Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor's Thesis Nr. 98b

Systems Group, Department of Computer Science, ETH Zurich

Applying the Multikernel Approach to a Heterogeneous OMAP4460 SoC

by

Claudio Föllmi

Supervised by

Timothy Roscoe
Stefan Kaestle

March 2013 – September 2013

Informatik
Computer Science

Abstract

Modern computers increasingly resemble networks, with processor and memory layouts becoming less symmetrical. Traditional operating systems were built on the abstraction of a single program running on identical cores with identical views of memory. A multikernel on the other hand does not depend on such assumptions, making it suitable to run on heterogeneous architectures.

In this thesis, we apply the multikernel approach to the OMAP4460 SoC. We ported the Barrelfish operating system to the Cortex-M3 microcontroller and ran it together with the existing Cortex-A9 port. We present the challenges with porting a general purpose operating system to a microcontroller and discuss our design decisions.

Contents

1 Introduction

Modern hardware is becoming more and more heterogeneous, and can be expected to become even more so in the future. Not just because of the benefits of having different processors to choose from, but also because many specialized devices like GPUs and network cards increasingly resemble specialized processors. Advanced microcontrollers already provide much of the functionality of a regular CPU. Microcontrollers included in a system in order to drive other devices may be repurposed and treated as a processor, if the operating system can cope with heterogeneity.

In this thesis, we will look at the Cortex-M3 subsystem on the OMAP4460 system on a chip. We discuss how a general purpose operating system can be made to run on that microcontroller, with the example of our port of the Barrelfish operating system to the Cortex-M3.

In a classically designed system, all processor cores are identical and connected to each other in the same way. A heterogeneous system on the other hand contains cores that are noticeably different from each other [1]. This can manifest in many forms:

- They may run at different clock speeds.
- They may support different instruction set extensions.
- They may have access to separate memory regions.
- They may be connected in a non-uniform topology.
- They may support different core instruction sets.

Introducing heterogeneity into a system allows to use the advantages of the individual architectures without committing to just one of them. The recent boom in GPU-computing for massively parallel workloads has shown both the advantages of and demand for using specialized hardware for specialized tasks [6].

As more internal devices – such as network cards – increasingly resemble specialized processors, the demand to use them for other purposes will grow. In mobile devices, having a small and low-power core for standby operation while also having a more powerful core for heavier workloads can decrease the overall power consumption without sacrificing performance. It may even increase peak performance, because it allows the big core to be much more powerful than what one would choose if it had to be powered all the time.

Increasing heterogeneity reflects the trend of single computers more and more resembling computer networks. But traditional operating systems still assume perfect uniformity and a single shared view of memory. The multikernel [3] approach on the other hand embraces heterogeneous and non-coherent memory architectures.

2 Related Work

In 2012, Le Sueur and Rodgers ported SMP Linux to the OMAP4430 (the direct predecessor of the OMAP4460) as a heterogeneous architecture, by providing a homogeneous abstraction to the application layer and most of the kernel [9]. This proof that it is possible to run a heterogeneous, general purpose operating system on the OMAP4460 was a big factor in our decision to try a similar project with Barrelfish. But where Le Sueur and Rodgers abstracted the heterogeneity away, we instead decided to expose it to all parts of the operating system.

The first Barrelfish port to ARMv7-A was done by Hitz for the Gem5 simulator [8]. This system was later ported to the OMAP4460 and is the basis for our ARMv7-M port. A heterogeneous Barrelfish port for x86_32 and x86_64 in one machine was done by Menzi [10].

3 Background

3.1 OMAP4460 SoC

The OMAP4460 [12] is a system on a chip (SoC) by Texas Instruments, intended for use in consumer devices like smartphones and tablet computers. It contains:

- A dual core ARM Cortex-A9 processor
- Two ARM Cortex-M3 processors
- A hardware spinlock module
- A mailbox module
- Many devices to process media input and output

The intention is that the Cortex-A9 will be running a general purpose operating system, while the Cortex-M3 processors will only be running a real-time operating system to control the imaging subsystem.

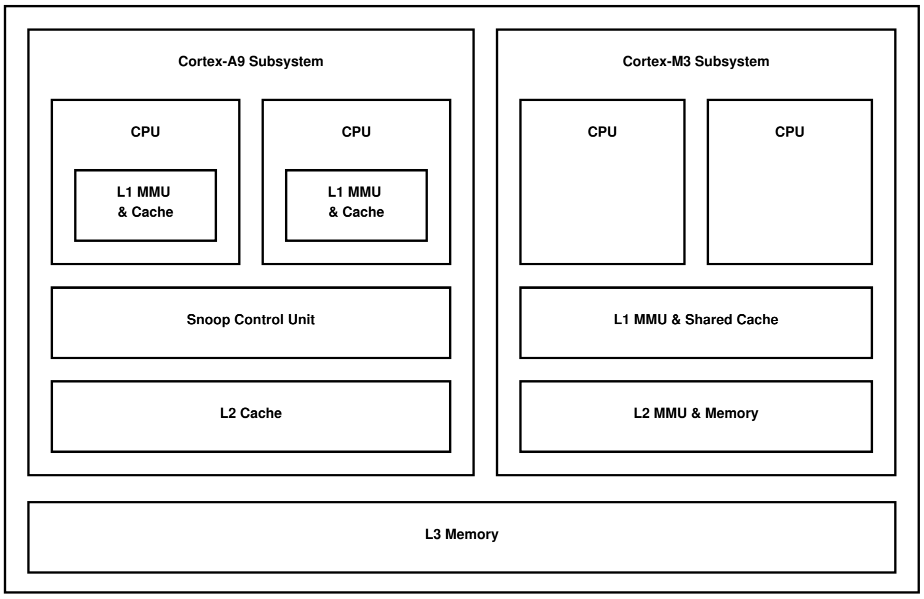

Figure 1: The memory layout for both the Cortex-A9 and Cortex-M3 subsystems.

Figure 1 shows the overall memory layout. Both subsystems can access all physical memory over the L3 interconnect. The A9 subsystem contains a snoop control unit to ensure cache coherence between the two cores. Both Cortex-M3 processors share the same cache module, which implies coherence. But there is no cache coherence between the two subsystems, making the use of shared memory both complicated and expensive.

For this thesis we used a PandaBoard ES revision B1, which is a USB-programmable development board for the OMAP4460.

3.2 Cortex-M3 subsystem

Because the Cortex-M3 [5] processor is intended for use as a microcontroller, it lacks many features that a general purpose processor would contain. On the OMAP4460, the Cortex-M3 subsystem contains additional devices to compensate for the lack of processor features, most notably a cache and a memory management unit (MMU) with a hardware pagetable walker. The two Cortex-M3 processors share all the devices of the subsystem.

3.2.1 Memory model

The memory model for the Cortex-M3 subsystem on the OMAP4460 is arguably its most unusual feature. As the processor itself does not contain a MMU, all memory translation is done through additional devices. Instead of one MMU doing all translation and permission work, there are two MMUs connected in series:

- A MMU for the cache, which is completely software loaded and does region based permission and translation. This can be bypassed by disabling the cache.
- A MMU for accesses on L2 and L3 memory. This MMU can walk two-level pagetables in hardware, and features a 32-entry TLB, but it does not handle access permission.

The combination of these two MMUs allows to use both page tables and memory protection, but fine grained permissions like per-page read-only have to be handled in software.

The different stages of a memory access can be seen in Figure 2.

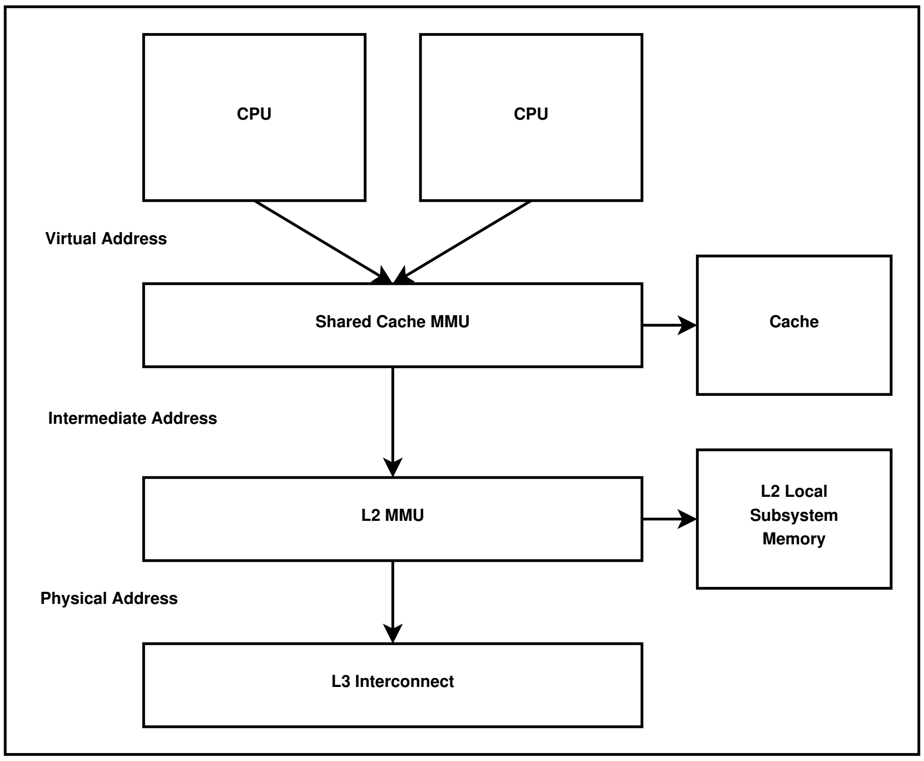

Figure 2: The different stages in address translation and access in the Cortex-M3 subsystem.

Because both MMUs can translate addresses, most addresses will be translated twice, with the notable exception of a few special sections that are not in L3 memory:

- A private section within the processor itself, containing control registers for the nested vectored interrupt controller (NVIC).
- A section in L2, containing control registers for the shared cache and L2 MMU as well as a bit of RAM.
- Two bit-banded sections of virtual address space, which are aliased to other parts of the virtual address space by the processor itself.

These special cases directly affect not only the kernel but also applications:

- Some virtual addresses will never point to memory, because they bypass the MMU. Unprivileged writes to control registers will be ignored, so untrusted code will not be able to interfere with the kernel. Applications that try to use these addresses will probably crash.
- Some configuration registers can not be mapped to a different part of the virtual address space.

Even without a memory protection unit (an optional processor feature), the Cortex-M3 checks all memory accesses using a default map of permissions. This means that all memory accesses need to satisfy two policies, one static and checked by the processor, the other configurable and checked by the shared cache MMU.

As both processors share the same cache module, they necessarily share their complete virtual address space (except for the local section within the processors themselves).

3.3 ARMv7-M Profile

The ARMv7-M [4] processor profile is specifically designed for microcontrollers, focusing on simplicity and predictability rather than general performance. It does not support regular ARM instructions, but only Thumb2, a more compact instruction set containing both 16-bit and 32-bit instructions.

Instead of ARMv7-A's eight execution modes, it only features two: a privileged handler mode and a thread mode that can be set to be privileged or unprivileged, with separate stack pointers for each.

It features the same registers as ARMv7-A, though with a few differences in usage. The general purpose registers (r0 - r12) are separated into low (r0 - r7) and high (r8 - r12) registers. For 16-bit instructions, only low registers may be specified. The three special purpose registers are the same as in ARMv7-A: the program counter (pc, r15), the link register (lr, r14) and the stack pointer (sp, r13). The pc and lr are not allowed to be loaded in the same instruction.

A big difference is the program state register xpsr, which stores:

- The current status flags
- The currently active exception vector (0 if in thread mode)
- A bit indicating that we are executing Thumb2 instructions
  (can be changed but generates exception if not 1)
- Information for interrupted load/store multiple instructions and If-Then-blocks. This part of the register is normally inaccessible, but can be set on exception return.

Thumb2 instructions are similar to ARM instructions and there is a unified assembly language that can be used for both instruction sets, but generally its immediates are smaller, branch ranges shorter, and fewer registers are allowed to be specified. Instead of directly allowing any instruction to be conditional, all conditional instructions have to be inside If-Then (IT) blocks, which are generated by the IT instruction.

To indicate that a given instruction is in Thumb2, every load to the pc (regardless of what instruction is used to achieve it) must set the least significant bit to 1 – effectively jumping to an odd address. Failing to do so will trigger a fault.

3.3.1 Exception model

One of the biggest differences between ARMv7-M and ARMv7-A is the simplified exception model. Whereas ARMv7-A uses several different modes depending on the type of exception, with differences in the banked registers, ARMv7-M uses one mode for all exceptions, with only the stack pointer optionally banked.

For each possible interrupt, the interrupt vector table holds the address of the corresponding exception handler. By default, the vector table is at address 0, but it can be relocated to addresses up to 0x2FFFFF00 (but must be aligned to its size).

When an exception is taken, the following sequence of events happens:

- The corresponding entry in the vector table is read.
- The beginning of the interrupt handler is prefetched.
- Registers r0-r3, r12, lr, pc and xpsr are pushed on the stack.
- The processor transfers into handler mode, a special return address is loaded into lr.
- The exception handler starts executing.

If these operations trigger a fault (e.g. the vector table entry is invalid), a hard fault is triggered. If a hard fault can not be handled, the processor stalls.

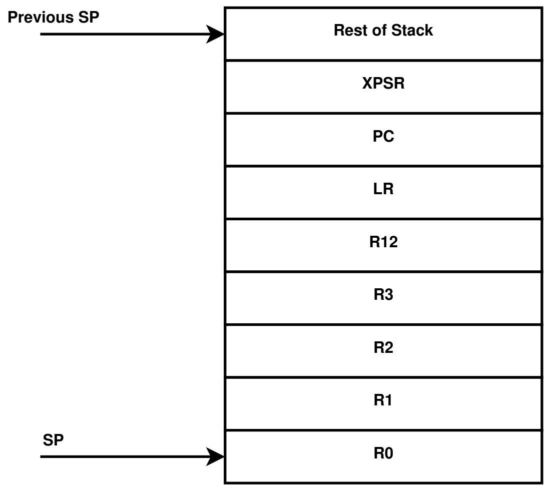

Figure 3: The layout of the stack when the exception handler is entered.

Figure 3 shows the layout of the pushed registers on the stack. The set of pushed registers corresponds to the caller-saved registers in the function call standard (with the addition of xpsr), so the exception handler could be a regular C function.

The only way to exit handler mode is to load the pc with a special value of the form 0xFFFFFFFX — with X being either 1, 9 or D to specify mode and stack to return to — which triggers the following sequence of events:

- The registers r0–r3, r12, lr, pc and xpsr are restored from the specified stack.
- The processor transfers into the specified mode, which can be either thread mode or handler mode.
- Execution resumes.

One very important detail is that the context is always pushed on the stack of the mode that triggered the exception, rather than that of the mode entered. This means that stack faults are generally unrecoverable, as the attempt to push the registers will trigger another fault.

3.3.2 Synchronisation

The ARM and Thumb2 instruction sets provide exclusive load and store operations (ldrex and strex), that are monitored by the memory bus. Because the subsystems use separate buses to connect to the L3 interconnect, they can only enforce exclusivity within a subsystem (either the Cortex-A9 or the Cortex-M3 subsystem), but not across the subsystems. They can therefore not be used for cross-architecture synchronisation.

To allow the different subsystems to synchronize easily, the OMAP4460 contains a hardware spinlock module. The module is very simple: it contains 32 memory mapped registers that each represent one lock.

When read for the first time, they read as "0", meaning the lock was free before and has now been acquired by this read operation. All subsequent reads will return "1", meaning the lock is already taken. Writing to the register frees it up again.

This very simple device allows spinning on what appears to be memory, using regular accesses instead of specialized instructions. As all the registers are in contiguous addresses on the same page, the individual spinlocks can not be isolated – a process with direct access to one of them is able to access all.

3.4 Barrelfish

Barrelfish is a research operating system, developed by researchers from ETH Zurich and Microsoft Research. Its core concept is that a multicore or multiprocessor machine can be treated as a distributed system – a multikernel [3].

Processor cores run independently, sending messages for communication instead of relying on shared memory. Common state information is replicated instead of shared. The goal is to build a system that will scale well with the increasing number of processor cores by treating a single machine more like a network.

Because the individual kernels (called CPU-drivers) are running independently, they do not all have to run on the same processor architecture or have a common view of memory (such as cache coherence). As all sharing of information is done explicitly through message passing, Barrelfish lends itself to heterogeneous architectures.

3.4.1 Devices

To facilitate the writing of device drivers, Barrelfish uses a domain specific language (DSL) called mackerel, which compiles device specifications into large header files of inline functions that access all the registers. This separates the logic of the driver from the definitions of the register layout and access rules.

3.4.2 Message Passing

Barrelfish uses different protocols for message passing, depending on what the architecture supports and whether the message is sent to a process on the same core or on a different core. The protocols are specified in a DSL called flounder, which separates the decision which protocol to use and protocol logic from the architecture specific implementation.

3.4.3 Dispatchers

Barrelfish not only schedules processes in the kernel, but all processes also schedule their own threads using a standardized system called a dispatcher [2]. At any given time, a process can either be executing a thread or the dispatcher. When a process is scheduled, execution resumes in the dispatcher instead of the thread that was interrupted.

3.4.4 Capabilities

Barrelfish uses capabilities [7] for memory management. A capability can refer to a kernelspace object or a region of physical memory. All allocation of virtual memory is achieved in userspace through the invoking and manipulating of capabilities.

3.4.5 Services

Similar to a microkernel, Barrelfish uses the CPU-driver only for scheduling, isolation and interrupt handling. Most architecture independent functionality is instead provided by userspace services such as the monitor (which communicates across cores) or a memory server.

3.4.6 Hake

Barrelfish uses a DSL called hake to specify how a target should be compiled. Every directory of the source tree contains a hakefile describing how to build the system of that subtree, and each target architecture defines a set of rules about what compiler options to use. From this information a big makefile containing the build rules for all necessary files is automatically generated.

4 Design and Implementation

In this section, we show how a general purpose operating system can be designed for hardware configurations like the Cortex-M3 subsystem on the OMAP4460, and discuss the implementation details of our Barrelfish PandaBoard port from ARMv7-A to ARMv7-M.

As there already was a working ARMv7-A port for the PandaBoard, we could heavily base the new port on existing code. The Barrelfish codebase is separated into architecture independent and architecture specific code. Some of the ARMv7-A specific code could be used without any changes at all, and many only needed small adjustments, since the external devices (such as the serial port) are the same.

The biggest changes were necessary in the code relating to kernel bootstrapping, exception handling, context saving/restoring and memory management. But even for these cases the ARMv7-A code was a helpful guide.

4.1 General design decisions

In principle, the Cortex-A9 and Cortex-M3 share a common subset of instructions and could potentially execute the same binaries. This is the approach taken by the SMP Linux port: all code was compiled to the common subset in order to abstract away the differences between the processors [9].

However, one of the biggest advantages of the multikernel approach of Barrelfish is that the individual CPU-drivers do not need to be identical or even similar at all. Our decision to build separate CPU-drivers for the Cortex-A9 and Cortex-M3 meant that we could use the existing gcc targets for these processors, and that the existing Cortex-A9 image could be largely unchanged.

We decided not to activate both Cortex-M3 processors, because they share their virtual memory space. There is no way to isolate two processes running simultaneously on both (though running two threads of the same process would arguably be possible). More importantly, because the shared cache MMU does not distinguish between privileged and unprivileged accesses, the two processors would have to synchronize all kernel entries and exits, leading to a massive overhead.

4.2 Memory Management

The L2 MMU contains a hardware walker for two-level page tables, whereas the shared cache MMU only supports region based mappings and is completely software loaded. To get any reasonable performance, the L2 MMU should be used for all translations, and the shared cache MMU only be used for access control.

Because the ARMv7-A MMU contains two separate bases for page tables, the existing code relied on the ability to reserve the second table for kernelspace mappings, only exchanging the first table when switching between processes.

To reuse this existing code with our L2 MMU, which only has one table base, we decided to replicate the kernelspace part on all page tables ever used. A counter is put into the very first table entry and is incremented on every change to a kernelspace mapping. If the page table is then exchanged for a table with a lower counter, the whole upper half of the table (which corresponds to all kernelspace mappings) is copied over.

This way, the currently active page table always contains the most recent kernel mappings. This approach is very coarse-grained, but it turns out that the kernelspace mappings are hardly changed at all and need to be replicated in some form anyway. Putting the counter directly into the table entry is not a problem, because the L2 MMU ignores at least a full byte of all entries, and we locked the mapping for page 0 in the TLB anyway (so the MMU will never look at that table entry).

4.3 Memory Protection

The kernel of any general purpose operating system must be able to protect itself from untrusted userspace programs, and also to protect these programs from each other (isolation). Since the L2 MMU does not feature any protection mechanism, we have to use the shared cache MMU instead. To protect kernel memory from userspace access, we can use the fact that only privileged writes can change the chache MMU policies:

- We reserve some of the regions in the shared cache MMU for the region of memory where the kernel resides. The number of regions necessary depends on how much of the kernelspace we are actually using.
- We use a small region at virtual address 0 for the vector table, and mark it as readonly.
- We use another small region for the exception entry- and exit code, and mark it as executable but not writeable.
- Whenever we exit the kernel, we first remap the kernel regions to point to userspace addresses.
- Whenever we enter the kernel, we first remap the kernel regions 1:1 to the correct physical addresses.

This way, whenever a userspace application tries to access kernel memory, it accesses userspace memory instead.

The smaller regions are necessary, so the code to change the kernel region's permissions can be executed from a trusted area. Optionally, the vector table and the exception handlers can be mapped by the same region.

4.3.1 Access Policies

Because the shared cache MMU is completely software loaded and does not support page tables, while the L2 MMU that can walk tables does not support permissions, the question arises how one can implement the classical permission bits (such as read only) in a reasonable way.

These permissions are not strictly necessary for general purpose operating systems, but they have been standard for so long that many advanced OS techniques rely on them (e.g shared libraries, copy on write).

A L2 page table entry contains many bits that are ignored by the MMU. These can be used by the kernel to store additional information about this mapping, such as access permission policies. Entries with policies other than "read, write, execute" need to be marked as invalid, so the first access will generate a pagefault. The pagefault handler then needs to perform the following operations:

- Look up the address that generated the fault in the page table.
- If the corresponding entry has the special permission bits set, add the mapping into the TLB.
- Add a mapping for the faulting address into the shared cache MMU, with the specified access policy and no translation.
- Restart the faulting instruction.

When the process resumes, it will be checked by the shared cache MMU and then hit the TLB instead of looking at the page table again.

The big advantage of this approach is that the page table itself is not modified. The mapping will be removed on the next TLB flush, such as during a context switch. As the shared cache MMU only has two permission bits (readonly, execute only), we can only enforce these permissions.

One subtlety of this approach is that context switching potentially becomes much more expensive. The next time we switch back to the old context, its special permission mappings will all have been flushed, so the first access will again lead to a page fault.

Because of time constraints, we were not able to bring up caches on the Cortex-M3. This also means we did not have time to actually implement the memory protection scheme as detailed here.

4.4 CP15

ARMv7-A processors contain a coprocessor 15 (CP15), that can be accessed with special assembly instructions. It provides direct access to system configuration registers, including the ones to control the MMU and cache. As the Cortex-M3 does not contain such a coprocessor, we have to use memory mapped registers instead.

One big design decision was whether we should keep the names of the CP15 functions, effectively emulating the device for the rest of the system, or if we should introduce new names for the new functions instead. We decided to introduce new names, for several reasons:

- The functionalities provided by our devices and the CP15 are not the same. Some distinction about what functions we can use must therefore be made anyway.
- Explicit invocation of either CP15 or devices actually simplifies porting, because it shows which parts of the kernel rely on the devices.
- Providing a device to the kernel that does not actually exist makes the architecture harder to understand.

One way to simplify the distinction would be to fundamentally change the hardware abstraction layer. When we only used ARMv7-A, it was a reasonable assumption that there would always be a CP15 (and a GIC, and a system timer). Now that we introduce ARMv7-M, which provides that functionality in a different way (on the OMAP4460 – other systems might not have any MMUs or caches at all), we might need to make the abstraction layer more generic.

4.5 Synchronisation

When implementing a driver for the hardware spinlock module, it became clear that the kernel and applications have very different requirements for synchronisation:

The kernel only needs to synchronise a fixed set of elements known at compile time. In the current Barrelfish PandaBoard port, this only consists of the serial connection. Applications can require an arbitrary number of locks, and can not be trusted with access to locks belonging to other processes.

4.5.1 Kernel Synchronisation

The synchronisation primitives provided in the ARM/Thumb2 instruction sets (ldrex and strex) only work within a subsystem, meaning they can be used to synchronize between the two Cortex-A9 cores but not with the Cortex-M3. This narrow scope of language features is a general problem of current heterogeneous systems, and the OMAP4460 solves this through a separate spinlock module. In the absence of such a module, it would for example be possible to run consensus algorithms over the message passing interface (provided that interface does not itself require synchronisation), but that would be very inefficient.

We implemented a simple kernelspace driver for the spinlock module, which allows the kernel to acquire or release a specified lock easily. The code to access the memory mapped registers is automatically generated from a device specification, using the mackerel language.

4.5.2 Application Synchronisation

The downside of the spinlock module is that all the locks are on the same page. We can not allow regular applications to directly access it, because they would be able to acquire locks intended for other processes or even the kernel.

It is out of the scope of this thesis to implement a userspace locking service, but there are two approaches that would integrate well with Barrelfish:

- Introducing lock capabilities, that could be invoked to acquire or release locks.
- Introducing a locking service that could be accessed through normal message passing.

Either one would be able to manage the locks for all applications and could abstract away the actual implementation for each architecture. If the locking service was centralized, the act of message passing would already serialize the request, so it would only need to keep track of the locks' state.

4.6 Exception Handling

As the ARMv7-M exception model is much simpler than the ARMv7-A one, writing exception handlers is very easy. In principle, the exception handler could just be a C function, but for Barrelfish we want more control, so the first part is written in assembly.

All handlers should start by looking at the link register. The special return address is different depending on whether the exception was triggered in thread mode or handler mode. For exceptions generated in thread mode, the handler should then save the interrupted context, part of which has to be read from the thread stack. In Barrelfish, the context save area depends on whether the process was executing a thread or dispatcher code.

After the context has been saved, all registers are free to be used. Because the first part is the same for all exceptions other than service calls, we can use the same handler for all of them, putting the exception-specific code into a C function called by the handler.

Returning to thread mode is achieved by writing a special value into the pc. The very simple and clean exception model would easily allow us to make the CPU-driver preemptive, but there is no obvious reason to actually do so.

4.7 Interrupt Controller

As the nested vectored interrupt controller (NVIC) is very different from the generic interrupt controller (GIC) used on ARMv7-A, we had to write the driver for it from scratch.

The NVIC register layout is specified using the mackerel language. The driver is mainly concerned with initialization, because the NVIC does not need to be regularly accessed if it is properly set up. While the GIC requires acknowledgements from the CPU, the NVIC is more tightly integrated and monitors the entering and exiting of exception handlers autonomously.

The NVIC also contains a SysTick timer module as a subdevice, which can be configured in the same memory mapped region. SysTick can generate timer interrupts, by counting the number of clock cycles of the CPU.

System faults (entries 2 - 16 in the vector table) are enabled as part of the kernel bootstrap procedure. External interrupts (entries 17 - 80) are not enabled by default, but can be enabled through the driver.

The much smaller set of external interrupt lines (64 on the Cortex-M3 compared to 128 on the Cortex-A9) raises the question of how a heterogeneous system should handle interrupt asymmetry. Two reasonable approaches would be to either have the Cortex-A9 forward interrupts to the Cortex-M3 through message passing, or to not spawn any applications on the Cortex-M3 that rely on interrupts it does not receive.

4.8 Entering Handler Mode at Startup

When the Cortex-M3 comes out of reset, it executes the reset handler in privileged thread mode rather than handler mode. To correctly separate the kernel from userspace, we need it to always be executing in handler mode. To achieve this first transition, the bootstrapping function that initializes exception handling is temporarily registered as the handler for system calls. Thread mode is restricted to unprivileged execution, and then a system call is triggered. The rest of the kernel bootstrapping is then done in handler mode.

4.9 Saving and Restoring Context

4.9.1 Basic Context Flow

In Barrelfish, there are several places where registers are saved or restored: A yielding thread will save its context, then the dispatcher will schedule and restore another thread. When entering an exception handler, all registers are stored to a safe location. There are two regions they can be saved to, depending on whether it was the dispatcher or a thread that was running at the time of the interrupt.

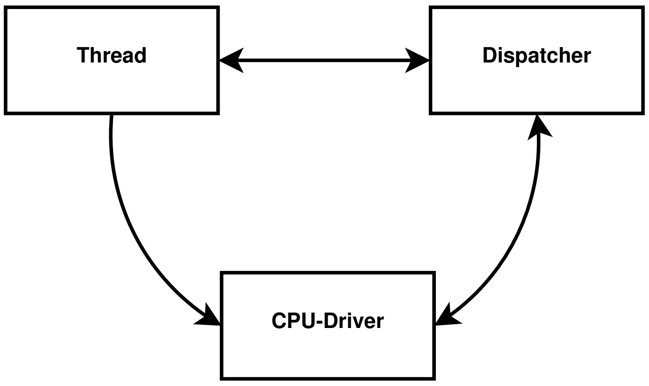

Figure 4: Control flow between threads, dispatchers and the CPU-driver. Instead of directly resuming preempted threads, the CPU-driver gives control to the dispatcher instead, which can then restore the thread or schedule another one.

Figure 4 shows the flow of control, which is also the flow of saved and restored contexts. The CPU-driver can be entered from either a thread or the dispatcher, but when returning to userspace, it will always give control to the dispatcher.

If the dispatcher was interrupted, it will be restored at the point of interrupt and continue from there. If it was a thread that was interrupted, the dispatcher is entered at a fixed location (upcalled) and it can decide which thread to resume.

To achieve these transitions we need to implement:

- A kernelspace function to save the interrupted context
- A kernelspace function to restore a context
- A userspace function to save a yielding thread's context
- A userspace function to restore a context

All four cases are written in assembly. As Thumb2 is more restrictive about the loading and storing of registers, the ARMv7-A code for the dispatcher could not be directly used but had to be modified.

When restoring a context in userspace, we have the problem that the stored pc value can not directly be loaded – not only does it contain the wrong value (lsb 0, indicating ARM mode), it can also not be used in a ldm instruction together with the lr register. To cope with this problem, we push the corrected pc on the stack, together with a register we use to point to the saved context.

```
/* r0 points to dispatcher struct, r1 to saved context */
/* Re-enable dispatcher */
mov     r2, #0
str     r2, [r0, #(OFFSETOF_DISP_DISABLED)]
/* restore sp and lr first, because they can not be used with ldr */
ldr     r0,  [r1, #(SP_REG  *4)]
mov     sp,  r0
ldr     r0,  [r1, #(LR_REG  *4)]
mov     lr,  r0
/* Restore apsr condition bits  */
ldr     r0, [r1, #(CPSR_REG *4)]
msr     apsr, r0
/* read pc and r1 values and push them on stack */
ldr     r2,  [r1, #(R1_REG  *4)]
ldr     r3,  [r1, #(PC_REG  *4)]
/* make sure lsb is one (force thumb mode) */
orr     r3,  #1
push    {r2, r3}
/* Restore registers */
ldr     r0,  [r1, #(R0_REG  *4)]
ldr     r2,  [r1, #(R2_REG  *4)]
ldr     r3,  [r1, #(R3_REG  *4)]
ldr     r4,  [r1, #(R4_REG  *4)]
ldr     r5,  [r1, #(R5_REG  *4)]
ldr     r6,  [r1, #(R6_REG  *4)]
ldr     r7,  [r1, #(R7_REG  *4)]
ldr     r8,  [r1, #(R8_REG  *4)]
ldr     r9,  [r1, #(R9_REG  *4)]
ldr     r10, [r1, #(R10_REG *4)]
ldr     r11, [r1, #(R11_REG *4)]
ldr     r12, [r1, #(R12_REG *4)]
/* pop r1 and pc, leaving no register clobbered */
pop     {r1, pc}
```

As we will discuss in 4.9.2, this is still not quite enough to restore all possible contexts, but it does cover all contexts that can be saved by the userspace save function. The userspace save function is much simpler, because it is allowed to clobber the caller-saved registers.

Restoring a context in the kernel is special, because it is done through a mode switch. This means that the caller-saved registers will be restored from the thread mode stack by hardware, and the callee-saved registers need to be restored beforehand. In some cases, such as when a process is run for the first time, the stack pointer might not be initialized yet. We then need to initialize the stack pointer, so we can push the necessary register values.

```
//r0 already points to saved context in memory
ldr     r1, [r0, #56]                    //stored stack pointer
sub     r1, #32                          //allocate stack space for 8
    registers
//copy the 8 expected registers
ldr     r2, [r0,#(R0_REG *4)]            //copy r0 entry
str     r2, [r1,#(R0_REG_PUSHED *4)]
ldr     r2, [r0,#(R1_REG *4)]            //copy r1 entry
str     r2, [r1,#(R1_REG_PUSHED *4)]
ldr     r2, [r0,#(R2_REG *4)]            //copy r2 entry
str     r2, [r1,#(R2_REG_PUSHED *4)]
ldr     r2, [r0,#(R3_REG *4)]            //copy r3 entry
str     r2, [r1,#(R3_REG_PUSHED *4)]
ldr     r2, [r0,#(R12_REG *4)]           //copy r12 entry
str     r2, [r1,#(R12_REG_PUSHED*4)]
ldr     r2, [r0,#(LR_REG *4)]            //copy lr entry
str     r2, [r1,#(LR_REG_PUSHED *4)]
ldr     r2, [r0,#(PC_REG *4)]            //copy pc entry
str     r2, [r1,#(PC_REG_PUSHED *4)]
ldr     r2, [r0]                         //copy xpsr entry
str     r2, [r1,#28]
//set thread stack pointer to saved context
msr     PSP, r1
//restore unpushed registers: r4-r11
add     r0, #(R4_REG *4)                 //point to r4 entry
ldmia   r0, {r4-r11}                     //restore r4 - r11
ldr     lr, =#0xFFFFFFFD                 //special return address to change
     modes
bx      lr                               //actual context switch
```

This is the only time in the whole system that we do not have to enforce the lsb of the pc value to be 1, because the information that we want Thumb mode is in the xpsr register instead. The epsr part of that register can normally not be read or written (reads return 0, writes are ignored), but it is pushed on exception entry and read on exception return, making this the only way to set the IT bits outside an IT block.

4.9.2 Special Cases

It is tempting to assume that the two context restore functions will only ever be called with contexts that have been created by the corresponding save functions. This would be especially convenient for the userspace one, because then we could use the fact that the userspace save function clobbers registers before saving.

In fact, the restoration code in both the kernel and the dispatcher is also called on artificially built contexts, the first time a new process or thread is scheduled. Most importantly though, the userspace restoration code can be called with a kernelspace saved context, which turned out to be a big issue.

The context restoration on exception exit is strictly more powerful than all other means to load registers, because it is the only time that the epsr part of the xpsr register can be written (all other attempts will simply be ignored). The epsr part is used to store information about interrupted ldm/stm instructions and IT blocks. Restarting an interrupted ldm/stm instruction is safe, because the processor reverts changes to registers that would lead to different outcomes on restart (such as address writeback). But interrupted IT blocks are not safe to restart, because they can contain arbitrary state-changing instructions (such as incrementing register values).

In short, the userspace context restoration code will be called on contexts that can only be restored by the kernel. To handle this, we had to introduce a new system call that can be used to restore arbitrary contexts.

4.10 Hake Target ARMv7-M

In order to compile the ARMv7-M port, we introduced a new hake target architecture called armv7-m. It is heavily based on the existing armv7 target, with adjusted compiler options to force Thumb output. Because hake allows several target architectures in the same build tree, compiling the files for both ARMv7-A and ARMv7-M is no harder than just compiling for ARMv7-A.

4.11 Creating a Bootable Image

To create a single image that can be transmitted to the PandaBoard over USB, the ARMv7-A port of Barrelfish uses a tool called molly. During compilation, molly parses a list of modules to include and generates C code that will set up a corresponding multiboot_info. The modules (ELF object files) are then stripped of unnecessary sections and transformed into opaque blocks of data. The final image consists of

- The functions to create a multiboot_info object
- Libelf functions to load and relocate ELF images
- Bootstrap code that loads the kernel into memory, writes the multiboot_info and starts the kernel
- Images of all the specified modules

To create a single image for our heterogeneous system, we decided to create two such molly images and nest one in the other. An otherwise unused field of the multiboot_info struct is used to tell the ARMv7-A kernel where the second image is located.

While the cores are not yet able to spawn applications on each other, this separation of modules allows them to each take care of their own applications. When cross-core application spawning becomes possible, this approach should be replaced by a single molly instance containing the modules for both architectures.

4.12 Starting a Cortex-M3 Core

Before the Cortex-M3 can start executing code, the following steps have to be taken by the Cortex-A9:

- Power on the Cortex-M3 subsystem
- Activate the Cortex-M3 subsystem clock
- Load the image to be executed into memory

- Enable the L2 MMU
- Set up mappings for the loaded image in the L2 MMU (can be written directly into the TLB)
- Write the first two entries of the vectortable (initial sp and reset vector)
- Take the Cortex-M3 out of reset

It is important to note that the Cortex-M3 is in a virtual address space from the very beginning, reading the vector table at virtual address 0. Inserting a 1:1 mapping for the kernel image greatly simplifies the bootstrapping of memory management on the Cortex-M3 once it is running, because it needs to know the physical address of the page tables it sets up.

4.13 Debugging

While the OMAP4460 contains many debugging features, we did not have the special hardware necessary to access them. All debugging was achieved through serial output.

For the early stages, we could rely on the working Cortex-A9 to print information on the Cortex-M3's status. As soon as serial output on the Cortex-M3 worked, we used it for all kernelspace related debugging messages. Because the exception and system call transitions were implemented later, we could not print in userspace right away. To find out if a line of application code was reached, we would insert a division by 0, and check if the exception was triggered.

This was especially important when implementing and debugging the system call transitions, because these had to be written in assembly and were therefore error-prone. While this way of debugging was far from convenient, it was sufficient to find out which part of the code needed reviewing.

5 Evaluation

Because we did not have time to bring up caches, the performance of the Cortex-M3 can not really be compared with the Cortex-A9 yet. While the Cortex-A9 finishes setting up all userspace services in under 3 seconds, the Cortex-M3 takes either 40 seconds (using -O2) or 2 minutes (using -O0) for the same task, depending on the level of compiler optimization.

Comparing userspace application performance of the two processors, preliminary tests suggest a factor of 30 for memory-unintensive workloads. Interestingly, workloads consisting almost entirely of integer divisions only show a factor of 10 (including the kernel overhead). This is because the Cortex-A9 uses a software division algorithm (up to 500 instructions), whereas the Cortex-M3 has a divider in hardware (up to 12 cycles). This suggests that, with proper caching, the Cortex-M3 might even surpass the four times faster clocked Cortex-A9 for very specific workloads (which are probably unrealistic).

In total, the implementation of the ARMv7-M port of Barrelfish required 5300 lines of code, but many of these are from new files that only slightly differ from their ARMv7-A counterparts. The code used for debugging (which has since been removed) required another 800 lines.

6 Conclusion

We successfully implemented a Barrelfish port for the Cortex-M3 subsystem on the OMAP4460, that can be executed on the PandaBoard.
More specifically, we can

- Bootstrap the CPU-driver, initializing all necessary devices
- Print to the serial console, synchronizing accesses using the spinlock module
- Start all the regular Barrelfish userspace services, such as init, monitor, mem_serv, skb, ramfsd, spawnd and startd
- Execute userspace applications, such as memtest
- Handle exceptions such as timer interrupts or page faults

Because of time constraints and the inconvenient debugging procedure, we could not implement as much as we would have wanted. The current implementation still lacks the following features:

- Support for caching and the shared cache MMU
- Support for the mailbox module
- A cross-architecture communication interface

Because of these limitations we currently treat the Cortex-M3 as a bootstrap processor, with its own multiboot_info and without knowledge of the Cortex-A9. This corresponds to running two separate instances of Barrelfish on the same machine, instead of one instance running as a multikernel. In the finished Barrelfish port, the Cortex-M3 should only be an application processor, getting its information about what processes to start directly from the Cortex-A9.

Because we have not enabled the caches, we can not implement the protection feature detailed in 4.3.1 yet. It also means that the system is currently running very slowly.

Our experience with the Cortex-M3 subsystem has shown that the gap between regular processors and microcontrollers can be bridged with external devices, such as MMUs and communication modules. But the bridging would be much easier if these devices were chosen with operating systems in mind [11] – in this case, not separating protection from translation in the MMUs.

7 Future work

There is much potential for future work, on support for heterogeneity in general and the OMAP4460 port of Barrelfish in particular.

- Implementing a cross-architecture message passing channel using the mailbox module.
- Based on that message channel, degrading the Cortex-M3 to an application processor and executing Barrelfish as a truly heterogeneous multikernel on both architectures.
- Activating caches and implementing the memory protection mechanism discussed in 4.3.1
- Activating caches and doing a systematic performance analysis, comparing the Cortex-A9 to the Cortex-M3:
  - Comparing the performance of services, such as system calls and messaging
  - Comparing the responsiveness to external interrupts
  - Comparing the performance of various workloads
  - Analysing the impact of merely having the Cortex-M3 running on the performance of the Cortex-A9

The last point is especially interesting, as this is where we expect the multikernel approach to shine. Because only messages and the spinlock for printing are shared between the processors, in the extreme case of the Cortex-M3 idling (only executing applications that do not print or send messages), the Cortex-A9 should not be performing any worse than if the Cortex-M3 had never been started up.

References

[1] Andrew Baumann Timothy Roscoe Paul Barham Tim Harris Adrian Schuepbach, Simon Peter and Rebecca Isaacs. Embracing diversity in the barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems*, June 2008.

[2] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. *ACM Trans. Comput. Syst.*, 10(1):53–79, February 1992.

[3] Pierre-Evariste Dagand Tim Harris Rebecca Isaacs Simon Peter Timothy Roscoe Adrian Schuepbach Andrew Baumann, Paul Barham and Akhilesh Singhania. The multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on OS Principles*, October 2009.

[4] ARM. *ARMv7-M Architecture Reference Manual*, 2006 - 2010. Revision C.

[5] ARM. *Cortex-M3 Technical Reference Manual*, June 2011. Revision r1p1, Version E.

[6] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: a gpu-accelerated software router. *SIGCOMM Comput. Commun. Rev.*, 40(4):195–206, August 2010.

[7] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, October 1988.

[8] Samuel Hitz. Multicore armv7-a support for barrelfish. Bachelor's Thesis, August 2012.

[9] Etienne Le Sueur and Simon Rodgers. Operating system support for the heterogeneous OMAP4430: A tale of two micros. In *13th Linux.conf.au*, Ballarat, Australia, jan 2012.

[10] Dominik Menzi. Support for heterogeneous cores for barrelfish. Master's thesis, ETH Zurich, July 2011.

[11] Baumann A.-Roscoe T. Mogul, J. C. and L Soares. Mind the gap: reconnecting architecture and os research. In *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, 2011.

[12] Texas Instruments. *OMAP4460 ES1.x Technical Reference Manual*, February 2011. Revision Y, March 2013.