Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Barrelfish on Netronome

by
Bram Scheidegger

Advisors
Simon Peter, Andrew Baumann
and Timothy Roscoe

due date

February 28, 2011

ETH Zurich, Systems Group
Department of Computer Science
8092 Zurich, Switzerland

Abstract

Commodity computer systems contain more and more specialized hardware tailored to perform certain operations efficiently in speed and power consumption. ARM became one major player in the market of embedded devices, and now even more commodity computers contain ARM based CPUs.

In current operating systems the power of such devices can only be used by accessing the components through the operating system kernel using a device driver. Barrelfish suggests a multikernel architecture that treats such a system containing heterogeneous hardware as a network. In a multikernel architecture, each component hosts its own kernel and together they form the operating system. This makes the system easier to understand and could even enhance performance.

In this thesis we describe the port of the Barrelfish kernel from an emulator to a powerful network card called the Netronome. This ARM based network card has an Intel XScale IXP2850 network processor and memory on board.

Contents

1 Introduction and motivation

In the early days of personal computers the central processing unit (CPU) was responsible for handling all computational work. Little by little, the devices surrounding the CPU became increasingly sophisticated and more powerful. Well-known representatives of this trend are disk controllers (e.g. SCSI, SATA) or the Graphics Processing Unit (GPU). Looking closer into the history of GPUs, we can point out two major steps in their evolution. At first, the highly specialized GPU supported the CPU by taking over the 3D rendering of objects on screen. Nowadays the graphics adapters have evolved one step further by offering their processing power to all applications. Today's mainstream trend goes towards even more specialized hardware. This has advantages like increasing the overall system performance by reducing their power consumption.

Building systems with low power consumption is currently a major ambition as the number of computer systems around the world grows fast. Because of this trend, ARM became increasingly popular, as their RISC (Reduced Instruction Set Computer) processors allow building power efficient systems. ARM based CPUs were mainly used to construct mobile and embedded devices, and they currently tend to move into the server market. There are different ARM implementations available, as ARM only sells the CPU design. Manufacturers like Intel can buy a license and start producing their own ARM based CPUs - so, the Intel XScale processor was born.

Computer networks became more and more important in the last ten years, so the Internet and intranet bandwidth progressed rapidly. With a growing bandwidth demand and an increasing number of clients, the server load also escalates. Therefore it makes sense to have specialized hardware for network processing which can advance to taking over other tasks like encryption. As a consequence, a server has more capacity to host its services. The Netronome network card is one step in the same direction as the GPU since this high performance network card is also fully programmable.

1.1 Introduction to Barrelfish

Barrelfish [1] is a multikernel OS developed by the systems group at ETH Zuerich in collaboration with Microsoft research. It is an approach to deal with the heterogeneity of modern computer systems in a new way. Apart from embedded devices even a modern processor like an Intel Core i7 processor with its QuickPath Interconnect [2] introduces heterogeneity into the system. Barrelfish is built upon several principles which we introduce in this section.

1.1.1 Multikernel

In a multikernel OS, each core has its own instance of a so called CPU driver. The CPU driver runs in privileged mode and its task is to mediate access to the associated hardware (for example the memory management unit). Device drivers and system services are not part of the CPU driver as they run in user

space. For instance, the interrupt handler of the CPU driver is very simple: it acknowledges the interrupt and delivers it to user space. This design increases the overall stability of the system since a crashed driver in user space cannot tear down the whole system.

The CPU driver is a very system dependent part of the Barrelfish OS intended to be highly optimized for the underlying architecture. On the other hand, the system services running in user mode can be built on top of the CPU driver and are therefore mostly system independent.

As we can see from the Barrelfish architecture, the core focus of this thesis lies in porting the CPU driver.

1.1.2 Message passing system

The data structures of different CPU drivers are purely local, there is no shared state between them. To keep the nodes synchronous, Barrelfish provides a message passing system. Traditional systems use a shared memory approach where different CPU cores are accessing the same memory region. Such an approach raises well known concurrency and cache coherency problems. Today's systems deal with this problem using locks and a cache coherence protocol. This is getting increasingly difficult, especially in a heterogeneous system with other components than processor cores attached to it.

Barrelfish distinguishes between two types of messages to increase the efficiency of the message passing system:

- Intra-core communication (messages on the same core): the CPU driver implements a lightweight inter-process communication system built over shared memory.
- Inter-core communication (messages between different cores): this type of communication takes place in user space and is performed by an OS process called the monitor, which is fully privileged.

1.1.3 Other OS components

Dispatcher Each process has its corresponding set of dispatcher objects (one on each core the process is intended to run on). The CPU driver schedules the different dispatcher objects and not the process itself. When a process has multiple threads, the dispatcher is responsible for thread scheduling. Inter-process communication evolves between dispatchers as well, it can use the underlying message passing facility for communication.

Capabilities Capabilities are used for right management. If a user space application requests device access, it has to request the capability for this access. The same holds for memory requests: the memory server mem_serv manages capabilities for memory. If a user space application requests memory, mem_serv checks if it can fulfill the request and then answers with a capability for the

requested piece of memory. From this moment on, the application can map the memory into its virtual address space and start using it.

System knowledge base The system knowledge base (SKB) provides information about the underlying hardware. The information for the SKB comes from many different sources like hardware discovery and static knowledge about the system. Example use cases of such a repository is efficient message routing or choosing a core near the device we want to access.

1.2 Netronome platform

Our hardware platform is a network card, shipped with a version of Monta Vista Linux as operating system (OS). It has an Intel XScale IXP2850 network processor and four 1000BaseT network interfaces on board. Such design allows performing complex network operations already on the network card. The preinstalled boot loader (RedBoot) is used to launch Barrelfish instead of Linux. The card itself is connected to its host PC over a PCIe connection.

In general, ARM processors support both Little and Big Endian. On the Netronome, the boot loader brings up the system in Big Endian mode and the preinstalled version of Linux runs in Big Endian mode as well.

Having a computer system contain a lot of different, optimized hardware components raises the question of how the interaction with such a system should be implemented. Currently, a software component has to communicate with such hardware by using a driver running on top of one OS kernel.

1.3 Aim

The aim of this thesis is to port the Barrelfish ARM version for QEMU to the Netronome platform, including the CPU driver and the user space applications shipped with Barrelfish. It is the first time Barrelfish runs on a hardware ARM based CPU and the first time it runs on a Big Endian platform.

Among other things, insight into the following questions is provided:

- How does an OS initially written for Little Endian need to be adapted to run in a Big Endian environment?
- Are different ARM implementations compatible to each other? What is the exact difference between an emulated ARM CPU (for example QEMU) and a hardware based implementation?
- How does Barrelfish behave on an embedded device?

Having Barrelfish run on a network card enables us to investigate further questions concerning the heterogeneity of a computer system, such as:

- How does the message passing system behave using the very fast PCIe bus?

- Can a system use the resources provided by the specialized networking hardware efficiently?

1.4 Document structure

Chapter two provides an overview over related work. We present the already available components and give a brief overview over the tools we used for our work. Chapter three explains some system components in detail, focusing on the ARM platform. One major topic is virtual memory, as this was one key component to get the kernel running. Chapter four describes the boot process and points out the major challenges we encountered. It builds upon the knowledge from the previous chapter. In chapter five, we discuss future work and in chapter six, we conclude.

2 Related work and building blocks

2.1 Porting on ARM

As already mentioned in chapter 1, the ARM platform becomes increasingly popular. As a consequence, there are already many ARM ports for different operating systems available, for example Linux (e.g. Debian [3]), FreeBSD [4] or the L4 microkernel [5].

NetBSD has been ported by Antti Kantee [6] to a single-board computer equipped with an XScale PXA255 and RedBoot as boot loader. He described the port process as easy. He encountered no problems setting up the cross compilation tool chain and furthermore pointed out the usefulness of a GDB connection for low level debugging.

Ming Chen [7] described the porting of the Jikes Research Virtual Machine (RVM), a research Java Virtual Machine, to XScale. His progress was relatively slow and he was unable to reach his goal to run a "Hello world" program in the end.

Porting to a Big Endian platform does not seem to be very common. The main reason for this is the support for both Little and Big Endian by ARM and the dominance of Little Endian systems.

2.2 QEMU ARM port

An ARM port of Barrelfish for QEMU was already available. In the current version, the CPU driver, the user space process "init" and the memory server are functional. However, the shell "fish" has not been ported to ARM yet.

2.2.1 Launch simulation

To compile Barrelfish for ARM, we set ARM as a target in our build system and as compiler we configure a Little Endian ARM compiler (freely available from

CodeSourcery [8]). After the compilation process is finished, we can launch the simulation by typing: "make sim ARCH=arm".

2.3 Simple serial console driver

There was already a very basic serial console driver available for the Netronome, which just prints out a string. The driver was useful to learn about the boot process on the Netronome and to study how to write a serial console driver for Barrelfish. However, we reimplemented the console driver with the help of a Mackerel specification file (subsection 2.4) for the usage in Barrelfish.

2.4 Mackerel shift driver

Mackerel is an interface definition language for device drivers. It allows specifying registers and offsets for device access and abstracts away the normally required bit manipulation. Mackerel hooks into the compilation system and generates a C header file out of the device specification. In order to access the fields the programmer has to include this header file and can now use the Mackerel naming schema to set the bits appropriately.

Behind the scenes, Mackerel 1 uses bit fields to generate a device access of word size (in our case 32 bit). Since bit fields are Endian dependent, it turned out to be a bad choice for our platform. However, there is a trick to use Mackerel 1 also for a Big Endian system: in the device specification file, one has to declare the order in which a register is specified. Normally, we take the bit order which is used in the documentation. In our case, we could just pass the "wrong" bit order to compensate for using Big Endian instead of Little Endian.

In Mackerel 2 this problem is corrected. It uses shifts instead of bit fields, which makes the access Endian independent. We therefore use Mackerel 2 in our port.

3 Porting to the IXP2800 hardware platform

In this chapter, we first describe how we managed to build a Big Endian compiler as this posed one major problem. Then, we have a look at different aspects of the underlying hardware platform. Whenever we were stuck, we had to understand the hardware platform in depth and therefore this knowledge is required to comprehend the crucial points in the port process.

3.1 Big Endian compilation system

Getting a Big Endian tool chain for ARM was much more difficult than initially expected. The compiler used for the simple serial console driver was GCC 3.4.5, which is too old to build Barrelfish.

We tried to build our own tool chain consisting of a Big Endian GCC and libgcc, whereas the most difficult part was building the libgcc for Big Endian. We tried a lot of different approaches, for example:

- A build script. There are several scripts available, we focused on crosstools [9] as the most promising solution.
- A manual compilation from the sources of GCC and libgcc.
- Combine a newer Big Endian compiler and the libgcc from the GCC 3.4.5 tool chain.

These approaches failed because we were unable to build libgcc. A free Big Endian compiler is offered from CodeSourcery [8] but without the libgcc for Big Endian. However, they offered a script to build the compiler. Using a try and error approach, we managed to build a libgcc before the script crashed. We then took our new libgcc and the free compiler from CodeSourcery to get our tool chain.

As an alternative, we considered introducing another stage after the boot loader to switch from Big to Little Endian mode before booting Barrelfish. However, after we managed to build the Big Endian compiler and because the preinstalled version of Linux also runs in Big Endian, we decided against switching to Little Endian. At that time, we did not suspect major drawbacks from this decision.

3.2 Virtual memory

The memory management of an operating system can be designed in several ways. One way is to physically address memory. Barrelfish uses physical addressing only during the early boot process to prepare for virtual memory mode.

Modern operating systems like Windows, Linux or BSD make use of virtual memory [10] [11] (pages 123 - 126). This allows user space programs to run in virtual memory as well. There are several advantages of using a virtual memory system [12] (pages 701 - 704): This simplifies linking and memory allocation for both the kernel and for user space.

In this chapter, we begin by explaining the motivation of using a memory split. Afterwards, the memory layout used in Barrelfish is explained in depth. We further have a close look at the implementation of virtual memory on ARM based processors. Finally, we learn how to perform the setup of a page table and how to activate the memory management unit (MMU). We focus on the implementation in Barrelfish.

3.2.1 Memory split

In virtual memory every process gets its own address space. The process itself runs in user mode, whereas the kernel runs in privileged mode. User space programs are not allowed to run privileged commands, like accessing a device or halting the machine. These restrictions are imposed by the operating system and enforced by the CPU [12] (pages 596 - 597). If a user space program needs to execute a privileged instruction (for example accessing a device), it has to make a system call.

The key point of a system call is to end up in privileged mode executing the kernel. This is usually implemented by traps, a software interrupt initiated by a user space program. When a software interrupt occurs, the CPU sets its mode bit to privileged mode and the execution jumps into the kernel using the interrupt vector associated with this interrupt. This makes it necessary to have the kernel mapped into the virtual memory space of each program. If the kernel did not reside in the same virtual address space as the user space application, it is impossible to jump to the kernel without switching to another virtual address space.

The motivation just explained reasons for the widely used memory split where the kernel resides in a dedicated region of each virtual memory space. Barrelfish and other operating systems like Linux or BSD make use of this technique.

The kernel is invisible from the viewpoint of a user space application and user space programs are prohibited from accessing the kernel data structures. Trying to access this region will lead to a protection fault. The memory access from the program will not succeed and the protection fault interrupt will hand over the control to the kernel. This mechanism protects the kernel and forces applications to use the system call interface.

Physical memory usage is not affected by mapping the kernel into each virtual memory space, because we can map the same physical memory region multiple times. This saves space and makes data sharing easier. If each process had a physical copy of the kernel, a change in the kernel's data structure would require an update of each copy.

3.2.2 Barrelfish memory layout

Figure 1 provides an overview of the memory mappings used in the ARM port of Barrelfish. We have a memory split at 2 GB. From 2 GB upwards, the memory is only accessible in privileged mode. The remaining lower 2 GB are available for user space programs.

There are three different mappings (marked with numbers) shown in Figure 1 which we will now discuss in detail. Please note that for the sake of readability, the proportions of the different sections are not correct.

1. Mapping the kernel and the kernel stack to virtual memory. The physical memory block referred to as "Expanded elf file" is the actual kernel code (see 4.2.3). The usual size of the kernel stack is around 64 KB. This is sufficient, since Barrelfish is a micro-kernel.

2. Mapping the page table. The initial setup takes place just before turning on the MMU. This page table will be used and enhanced while the kernel is booting. Once we go on to user space, this page table serves as a master copy for each new virtual address space. It contains the necessary mappings for the kernel but no mappings from user space programs.

3. To access memory mapped devices, we need to map them into virtual memory. Barrelfish maps them as sections (see 3.2.3).

The box "All memory" maps all available physical memory continuously into the kernel's virtual address space.

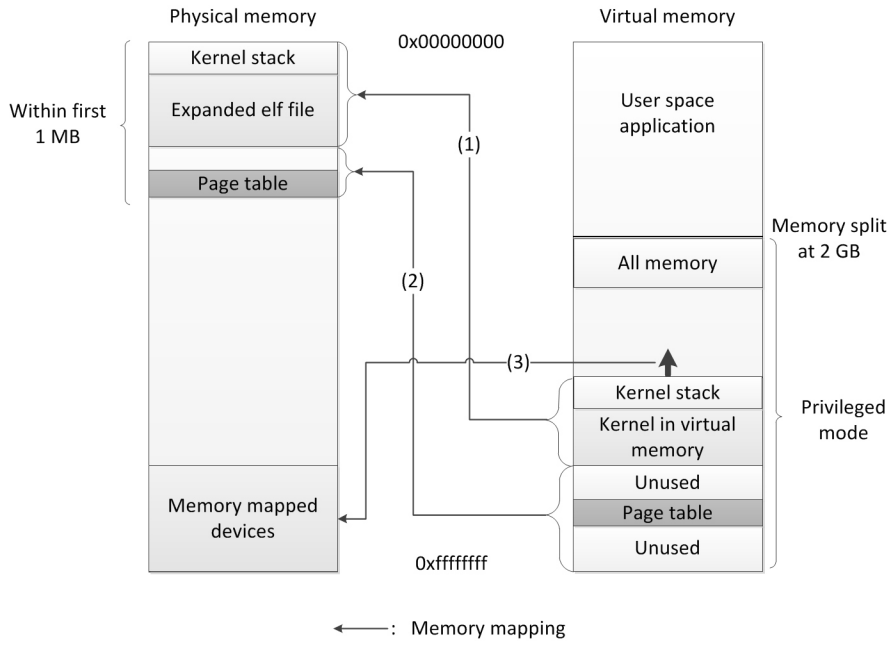

Figure 1: Barrelfish memory layout

3.2.3 ARM memory management unit

As on most x86 systems [13], [12] (pages 715 - 720), the ARM virtual memory system has a MMU with associated Translation Lookaside Buffer (TLB). The MMU translates virtual into physical addresses and since this is implemented in hardware, the translation is fast and there is no need for any software routine. The TLB caches a few frequently used entries to avoid a slow look-up in main memory.

Virtual memory is divided into pages. The mapping between a page in virtual memory and its associated page frame in physical memory is stored in a page table. On ARM, there are different types of pages and page tables [14] [15] (pages 491 - 546).

Level 1 page table The level 1 (L1) page table always translates sections of 1 MB size. This divides the 4 GB virtual memory space into 4096 pages and each page table entry has a size of 4 Byte. A complete L1 page table for one virtual address space therefore consumes 16 KB of memory.

The Barrelfish CPU driver uses the L1 page table to translate sections between virtual memory and physical memory. L1 page table entries can also serve

as pointers to a level 2 (L2) page table, allowing a more fine grained control over physical memory. It can be especially useful on embedded systems with limited physical memory available.

Level 2 page table On ARM we have "coarse" and "fine" L2 page tables. A "coarse" L2 page table divides a section in virtual memory in either small (4 KB) or large (64 KB) pages, holds 256 entries and consumes 1 KB of memory. The "fine" L2 page table supports tiny (1 KB), small and large pages, holds 1024 entries and consumes 4 KB of memory.

For both small and large pages, the access permissions (see subsection 3.2.4) can be configured more fine grained on the granularity of a quarter of a page. In Barrelfish, we use a page size of 4 KB.

3.2.4 Access permissions

The associated access permissions, which can be configured per page, control the access to a page depending on the system mode. For example, this helps us to protect the kernel from user space programs. As mentioned in chapter 3.2.1, a user space program generates a protection fault when accessing a kernel page. In this case, the page has been configured to allow full access from privileged mode and no access from user space.

3.2.5 Context switch

Figure 2 shows the three steps involved in a context switch.

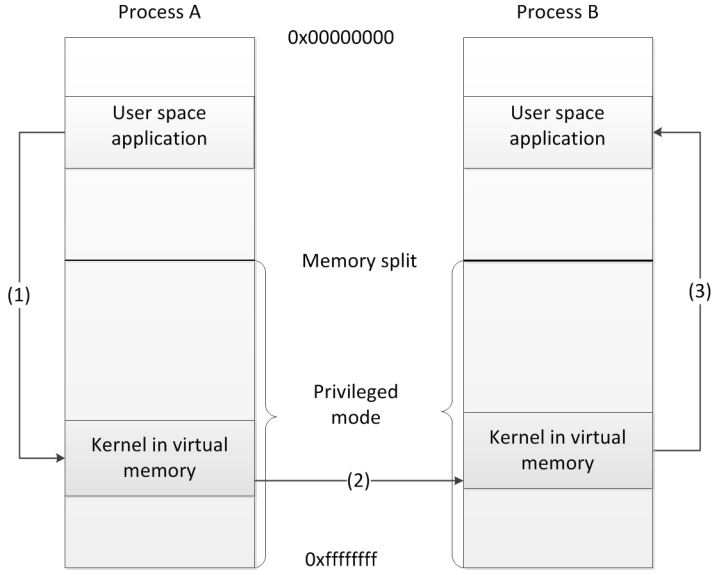

Figure 2: Switching between two user space applications

1. Process A makes a system call or gets preempted.

2. The kernel saves the process control block (PCB) of process A and restores the PCB of process B. In Barrelfish we use a dispatcher object instead of a PCB.

3. The kernel hands over control to process B

Our focus is on the role of virtual memory and the MMU during a context switch.

In step two, we have to switch from the virtual memory space of process A to the one of process B. Therefore we have to change the page table mapping to point to the page table of process B. To get rid of cached values from the old mapping, the TLB gets flushed. Furthermore, we have to flush the instruction and data caches to invalidate old entries (see section 3.3).

After changing the memory mapping, the kernel executes the next instruction as if nothing happened. At the first glance, this is somewhat surprising because the program counter (PC) contains the address of the next instruction but in the context of the memory mapping of process A. It is not required to adopt the PC because the kernel is mapped to the same location of each new page table. For this purpose, Barrelfish provides a function "paging_make_good".

3.2.6 Allocating pages

A page table is nothing more than an array. In order to insert a mapping in the page table, we need to know three things: physical address, virtual address and the start location of the page table.

First, we set up the page table entry. The page table entry contains the page configuration and the aligned physical address. This is the address of the beginning of the page frame we want to map. As Barrelfish uses a single level L1 page table, the page frame is aligned to one MB and one MB in size. The physical address to be mapped is located somewhere within this page frame. We can obtain the aligned physical address by a logical right shift of 20 (listing 1, line 2). There is no need to store the complete physical address into the page table entry because of the alignment.

The second step is to insert the page table entry into the page table (listing 1, line 3). We use the corresponding aligned virtual address as a position in the page table array. We can immediately see that the look up time for a translation from virtual to physical memory is O(1).

Listing 1: Insert page table entry into page table

```
1  // Some more configuration of l1
2  l1.section.base_address = pa >> 20u; // Insert physical address
3  l1_table[va >> 20u] = l1; // Write page table entry into page table
```

3.2.7 MMU setup

Before turning on the MMU, we need to setup our first page table. We have to consider two things:

1. Make sure that the code is mapped into our virtual address space.
2. After turning on the MMU, the PC should point to the next valid instruction. There are two ways to achieve this:
   (a) We perform a one-to-one mapping. In this case, there is no need to change the PC.
   (b) We can calculate the new value for the PC and store it in a register. After turning on the MMU, we can just update the PC.

In Barrelfish, we use approach (b) because we have an ELF file which has already been linked to an address in virtual memory. If we map our code to this location in virtual memory, we can now use (e.g. in assembly) a label to calculate the new address.

Finally, we can turn on the MMU as shown in listing 2. On ARM, we have coprocessors in the range p0 to p15. The instruction "move to ARM register from coprocessor" (mrc) is used to read out the current configuration, whereas "move from ARM register to coprocessor" (mcr) is used to write to the coprocessor. The arguments for mcr are:

1. The coprocessor to access.
2. Opcode 1: First coprocessor instruction (ignored by CPU).
3. Source register: The value to transfer into the ARM coprocessor
4. First coprocessor register (ignored by CPU)
5. Second coprocessor register (ignored by CPU)
6. Opcode 2: Second coprocessor instruction (ignored by CPU), optional to the mcr instruction.

Listing 2: Activate the MMU

```
1  ldr     r1, =0x1007 // Enable: D-Cache, I-Cache, Alignment, MMU
2  mrc     p15, 0, r0, c1, c0, 0 // Read out current setting
3  orr     r0, r0, r1
4  mcr     p15, 0, r0, c1, c0, 0 // Write back new configuration
```

The CP15:c1c0 register is utilized to activate the MMU. The various steps are shown in listing 2:

1. Load r1 with the bit pattern to enable caching, alignment checking and the MMU.

2. Load the current configuration from the control register cp15:c1 into r1.

3. Set the bits to activate caching, alignment checking and the MMU and store the result in r0.

4. Write r0 back to cp15:c1.

More information and configuration options are available in the ARM or XScale architecture reference manuals as well as in the ARM System Developer's Guide [14, 16] [15] (pages 513 - 515).

3.3 Caching

3.3.1 IXP2800 platform

On the IXP2800 platform, we have a separate data cache (d-cache) and instruction cache (i-cache). This is called a Harvard cache architecture. Apart from that, the XScale core has also a mini d-cache. According to Intel, this should avoid "thrashing of the d-cache for frequently changing data streams" [17, 18]. The i-cache and d-cache can hold 32 KB whereas the mini d-cache holds only 2 KB [16].

The write behavior ("write-through" and "write-back") of the cache can be configured per page. In case of a write-through policy, a write to data stored in d-cache will immediately be forwarded to main memory. If we have a write-back policy, the written data will be marked as dirty in the cache. The dirty cache line will be written to main memory as soon as it gets evicted from cache.

3.3.2 Clean and flush

On ARM, there is a difference between cleaning and flushing the cache.

If we flush a cache, all cache entries will be invalidated. This can be useful in case the data in main memory has changed. However, if we use a write-back policy and some data has been marked as dirty but has not yet been written back to main memory, these updates will be lost.

Using a write-back caching policy, we therefore need to clean the cache before flushing it. When cleaning the cache, we force it to write back dirty cache lines. Now, we can safely flush it.

Sloss at al. [15] (pages 423 - 443) describe three methods to clean the cache. Only one of them is applicable to the Intel XScale core. Cleaning and flushing the d-cache and flushing the i-cache involves four steps [16]:

1. Clean the d-cache

2. Clean the mini d-cache

3. Invalidate the d-cache and mini d-cache

4. Invalidate the i-cache

To clean the d-cache, we iterate through all cache lines and use a special line-allocation command on a dedicated, cached line of memory. This cached line of memory needs to exist in virtual memory but the line-allocation command will never overwrite the corresponding address in physical memory. While cycling through all cache lines, dirty data will be written back to main memory.

Cleaning the mini d-cache is similar to cleaning the d-cache. For the mini d-cache, there is no line-allocation command available. We therefore write 2 KB of unused data to all cache lines of the mini d-cache.

The invalidation process is much simpler as we can instruct the coprocessor to invalidate all cache lines. We need one command for the d-cache and mini d-cache and one more for the i-cache.

3.4 Processing modes

On ARM, the Current Program Status Register (CPSR) [15] (pages 22 - 29) stores information like condition flags, interrupt masks and the processor mode. There are several processor modes available, like interrupt request, system, supervisor or user mode. For the moment we ignore the system mode and come back to it later, as it is only a privileged version of the user mode. Please note that the discussion about shadow registers does not hold for system mode.

3.4.1 Shadow registers

In case of a change from user mode to any other mode, some registers are automatically saved (also referred to as banked). In the event of a fast interrupt request (FIQ), r8 to r12 are banked, otherwise only r13 (stack pointer, abbreviated by SP) and r14 (link register, abbreviated by LR). For instance when switching from user mode to supervisor mode, register r13 is automatically replaced by r13_svc by the processor.

In Barrelfish, each mode has its own SP. It raises no problem on ARM, since the SP is a banked register in all modes. To initialize the SP, we switch to each mode, set the stack pointer and switch back.

The CPSR register is special. When coming from user mode, the users CPSR will be stored in SPSR (Saved Program Status Register) and replaced with a version for the new mode. In Barrelfish we use the SPSR register to determine whether the interrupt came from user space or from the kernel.

3.4.2 Change processing mode

There are two ways to change the processing mode. In case of an interrupt, the mode will be changed by the processor. In privileged mode, we can write to CPSR to change our mode.

When entering the interrupt we need to save all registers which will be used by the interrupt handler and that are not banked.

3.4.3 Return from interrupt

To return from an interrupt, we restore the saved registers. To restore the CPSR from SPSR , there is a special instruction denoted by "^" . Listing 3 provides an example on how to restore r0 to r15 and the CPSR. In that example, "regs" stands for the location of the struct holding the register values to restore.

Listing 3: Restore registers and CPSR

```
1  ldmia  %[regs], {r0-r15}^
```

3.4.4 System mode

In system mode, there are no banked registers. Even this mode is fully privileged and can set its own CPSR to any value, we cannot restore CPSR from SPSR in case an interrupt sets system mode. The "^" instruction has no effect.

4 Barrelfish implementation

4.1 Boot loader

4.1.1 Launching Barrelfish

The Netronome card runs by default Monta Vista Linux. We cancel the start up of Linux and get the boot loader shell.

Listing 4: Boot loader launch instructions

```
1  ip -b
2  load -r -b 0x20000000 brams/romfs.cpio
3  load -r -b 0x10000000 brams/cpu.bin
4  exec -r 0x800000 -s 0xd061ff 0x10000000
```

Listing 1 shows how we can use RedBoots Trivial File Transfer Protocol (TFTP) support to run the Barrelfish kernel. The first line is necessary to fetch an IP address from DHCP. In line 2 and 3, we load romfs.cpio and cpu.bin from the TFTP server into the memory address specified by "-b". These files are loaded as raw images using the parameter "-r". In the last line the kernel gets executed as Linux image. The first parameter specifies the ramdisk location (our case is different, see subsection 4.1.3), whereas the second parameter is the size of the ramdisk. The memory address at the end is the entry point of our executable code (in our case cpu.bin).

4.1.2 RedBoot features

RedBoot is a rich boot loader with many features [19]. This includes:

GDB connection Running GDB on our local machine, we can use "target remote |ssh emmentaler console −f nos2−nfe" to connect to the boot loader. Note that in our case we have to redirect the connection through an SSH server because the Netronome card (nos2-nfe) has no public IP address.
It seems that the part of the boot loader which allows a GDB connection resides somewhere lower than approximately 0x40000 in memory. As soon as the kernel starts writing on these addresses, we cannot establish a GDB connection anymore.

Endian switch The exec command offers a parameter to switch to a different endian mode. Unfortunately, this option is not supported on the Netronome, although our CPU can run in both Little Endian and Big Endian mode.

Hex dump RedBoot has some basic hex dump functionality. This turned out to be useful for verifying the memory content just before starting the kernel.

Auto start We can even configure RedBoot to automatically launch Barrelfish. This can be done directly using the boot loader prompt.

In a first step, we set up an alias, giving our boot script a name. In listing 5 line 1, we can see the launch script from subsection 4.1.1 as alias. Please note that the length of an alias is bounded. However, it is possible to link multiple boot scripts together.

Now we can specify the default boot script, timeout, GDB connection port and much more by calling fconfig (listing 5, line 2). One has to work through several questions and in the end we can either write the changes into flash disk or discard them.

Listing 5: Configure boot loader for auto start

```
1  alias barrelfish 'ip -b; load -r -b 0x20000000 brams/romfs.cpio;
       load -r -b 0x10000000 brams/cpu.bin; exec -r 0x800000 -s 0
       xd061ff 0x10000000'
2  fconfig
```

4.1.3 Ramdisk

All user space programs (for example the shell “fish”) are currently stored in a ramdisk called “romfs.cpio”. The kernel is separate file called “cpu.bin”. As shown in subsection 4.1.1, we can load the kernel and ramdisk into main memory using RedBoot.

Normally, the boot loader would pass the location of the ramdisk to the kernel. On our Netronome with RedBoot version 1.31.1 at hand, we experienced the problem that the ramdisk gets altered when passing the correct location to the execute command. As a workaround, we have hard-coded the location of the ramdisk and gave RedBoot the wrong location. Not passing a ramdisk to the exec command is not an option in our case, since this leads to missing

ATAG headers. We decided against changing the code due to the boot loader misbehaving.

The second problem is the size of the ramdisk. As we have seen, it has to be passed as a parameter. It poses no problem for a static system, but for a continually changing research operating system, this gets cumbersome. As a workaround, we wrote a script which hooks into the compilation system. It first compiles the system, then writes the new ramdisk size into a header file and finally compiles the kernel again to include the changes to the updated header file.

4.2 Kernel bootstrap

This section describes the initial boot process until we can jump into init.c. The code for booting is written in assembly and is located in boot.s. The initial boot process relocates the kernel, sets up the page table and activates caches and MMU.

4.2.1 File formats

The boot loader expects a Linux image. This is a binary file which contains no header information, just machine instructions. The boot loader loads this file into memory and starts executing it.

When compiling the Barrelfish kernel, we get an ELF image. As an ELF image contains header information (e.g. ELF header and program header), we cannot pass this directly to the boot loader.

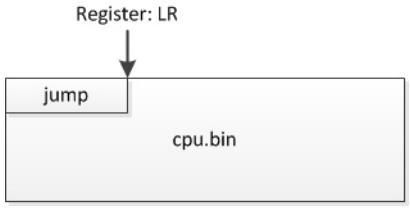

Figure 3: Boot image layout

To overcome this issue, the first word of the ELF image is overwritten with a jump instruction (Figure 3). This jump instruction points to the text section of the ELF image which contains executable code. As the jump instruction sets the link register (LR) to the instruction after the jump, we still know where cpu.bin, the modified version of the generated ELF file, starts. Later on, we need this to access the section header table which contains information about the location and size of the different sections (e.g. .text or .data) [12] (pages 544 - 548). This information is necessary to setup the page table later on.

A shell script inserts the jump instruction after the compilation system has created the ELF file. This script was initially written for Little Endian ARM systems. As we are in a Big Endian environment and the encoding of the jump

instruction is endian dependent, that script had to be adapted. Listing 6 shows how we can use OBJDUMP (line 1) to obtain information about the endianess of the kernel. Depending on the outcome, the byte order gets reversed. As a result we obtain an endian independent version of the script.

Listing 6: tools/arm-mkbootelf.sh

```
1  ${OBJDUMP} -f ${KERNEL} | grep -q bigarm
2  if [ $? -eq 0 ]; then
3          # Big Endian - reverse byte order
4      BL=${BL:6:2}${BL:4:2}${BL:2:2}${BL:0:2}
5  fi
```

4.2.2 Base setup

After the jump, the program executes the program code located in the middle of the ELF file. First of all, we have to set up the stack and learn some basic parameters.

To set up the stack, the stack pointer has to point to a memory address known to be unused and in memory. This parameter is located in the file "offsets.h".

Next, we test the ELF header for validity. We know that the first word, being the previously inserted jump instruction, is invalid, so we can safely omit it. Apart from this missing first byte, the subsequent three bytes should be a valid ELF magic number [20]. In case of an invalid ELF header, the execution of the code is halted.

After successfully reading the ELF header, we obtain the file size of the ELF image and expanded elf file.

4.2.3 Physical memory layout

The boot loader can put the kernel anywhere in memory. Therefore, the kernel can make no assumptions about its location in memory. This raises two main problems: The first one, address dependency, can be solved by writing address independent code for all parts running in physical memory. The second problem is the possibility of overwriting our own code, which could happen for example while extracting the kernel. This can be solved by copying the kernel to a known and therefore safe location.

Figure 4 gives an overview over the steps involved. The whole process of preparing the image to run in virtual memory is performed in the lower address range. In the end, the final, expanded version of the kernel should reside within the first 1 MB section. This is due to the configured kernel offset of 0xfff00000. The numbers in Figure 4 will give a rough idea about the size and the location of the different memory blocks. Please not that these numbers are not hard coded. The memory settings and therefore the addresses can be configured in a file called “offsets.h”.

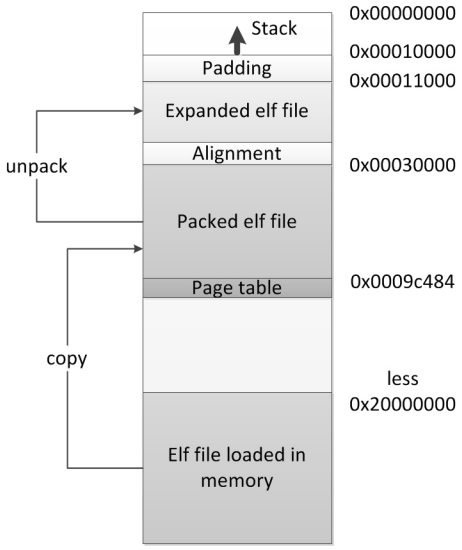

Figure 4: Prepare kernel for virtual memory

4.2.4 Physical memory setup

First we check where the boot loader has loaded the kernel into memory. If the image lies below the 512 MB memory barrier, we perform a first copy operation. The target address is chosen in such a way that there is enough room for the expanded file later on. In the other case where the image is loaded to an address higher than 512 MB, there is nothing to do. One may ask why this first copy operation is necessary. In Figure 4, we come across an ideal situation where the copy process is indeed not required. Yet imagine that the boot loader loads the kernel on 0x11000. In this case, the copy is necessary.

As shown in Figure 4, we unpack the kernel just before the previously copied packed ELF file. Although "expanding" suggests otherwise, the resulting unpacked memory block is much smaller than the original file. The packed ELF file contains a lot of debugging and header information which is not essential after switching to virtual memory.

An interesting question still remains: where does the kernel get all this information from? To answer this question, we again take the example from Figure 4 and have a look at all addresses involved:

- Stack: the start of the stack is located at 0x10000. This is preconfigured and can be read from "offsets.h"
- Expanded_elf: although the start location for the expanded ELF file is configured in "offsets.h" (here: 0x11000), boot.S takes a different approach to calculate the start location for the expanded ELF file. We know two things:

1. kernel_v_addr: from reading the ELF header
2. kernel_v_offset: the kernel's virtual base address, obtained from "offsets.h"

By using a bitwise AND (denoted by &), we get:
$Expanded\_elf = kernel\_v\_addr \ \& \ kerne\_v\_offset$

- Elf_size: we can calculate the size of the expanded ELF image using the ELF header information. This gets aligned to four KB, therefore we need some extra space here.
- Finally, we can obtain the start location for the first copy operation:
$Elf\_copy = Expanded\_elf + Elf\_size$

4.2.5 On to the C code

After the whole copy and expanding process has been finished, we can allocate our page table as shown in Figure 4 and insert the following three mappings:

- The expanded elf file gets mapped to the virtual address used by the linker to create the ELF file.
- We use a one-to-one mapping for the program counter so as to have the page table in virtual memory. There is no need for the mapped code associated with the program counter because after turning on the MMU we will continue execution from the expanded elf file.
- The ATAG headers are also mapped one-to-one into virtual memory. This was not necessary for the QEMU ARM port because the ATAG headers were located within an already mapped region.

After preparing the program counter and stack pointer for relocation, we can turn on the caches, alignment checking and the MMU.

Before calling the first C function, we unmap memory address zero and prepare the arguments for the C function. Now we can finally branch to C code.

4.3 Prepare for user space

The boot process continues in a file called "init.c". It is responsible for device and memory setup. The remaining part of the start up process until user space is located in the file "startup_arch.c".

4.3.1 Virtual memory in C

The memory mapping functions in C (located in "paging.c") were incompatible with our Big Endian system. As soon as we tried to map memory and devices to virtual memory, the kernel crashed as soon as we attempted to access the

newly mapped region. It was difficult to locate the problem, as the mapping of the serial console to virtual memory is necessary to produce output, and without output, it was unidentified if the assembly mappings worked. It was even unknown how far the kernel progressed.

As we inserted a one-to-one mapping for the serial console into the assembly file, we were able to generate output. After analyzing the problem for quite some time, we noticed that the bit fields used to setup the page table entries are Endian dependent. After changing the order of the bit fields, the virtual memory system worked.

4.3.2 ATAG header information

In the first part of the C code, we process the ATAG headers to find all available memory and the ramdisk romfs.cpio. Furthermore, we check for certain devices like the command line. If they are available, they will be initialized. Now that we know where the memory is located, we can map all available physical memory to virtual memory.

4.3.3 UART setup

To get some output on the serial console (UART), we need to initialize it and provide a simple driver.

In the configuration step, we enable the UART device and disable parameters like a "non return to zero encoding". During the start phase, we furthermore disable all interrupts.

The serial output driver is very simple. As soon as the driver function gets called, we spin until the UART is ready to receive a character. We transmit one character when ready. To send strings, we can simply loop over this simple send function.

4.3.4 Programmable interrupt controller setup

The programmable interrupt controller (PIC) is responsible for managing the interrupts originated from different devices. Apart from masking interrupts per device, we can also mask them in the PIC. Some devices (like the timer) do not support masking their interrupt at device level.

Although the PIC manages the interrupts, we do not have to acknowledge them here. On the ARM platform, it is sufficient to clear the interrupt at device level. For example if we have a timer interrupt, we can clear it on the timer device. The interrupt on the PIC will be cleared automatically.

During initialization, we define which interrupts we are going to use. In our case, we use the UART, timer and the error sum. The error sum is the logical OR of all error status registers. If any other operating system component tries to turn on an interrupt defined not to use, a kernel panic will occur.

Next, we disable all interrupts. They will be activated one by one as we need them.

4.3.5 Programmable interval timer setup

We need a programmable interval timer (PIT) to implement preemptive scheduling and other system services [11] (pages 57 - 60).

The operation principle of the PIT is pretty simple: it counts down from a previously configured value. If it reaches zero, it will set the timer interrupt. The timer itself will wrap around and restart its count-down.

On the IXP2800 platform, the timer is attached to the Advanced Peripheral/Bus Clock (APB-CLK). It has an operation frequency of 50 MHz [16].

During initialization, we first map the timer device to virtual memory. Afterwards, we load the timer with an initial value and set the divisor. The divisor is applied to the APB-CLK to reduce the number of ticks. We activate the timer interrupts on the PIC, but we do not yet start the timer.

4.3.6 Load user space application

The location of the ramdisk is hard-coded. We first check the validity of the archive as a whole. Because of this checking, we found out that the boot loader changed our image (see chapter 4.1.3).

A bit later, when unpacking the archive, we get all user space programs. They are checked again for validity. At first, these checks failed as well on our platform. The reason was an endianess check which verified that the user space programs are compiled for Little Endian. We replaced this by a function which determines the endianess of our platform and verifies that the user space programs are compiled using the same endian as the kernel.

4.3.7 Launch init.c

Before we can launch the first user space program called "init", we need to perform some more preparatory work. Amongst others, this involves:

- Creating a multilevel page table and mapping the kernel into the new page table.
- Setting up a struct called "BOOTINFO" which contains a lot of information (e.g. memory location). The struct needs to be mapped to virtual memory so that it becomes accessible by init.
- Preparing command line arguments, which also contain the location of BOOTINFO.
- Creating a DCB (Dispatcher Control Block), preconfiguring and mapping it to init.
- Context switching to init and flushing caches.
- Configuring capabilities, for example for physical memory and the page table of init.

The QEMU implementation behaved strangely just after the context switch. The execution jumped back to an earlier stage followed by the kernel crashing a bit later. This anomaly turned out to be a caching problem. The original QEMU implementation merely invalidates the cache without cleaning it first.

4.4 User space

Init is fully privileged and responsible for bringing up the other user space programs (e.g. monitor or mem_serv). In this chapter we describe the problems encountered during startup of the user space processes.

4.4.1 Processing mode

The Barrelfish interrupt handler checks whether the software interrupt came from user space or from privileged mode. A software interrupt originated from privileged mode does not make sense since there is no need to use the system call interface if we are executing the kernel. Although the system call came from init, the kernel detected privileged mode which led to a kernel panic.

We found out that init was running in system mode instead of user mode. However, the resume process was correct and should have set user mode instead of system mode. However, checking the current execution mode revealed that the kernel was running in system mode as well. As described in subsection 3.4.4, we cannot restore the SPSR in this mode.

To correct the problem, we configured privileged mode in the very beginning of the kernel boot process. We furthermore had to move the stack for privileged mode just above the IRQ save area to make things work.

4.4.2 Caches and system calls

We inserted multiple system calls called "sys_print" to track program execution for debugging purposes. Yet inserting and removing these print instructions as well as allocating memory on the stack altered the execution. Sometimes the kernel crashed later when inserting a print statement as without the print statement.

In this situation we encountered a caching problem once more. Turning off all caches resulted in a more predictable system behavior. What caused trouble here is the virtually addressed cache. If we map a memory region twice, once from the kernel and once from user space, they have a different virtual memory address. If init and the kernel access the same memory region, it is cached twice. Hence, a write from init may be cached and is invisible for the kernel, as the kernel accesses its own cached value.

As a temporary solution, we turned off the d-cache just before switching to user space. This allows a fast system startup and corrects the problem in user space. However, this needs to be changed in future versions.

4.4.3 Stack alignment

Calling a certain function named "memobj_create_pinned" resulted in a page fault. Although this region should have been in virtual memory and was supposed to be accessible (tested using a memory sweep), the function call resulted in a page fault.

After commenting the function out, we received a page fault a bit later on. However, it was always the STRD instruction (load a 64 bit value from memory) which led to a crash. A function call to "memobj_create_pinned" without function body also resulted in a page fault. Therefore, the function call itself had to be the problem.

The ARM platform requires an 8 byte stack alignment [21]. After aligning the stack pointer of init, the problem was gone.

5 Future work

As a first step to make Barrelfish operational, the remaining user space applications should be ported as well. This includes "init", the memory server, "monitor", "spawnd", "ramfsd" and the shell, "fish".

Because of the virtually addressed cache, the exact places to flush and clean the cache will have to be determined so that the d-cache can be activated again. To increase performance further, the cache should be cleaned only partially. This optimization may be tricky as one has to know exactly which addresses in virtual memory need to be cleaned, for example in case of a system call.

Running a micro-benchmark to measure the context switch time on the Netronome platform would provide insight into the performance of the Barrelfish ARM port compared to Barrelfish running on other platforms. To get a reference value on the same platform, one could run a Benchmark on Barrelfish and on the preinstalled Monta Vista Linux.

In a next step, a network driver for the Netronome could be written. This allows testing the network stack and the overall IO-performance of Barrelfish under heavy load.

The ultimate goal is to run one instance of Barrelfish on a host computer and on the Netronome simultaneously. To enable communication between them, both kernel instances need a PCIe driver.

6 Conclusions

The boot process and the CPU driver are working on the Netronome. Drivers for the UART and PIT are available and written using Mackerel 2, which ensures that the drivers are Endian independent. The virtual memory system functions properly, thus making the CPU driver complete. Furthermore, for developers working on the ARM port of Barrelfish, this thesis can serve as a documentation for the boot process.

Due to the obstacles encountered and the time limitation we were unable to reach our initial goal to see the shell, "fish", running on the Netronome. However, as many problems have already been identified and corrected, the current status of the port sets a solid foundation to achieve this goal in a later step.

Some answers to the initially posed questions from subsection 1.3 have been found. First, a port from Little Endian to a Big Endian system is tricky as C code is not completely Endian independent and the problems are hard to locate. It is furthermore difficult to get a Big Endian tool chain and we therefore recommend to avoid Big Endian whenever possible.

Referring to the second question, different ARM hardware platforms are compatible to some degree. Most of the problems we encountered were caused by the porting from a simulator to real ARM hardware. When there is only a simulator port available, we strongly recommend to first port the OS to a platform as similar as possible to the simulator. This narrows down the source of error and makes debugging easier. During our work we have encountered problems like caching and alignment, which did not show up on the simulator.

Due to the fact that user space is currently not fully implemented and we have no long running programs, we cannot make a clear statement about the overall stability of Barrelfish on the Netronome. As far as we can tell both the boot process and to some point init are running smoothly after our adaptions.

References

[1] P.-E. Dagand T. Harris R. Isaacs S. Peter T. Roscoe A. Schuepbach A. Baumann, P. Barham and A. Singhania. The multikernel: A new os architecture for scalable multicore systems. *In 22nd ACM Symposium on OS Principles, Big Sky, MT, USA*, October 2009.

[2] Intel quickpath architecture. `http://www.intel.com/pressroom/archive/reference/whitepaper_QuickPath.pdf`. White Paper.

[3] Debian gnu/linux on arm. `http://www.debian.org/ports/arm/index.en.html`.

[4] Freebsd on arm. `http://www.freebsd.org/platforms/arm.html`.

[5] Arm port of the l4 microkernel. `http://www.l4hq.org/arch/arm/`. L4 community.

[6] Antti Kantee. Porting netbsd/evbarm to the arcom viper. Technical report, Helsinki University of Technology.

[7] Ming Chen. A java virtual machine for the arm processor. Master's thesis, University of Manchester.

[8] Code sourcery. `http://www.codesourcery.com/`.

[9] Building and testing gcc/glibc cross toolchains. `http://www.kegel.com/crosstool/`.

[10] Gaurang Khetan. Comparison of memory management systems of bsd, windows, and linux. Technical report, Department of Computer Science, University of Southern California, Los Angeles, CA., December 16, 2002.

[11] Michael J. Karels John S. Quarterman Marshall Kirk McKusick, Keith Bostic. *The Desing and Implementation of the 4.4 BSD Operating System*. ADDISON WESLEY, 2006.

[12] Randal E. Bryand and David R. O'Hallaron. *Computer Systems A PROGRAMMER'S PERSPECTIVE*. Pearson Education, 2003.

[13] *Intel 64 and IA-32 Architectures Software Developers Manual System Programming Guide, Part 1*, 3a edition, January 2011.

[14] *ARM Architecture Reference Manual*, 2005.

[15] Chris Wright Andrew N. Sloss, Dominic Symes. *ARM System Developers Guide, Designing and Optimizing System Software*. Morgan Kaufmann publications, 2004.

[16] *Intel IXP2800 Network Processor Hardware Reference Manual*, November 2003.

[17] Intel xscale microarchitecture. `ftp://download.intel.com/design/intelxscale/XScaleDatasheet4.pdf`. Technical Summary.

[18] Intel i/o processors based on intel xscale technology: Single chip processor, validated chipset and standalone core. `ftp://download.intel.com/design/network/papers/25286901.pdf`. White Paper.

[19] Redboot user's guide. `http://ecos.sourceware.org/docs-latest/redboot/redboot-guide.html`.

[20] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*, May 1995.

[21] ARM. *ABI for the ARM Architecture Advisory Note - SP must be 8-byte aligned on entry to AAPCS-conforming functions*, October 2009.