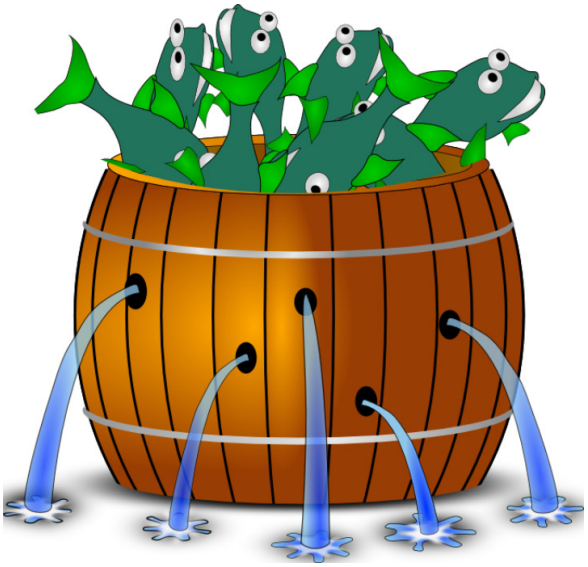
Barrelfish Project
ETH Zurich



Barrelfish on ARMv7-A

Barrelfish Technical Note 017

Simon Gerber Stefan Kaestle Timothy Roscoe
Pravin Shinde Gerd Zellweger

31.05.2016

Systems Group
Department of Computer Science
ETH Zurich
CAB F.79, Universitätstrasse 6, Zurich 8092, Switzerland
`http://www.barrelfish.org/`

Revision History

Revision	Date	Author(s)	Description
0.1	05.12.2013	SK	Initial version
0.2	08.12.2015	TR	Rewritten for new ARMv7 code
1.0	31.05.2016	TR	Newly-factored ARMv7 platform support

Contents

1 Introduction **5**

2 Compilation **6**
2.1 Building for GEM5 6

3 Hardware assumptions and limitations **8**
3.1 No support for Large Physical Address Extensions 8
3.2 Physical RAM starts at 2GB 8
3.3 Physical RAM is limited to 1GB 8

4 Organization of the address space **9**

5 Boot sequence **11**
5.1 BSP (initial) core 11

6 Exception code paths **12**
6.1 Reset exception 12
6.2 Undefined Instruction exception 12
6.3 Supervisor call (software interrupt) 13
6.4 Prefetch Abort exception 13
6.5 Data Abort exception 13
6.6 Hyp Trap, or Hyp mode entry 13
6.7 IRQ interrupt 13
6.8 Fast interrupt 13

7 The Dispatch mechanism **14**

8 Key data structures **15**

9 Hardware abstraction layers **16**
9.1 The ARMv7-A HAL 16

10 Code organization **17**

11 Versatile Express platform **18**

12 GEM5 specifics **19**
12.1 Boot process: first (bootstrap) core 19
12.2 Boot process: subsequent cores 20

13 OMAP44xx platform **22**
13.1 Compiling and booting 22
13.2 Booting the second OMAP A9 core 22
13.3 Physical address space 23
13.4 Interconnect driver 23

13.5 M3 cores 23

Chapter 1

Introduction

This document describes the state of support for ARMv7-A processors in Barrelfish.

ARM hardware is highly diverse, and has evolved over time. As a research OS, Barrelfish focusses ARM support on a small number of platforms based on wide availability, ease of maintenance, and research interest. However, since management of hardware complexity and diversity is also a research goal of the Barrelfish project, we aim to make it easy to add new ARM-based platforms with a mixture of traditional and non-traditional engineering techniques.

The principal processors with 32-bit ARM support in Barrelfish at present are ARMv7-A (Cortex A-series), in particular the Cortex A9.

Past support for older ARM 32-bit architectures in Barrelfish included:

- ARMv7m (Cortex M-series), in particular the Cortex M3.
- ARMv5 processors, in particular the Intel iXP2800 network processor (which uses an XScale core).
- ARMv6 (ARM11MP) processors running under simulation in `qemu`.

The main 32-bit ARM-based systems we target at present are:

- The Texas Instruments OMAP4460 SoC used in the Pandaboard ES platform.
- The ARM VExpress_EMM board, under emulation in the GEM5 simulator.

Chapter 2

Compilation

Building Barrelfish with ARMv7 is straightforward; detailed requirements for packages are described in the latest README file.

Compiling ARM support in Barrelfish requires a cross-compilation toolchain on the programmers `PATH`. For ARMv7 support we track the GNU toolchain shipped with Ubuntu LTS (14.04.3 at time of writing).

Once you have the right tools, run hake with the correct options, e.g.:

```
$ cd /build/barrelfish
$ /git/barrelfish/hake/hake.sh -a armv7 -s /git/barrelfish
...
$
```

After running `hake` with appropriate architecture support (i.e. use `-a armv7`), you can ask the Makefile what platforms it supports:

```
$ make help-platforms
------------------------------------------------------------------
Platforms supported by this Makefile.  Use 'make␣<platform␣name>':
 (these are the platforms available with your architecture choices)

 Documentation:
         Documentation for Barrelfish
 PandaboardES:
         Standard Pandaboard ES build image and modules
 ARMv7-GEM5:
         GEM5 emulator for ARM Cortex-A series multicore processors
------------------------------------------------------------------
$
```

Then build:

```
$ make -j 8 PandaboardES
```

2.1 Building for GEM5

To boot Barrelfish in GEM5, in addition to the previous steps you will need a supported version of GEM5. The GEM5 website (`gem5.org`) has comprehensive information.

Unfortunately, different versions of GEM5 manifest different subtle bugs when emulating ARM systems. We recommend revision 0fea324c832c of GEM5 at present; please let us know if you find a more recent version that works well.

To fetch and build GEM5 on Ubuntu LTS:

```
$ sudo apt-get install scons swig python-dev libgoogle-perftools-dev m4 protobuf-compiler libprotobuf-dev
$ hg clone http://repo.gem5.org/gem5 -r 0fea324c832c gem5
adding changesets
adding manifests
adding file changes
added 9356 changesets with 53499 changes to 6576 files
updating to branch default
3269 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ cd ./gem5
$ scons build/ARM/gem5.fast
...

$
```

GEM5 is a large system and may take some time to build. In addition, you may have to install minor fixes to ensure compilation (I had to add some initializers to `mem/ruby/network/orion/Wire.cc`, for example).

After the compilation of GEM5 is finished, add the binary to your PATH.

Now, build Barrelfish like this:

```
$ make -j 8 ARMv7-GEM5
```

It's a good idea to set `armv7_platform` in `<build_dir>/hake/Config.hs` to `gem5` in order to enable the cache quirk workarounds for GEM5 and proper offsets for the platform simulated by GEM5.

You can also build Barrelfish and boot inside GEM5 in a single step:

```
$ make help-boot
----------------------------------------------------------------------
Boot instructions supported by this Makefile.  Use 'make␣<boot␣name>':
 (these are the targets available with your architecture choices)

 gem5_armv7:
         Boot an ARMv7a multicore image in GEM5
 gem5_armv7_detailed:
         Boot an ARMv7a multicore image in GEM5 using a detailed CPU model
$ make gem5_armv7
...
```

To get the output of Barrelfish you should:

```
$ telnet localhost 3456
```

GEM5 is a highly configurable simulator. You can print the supported options of the GEM5 script as follows:

```
$ gem5.fast gem5/gem5script.py -h
```

Note that if you boot using `make arm_gem5_detailed` rather than `make arm_gem5`, the simulation takes a long time (depending on your machine up to an hour just to boot Barrelfish).

Chapter 3

Hardware assumptions and limitations

The current state of ARMv7 support in Barrelfish makes a number of assumptions about the underlying hardware platform, and also imposes some limitations. We discuss these here.

3.1 No support for Large Physical Address Extensions

The current Barrelfish design does not support LPAE for 32-bit ARM processors. Instead, it assumes a 32-bit physical address space. Supporting LPAE would require changes to the paging code, but would also require a mechanism to address user memory from the kernel effectively (see below).

3.2 Physical RAM starts at 2GB

Within the 32-bit physical address space, RAM is assumed to start at the 2GB boundary (i.e. `0x80000000`). This is the architectural recommendation for Cortex-A series processors, and we have yet to encounter non-LPAE ARMv7-A hardware which does not do this. Changing this assumption in the code should be possible, but in practice is likely to be dominated by the other limitations mentioned here.

3.3 Physical RAM is limited to 1GB

The Barrelfish ARMv7 CPU drivers can handle up to 1GB RAM, contiguously situated in the physical address space starting at 2GB. This limit could be raised by half a Gigabyte or so, at the cost of space for mapping kernel devices. In practice, the CPU does not need to map many kernel devices since most drivers run in user space on Barrelfish. Consequently, the allocation of the top 2GB of the virtual address space betwen 1-1 mapped RAM and kernel hardware devices could easily be moved.

However, it remains that the total RAM visible to the CPU *plus* the mappings for any devices needed by the CPU driver must fit into the top 2GB of the address space (mapped by the TTBR1 register).

In particular, the CPU driver assumes that all physical RAM is mapped 1-1, and relies on this when performing capability invocations. If the system had more RAM that could be mapped 1-1 into kernel virtual address space, we would need a method for the CPU driver to quickly access arbitrary physical addresses, entailing some kind of paging system.

Chapter 4

Organization of the address space

Like many other popular operating systems, Barrelfish employs a memory split. The idea behind a memory split is to separate kernel code from user space code in the virtual address space. This allows the kernel to be mapped in every virtual address space of each user space program, which is necessary to allow user space code to access kernel features through the system call interface. If the kernel was not mapped into the virtual address space of each program, it would be impossible to jump to kernel code without switching the virtual address space.

Figure 4.1: Barrelfish virtual address space layout for ARMv7-A

Additionally ARMv7-A provides two translation table base registers, TTBR0 and TTBR1. We configure the system to use TTBR0 for address translations of virtual addresses below 2GB and TTBR1 for virtual address above 2GB. This saves us the explicit mapping of the kernel pages into every L1 page table of each process. Even though the kernel is mapped to each virtual address space, it is invisible for the user space program. Accessing memory, which belongs to the kernel, leads to a pagefault. Since many mappings can point to the same physical memory, memory usage is not increased by this technique.

Figure 4.1 shows the memory layout of the complete virtual address space of a single ARMv7-A core running Barrelfish.

We have a memory split at 2GB, where everything upwards is only accessible in privileged modes and

the lower 2GB of memory is accessible for user space programs.

The kernel runs out of the kernel virtual address space where system RAM is mapped 1-1; in the region between `0x80000000` and `0xC0000000` RAM is mapped directly physical-to-virtual.

The L1 page table of the kernel address space is located inside the data segment of the kernel right after the kernel and naturally aligned to 16KB.

We map the whole available physical memory into the kernels virtual address space using "sections" (1MB large pages), obviating the need for a kernel L2 page table.

Above `0xC0000000`, the CPU driver maps regions of physical memory corresponding to hardware devices it needs to directly access (typically the UARTs, interrupt controller, timers, Snoop Control Unit, and a few others). These are also mapped using sections. Virtual address regions are allocated in 1MB increments (the size of a section mapping) working down from the top section, which is used to map the area of RAM containing the CPU driver's exception vectors.

Below the `0x80000000`, all mappings are handled by TTBR0 and changed on every context switch. At startup, the kernel uses another page table (also 16kB-aligned and located inside its data segment) to map low memory virtual-to-physical as well, as a way to access hardware devices in this region before the rest of the system has come up. However, after the early stages of bootstrap this table is no longer used.

Instead, TTBR0 is always loaded with the address of a user domain's hardware page table and changes on a context switch. TTBR1 does not change, ensuring the kernel mappings are static after boot.

Chapter 5

Boot sequence

5.1 BSP (initial) core

1. `boot.S:start` is called by the bootloader. It sets the processor `System` mode, sets up the (single) kernel stack, the global object table pointer, and jumps to `arch_init`.
2. `init.c:arch_init` is called with a single argument: the address of the multiboot info block. It first initializes the serial console `serial_early_init` and checks to see if this is the BSP. If so, it calls `bsp_init`.
3. `init.c:bsp_init` reads information from the multiboot info into the global data structure, initializing it. It also resets global spinlocks, and sizes RAM (though this information is not yet used). It returns.
4. `init.c:arch_init` continues by initialzing paging, calling:
5. `paging.c:paging_init` populates the two initial page tables (one for each base register). The kernel (upper) page table is initialized to map 1GB of RAM at 0x80000000, and the exception vectors at the top of memory. The initial user (lower) page table is set to map the lower 2GB of the physical address space 1-1 to enable early device access. The MMU is then enabled.
6. `init.c:arch_init` continues with the MMU enabled by jumping at:
7. `init.c:arch_init_2` which initializes exceptions, relocating the current KCB, parses the command line arguments, and re-initializes the serial ports so that the UART hardware is now mapped correctly into kernel address space with a section mapping.

   It then initializes the GIC, the Snoop Control Unit, the Global Timer, and the Time Slice Counter. Cycle counter access from `User` mode is enabled, and the coreboot spawn handler set up. It then calls:
8. `startup_arch.c:arm_kernel_startup` which initializes a simple memory allocator from the global structure, allocates the a new KCB, and calls:
9. `startup_arch.c:spawn_bsp_init` which creates the initial kernel data structures for spawning the init process. It also creates the initial capabilities for init to use to allocate memory, and returns.
10. `startup_arch.c:arm_kernel_startup` continues but calling `dispatch` on the init DCB, and we are now up and running.

Chapter 6

Exception code paths

ARMv7-A exceptions are initialized in `exceptions.S:exceptions_init`, which for some reason is written in assembly. It assumes the core is running in `System` mode.

There is a 256-byte statically-allocated stack for each exception mode, and an 8kB stack used for subsequently calling into C in `System` mode, all defined in `exceptions.S`.

Most exception handlers in the vector table start by checking whether the processor was in `User` mode or not when the trap happened. In most cases, if the processor was not in `User` mode, the result is that `System` mode is entered and the processor jumps to `exn.c:fatal_kernel_fault`, which panics. Exceptions to this rule are noted below.

For exceptions taken while the processor is in `User` mode, the address of the current (user space) dispatcher is loaded (macro
`get_dispatcher_shared_arm`), and a check is made to see if the dispatcher is "enabled" (in other words, whether the dispatcher should be upcalled when next dispatched).

This latter check is performed by the macro `disp_is_disabled`, and returns non-zero if:

1. The `disabled` value in the dispatcher (at offset `OFFSET_OF_DISP_DISABLED`) is non-zero, *or*
2. The PC lies between the two values in the dispatcher with offsets `OFFSETOF_DISP_CRIT_PC_LOW` and `OFFSETOF_DISP_CRIT_PC_HIGH`[1].

Depending on this, context is saved in a different area of the dispatcher, `System` mode is entered, and a call is made to C code as noted below.

Taking each exception in turn:

6.1 Reset exception

This is vector 0x00, and is not used.

6.2 Undefined Instruction exception

This is vector offset 0x04, and is referred to as `ARM_EVECTOR_UNDEF` in the source. The processor enters `undef_handler` in `Undefined` mode. Context is saved in either the `ENABLED` or `TRAP` area. C is entered at `exn.c:handle_user_undef`.

[1] A trick suggested by Justin Cappos to allow an atomic resume of a user-level thread without entering the kernel

6.3 Supervisor call (software interrupt)

This is vector offset 0x08, and referred to as `ARM_EVECTOR_SWI` in the source. The processor enters `swi_handler` in `Supervisor` mode.

If the syscall was issued from user space, context is saved in either the `ENABLED` or `DISABLED` area. C is entered at `syscall.c:sys_syscall`.

If the syscall was issued from kernel space, no context is saved and C is entered at `syscall.c:sys_syscall_kernel`.

6.4 Prefetch Abort exception

This is vector offset 0x0C, and referred to as `ARM_EVECTOR_PABT` in the source. The processor enters `pabt_handler` in `Abort` mode.

Context is saved in either the `ENABLED` or `TRAP` area. C is entered at `exn.c:handle_user_page_fault`.

6.5 Data Abort exception

This is vector offset 0x10, and referred to as `ARM_EVECTOR_DABT` in the source. The processor enters `dabt_handler` in `Abort` mode.

Context is saved in either the `ENABLED` or `TRAP` area. C is entered at `exn.c:handle_user_page_fault` with the faulting address in `r0`.

6.6 Hyp Trap, or Hyp mode entry

This is vector offset 0x14, and is not used in Barrelfish.

6.7 IRQ interrupt

This is vector offset 0x18, and referred to as `ARM_EVECTOR_IRQ` in the source. The processor enters `irq_handler` in `IRQ` mode.

If the syscall was issued from user space, context is saved in either the `ENABLED` or `DISABLED` area. C is entered at `exn.c:handle_irq`.

If the syscall was issued from kernel space, context is saved in `irq_save_area`, `System` mode is entered, and C is called at `exn.c:handle_irq`.

6.8 Fast interrupt

This is vector offset 0x1C, and referred to as `ARM_EVECTOR_FIQ` in the source. The processor enters `fiq_handler` in `FIQ` mode.

If the syscall was issued from user space, context is saved in either the `ENABLED` or `DISABLED` area. C is entered at `exn.c:handle_irq` (as for IRQ).

If the syscall was issued from kernel space, context is saved in `irq_save_area`, `System` mode is entered, and C is called at `exn.c:handle_irq` (as for IRQ).

Chapter 7

The Dispatch mechanism

Each time a CPU driver decides to switch to running a domain, it dispatches the domain in one of two ways:

RESUME , also known as "disabled": in this mode, the domain is resumed exactly where it was preempted before, much as in operating systems like Unix.

UPCALL , also known as "enabled": as with Scheduler Activations, the domain is upcalled at a fixed address with a new context on a small, dedicated stack. The context of the previously-running thread in teh domain is available to be resumed in user space, if the user-level scheduler (also known as the activation handler) decides to.

Which one of these happens depends on the state of the domain.

When a domain is running in user space (i.e. the kernel is *not* executing) the domain is in one of two states, indicated by a combination of:

- the `disabled` field of the `struct dispatcher_shared_generic` structure,
- the current program counter,
- the `crit_pc_low` and `crit_pc_high` fields of the `struct dispatcher_shared_generic` structure.

Note that all of these values can be written by the user program.

Specifically, the domain is in `RESUME` state *iff*:

1. `disabled` is `true`, *or*
2. the current program counter lies between `crit_pc_low` and `crit_pc_high`

Otherwise, it is in state `UPCALL`.

Once the kernel is entered, the `disabled` flag of the domain's `struct dcb` structure (as opposed to the `struct dispatcher_shared_generic`) is updated to reflect the state of the preempted domain.

Chapter 8

Key data structures

- `struct dcb`: in `kernel/include/dispatch.h`; the main domain control block.

  `dcb_current` is a global pointer in the CPU driver that points to the current DCB.

  If `dp` is of type struct dcb *, then `dp->disabled` is a flag which is 1 if the current DCB has activations disabled (i.e. it should be resumed when next scheduled to run) and 0 otherwise (in which case it should be upcalled) - the analogy is with enabling and disabling interrupts. The flag is set on entry to the kernel.

- `struct dispatcher_shared_generic`: in `include/barrelfish_kpi/dispatcher_shared.h`: the architecture-independent part of the a dispatcher, the user-space datastructure corresponding to a DCB. This is the first struct in architecture-dependent variants, such as `struct dispatcher_shared_arm`.

  If `dp` is of type struct dispatcher$_shared_generic*, then$`dp->disabled`$isaflagwhichis1ifthecurrentDCBhas$

Chapter 9

Hardware abstraction layers

Barrelfish distinguishes between:

- General code
- Architecture-specific code (e.g. ARMv7-A code)
- Platform-specific code (e.g. code for the OMAP4460 SoC)

Since most Barrelfish device drivers run in userspace, the difference between "platform" as a chip (such as the OMAP4460) and "platform" as a board or complete machine (such as the PandaBoard ES) are relatively unimportant inside the CPU driver, since most of the platform-specific CPU driver code is actually specific to a chip or SoC.

Barrelfish CPU driver source code for ARMv7-A systems therefore consists of the following categories:

- Portable, architecture-independent code.
- ARMv7-A-specific code which common to all ARMv7-A platforms
- Code for particular devices or macrocells which are only used on ARMv7-A, but might appear on multiple ARMv7-A platforms.
- Platform-specific code.

We restrict platform-specific code to a single source file, which roughly corresponds to ARM's concept of an "integrator", and acts as a compilation-time indirection layer between commmon ARMv7-A-specific code and individual device and macrocell drivers.

9.1 The ARMv7-A HAL

Platform code for a Barrelfish ARMv7-A CPU driver must implement the following interfaces:

serial.h : Low-level drivers for a multiple UART devices.

spinlock.h : Some number of static spinlocks, used for coordinating access to e.g. serial devices between CPU drivers on different cores.

Chapter 10

Code organization

The variety of ARM platforms make organizing source trees to maximise code reuse across different platforms a challenge.

Barrelfish distinguishes between *Architectures*, which are typically processor architectures like "ARMv7-A", and *Platforms*, which are complete system targets, like "PandaBoard-ES".

Code and headers specific to a particular architecture are found in the source tree is various subdirectories of the form `../arch/armv7/`.

Chapter 11

Versatile Express platform

Chapter 12

GEM5 specifics

The GEM5 [1] simulator combines the best aspects of the M5 [2] and GEMS [3] simulators. With its flexible and highly modular design, GEM5 allows the simulation of a wide range of systems. GEM5 supports a wide range of ISAs like x86, SPARC, Alpha and, in our case most importantly, ARM. In the following we will list some features of GEM5.

GEM5 supports four different CPU models: AtomicSimple, TimingSimple, In-Order and O3. The first two are simple one-cycle-per-instruction CPU models. The difference between the two lies in the way they handle memory accesses. The AtomicSimple model completes all memory accesses immediately, whereas the TimingSimple CPU models the timing of memory accesses. Due to their simplicity, the simulation speed is far above the other two models. The InOrder CPU models an in-order pipeline and focuses on timing and simulation accuracy. The pipeline can be configured to model different numbers of stages and hardware threads. The O3 CPU models a pipelined, out-of-order and possibly superscalar CPU model. It simulates dependencies between instructions, memory accesses, pipeline stages and functional units. With a load/store queue and reorder buffer its possible to simulate superscalar architectures as well as multiple hardware threads.

The GEM5 simulator provides a tight integration of Python into the simulator. Python is mainly used for system configuration. Every simulated building block of a system is implemented in C++ but are also reflected as a Python class and derive from a single superclass SimObject. This provides a very flexible way of system construction and allows to tailor nearly every aspect of the system to our needs. Python is also used to control the simulation, taking and restoring snapshots as well as all the command line processing.

We use a VExpress_EMM based system to run Barrelfish. The number of cores can be passed as an argument to the GEM5 script. Cores are clocked at 1 GHz and main memory is 64 MB starting at 2 GB.

12.1 Boot process: first (bootstrap) core

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

1. Setup kernel stack and ensure privileged mode
2. Allocate L1 page table for kernel
3. Create necessary mappings for address translation
4. Set translation table base register (TTBR) and domain permissions
5. Activate MMU, relocate program counter and stack pointer
6. Invalidate TLB, setup arguments for first C-function arch init

7. Setup exception handling
8. Map the available physical memory in the kernel L1 page table
9. Parse command line and set corresponding variables
10. Initialize devices
11. Create a physical memory map for the available memory
12. Check ramdisk for errors
13. Initialize and switch to inits address space
14. Load init image from ramdisk into memory
15. Load and create capabilities for modules defined by menu.lst
16. Start timer for scheduling
17. Schedule init and switch to user space
18. init brings up the monitor and mem serv
19. monitor spawns ramfsd, skb and all the other modules

12.2 Boot process: subsequent cores

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

1. Pass argument from bootloader to first C-function arch init 18
2. Make multiboot information passed by bootloader globally available
3. Create 1:1 mapping of address space and alias the same region at high memory
4. Configure and activate MMU
5. Relocate kernel image to high memory
6. Reset mapping, only map in the physical memory aliased at high memory
7. Parse command line and set corresponding variables
8. Initialize devices
9. Initialize and switch to inits address space
10. Load init image into memory
11. Create capabilities for modules defined by the multiboot info
12. Schedule init and switch to user space
13. init brings up the monitor and mem serv
14. monitor spawns ramfsd, skb and all the other modules
15. spawnd parses its cmd line and tells the monitor to bring up a new core
16. monitor setups inter-monitor communication channel
17. monitor allocates memory for new kernel and remote monitor
18. monitor loads kernel image and relocates it to destination address

19. monitor setups boot information for new kernel
20. spawnd issues syscall to start new core
21. Kernel writes entry address for new core into SYSFLAG registers
22. Kernel raises software interrupt to start new core
23. Kernel spins on pseudo-lock until other kernel releases it
24. repeat steps 15 to 23 for each application processor

Chapter 13

OMAP44xx platform

The OMAP4460 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.

The processor configuration in the OMAP4460 is somewhat unconventional; for example, the Cortex-M3 processors share a custom MMU with page faults handled by code running on the Cortex-A9 processors and hence are constrained to run in the same virtual address at all times. They are also not cache-coherent with the Cortex-A9 cores.

13.1 Compiling and booting

To compile Barrelfish for the Pandaboard, first configure your toolchain as described in Section 2. Then execute:

```
cd SRC
mkdir build
cd build
../hake/hake.sh -a armv7 -s ../
make pandaboard_image
```

The resulting image can be booted on the Pandaboard over the USB OTG connector using the standard `usbboot` utility. It will generate console output on the Pandaboard's serial connector.

13.2 Booting the second OMAP A9 core

Here is a brief overview of how the bootstrapping process for the second core works: it waits for a signal from the BSP core (an interrupt), and when this signal is received, the application core will read

an address from a well- defined register and start executing the code from this address.

To boot the second core, one can write the address of a function to the register and send the inter-processor interrupt. Following are some pointers to the documentation to help understand the boot-strapping process in more detail:

- Section 27.4.4 in the OMAP44xx manual talks about the boot process for application cores.
- Pages 1144 *ff.* in the OMAP44xx manual have the register layout for the registers that are used in the boot process of the second core.

Note that the Barrelfish codebase distinguishes between the BSP (bootstrap) processor and APP (application) processors. This distinction and naming originates from Intel x86 support where the BIOS will choose a distinguished BSP processor at start-up and the OS is responsible for starting the rest of the processors (the APP processors). Although it works somewhat differently on ARM, the naming convention is applicable here as well.

Note also that the second core will start working with the MMU disabled, so is running in physical address space. The bootstrapping code sets up a stack, initial page tables and an initial Barrelfish dispatcher.

13.3 Physical address space

At present, a temporary limitation in the core boot protocol means that running Barrelfish on both A9 cores requires static partitioning of the available RAM into two halves, with an independent memory server running on each core. This is will fixed in a subsequent release.

13.4 Interconnect driver

Communication between A9 cores on the OMAP processor is performed using a variant of the CC-UMP interconnect driver, modified for the 32-byte cache line size of the ARMv7 architecture. A notification driver for inter-processor interrupts exists.

The OMAP4460 also has mailbox hardware which can be used by both the A9 and M3 cores. Barrelfish support for this hardware is in progress.

13.5 M3 cores

Barrelfish also has rudimentary support for running on both the A9 and M3 cores. This is limited by the requirement that the M3 cores must run in the same virtual address space, and do not have a way to automatically change address space on a kernel trap. For this reason, we only execute on a single M3 core at present.

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

1. Power on the Cortex-M3 subsystem
2. Activate the Cortex-M3 subsystem clock
3. Load the image to be executed into memory
4. Enable the L2 MMU
5. Set up mappings for the loaded image in the L2 MMU (can be written directly into the TLB)
6. Write the first two entries of the vectortable (initial sp and reset vector)

7. 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.

References

[1] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.

[2] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The m5 simulator: Modeling networked systems. *Micro, IEEE*, 26(4):52–60, 2006.

[3] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *ACM SIGARCH Computer Architecture News*, 33(4):92–99, 2005.