capabilities, and the Barrelfish OS. Afterwards, we introduce the most important parts of the platform used during this thesis, followed by a summary of the different exception implementations relevant to this thesis.

3.1 Classical Memory Model

Traditionally the memory in a system has been viewed as a contiguous range starting at 0. This single range is then shared among all cores in the system. While this provides a simple abstraction that is easy to work with, it no longer resembles the reality in modern systems. Many devices (e.g. NIC, GPU) have their own dedicated memory, that is also accessible by the CPU.

Another example of where this model falls short is Non-Uniform Memory Access (NUMA). In a NUMA system each CPU has a region of local memory that is located close to it. Each CPU can also access the memory of the other CPUs, albeit slower. Even though these memory regions are physically disjoint, logically they are still considered part of the same contiguous range of memory.

These physical memory regions are then mapped to virtual address spaces for each application running on a system. The virtual address space provides a uniform view of memory to the application, while also providing protection from other applications. This model has been extended for a specific system (Barrelfish) in [4]. They propose to extend the address translation to include an identifier as to which physical address space is referenced.

While this model has been used for a long time, it is not the only model still in use. Many simpler processors do not include any support for address

translation. This can be viewed as if there were a fixed 1-to-1 mapping from virtual to physical addresses.

3.2 Capabilities

Capabilities are a way of managing resources in a system as well as enforcing access control to them. They are a sort of token that can be thought of as an unforgeable key, that provides access to the resource they describe. If an actor in the system holds a capability, they can use the resource it represents, and as such if they don't hold any capability, they cannot access anything.

The operations that can be performed on capabilities are often called invocations. When a capability is invoked for an action, the capability system checks the validity of the invoked capability, and then performs the action only if the capability provides sufficient permissions to do so.

A more complete overview of different capability systems can be found in [3].

3.3 Barrelfish

This section provides a brief overview of Barrelfish. Barrelfish OS is a research OS built by the Systems Group at ETH Zurich. It is an implementation of a multikernel OS design. This means that the OS is designed as a distributed system where each core has its own kernel. These kernels share no memory except the message passing channels. This design allows the underlying hardware to be heterogeneous fairly easily as each unique core can have their own kernel implementation.

In the rest of this section we provide an overview of the relevant parts of Barrelfish.

3.3.1 Capabilities in Barrelfish

Barrelfish uses capabilities for resource management. These capabilities are so called typed capabilities, which means that for each use case there exists a dedicated type of capability. Capabilities are stored in a special region called a CSpace with the capability type for it being CNode. RAM that can be mapped by a user application is called a Frame.

An application running on Barrelfish needs to have access to their capabilities to be able to use them. They refer to them via a capability address. This is an address in the applications CSpace which is a region of memory that cannot be mapped into the applications address space. As such the application cannot manipulate their own capabilities.

A reader interested in reading more about the capability system in Barrelfish might want to read [3].

3.3.2 Memory management

The memory management in Barrelfish is handled entirely in user space. Through the capability system, Barrelfish allows applications to control their own virtual memory system. The application only invokes the kernel to perform the actual mapping in the system.

3.3.3 Kernel

The kernel in Barrelfish is called a CPU driver. This CPU driver is designed to be stateless with no dynamic memory allocation. It provides services to the applications such as mapping a frame into the virtual address space. The CPU driver only needs to check the capabilities provided to the system call and verify that the application is allowed to perform the operation. Any operation that needs additional memory, the application needs to provide the required memory. Additionally, each CPU driver keeps a *mapping database* that holds all the capabilities for that core. The mapping database enables efficient operations on the capabilities.

3.4 i.MX 8X

This thesis uses the NXP i.MX8 platform as a basis. In particular, we use an i.MX 8QuadXPlus (i.MX 8X). The following sections introduce the components of the i.MX 8X that are relevant to this thesis.

3.4.1 Cortex-A35 Subsystem

The i.MX 8X features four Cortex-A35 cores that are the main processing power. These cores implement the ARMv8-A ISA. They implement an MMU enabling software to utilize virtual memory.

3.4.2 Cortex-M4 Subsystem

The i.MX 8X also features an ARM Cortex-M4 as a separate subsystem. Unlike the A35's the M4 does not implement ARMv8-A, but it instead implements the ARMv7-M ISA. As such the M4 does not have support for a Memory Management Unit (MMU), and as such has no notion of paging and virtual memory. To still support a model of unprivileged and privileged software execution the M4 supports the Protected Memory System Architecture (PMSAv7). This is done by providing a Memory Protection Unit (MPU).

The MPU allows us to split the memory into 8 regions. Each region has a size between 32 bytes and 4GiB, and allows us to configure access permissions both for privileged and unprivileged execution, as well as caching behavior for the entire region. Additionally, each region of at least 256 bytes can be divided into 8 subregions that can be disabled individually. The MPU also allows us to use the default memory map as an additional background region that can be used by privileged execution, however any permissions specified in active regions take precedence.

While the M4 does not implement a core cache, NXP implements two system-level caches around the M4. These caches are both 16KB in size, have 256 sets, 32B cache lines and are 2-way set-associative. The first cache covers the Memory region of `0x00000000` to `0x1FFFFFFF` and is called the Code cache. The second cache covers the Memory region of `0x20000000` to `0xFFFFFFFF` and is called the System cache. Despite the name, both caches can cache instructions and data, however it is intended to have instructions use the Code cache with data accessing the System cache, as both caches can be accessed simultaneously.

The M4 version included in this SoC also includes a floating-point unit (FPU). It implements the single precision ARMv7-M floating-point extension FPv4-SP-D16. As such it implements 32 single-precision registers, that can be accessed as 64-bit registers for loads and stores.

3.4.3 SCU

The SCU is responsible for booting the system, interfacing with the external PMIC, managing power, clocking, and reset of the i.MX 8X subsystems, controlling the pin multiplexing and IO control, resource partitioning, and thermal management. These services are provided to the AP cluster and Cortex-M4s through an API that is explained in section 5.6.

3.4.4 MU

The i.MX 8X provides several MU's that can be used for communication between cores by passing messages through the MU. Each MU exposes two interfaces: Processor A-facing and Processor B-facing. While these are mostly identical, the A-side has the additional capability of resetting the entire MU.

Both sides contain ten device registers that are 32-bits wide each. Of those four are transmit registers, four are receive registers, one is a status register and the last one is a control register. The A-side transmit registers are mapped to the B-side receive registers and vice-versa. The MU also allows both sides to send interrupts to the other side.

3.5 ARM Exception handling

In this section we give an overview of the different exception handling structures present in ARM processors. In the i.MX 8X we have both ARMv8-A, and ARMv7-M cores, and as such we focus on those two ISAs. That being said, the A-profiles all have a similar design to each other with only slight differences in the details. Additionally, the M-profiles also share the exception model with each other.

3.5.1 ARMv8-A exception handling

ARMv8-A processors have four execution levels (EL0-EL3). These correspond to different privilege levels, with EL0 being the lowest, and EL3 being the highest. Typically, EL0 is used for user space, EL1 for an OS kernel, EL2 for a Hypervisor, and EL3 for the secure monitor.

When an exception is taken, the EL is potentially updated and saved in the corresponding PSTATE register alongside the return address. The processor then branches to the vector table. The vector table is unique to an EL. As EL0 does not handle exceptions, we have 3 vector tables. Each vector table consists of four sets containing four entries each. Each set contains a vector for a synchronous exception, an IRQ, a FIQ, and SError. Which set is used depends on the execution while the exception occurred. Two sets handle exceptions from the same EL, and two sets handle exceptions from lower ELs.

Each entry in the table is 16 instructions long and as such can be used for the first part of the exception handler.

The exception return restores the previous EL and PSTATE register and restarts execution on the saved return address.

3.5.2 ARMv7-M exception handling

ARMv7-M processors have two different execution "levels": Handler mode and Thread mode. Handler mode is used to execute exception handlers and executes as privileged. Thread mode is the default execution mode and can be either privileged or unprivileged.

The vector table holds an entry for each exception that can occur, as well as an entry pointing to the initial stack pointer. This means that we have 16 entries for the exceptions defined by ARM plus any interrupts defined by the implementation. In our case, we have a total of 610 defined entries including vectors for interrupts from various other subsystems on the i.MX 8X such as a UART. Note that this is more than the 496 external interrupts specified on [2, Page B1-524]. The vector table entries are taken from the NXP provided SDK for the i.MX 8X Cortex-M4. Each entry is a 32-bit word that points to the location of the handler.

3. BACKGROUND AND RELATED WORK

When an exception is taken, the processor saves context onto the current stack. This content consists of at least eight 32-bit words: xPSR, ReturnAddress, LR, R12, R3, R2, R1, R0. If the processor implements the Floating-point Extension, and it is currently being actively used, the processor also pushes some FP state. In particular, it pushes an additional 18 32-bit words comprising the FP registers s0-s15 and the FPSCR.

After saving the context, the processor mode is changed to Handler mode, the vector table entry is read and used as a jump value. Additionally, the LR is filled with the return value that can be used to return to the previous state.

Exception return occurs if a value of `0xFXXXXXX` is loaded into the PC via a POP, LDR, or BX on a register containing the value. The processor then changes execution mode, stack and restores the context from the stack. If bit[4] is 1, the saved context includes the FP state and as such that is also popped. If bit[3] is 1, we switch execution mode to Thread mode, otherwise we stay in Handler mode. If bit[2] is 1, we also switch to using the PSP as the SP, otherwise we use the MSP as SP. Handler mode always uses the MSP for its stack.

Due to the way the context is saved and restored, the code must ensure that the stack area is accessible. If it is not accessible by the code, we enter another exception on entry or return.

Chapter 4

Design

In this chapter we first give a high-level overview of the Barrelfish implementation on the M4. We then introduce the memory layout of the i.MX 8X and then discuss the capability transfers and transformations resulting from said layout. We then follow this with a discussion on the way the M4 is booted.

4.1 General design of the Barrelfish implementation

The design of Barrelfish on the Cortex-M4 follows the same design as for the other architectures. There is a Boot driver responsible for starting up the core, the CPU driver that forms the actual kernel implementation, and a Monitor that forms the user space parts of the kernel.

The CPU driver is based on the ARMv7-A CPU driver from the older AOS version. The monitor is based on the ARMv8-A implementation of the AOS version, and the code is (mostly) shared with the ARMv8-A version. Both have been adapted as detailed later. While the boot driver is also based on the ARMv7-A version, it is only very loosely so.

One of the most significant differences is due to the lack of a MMU. This leads to some design changes.

As mention in subsection 3.4.2, our implementation of the M4 contains an MPU. This means, that we only have memory protection, but no address translation. Therefore, we do not have virtual memory, but instead use a 1-to-1 mapping.

While this does not have a large impact on the CPU driver, it greatly affects user space. This begins with the program binaries. The programs are compiled into an ELF format that expects the segments to start at specific addresses. With virtual memory, the actual location of the memory is hidden from the program, and it can just be placed anywhere. For the M4, we need to compile

it in a way that it can be put anywhere in memory, as well as modify the data after loading the ELF.

To enable users to write code like they would for other OSes, Barrelfish provides support for standard features such as a heap for dynamic memory allocation. This is usually done by allocating a region of virtual memory, and then backing it on demand with physical memory. This is once again not possible on the M4. Therefore, we just allocate a larger region of memory and then pass it to the heap gradually.

4.1.1 Paging

In Barrelfish, the paging code is responsible for managing the virtual memory of a domain. This includes reserving regions for memory and then later backing them with physical memory by mapping the virtual address space onto the physical address space. This is done by the user code on demand with page faults. If a page fault occurs, it is passed to the user code responsible for causing it. That domain then allocates a Page in memory and maps it to the faulting address.

On the M4 however, we need to hold the memory before we can know at what address we need to access. As such when we allocate a region in virtual memory, we actually allocate the memory, and then "map" the memory to the same address in virtual memory. The problem with this is that we only have a total of 8 MPU regions on the M4 core. Each of these regions can cover one memory capability to allow the domain to access this memory. Many programs however need more than 8 capabilities, and as such we cannot allow access to all of them at the same time. Instead, we "map" the capabilities into the MPU regions until we have occupied all of them. At that point we "unmap" one of the capabilities from the MPU and "map" the new capability into the newly freed MPU region.

When we now try to access the memory of the region that was just "unmapped", we generate an exception. We treat this as if it were a page fault on a system with virtual memory: the exception is reflected to the programs paging code, where a page fault handler runs. The page fault handler then needs to "map" the region containing the fault address again. It does so by "unmapping" another region and "mapping" the current region into the MPU. Which region is replaced is up to the implementation to decide. The simplest variant is to use a round-robin replacement policy. In this case each usable slot will experience replacement the same amount as the other slots. This policy however, will lead to an increased number of page faults as regions that are used constantly will be replaced by regions that might be used only once.

A better policy would be to use a least-recently-used(LRU) replacement policy. In such a policy, the regions used most often are kept "mapped" for the longest

possible time. To determine which regions was used the least recently we would need to keep track of each address accessed. However, the only way to determine the address of any access is to trap it. This would require the regions not to be mapped, which would defeat the entire purpose.

An alternative similar to LRU that would be feasible to implement, would be to keep track of the number of times a region was already "mapped" in. Then we would replace the region that was "mapped" in the least number of times. This would result in the preferred regions to be replaced being new or very seldom used regions. However, to be marked as being "mapped" in often, it also needs to be "unmapped" often. In an application where we have more than 8 regions that are used over and over, eventually each region will be marked as equally often "mapped". Therefore, this would eventually default back to round-robin replacement.

As we do not have a better policy than round-robin, we have implemented it as the replacement policy.

That being said, there are some regions of memory that are simply more important than others, and have to always be "mapped" in. The number of regions that have to always be "mapped" in should be kept to a minimum to allow for more application specific regions to be "mapped" in at any given time. We have kept the number of always "mapped" regions to 4, or half the available MPU regions. These are kept in the upper four MPU regions as these take priority over the lower MPU regions. The regions we keep mapped are the code segment, the data segment, the dispatcher, and the thread structures. The code and data segments obviously need to be mapped in order to enter the application. The dispatcher also needs to be "mapped" as certain parts of it need to be accessed to enter the application as explained in subsection 4.1.5. Finally, the reason for the thread structures to be always "mapped" is that when it is "unmapped", and the thread causes a page fault, we enter the thread through and upcall and set values in the struct before we handle the page fault. However, as it is not "mapped" we cause another page fault which leads to an unrecoverable failure.

4.1.2 Threads

Like many other OSes, Barrelfish provides support for threads. Each thread was designed to have their own separate stack for handling any exceptions such as a page fault. This stack gets allocated on creation of the thread and is mapped at that time. As mentioned previously, on the M4 we "unmap" regions often even though we still use the region, and when it is needed again we let the thread causing the exception map it back in with the page fault handler. If however we "unmapped" the exception stack of a thread, and it causes a "page fault", the thread will then try to handle the exception. This uses the exception stack and as it is not mapped will cause another exception,

and enter a loop of exceptions that cannot be broken. We therefore need to ensure that the exception stack is always mapped.

As we have only 8 MPU regions per domain, we cannot permanently map each threads exception stack as we allow up to 256 threads in a domain. For the first thread in the domain we have a statically allocate exception stack in the ELF. For any further thread on the M4 we could choose to use the same statically allocated exception stack as the first thread. However, this would lead to an issue when more than one thread is handling an exception at a time, as when an exception is entered the stack is reset. Another solution would be to include the exception stack in the thread struct on ARMv7-M. This solves the issue as we always keep the region of memory containing the structs for all threads mapped.

4.1.3 CPU driver

The CPU driver runs almost exclusively in Handler mode. This means that it is only entered when an exception occurs. This can either be an exception triggered by an external device, wrong execution, or a system call (via SVC instruction). For the system call, the CPU driver handles the users request if they have the necessary permissions to do so. For all other exceptions, the CPU driver reflects the fault back to the user code to handle. This includes faults such as memory access violations(treated like a page fault), and any interrupts from devices used by the code.

4.1.4 Message passing

Barrelfish distinguishes between two types of message passing: messages between domains on the same core, and messages between the monitors on different cores. These are both used to build an RPC system based on certain messages. These messages first get sent to the monitor, and if it cannot be handled on this core, or is not intended for this core, the message is forwarded to the other core. This means that the message structure must be consistent across the cores. As such, only data types that have a size not defined based on the architecture can be used. In our case we are using data types that are equivalent to the data types used on the ISA with the largest word size in the system.

4.1.5 Upcalls

Barrelfish has two ways of switching from the CPU driver to user code. The first way is to resume the user code from where it entered into an exception. The second way is to upcall either to a user space scheduler, or a specific entry like a page fault handler. Which method is used depends on whether the associated dispatcher is disabled, which means that it is currently in a

critical section and needs to be resumed. Both of them are implemented as an exception return. As mentioned in subsection 3.5.2, the exception return pops values from the stack to restore the state. Therefore, the stack needs to be accessible to user code. For a return this is not a problem as we entered the CPU driver through an exception and as such have the stack already configured.

For an upcall, we have two options. The first one would be for the CPU driver to find the appropriate stack in the user code, build up an exception return stack on it, and then "return" from the exception. This would require the user code to adhere to specific naming conventions and/or specific offsets for their stack. Alternatively, we can add a dedicated stack to the dispatcher that is just large enough to hold an exception frame. This stack can then be used to build up an exception frame for all upcalls. The user code is then be required to set the appropriate stack on entry.

In the ARMv8-A version of Barrelfish, an upcall uses the CPU driver stack for the exception return, and has the user set the correct stack on the upcall entry. As it is simpler and keeps the codebase consistent with other implementations, we have extended the dispatcher with a dedicated upcall stack.

4.2 Memory Layout on i.MX 8X

In this section we provide an overview of the relevant parts of the i.MX 8X memory layout. While the A35 cores can see higher addresses, I limit our focus on the 32-bit space seen by the M4. For details on the devices located in the regions, refer to [7, Chapter 2] Figure 4.1 shows an overview of the different memory sections present with the M4 view on the upper half, and the A35's view on the bottom half. The light green section depicts the 2GB of RAM on our system. It starts at the address `0x80000000` and ends at `0xFFFFFFFF`. The M4 however, cannot access any RAM above `0xDFFFFFFF`. Instead, the Private Peripheral Bus (PPB) occupies that space.

The dark blue section, from `0x60000000` to `0x7FFFFFFF`, contains the PCIe mapped IO and is shared between all cores. The light blue section is used for on-chip peripherals and is shared between all cores. The striped light blue section is also used for on-chip peripherals, however these are used for core specific on-chip peripherals, and is mostly unused.

A more interesting section is depicted in red. On the A35 cores the 64MB large region starting at `0x34000000` is mapped to two separate regions on the M4. These are the region from `0x1F000000` to `0x21000000` which contains the Tightly-Coupled Memory (TCM), and the region from `0x40000000` which contains M4-specific on-chip peripherals such as the INTMUX or some MU's.

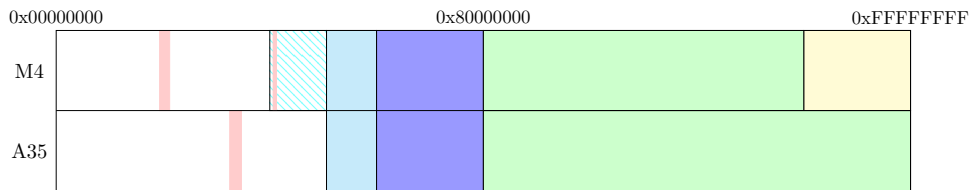


Figure 4.1: Partial representation of i.MX 8X Memory space.

As Figure 4.1 shows, the only regions with a different view of memory are above `0xDFFFFFFF`, and the TCM and some private devices of the M4.

4.3 Capability transfer

In Barrelfish, we have the ability to transfer capabilities between the cores. This is mostly used on booting another core, or when spawning a domain on another core. As shown in Figure 4.1, we have sections of memory, where the different cores might not see the same memory at the same address. If we were to transfer a capability within that memory region, we have to modify the capability as required by the memory map. On the i.MX 8X we have two types of different views on the memory:

1. Too high to be read by the M4
2. Memory local to the M4 at different address on the A35

These two types can be handled differently.

4.3.1 Too high to be read by the M4

This case has two possible ways to handle. The first option would be to refuse to create the capability on the M4 as it is not usable. This would prevent any code that might get a RAM capability to such a region on the M4 from trying to access the memory, which would lead to a crash of the code. It would instead allow the domain providing the capability to see that an error occurred, and then retry with a capability in the correct range.

However, capabilities can provide more than just a memory access. They can also signify that a domain is allowed to perform an action on a certain resource. An example of this could be that the application is allowed to power of a specific core. Limiting this to only certain cores could vastly limit the usability of a core. Therefore, we could allow the capabilities to be copied to the M4 as they are, and if we need to perform an action on the capability that is not possible physically, such as reading it, we could get the A35s to perform it for us.

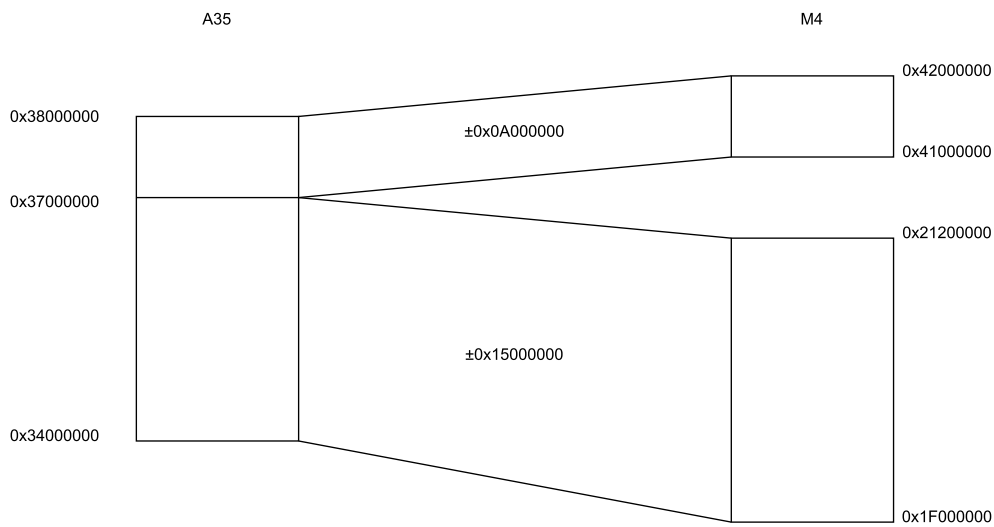


Figure 4.2: Memory transformations between A35 and M4

One example for this would be the multiboot modules. On our implementation the modules containing the binaries are located at address `0xFXXXXXXX`. However, we still should be allowed to spawn these modules on the M4 and as such should still have the capabilities for them.

4.3.2 Memory local to the M4 at different address on the A35

In this case, both cores can have access to the memory. As such, we need to transform the address of the capabilities on transfer. We assume that each address maps to exactly one address on the other core. As such, we have a constant offset for each memory location. While this could be on a granularity of a single byte, in our case the smallest granularity is 16MB.

On the i.MX 8X, the only region where this is the case is shown in red in Figure 4.1. This is split in two different mappings. While they are contiguous on the A35s, on the M4 one is located at a lower address, and one at a higher address. This results in the conversion shown in Figure 4.2.

4.3.3 Complete transformation

Putting these two options together, we arrive in the following capability transformation. If the capability we receive has an address that falls within the range where we have a different view, we convert the address according to the previous mapping. If not, we accept the capability as is. While this does mean that the M4 can get capabilities that are not locally usable, we do think it is beneficial to allow such capabilities.

4.3.4 General design

Based on the example of the i.MX 8X, we can generalize the design to work for an arbitrary platform. We will now consider two arbitrary cores A and B which have an arbitrary ISA and assume that the system has a known memory map. For each address in the address space of core A, we have three options of how core B can see the associated memory location:

1. At the same address.
2. Sees memory location at a different address.
3. Cannot access memory location.

We list how we treat these cases in the following.

Same address

This case is trivial. We can simply accept the capability as is.

Different address

In this case, the memory location is mapped into the address space of core B, but at a different address. Therefore, to transform the capability to a capability for core B, we need to add or subtract the difference between the addresses. While doing this, we need to ensure that the region of memory that is covered by the capability can be handled by the same transformation. If this is not the case, we could take one of two options.

The first option is to refuse to accept the capability and return an error. This would signify to the sender that they need to split the capability into multiple capabilities and resend them. This would however require the sending application to have knowledge of the exact memory map, which would be overly complex. Alternatively, we could split it into multiple capabilities transparently. The second option would be more user-friendly. However, we would need to make sure that all created capabilities are considered copies or descendants of the original capability.

Cannot access memory location

This case is the result of the two cores having a different address range. As such the memory location is simply outside the address range of Core B. In this case, the capability might not be for memory, but instead it could signify the permission to perform a certain API call. This permission should still be valid even if the action might require a different core to perform it for core B. As such we should keep the capability as is.

Resulting program flow

The actions for each of the three cases can then be combined into the following program flow. As cases 1 and 3 result in the same inaction being taken, we can treat them as the same. Therefore, when receiving a capability from core A, core B needs to check if the capability is for an address that falls into the second case. If that is the case, the corresponding transformation needs to be applied. If it does not fall into case 2, we have nothing to do.

4.4 Booting the M4

Booting the M4 from the first A35 core is similar to booting a second A35 core. To boot a second core we first allocate memory for the communication channel across the cores. Then we allocate memory for the boot driver, CPU driver, the monitor domain, and some RAM for the allocator on the new core. We then load and relocate both the boot and CPU drivers, and then do a secure monitor call to start the second core.

Booting the M4 as a second core differs in two main points:

1. Memory allocation
2. Start command

4.4.1 Memory allocation

Firstly as shown in Figure 4.1, the M4 requires memory to be located below `0xDFFFFFFF` to be accessible. As such, we need to make sure that all allocated memory falls below said address. Additionally, we have additional restrictions on the position of the boot and CPU drivers. This is because these both include a vector table. The vector table included in the CPU driver is used during normal operation for all interrupts and exceptions, including system calls, and as such needs to be usable. The only time the vector table in the boot driver is used, is during boot. As mentioned in subsection 3.5.2, the first two words of the vector table are the location of the stack, and a pointer to the reset handler. On start, these are loaded into the `sp` and `pc` registers respectively.

As per [1, Chapter 2.3.4] the default location for the vector table is at address 0, but it is configurable at runtime to start at an address in the range from `0x00000080` to `0x3FFFFFF80`. On the i.MX 8X implementation of the M4, the default address is `0x1FFE0000`, which is the start of the TCM. As such, we need to place the boot driver at the start of the TCM, and also place the CPU driver into the TCM to be able to use the vector table.

4.4.2 Start command

The secure monitor has no command defined to boot the M4 on the i.MX 8X. Instead, the M4 is controlled by the SCU. This is done in two separate steps. First the SCU needs to turn on the power to the M4. Only after that can we actually write to the TCM and prepare the memory for booting. Once that is done, we leverage the SCU to start the M4.

Chapter 5

Implementation

This chapter provides more detailed insights into the implementation of the design described in chapter 4. We start this with the implementation of the Barrelfish OS on the M4. Afterwards we take a look at the implementation of atomics in Barrelfish, and the hardware quirks associated with it on the M4. We follow this with an explanation of the implementation of the capability transfer. Then we provide a detailed overview of the process of booting the M4 and what needs to happen for this. After that, we present the process of spawning applications on the M4 and the issues we are facing with it. And lastly, we present the implementation of the SCU API into Barrelfish OS.

5.1 Barrelfish implementation

In this section we present an overview of the challenges faced during the implementation of Barrelfish on the M4. We start with our interpretation of the paging system. Then we explain the differences between the application binaries on the A35 and M4. We follow this with some details on the CPU driver and the exception handling, followed by the required differences to the monitor application. We then end this section by detailing the process of passing messages and performing upcalls to the application.

5.1.1 Paging

As mentioned in subsection 4.1.1, we do not have true paging. Instead, we constantly change which regions are currently accessible. In order to adhere to the capability system, this is done using mapping operations. This is done so that any capability deletions will actually remove all derived capabilities, and as such also remove any access via mappings.

On systems with an MMU the mappings are done using page tables. These provide both address translations and what kind of permissions one has on the

memory. These page tables are then walked by the MMU on accesses to the virtual addresses. Barrelfish requires a capability type for each level of page table. Additionally, Barrelfish has a capability type for a mapping into each level of the page table which is called a VNode. When a frame is mapped into the virtual address space, we first need to find the page table that translates the virtual address and the corresponding VNode. We then map the frame into the VNode with a system call. This system call configures the MMU and creates a mapping capability in a provided location. The created mapping capability is used to ensure that a mapping is removed when either the frame capability or the VNode capability is destroyed. The page tables (and the corresponding VNode) are created in user space on demand and then mapped into the VNode of the higher level.

On the M4 the access is provided by the MPU by configuring memory regions with the permissions. The regions are set individually by writing two registers totaling 64 bits. To treat these MPU regions in the same way we would a page table, we create a new VNode capability type with a corresponding mapping capability type. The VNode capability contains 8 mapping slots (one for each MPU region). Each of these mapping slots represent one region of the MPU, and is 64 bits large. In each of these slots we store the bits we write into the RBAR and RASR registers.

Context switching

In a system with an MMU a context switch consists of changing the root of the page table and invalidating the TLB. This results in the address translation now using the mappings installed by the new domain.

For the M4 a context switch needs to disable any MPU regions mapped by the previous domain, and install the "mappings" of the new domain. This is done by iterating over the 8 slots of the VNode, and copying the saved values into the corresponding MPU region. By writing the values into the MPU registers, the new mapping overrides the old one. As such we do not need to explicitly remove the previous mapping.

5.1.2 Applications

The lack of an MMU also has consequences for the way we compile applications. For the A35s we can compile and link the applications statically. This means that the virtual addresses of most variables and functions are known statically. Therefore, Barrelfish can just load the binary, and relocate a couple addresses dynamically. This is possible as the address translation allows us to place the application at any physical address without changing the virtual addresses.

On the M4 we don't have address translation. As such, the only way we could statically compile and link the applications would be to reserve regions of memory for each program in advance. This would also mean that any time the application is changed, the space requirements might change, which could intern require more memory to be blocked of statically. Additionally, this would mean that the change to one application might necessitate changes to other applications as they could then overlap. We therefore chose to not attempt to statically allocate space for the applications, but instead relocate them dynamically.

This leads to several implications. The first one is that the binary expects a certain offset of the code and data segments that are loaded by default. In a system without address translation this requires the physical memory allocated for the segments to adhere to the same offset. Depending on the already allocated and potentially freed regions this might not be possible even though there is enough free memory to load each segment individually.

Alternatively, we can compile the application without PC-relative data. This results in each variable access going through an additional layer of indirection through the Global Offset Table (GOT). As such, each variable that is not on the stack now has an entry in the GOT, and not only global variables. This allows us more flexibility on the placement of the segments into memory. This however results in a more complex ELF loader.

To be able to handle an arbitrary offset for the relocation, we need to keep track of both the physical address where it was loaded into, and the virtual address that the segment requested. During relocation, the relocation entry specifies a virtual address of the symbol in the binary (where the variable is referenced), as well as an offset that the symbol has from virtual address 0. In a normal relocation, the offset would have to be added to the base address of the segment. However, due to the differing offsets of the segments, we need to only add the offset within the segments. For each relocation entry we have to apply this same logic to both the relocation address and relocation value. This is because the relocation address can be in either the data or the code segment. If it is in the data segment, the offset in physical memory will be different to the offset from 0 in the elf. When we have the correct address, we first read the value located there. This value will be the address of the symbol we are relocating. This can again be in the other segment, and as such will take a different offset.

5.1.3 CPU driver

Many parts of the CPU driver are shared with the ARMv8-A driver. This includes any capability operations and schedulers. These shared parts are called from architecture specific handlers.

Exception entry

As mentioned previously, the only time the CPU driver is entered is through exceptions. This results in some context being saved on the users stack. On entry, we need to also save the rest of the context in the dispatcher. When saving the context, we also save most of the context from the stack. We save it in order to have a consistent view of the execution state in the CPU driver to enable easier debugging. There are two registers we do not copy from the stack into the dispatcher save area. These are the xPSR, and the link register. We do not save the xPSR from the stack but instead save the xPSR after exception entry. This is because on exception return, the hardware does some checks on the current state, and that includes that the xPSR includes the current exception number that is not included in the one saved on the stack. The link register on the stack is used for function return addresses in the user code. On exception entry, it is overwritten with the exception return value that needs to be used on the return to restore the state correctly. As such we store the exception return value as we might dispatch another domain in between and would lose that information.

Waiting for work

When a user domain has no more work left, or wants to wait for some time, it can yield it's time to another domain. To do so it makes a system call, and the CPU driver can then dispatch a domain. If no domain has any work, we do not schedule any of them. Instead, we want to go into an idle state. ARM provides two instructions that can be used in such a case: `wfi`, and `wfe`. Both instructions let the processor enter into a low-power state after suspending execution, however `wfe` has a more limited wake-up events. As the CPU driver on other architectures we use `wfi` and loop on it. To resume from this state we need an asynchronous exception that would preempt the current execution. A synchronous exception cannot occur, as that requires the execution to cause it.

The most likely exception to cause us to resume is a SysTick exception as it is a regular timer. This leads to a problem as we might enter that state from a SysTick exception. As we would then be waiting during a SysTick exception, another SysTick cannot preempt us as an exception cannot preempt itself. Therefore, we need to go to a state where any exception can be taken. This means that we need to switch to Thread mode. As we are still in the CPU driver, and we loop on the instruction, we need to be able to execute the code there. This could be achieved by "mapping" that part into a user domain. In order to not disrupt any actual domain, we would have to create a dedicated domain to do so. This would be quite some overhead.

Instead, we switch to privileged Thread mode. This allows us to still execute the code of the CPU driver while not in an exception without needing to

configure the MPU. To do so we need to use an exception return with a stack built to resemble an exception entry. For this we first reset the stack to the beginning of the CPU driver stack, and build it up with nonsense parameters for the regular registers, and the address of the looping function as the return address. We can reset the stack as the CPU driver is stateless, and as such does not need the information stored on it previously. Of course by doing this we need to switch Thread mode back to unprivileged on exception entry.

5.1.4 Monitor

The monitor shares almost all code with the monitor for the ARMv8-A implementation. The most noticeable exception involves the passing of the urpc frame. On the ARMv8-A implementation and any other implementation using virtual memory, the frame is mapped into the monitor domain at a fixed virtual address. On ARMv7-M this is not possible as we do not have any address translation, and the frame is also only allocated at runtime and as such the address is not known at compile time. Instead, we need to find the address in an alternative way. As the capability is passed to the monitor at a well known address, we can use the capability to find the address. This requires a system call to get the physical address at which we can then initialize the urpc structures.

5.1.5 Message passing

As mentioned previously, we separate message passing into two distinct versions. The UMP (across the cores) is done almost equivalent to the way it is implemented for the A35s. We have a frame of shared memory, that includes 4 distinct channels. This is divided into two pairs of channels, one where the M4 is the server responding to requests, and one where the M4 is the client issuing the requests. These channels all are made up of 31 bytes of payload, and one byte for flags. This size was picked based on the cache line size of the A35s because the UMP relies on the cache coherency for performance, and as such we need the channel to fit into one line of cache. In contrast, the M4 cache does not have a cache coherence protocol implemented. Additionally, the cache on the M4 is not currently activated due to the issue explained in section 5.2. As such, the UMP is no longer really a form of message passing but rather a generic shared memory channel. For consistency, we continue to call it UMP.

For the LMP we rely on system calls to transfer the data. As such we pass the messages via arguments to the system call. These arguments are of size word size, which is 32-bits on the M4, and 64-bits on the A35s. Most RPC messages implemented on top are designed to fit into one LMP message or be at most $8 * 64 - bit$ long. This is too long for the M4 which is why we use double the LMP messages on the M4. Each LMP message includes the RPC

header, which is also 64-bit long. As such we split it in half and as such use one word more than on the A35s for each actual LMP transfer.

5.1.6 Upcalls

As mentioned in subsection 4.1.5, we require a dedicated stack in the dispatcher to perform upcalls to the domain. This allows the user code to decide what the appropriate stack for the entry point is. At some point, the entered thread will either have handled the exception, or we just entered the user space scheduler, and we resume a thread. This results in the registers being restored either from the threads save area where it was also stored from the user space and can just be reused, or it is restored from a dispatcher save area. That save area however, might be populated by the CPU driver on exception entry. This means that the xPSR, link register, and the stack register are not at the state that it was in on exception entry. As such, we need to simulate the exception return in user space. To do so, we need to check the value in the saved link register.

If it is an exception return value, we need to check which value it is to determine if only a basic exception entry context was saved or if also floating-point registers were pushed onto the stack. Depending on which one was used, the stack pointer needs to be increased by different values. Additionally, we might increase the stack pointer by an additional 4 as it was aligned on entry before pushing values. This is decided based on bit[9] of the xPSR. Additionally, as we are not returning from an exception to the return address, but rather as an unconditional branch, we need to make sure that the least significant bit of the value we load into the PC is 1. This is so that when we branch on that value, we do not attempt to switch to ARM mode as it is not supported on the M4.

5.2 Atomic operations in Barrelfish

As many other OSes, Barrelfish deals with a great level of concurrency. To make sure that this is handled correctly, atomic operations such as Test-and-Set (TAS) are used. This instruction sets a memory location to 1 and returns the previous value as if it were done in a single operation. In ARM this is done using load and store exclusive instructions. The load exclusive instruction (LDREX) loads the value stored in memory into a register and marks the physical address as exclusive access. The store exclusive instruction (STREX) stores the value in the register to memory if the processor has exclusive access to the address. If it does not have exclusive access, the store will not take place. STREX returns a 0 if the operation succeeded, and a 1 if it failed.

The TAS instruction is then compiled into the code shown in Listing 1. The code assumes that the address of the variable is located in r0, r2 is used to

hold the previous value used later, and `ip` is a scratch register used to check if the store failed. If it failed, line 5 jumps back to line 2 to load the value exclusively again. This ensures that execution cannot move past line 5 until the memory location has been updated.

```
1      mov.w   r1, #1
2      ldrexh  r2, [r0]
3      strexh  ip, r1, [r0]
4      cmp.w   ip, #0
5      bne.n  2
```

Listing 1: ARMv7-M assembly of TAS

On the M4 in the IMX8X however, this operation did not produce the expected result. After using the TAS instruction to set a variable to 1, if it was previously 0, we run into an assertion that the variable is now indeed set to 1. This assertion failed.

As previously seen, execution does not pass the TAS if it has not set the memory location to 1. As I had previously had some issues related to the cache, I decided to disable it to rule out any strange caching behaviors. With the cache disabled, the assertion succeeded.

A similar problem has been encountered by the development team of FreeRTOS as detailed in [8] and [5]. In their case, a Compare-and-Swap (CAS) was used to set a variable with more than two states. They found that while setting and reading the variable worked fine, the CAS did not see the updated value. This led them to suspect incorrect cache behavior.

To test this, they ran the same code in regions of cacheable memory, as well as the non-cacheable TCM. As the code in the TCM did not show the behavior, they tested the theory by manually controlling the cache.

If they write back (flush) the cache before the CAS, the CAS now sees the updated value. After the CAS the old value is still seen until the cache is invalidated.

This behavior was then reported to NXP in [5]. In that thread NXP confirmed that this was a hardware issue as the LDREX and STREX instructions overlooked the cache. This means that they always access the memory directly which leads to inconsistent states for the data. While this applies to the i.MX7 processor's M4, NXP said that it will be fixed for future M4 integrations. This leads to the assumption that they use the same M4 subsystem on all their SoC's and as such could also be the case for our i.MX 8X processor, especially because the error was only found a couple of months before our processor revision was released.

To confirm the results, we decided to test if the TAS modified the memory while not modifying the cache. For this we read out the value as seen on the M4, which will be the cached value, as well as from an A35 core where we mapped the address non-cacheable to read the value from memory. This resulted in the M4 reading a value of 0, with the A35 reading a value of 1. As such we concluded that our i.MX 8X has the same hardware issue as described above.

Since we still need the TAS for ensuring correctness of the execution, we need to explore options for overcoming the hardware issue.

The simplest solution would be to disable the cache entirely. This ensures correctness by ensuring every access is made directly to memory. However, it provides a performance penalty for the rest of the system. Due to time constraints, this is the solution I used.

Now we present other solutions that could be explored.

The most obvious solution would be to ensure that the MPU region that covers the variable accessed by atomic operations is configured as non-cacheable. This would ensure that all loads and stores to the variable go directly to memory, while other regions still benefit from caching. However, configuring the region attributes to non-cacheable (all possible ways to set this), does not change the behavior. The M4 still reads a 0, while the A35 still reads a 1.

The only way to change the behavior using the MPU, is by setting the memory type to Device or Strongly-ordered. These two types are never allowed to be cached. This is because they are used for memory regions where reads and writes can have side effects. This indicates that the implementation of the cache ignores the caching policy of the MPU for regions with memory type Normal, and always caches them.

Another way would be to ensure that the cache where the variable is located is invalidated after each atomic operation. This would require the entire cache line to be invalidated. As the variable most likely does not fill the entire cache line we might invalidate data that has not yet been saved. Therefore, we must clean the cache line before the atomic instruction as well. These cache operations are only possible from privileged execution. As such the user code would have to make a system call for each of them. This would result in a significant overhead for each atomic instruction.

Alternatively, we could make use of the hardware provided semaphore. While it is intended to provide synchronization across different cores, it could be used to provide a lock for modifying variables. By using a locking mechanism we could use normal load and stores, enabling caching to be enabled. However, as described in [7, Section 12.8.3], the semaphore can only be locked in supervisor mode. As such, this would also require two system calls for each atomic operation and not provide a benefit over the previous solution.

The previous two solutions required two system calls each. This could be reduced to one by replacing the atomic instruction with a system call. This would enable us to clean the cache, perform the operation and invalidate the cache and only then return from the system call. While this still includes the overhead of a system call, it removed half of the overhead.

Finally, the solution that NXP suggested in [5]. As the TCM is not cached, all atomic variables could be placed in it. In the typical application of the M4 this could be achieved using a linker script that ensures all variables are statically placed in the TCM. We however allow domains to be allocated dynamically at runtime, with each requiring at least one atomic variable in the dispatcher. Additionally, if the code would require more variables, a mechanism would be required to allocate a new variable at runtime. As we still want to protect the different domains from each other, each variable would require an MPU region to be allocated. The smallest allowed region is 32 bytes large, resulting in wasted space as most atomic variables used only require one bit. Additionally, we use the TCM for the boot and CPU drivers.

As mentioned in previously, the boot driver needs to be placed at the start of the TCM. But as it is not needed after we enter the CPU driver, the space taken up by the boot driver could be reclaimed. With the amount of memory the boot driver uses at the time of writing, we could allocate about 1400 atomic variables, assuming each variable gets their own 32 bytes. While this seems to be the solution that impacts performance the least, it would require the most modifications of the Barrelfish code. Additionally, with the variables now requiring their own MPU region, it might increase the number of system calls required to modify the MPU regions, potentially nullifying any performance benefit.

5.3 Capability transfer

As mentioned in section 4.3, we enable capability transfer between the cores by modifying the address for certain ranges. In Barrelfish this is done by sending the Metadata of capability being copied to the other core. This message is then received by the monitor on the target core, where it then invokes the CPU driver to create a capability from the Metadata. The CPU driver then creates the capability from the provided Metadata in the provided CNode slot. After creating the capability we check if the capability came from the other core, and if it did, we check if we need to transform the address of the capability. We check if the capability came from the other core by checking if the owner is not the current core. This is done, as we also use the same invocation when creating the capabilities when booting the first core. After determining that we might need to transform the capability, we call a platform specific function on the address of the capability. It is platform specific because each system has

their own memory map and as such can have a unique memory transformation. Therefore, if we want to use the same CPU driver for several platforms, as Barrelfish is intended, we cannot hard code such a transformation. This also allows for systems with a uniform memory layout. These will just have to implement the function so that it always returns the input.

5.3.1 Issues with current implementation

The current implementation assumes that we only ever transfer capabilities that need a transformation between cores where it is actually necessary. If we were to send a capability from between two A35s with an address of `0x1FFE0000`, the address would be transformed. While this situation is unlikely to occur on the i.MX 8X as the A35s do not have any resource there, other systems could have a map where different cores see different resources at the same address. In order to solve this problem, we would need to know which type of core the capability was transferred. Barrelfish already provides a way to identify the platform information. This includes both the ISA, and a platform identifier. This would have to be included into the transfer and invocation so that the transformation can be implemented without relying on the assumption mentioned before.

5.4 Booting the M4

This section explains the details of the boot procedure outlined in section 4.4. We first detail what the A35 is required to do to prepare the M4 for boot. Then we explain the actions the M4 is required to perform to boot and start the monitor application.

5.4.1 A35 side

The overall boot process is quite similar to the boot process of the A35s. Before we actually boot the M4 we need to allocate various objects. First we allocate a frame on the A35 that will be used for the communication between the cores. While on the A35s this frame can be mapped cacheable to take advantage of the cache coherency protocol, we cannot map it cacheable for usage with the M4. In part, this is because we do not use cache on the M4, and as such have no access to a cache coherency protocol. To allow most of the code to be reused, the mapping of the frame is differentiated based on the first character of the CPU driver name. This is possible as the implementation assumes that any multiboot module not starting with `"/` is prepended with `"/armv8/sbin/`. As the M4 is an ARMv7-M core, the binaries are located in `"/armv7m/sbin/` and therefore require the absolute path to be used, resulting in the first character always being `"/`. Additionally, the frame is used to

communicate the initialization parameters consisting of the RAM capability allocated to the M4, the bootinfo, and the multiboot info.

In the function `coreboot()` the architecture specific parts of booting the core are done. We first use the SCU to put power on the M4 core and the MU used to communicate with the SCU from the M4. This uses the SCU API function `sc_pm_set_resource_power_mode()` with the resources: `SC_R_M4_0_PID0`, and `SC_R_M4_0_MU_1A` respectively. As previously mentioned, the M4 always starts execution at the address `0x1FFE0000` which is the start of the TCM. To put the Boot driver there, we forge a capability for the TCM. Since the TCM is a region where the cores have a different view, we forge the capability with address `0x34FE0000`. The reason we forge it instead of just allocating it is that the A35s have no knowledge of the Memory region.

After we have allocated space for the Boot driver we also need to allocate space for the CPU driver. As the Boot driver most likely does not take up the entire TCM, we check to see if the CPU driver fits into the remaining TCM space. To ensure that any alignment requirements in the CPU driver are still valid, we align the start of the CPU driver space to the next page size boundary(4096 bytes). If it does not fit into the TCM, we just allocate space in RAM in the range accessible by the M4.

We also need to allocate space for the monitor after relocation with all other structures needed for a user domain to run such as a dispatcher frame. These structures will be allocated by the CPU driver on the M4 as well as the monitor relocation will occur there. As such we only need to allocate the space and pass it. However, in the current implementation, all the multiboot binaries are located in an address space that is too high to be read from the M4. To circumvent this we increase the size of the region allocated and copy the binary into the start of that memory region.

Also, we allocate the Kernel Control Block(KCB) used to store all the per core state. We additionally allocate space for the `core_data` structure together with the kernel stack. This structure is architecture specific, and as such requires certain datatype sizes. As we do not have access to architecture specific types except for the current architecture, all types have been replaced with `stdint` types. As an example, the type `lvaddr_t` that represents a virtual address on the local core is 32-bit wide on ARMv7-M, but 64-bit on ARMv8-A. For this case we replace it with the type `uint32_t` to be able to write it correctly from ARMv8-a. Alternatively, one could create these types for each architecture, and replace any usage of `lvaddr_t` with architecture specific types.

After allocating all required Memory we load and relocate the boot and CPU drivers. This is done using customized load and relocate ELF functions. The load function copies a single loadable segment into the previously allocated memory, and returns the address of the entry point in memory. If the ELF has more than one segment, or the provided entry point is not in the loadable

segment, we fail. After loading the binary into memory, we need to relocate them. As we do not have a means of address translation on the M4 we need to ensure a direct mapping. However, if we are relocating a binary loaded into the TCM, we need to ensure that the addresses are correct for the M4, and as we have a different view, we need to offset the relocation difference by the difference of the views on the Memory locations. We also have to do this for the entry point addresses that we pass to the M4. An additional problem with the relocation is that the TCM only supports 64-bit aligned writes from the A35s. The relocation addresses in an ELF32 however are only 32-bit wide. As such we make sure to first read the 64-bit region containing our relocation address, modify the relocation value, and write back the updated 64-bit region. In addition to the two entry points we also return the location of the boot arguments struct in the boot driver. This struct contains a pointer to the `core_data` and is used to identify if we are the first core to boot.

Now that we have all the accurate information about the location of the boot and CPU drivers, we fill out the `core_data` struct that we will pass to the M4. This struct includes the locations of the CPU driver stack, urpc frame, monitor binary, KCB, and global lock, as well as identifying information such as the ID of the new core and of who spawned it.

For the A35s, we start the core via a secure monitor call. This requires a call to the CPU driver where first the global lock data is filled out as it is a kernel only data structure, and then boots the core. For the M4, starting the core is again done via SCU API call. In our current state this is done in user space. Therefore, the only reason to invoke the kernel is to set the global lock data and then return to the monitor. There we start the core by calling `sc_pm_cpu_start()` with the resource `SC_R_M4_0_PID0`, and the address `0x34FE0000`(start of TCM as seen by A35s).

The A35 then waits for the M4 to boot by polling the urpc frame for the signal from the monitor on the M4. After receiving the signal we initialize the communication loops on said frame.

5.4.2 M4 side

Now for what we do on the M4 side.

We keep the Boot driver very simple. The only things the boot driver has to do is enabling the cache, enabling the MPU, and setting the correct stack. However, as we currently have the cache on the M4 disabled, we only really enable the MPU, after which we jump into the CPU driver.

In the CPU driver we first save the constants provided with the core data. This includes the core data itself, the core ID, the stack to enable stack resetting, the KCB, and the global locks.

We then initialize the serial console to enable output and change the exception vector table. It needs to be remapped as the vector table in the Boot driver has no knowledge of where the handlers will be, so we cannot use that table. We change the table by configuring the Vector Table Offset Register (VTOR) to hold the base address of the new vector table. After changing the M4 will look for the vector table at the address specified in the VTOR.

In combination with that, we also need to enable some faults. In particular, we need to enable the *UsageFault*, the *BusFault*, and the *MemManage fault*. The *UsageFault* catches any undefined instructions, or invalid execution states. The *MemManage* fault handles memory protection faults that are caused by instruction or data accesses to memory not mapped through the MPU. The *BusFault* catches the memory-related faults that are not covered by the *MemManage* fault. If these faults are not enabled, we will instead cause a *HardFault* exception, making exception handling more complicated. Additionally, we configure the *SysTick* timer, and enable the FPU for use in all privilege levels.

At this point the CPU driver spawns and runs the monitor. The first thing we do here is to create the capability database for this core. Then we create the *CSPACE* for the monitor along with a new dispatcher for it. We then create the capability for the cross-core communication as specified by the spawning core.

Afterwards, we load and relocate the monitor binary as described in subsection 5.1.2. It is loaded into the memory allocated by the A35 for this purpose. After relocation, we still need to get the location of the GOT. The address we get back from the ELF also still needs to be adjusted to the actual address. We then save the entry point, the GOT address, and the location of the user space dispatcher into the appropriate register save area of the dispatcher.

Now the only thing left to do in the CPU driver is to start executing the monitor. In Barrelfish, to change from the CPU driver to user space an exception return is used. As mentioned in subsection 3.5.2, ARMv7-M pops certain registers from the stack used. Therefore, we need to simulate exception entry by building the stack as if we entered using an exception. For this we use the upcall stack in the dispatcher that has enough space for the eight words, and is properly aligned.

The problem we have now, is that on starting we are not in Handler mode, so we cannot do an exception return. Instead, we are in privileged Thread mode. Therefore, we need to enter Handler mode, switch Thread mode to unprivileged, and "return" into the monitor. This is done using the software triggerable exception *PendSV*. In the handler for this we set the least significant bit of the control register to 1, and call *dispatch* with the monitor dispatcher.

When we enter user space, we update the stack to a stack in the data section and switch to C code. We then save the address of the user space dispatcher as we do not have a better way to let the user space know where it is. We then create the first thread and initialize the paging state and heap allocation.

Once that is done, we do a context switch into the created thread, and start executing the `main()` function and initialize our RAM allocation. At this point we signal the A35 that we are running and initialize the UMP with the A35.

5.5 Spawning on the M4

As mentioned previously, the spawning procedure is almost identical to how it is on other architectures. The main differences are how we load the binaries. The first problem with the binaries is their location. In the current implementation the multiboot modules are located in memory that is too high to access on the M4. This leads to two possible solutions.

The first is to ask the A35s to load and relocate the binary for us. The problem with this comes during the load of the ELF. While loading, memory is allocated for each loadable segment. This memory is also mapped into the paging state of the domain to be spawned. To do that, we would need a way to map regions on a remote core. That would require the A35 to invoke the CPU driver on the M4 at least twice. Additionally, as we also need to map them temporarily on the A35, we would have to deal with a mix of physical and virtual addresses during loading and relocation, leading to a more complex implementation.

Alternatively, we could ask the A35s to copy the binary into a specified location, and then handle the loading and relocating on the M4. To do this we need to send two requests to the A35. The first one is to request the memory region information for the multiboot region containing the binary. We need to do this, because while we do have all the information about location and size of each module, the name strings of the modules are located in too high memory. As such we cannot see if a region is the correct binary. After receiving the information, we allocate a region of memory to copy the binary into, and send it to the A35. Then we ask the A35 to copy the binary into the memory we sent over. While we could ask the A35 to allocate the memory directly and send it to the M4, that would require us to either revoke the memory after loading from it, or we would run out of memory eventually.

During loading the ELF, we also need to keep track of the addresses we mapped the segments into. This includes the physical address of the start of the segment for both the data and code segments. We additionally save the virtual address that the segment wants to be placed at. By using these, we can

calculate the address of any symbol in the ELF. This is needed to get the correct location of the GOT for the spawned domain.

After we have all the needed data at the correct location, we need to prepare the dispatcher for that domain for the first entry. As mentioned previously, we enter programs through an exception return. This means, that we need to simulate the exception entry. To do this, we use the upcall stack that is contained in the dispatcher structure, using the entry point of the ELF as the return address.

5.5.1 Issues with current implementation

In the current implementations there is still a very big issue, resulting in spawn being somewhat broken on the M4. The issue is that we either get a UsageFault or MemManage exception on the first print statement. This issue however, does not occur all the time. Instead, running the same binary multiple times results in one of the issues occurring approximately 50% of the time. The rest of the time, the program runs normally.

The following two sections provide more detail on the encountered faults.

UsageFault

The UsageFault exception is caused by an undefined instruction being executed. However, the faulting address points to the first instruction of the function `__get_locale()`, which is part of the libc implementation. The first instruction is supposed to be a push instruction, that saves some registers onto the stack that will be restored on return. From reading the faulting address, we see the word `4630f875`, which is not the push instruction (and also does not include it as many are half-word instructions). The fact that the same binary run multiple times only sometimes produces the fault, leads me to believe that this might be a case of memory corruption, or the relocation of the ELF might not have worked.

The first steps on debugging this issue, I would first check if the instruction is correct after relocating the ELF. If it is faulty the problem is either with the loading and relocating, or with the copying done by the A35. If it is correct then, I would put a watchpoint on the address of that faults and check where it gets changed.

MemManage

The MemManage exception is caused by a data access. This access occurs on addresses in the range of `0x100` to `0x110`. The exception is caused by the function `_once()`. This function is called by `__get_locale()` and is used to get some thread local data or if it is not present, initializes it. As this also

only sometimes happens, I assume that this happens to be the result of a race condition on setting the location of the data, or the result of the GOT entry being wrong.

The first steps on debugging this issue would be to check the GOT for correctness. If that is not the issue, I would find the address that points to the data and put a watchpoint on it. That way, we see when it is set to the incorrect location resulting in it being accessed. If it is never changed there might be an issue with the initialization of the structure that would have to be fixed.

5.6 SCU support in Barrelfish

As the SCU is the only component that can power on the Cortex-M4 we needed a way for Barrelfish to leverage the SCU to power up the Core for use. The System Controller Firmware (SCFW) running on the SCU exposes an API that can be used by all cores in the system. The clients (in our case the AP cores) make function calls using a RPC mechanism. This is built on an IPC layer. This allows the RPC layer to be independent of the hardware it is running on and depend only on the SCFW API version.

5.6.1 IPC

The IPC mechanism that is implemented by the SCFW uses the MU. The SCFW IPC uses all four transmit and receive registers to create a single bidirectional channel. Each message starts in the first MU transmit register. If the message contains more than one word, they are written into successive registers, and if the message is greater than four words, we wrap back to the first register. Should the next transmit register not be empty, the transmission blocks.

The response is handled in the same way. It always starts in the first receive register while the following words go into the successive receive registers and wrap back to the first if the message is greater than four words. If the next receive register is not empty, the transmission blocks.

The API is fully synchronous and should not be preempted or be timed out to not confuse the IPC state in the SCFW. Not following this could result in an error. The SCFW does allow multiple RPC provided that each uses a unique MU. However, this does increase the latency of one RPC if it is already processing another RPC.

5.6.2 RPC

The API calls all construct a series of 32-bit words to form an RPC request. The request consists of a one word header and up to 7 additional words containing parameters. The call then send the request, and almost all calls receive

a response of up to 8 words. The response is then deconstructed and returned to the caller.

The header consists of four bytes:

- version: Protocol version number
- size: Number of words in message including header
- svc: Service index. Differentiates what type of call it is.
- func: Function index within the service.

5.6.3 Integration into Barrelfish

As the API is used to control the entire SoC, access should be restricted. The best way to do this, would be to integrate this into the CPU driver. We could then create capabilities for the API with different access permissions and give programs the appropriate capability. Any prohibited call would then be prevented during the capability invoke.

This would however have taken a lot of time, and made using it more difficult. As Barrelfish is a research OS, security is not of much concern. I therefore implemented the API as a user space library. This means that any application could include it and get access to the SCFW API, provided that they can access the MU.

For the actual implementation I used the SCFW porting kit version 1.7.4. The porting kit contains ports of the client API for ATF, FreeRTOS, Linux, QNX, and u-boot. I chose to use a slightly modified version of the ATF code as it was the most compact implementation. Nevertheless, I still consolidated all RPC-related headers into a single header.

For the underlying RPC implementation I wrote a simple device driver for the MU. The only functionality the driver needs for the API is the ability to write to the transmit registers, and to read from the receive registers. This is implemented in a blocking manner that spins on the flags indicating register full/empty respectively.

To make use of the API the following code has to be executed first. For simplicity error handling has been omitted.

5. IMPLEMENTATION

```
1  sc_ipc_t ipc;
2  errval_t err;
3
4  struct capref ipc_cap;
5  err = slot_alloc(&ipc_cap);
6
7  err = dev_caps_acquire(IMX8X_AP_TO_SCU_MU1_A, IMX8X_MU_SIZE,
8  ↪ &ipc_cap);
9
10 void *scu_mu_base;
11 err = paging_map_frame_attr(get_current_paging_state(),
12 ↪ &scu_mu_base, IMX8X_MU_SIZE, ipc_cap,
13 ↪ VREGION_FLAGS_READ_WRITE_NOCACHE);
14
15 sc_err_t sc_err = sc_ipc_open(&ipc, scu_mu_base);
```

Listing 2: Initialize SCFW API

Afterwards all API calls can be used. As an example to turn on UART 0 the following code would be used as shown in the API porting guide.

```
1  sc_pm_clock_rate_t rate = SC_160MHZ;
2  /* Powerup UART 0 */
3  sc_pm_set_resource_power_mode(ipc, SC_R_UART_0,
4  ↪ SC_PM_PW_MODE_ON);
5
6  /* Configure UART 0 baud clock */
7  sc_pm_set_clock_rate(ipc, SC_R_UART_0, SC_PM_CLK_PER, &rate);
8
9  /* Enable UART 0 clock */
10 sc_pm_clock_enable(ipc, SC_R_UART_0, SC_PM_CLK_PER, SC_TRUE,
11 ↪ SC_FALSE);
```

Listing 3: SCFW usage example taken from [6, Section 13.4.1]

For a complete list of API calls please refer to the reference guide [6].

Chapter 6

Evaluation

In this chapter we present an evaluation of the performance of some parts of the Barrelfish implementation on the M4. We will compare the performance to the equivalent parts of Barrelfish on the A35s.

6.1 Communication performance across cores

As Barrelfish relies on passing messages between the monitors on each core to keep the shared state, such as the capabilities in the system, consistent, the communication channel should be as performant as possible. This benchmark aims to determine how performant both a message channel from the first A35 to the M4, and from the first A35 to a second A35 is. Since the cores do not share performance counters or timers, we cannot measure the timing on the receiving side. We therefore measure roundtrip time on the first A35 core. This is measured in cycles using the PMCCNTR_ELO performance counter from ARMv8-A. As we are measuring a communication channel, there are two interesting performance parameters: latency and throughput. These two parameters are handles in the following subsections.

6.1.1 Latency

To determine the latency we have, we send the smallest possible message to the second core, and wait for the answer that is just as small. The smallest message available in our implementation consists of only a header.

The data generated by this experiment is plotted in Figure 6.1. This figure uses box plots to visualize the data. The whiskers represent the 1.5 IRQ, and the notches indicate the 95% CI around the median. Additionally, we use a logarithmic scale ranging from 10^4 to 10^6 .

As Figure 6.1 shows, the latency to the M4 is around 400000 cycles while the latency to the second A35 is around 17000 cycles. This means that the latency

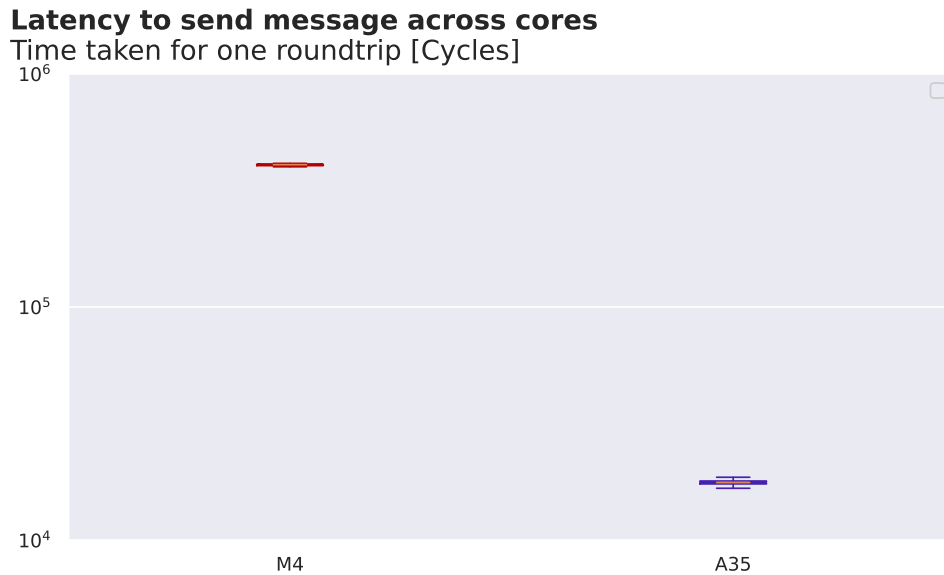


Figure 6.1: Latency of sending an empty message from an A35 core to a second core.

to the M4 is more than a full order of magnitude higher.

6.1.2 Throughput

To determine the throughput we have, we send messages of varying size to the second core, and wait for an answer that is as small as possible. The payload we are sending, are strings of varying sizes. The smallest string we send is 64 bytes long, and we then increase the payload size by 128 bytes up to 1728 bytes. We choose 1728 as the largest payload, as it is just 7 bytes smaller than the largest supported payload, and it is a nicer succession. Additionally, this should show our maximum possible throughput as the header overhead is amortized. However, as many messages are shorter, we are also interested in the smaller payload sizes.

While we measure the time it takes to send the message to the other core in cycles, we want to present the throughput in bytes per second. To achieve this we first divide the cycles by the clock speed on the core. We then divide the payload size by the time in seconds and adjust it to MB per second to make the numbers more manageable.

Figure 6.2 shows the results of this experiment. We can see that for smaller payloads we are almost two orders of magnitude worse when sending to the M4. We can also see that from payloads larger than 800 bytes the throughput barely increases. This results in a maximum throughput of 6.6 MB/s for

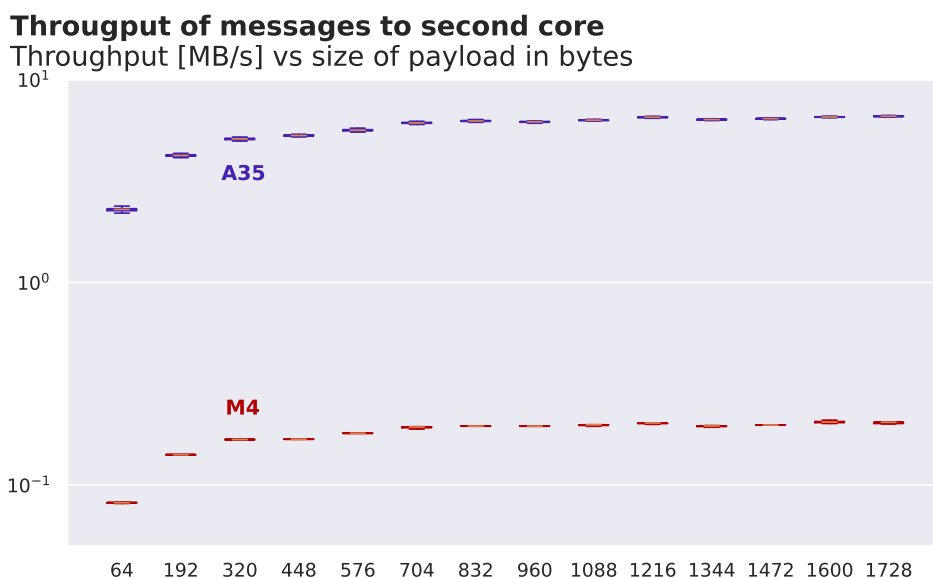


Figure 6.2: Throughput of sending messages from an A35 core to a second core.

sending data from the A35 to the second A35, and 0.2 MB/s for sending data from the A35 to the M4.

6.1.3 Conclusion

As seen from the two previous sections, sending data to the M4 performs at least an order of magnitude worse than what is achieved for the A35s. The reason for this is fairly simple. The communication between two A35s benefits from the cache. They write data into the cache, and the cache coherency protocol ensures that the receiver only accesses memory when the data is already there. The rest of the time they poll on the cached data.

With the M4 however, we always have to access memory directly. This results in expensive access while waiting for the channel to be free, so we can write the next part of the data.

This can be seen in Figure 6.3. There we repeat the throughput experiment again with two A35 cores. The difference however, is that we mapped the channel as non-cacheable. This shows that while it is significantly worse than cached, it is still an order of magnitude better than with the M4.

This shows that, while caching is a contributing factor, it is not the main bottleneck. Instead, I suspect that it is rooted in the clock speed difference. The A35 needs to wait until the M4 has read the data, and cleared the channel flags. As the clock speed is 4.5 times as slow, each of these operations takes 4.5

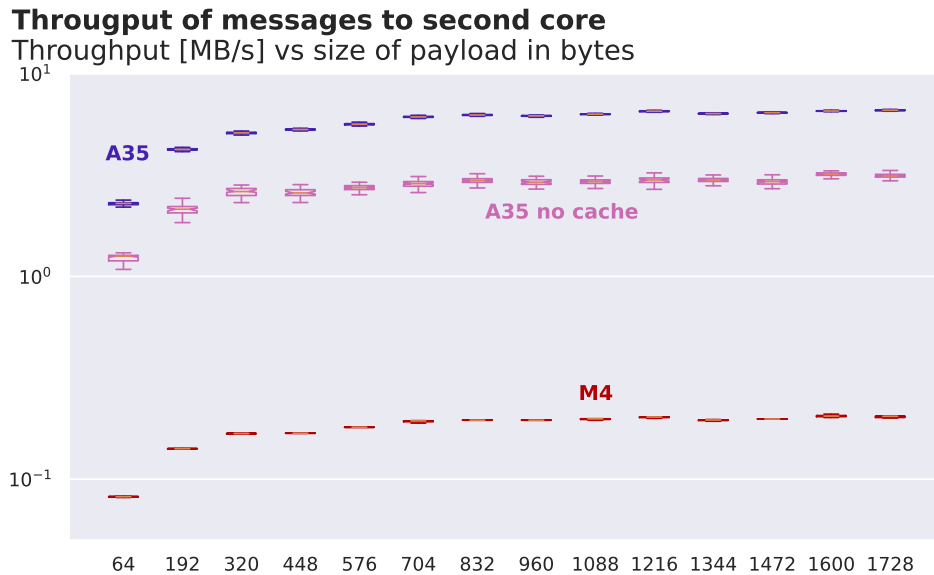


Figure 6.3: Throughput of sending messages from an A35 core to a second core with added no cache variant of A35.

times as long. Another contributing factor is the compilation of the binaries. For the A35, we compile and link the binaries statically. This means that most memory locations are known before running the application. For the M4 we cannot compile and link the binaries statically. This results in any access to the variables in the code having to go through an extra level of indirection. First we read an entry in the GOT, and then we have the address of the variable and can access it.

6.2 Time to boot

In this benchmark we take a look at the time it takes to boot the M4 vs how long it takes to boot a second A35. We want to know if a simpler core such as the M4 is able to boot quicker than the A35 core. We measure the booting process in two parts. The first part measures the time from when the start signal is sent, until the second core lets us know that they are ready via the UMP channel. The second part measures the preparation before the start signal to see if the different requirements make a difference in performance. For both of the boots we measure the time in cycles on the first A35 core using the PMCCNTR_ELO performance counter. To get accurate data we do poll on the flag signalling that the second core is up instead of sleeping as this results in the processor entering a `wfi` making the counter inaccurate.

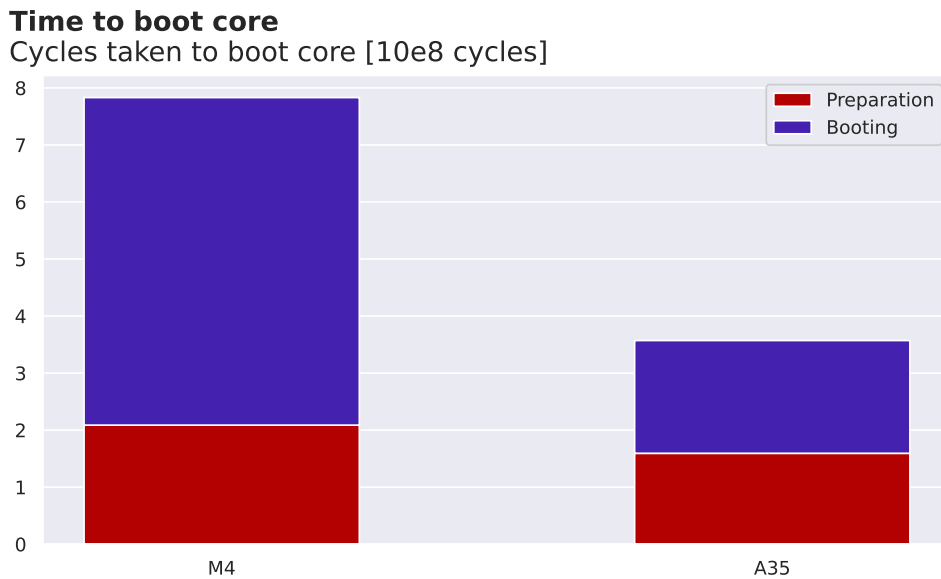


Figure 6.4: Time to boot cores in cycles

In Figure 6.4 we can see the time taken to boot both an A35 core and an M4 core. The red bar includes all preparations required for booting a core up to sending the start signal. As can be seen, preparing the M4 for boot takes about 30% more cycles than preparing the second A35.

The blue bar shows the number of cycles taken from sending the start signal until the booted core signals they are up and running. This shows that the M4 takes 2.9 times as many cycles as seen from the A35s.

While the difference in time might seem large, we need to consider that the booting process, while simpler on the M4, does a similar amount of work. As the clock speed of the M4 is about 4.5 times slower than that of the A35, taking only 2.9 times as long is quite good. As for preparing the core for boot, we can see that the M4 is more complex to work with. This is mostly due to the boot and CPU driver having to be located in the TCM. The TCM requires all accesses from the A35 to be aligned in a way that relocating an ELF32 to read and write twice the amount of data for each location. Additionally, it is located farther from the A35 than normal memory, and as such takes a bit longer to access.

6.3 User code performance

In this benchmark we compare the performance of user code on the M4 vs the A35. Having a reference on how much of a performance difference we have between the M4 and the A35 can help decide for which kind of code would be best allocated on which core. In this benchmark we use different performance counters as we measure on the core we run the code on. That means that on the A35 we once again use the `PMCCNTR_ELO` performance counter, while on the M4 we use the `CYCCNT` register in the DWT module. Both of these count CPU cycles, but as we have different clock speeds, we convert the result to seconds.

In order to have a comparable situation, we run the benchmark on the second core, as there might be some interference from the first core. As mentioned in subsection 5.5.1, `spawn` is currently broken on the M4. Therefore, we run the code directly in the monitor on both the M4 and A35. While this means that the performance measured here might not be the exact same as when run in a separate domain, it should be a close enough approximation. If anything, the performance should be slightly better, as the monitor domain has the entire core for itself, and other domains cannot interfere with it.

In this benchmark we used two simple programs to see how much slower the M4 is for a user application. The first application is a simple greatest common divisor implementation. The second application is intended to compare the floating point execution on the cores. In order to keep it simple it computes the average of an array of floating point values. As the M4 only supports single precision floating point operations, we stick to using the data type `float`.

In Figure 6.5 we can see the time taken to calculate `gcd(10000, 11)`, and for calculating the average of 12000 floating point numbers. The results from running on the M4 are depicted in red, while the results from the A35 are depicted in blue. This shows that the M4 takes two orders of magnitude longer for the same calculations.

While it was expected that the programs on the M4 are significantly slower, it was not expected to be this significant. With a 4.5 times slower clock speed, we would expect the programs to take around 4.5 times as long. To this we can add some more factors that might have contributed. The first one is the disabled cache. While we do not have data on the benefits of caching on the hardware, if it is comparable to the cache in the A35, that would only mean a speedup of 2x. Another contributing factor will also be the added indirection from the GOT, although this would also not account for this type of overhead. At the time of writing I do not know the root cause of this massive overhead.

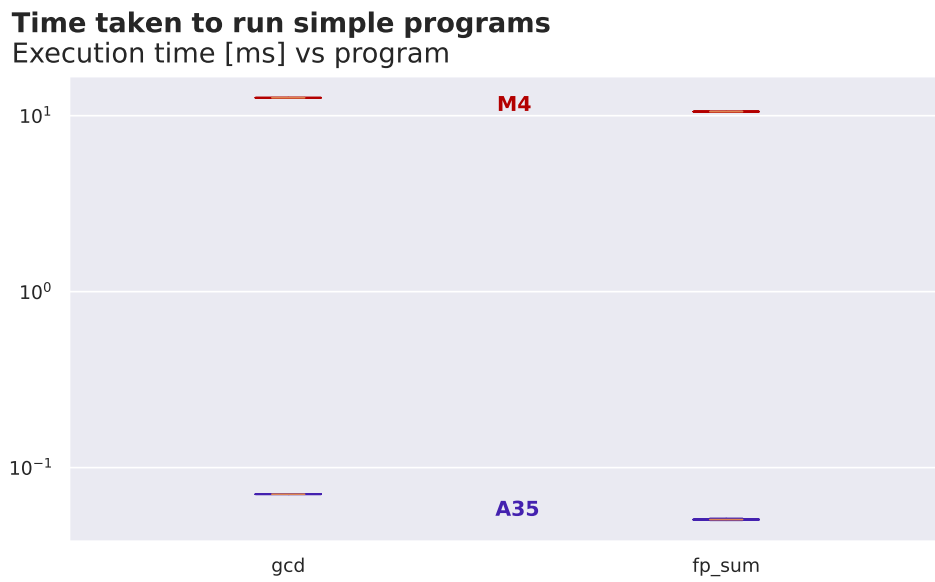


Figure 6.5: Performance results of user programs run in the monitor domain

Chapter 7

Conclusion

This chapter provides an overview of what was achieved in this thesis, and a list of future work that will need to be completed in order to fully utilize the M4 on the platform.

7.1 Summary

In this thesis we have extended the Barrelfish implementation used in an OS course at ETH Zurich to run on a heterogeneous platform with cores of different architectures. In the process of this thesis we have ported the Barrelfish CPU driver to a new ISA in form of ARMv7-M in order to also run on the ARM Cortex-M4. Additionally, the Barrelfish implementation on the M4 can communicate with the A35 cores present in the system. This required several changes to the communication infrastructure to deal with the two different word sizes. It also required the system to handle the different views of memory from the cores. This resulted in the design and implementation of a capability transformation that is used when capabilities are copied to another core.

7.2 Future work

In this section we provide an overview of the work that still needs to be done. Some of them are required to complete the implementation, some of them are performance optimizations, and some are features to make the system more user-friendly.

- **Fixing spawn:** In the current implementation, spawning on the M4 is partially broken. For a detailed description of the problem please refer to subsection 5.5.1. To summarize, the spawn has three outcomes: the application runs normally, the application generates page faults on inac-

cessible addresses, or the application executes an undefined instruction. Which outcome we see is nondeterministic.

This issue is major as spawning is a very large part of a usable Barrelfish implementation. While this is not fixed, we are limited to running code in the monitor on the M4.

- **Capability transfers:** As mentioned in subsection 5.3.1, the capability transformation on transfer makes some assumptions on the number and type of cores running in the system. While this works on the i.MX 8X, other systems could benefit from such a capability transformation in the future. In order to deal with arbitrary systems and core numbers, the system call would have to be extended with information about the sending core.

While this is currently a minor issue, this would turn into a major issue when we want to use this on other platforms.

- **Locks on the M4:** In the current state of implementation, the M4 runs without caching enabled. This is because the current hardware does not support locks at arbitrary addresses with caching enabled. The reason for this is that, as described in section 5.2, the load- and store-exclusive instructions used to implement locks ignore the cache entirely and always go to memory. The simplest solution for this was to disable the cache entirely, but this has performance implications for all software running on the M4. As such, this issue falls into the category of performance optimizations, that would make using the M4 more appealing.
- **Communication across cores:** As mentioned previously, Barrelfish depends on message passing for communication between the cores. To be performant, this relies on a cache coherency protocol to notify the receiving core about new data. If one core is the M4, we cannot use caching for this protocol, and as such achieve an order of magnitude lower performance. The i.MX 8X however has hardware components that are intended for communication across the cores. These MUs might be able to improve the performance of the UMP implementation and as such would be worth exploring. This would however require a redesign of the UMP implementation as we need to change the protocol based on the type of the target core, resulting in multiple protocols being in use at the same time. This issue also falls into the category of performance optimizations, and could make the M4 more usable.
- **LMP:** This point is a consequence of the change to UMP mentioned before. In the current implementation all messages are optimized for the ARMv8-A implementation. If we were to redesign the UMP implementation, it might be worth it to also split the LMP implementation to be core specific. This would then require a conversion of message

formats for sending messages to another core, but could result in faster message passing between domains on the M4.

- **Locks across cores:** The current implementation requires the M4 to have a dedicated UART output. The reason for this is that the CPU driver (and as such also debugging output) uses global locks to serialize their output onto one UART. This global lock is a simple memory location, where to take the lock the processor uses the load- and store-exclusive instructions provided by the ARM ISAs. While this should also work to prevent the M4 writing to the lock while the A35 is also writing, and as such prevent both cores from holding the lock, this does not happen. There are two possible causes for this. The first one is that the exclusive instructions are not working as intended. The second one is that the A35 caches the lock, and as such can have a different view of the memory than the M4, and as such both can take the lock simultaneously. The second issue could be solved by simply not caching it on the A35. The first one would require hardware locks to be used such as the semaphore explained in section 5.2.

This issue is in the category of making the system more user-friendly as solving it would allow all output to be delivered over one cable instead of two.

- **Program performance:** As seen in the benchmarks of the user code we can see that in the current implementation, the M4 takes more than two orders of magnitude longer than the same code on the A35. With the M4 being this slow, it is not very usable to run applications on the M4 instead of the A35. Solving this issue would be very helpful in making the M4 more usable.

Bibliography

- [1] ARM. *Cortex-M4 Devices Generic User Guide*, Version 1.0 b.
- [2] ARM. *Armv7-M Architecture Reference Manual*, Version E.e.
- [3] Simon Gerber. *Authorization, Protection, and Allocation of Memory in a Large System*. Doctoral thesis, ETH Zurich, Zurich, 2018.
- [4] Nora Hossle. Multiple address spaces in a distributed capability system. Master thesis, ETH Zurich, Zurich, 2019.
- [5] NXP. i.mx7d: atomic compare and swap instructions don't work with cache. <https://community.nxp.com/t5/i-MX-Processors/i-MX7D-atomic-compare-and-swap-instructions-don-t-work-with/m-p/700599>. Accessed: 08.08.2023.
- [6] NXP. *System Controller Firmware API Reference Guide i.MX8QXP Die (Ver 1.29)*.
- [7] NXP. *i.MX 8DualX/8DualXPlus/8QuadXPlus Applications Processor Reference Manual*, Rev. 0, 05/2020.
- [8] rschaefer.tech. i.mx7 m4 atomic cache bug. <https://rschaefer.tech.wordpress.com/2018/02/17/imx7-hardware-bug/>,. Accessed: 08.08.2023.



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Exploring heterogeneous OS architecture

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):

Buitendijk

First name(s):

Dennis

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

Zürich, 13.09.2023

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.