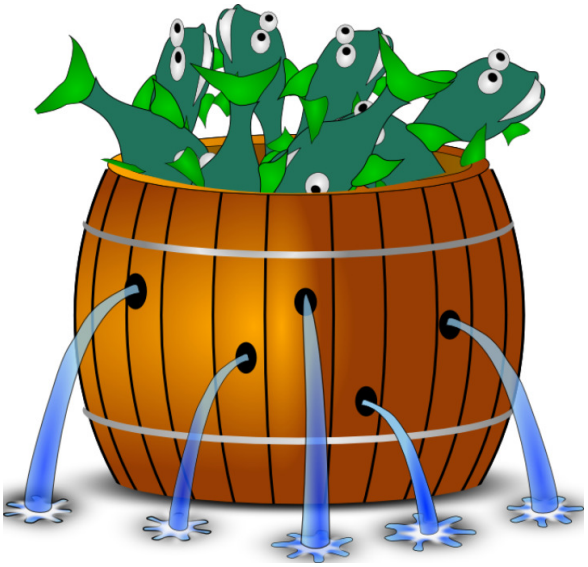
Barrelfish Project
ETH Zurich



Barrelfish on ARMv8

Barrelfish Technical Note 022

David Cock

11.04.2016

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
1.0	11.04.2016	DC	Initial version

Chapter 1

Summary

Barrelfish now supports ARMv7 and ARMv8 as primary platforms, and we have discontinued support for all older architecture revisions (ARMv5, ARMv6). The current Barrelfish release contains a port to a simulated ARMv8 environment, derived from the existing ARMv7 codebase and running under GEM5, with generous contributions from HP Research.

Simultaneously, we are undertaking a clean-slate redesign of the CPU driver for ARMv8, as it presents a number of novel features, and greatly improved platform standardisation (Section 2.1.1), that should allow for a much cleaner and simpler implementation. This redesigned CPU driver will for the basis for ongoing research into large-scale non-cache-coherent systems using ARMv8 cores. This document presents the new CPU driver design (Chapter 3), briefly covering those features of ARMv8 of greatest relevance (Chapter 2), and discusses a number of technical challenges presented by the new architecture (Chapter 5).

Chapter 2

Background

The Barrelfish research operating system is a vehicle for research into software support for likely future architectures, where large numbers of non-coherent (or weakly-coherent) heterogeneous processor cores are assembled into a single large-scale system. As such, support for a common non-x86 architecture has always been part of the project, beginning with the ARMv5 (XScale) port, which permitted the embedded processor on a network interface card to be integrated as a first-class part of the system, with its own CPU driver. We have also actively maintained an ARMv7 port, to the OMAP4460 processor on the Pandaboard ES, which we use as a teaching platform in the Advanced Operating Systems course at ETH Zürich. These ports are described more fully in the accompanying technical report, Project [2013].

2.1 The ARMv8 Architecture

The ARMv8 architecture is quite a radical departure from previous versions, and represents the culmination of a trend that has been developing for quite some time. While the first wave of ARM-based microservers, based on the 32-bit ARMv7 architecture, was largely a commercial failure, it's clear that ARM is now actively targeting the server market, where Intel currently has near-total dominance.

ARMv8 discards some long-standing features of the ARM instruction set: universal conditional execution, multiple loads/stores, and the program counter as a general-purpose register. These most likely caused difficulty in scaling the processor pipeline to high clock rates, and we present some consequences of their loss in Section 5.1 and Section 5.2. The instruction-set changes, however challenging to the systems programmer, are ultimately of little consequence compared to the consolidation of the ARM ecosystem into a serious server platform. The two features of most interest at this stage in the design process are the standardisation of hardware features and memory maps, and of the boot process.

2.1.1 ARM Server Base System Architecture

ARM has long been criticised by systems programmers for its highly fragmented, and non-uniform programming interface. Linux, in particular, has struggled for years with supporting the great multiplicity of ARM platforms. The principal reason for this is the lack of any concept of a *platform*: a set of assumptions (available hardware, memory map, etc.), that programmers can rely on when initialising and managing a system. The Linux source tree famously contained a vastly greater amount of code in the ARM platform support subtrees, than that for x86.

The relative standardisation of the x86 platform is largely a historical accident, due to the rapid proliferation of PC/AT clones in the early 1980s. The x86 platform thus contains layers of ossified legacy interfaces, necessary to ensure broad cross and backward compatibility. ARM's business model, on the contrary, has long emphasised the specialisation of implementations: an ARM licensee would take

Supplier	Processor	Name	
APM	APM883208	Mustang	1P 8-core X-Gene 1 with serial trace.
	APM883408-X2	X-C2	1P 8-core X-Gene 2.
Cavium	CN8890	StratusX	1P 48-core ThunderX.
		Cirrus	2P 48-core ThunderX.
ARM	AEM	Fixed Virtual Platform	The *architectural envelope model* covers the range of behaviour permitted by ARMv8. Bare-metal debug.
		Foundation Platform	Freely available, compatible with FVP.

Table 2.1: ARMv8 platforms of interest

their ARM-designed CPU core, and integrate it themselves in to a complex SoC (system on a chip), with their own specialised, proprietary interfaces. The upsides of this were the possibility to highly optimise a particular design, and no requirement on ARM itself to maintain a coherent platform.

While ARM's customisable platform worked well for embedded devices, and scaled reasonably well to relatively powerful smartphones, it's a disaster for producing high-quality systems software, able to execute on a broad range of hardware from competing vendors: exactly what a competitive server platform requires. ARM clearly know this, and since 2014 have published the Server Base System Architecture [ARM, 2016]. To the extent that manufacturers adhere to these guidelines, our job as systems programmers is significantly simpler: it should be possible to write a single set of initialisation and configuration code for ARMv8, that will run on any SBSA-compliant system, much as we already do for x86-64.

Our target platforms listed in Table 2.1 all support SBSA to some extent, and absent any compelling reason, we will only support SBSA-compliant platforms.

2.1.2 UEFI

One aspect of the SBSA which eases portability is the specification, for the first time, of a boot process for ARM systems. ARM has specified that SBSA systems must support UEFI [UEFI, 2016b] (the unified extensible firmware interface). UEFI is a descendant of the EFI specification, developed by Intel for the Itanium project. While Itanium is no longer a platform of any great commercial interest, UEFI support is now widespread in the x86-64 market. UEFI, in turn, specifies the use of ACPI [UEFI, 2016a] (the advanced configuration and power interface) for platform discovery and control.

Supporting ACPI and UEFI requires a one-off investment of effort to design a new boot and configuration subsystem, but should pay off in the long term, as ports to new ARM boards will no longer require extensive manual configuration. The code should also be largely reusable for x86 UEFI systems. Our new UEFI bootloader is described in Section 4.1.

2.2 A Direct Port from ARMv7

As already described, the Barrelfish release current at time of writing includes an initial ARMv8 port to the GEM5 simulator. This port contains code generously contributed by HP Research.

Being developed from the existing codebase, this ARMv8 port follows the structure of the existing ARMv7 code closely. While it is highly useful to have a running port, we are nevertheless continuing with a significant redesign of the CPU driver, as significant improvements and simplifications will be possible, once we no longer need to follow the existing structure, originally developed for a significantly different platform.

The GEM5 simulator's model of an ARMv8 platform is relatively primitive, and does not conform to modern platform conventions, for example placing RAM at address `0`, rather than `0x80000000` as mandated by the SBSA. For this reason, in addition to better integration with ARM debugging tools, we have switched to the ARM Fixed Virtual Platform as our default simulation environment, with the Foundation Platform supported as a freely available simulator.

2.3 Registers

2.3.1 General-purpose Registers

In total there are 31+1 general purpose registers (`r0-r30`) of size 64bits(Table 2.3). They are usually referred to by the names `x0-x30`. The 32-bit content of the registers are referred to as `w0-w30`. The additional stack pointer `SP` register can be accessed with a restricted number of instructions.

Register	Special	Description
`X0-X7`	Caller-save	function call arguments and return value
`X8`		indirect result e.g. location of large return value (struct)
`X9-X15`	Caller-save	temporary registers
`X16`	IP0	The first intra-procedure-call scratch register[1]
`X17`	IP1	The second intra-procedure-call temporary register
`X18`		The Platform Register (TLS), if needed; otherwise a temporary register.
`X19-X28`	Callee-save	need to be preseved and restored when modified
`X29`	FP	frame pointer
`X30`	LR	link register
`SP`		stack pointer (XZR)

Table 2.2: ARMv8 General purpose Registers

Procedure call

- The registers `x19-28` and `SP` are callee-saved and hence must be preserved by the called subroutine. All 64 bits have to be preserved even when executing in the 32-bit mode.
- The registers `x0-x7` and `x9-x15` are caller saved.
- During procedure calls the registers `x16`, `x17`, `x29` and `x30` have special roles i.e. they store relevant addresses such as the return address.
- Arguments for calls are passed in the registers `x0-x7`, `v0-v7` for floats/SIMD and on the stack

Indirect result This register is used when returning a large value such as declared by this function: `struct mystruct foo(int arg);`.

Platform specific The use of register `x18` is platform specific and needs to be defined by the platform ABI. This register can hold inter-procedural state such as the thread context.

Linker The registers `IP0` and `IP1` can be used by the linker as a scratch register or to hold intermediate values between subroutine calls.

2.3.2 SIMD and Floating point

There are 32 registers to be used by floating point and SIMD operations. The name of those registers will change, depending on the size of the operation.

Register	Description
`v0-v7`	function call arguments, intermediate values and return value, caller save registers
`v8-v15`	Callee-save registers. They need to be preserved
`v16-v31`	Caller-save registers

Table 2.3: ARMv8 General purpose Registers

Chapter 3

Design and Implementation

3.1 Redesigning the CPU Driver

Given that ARMv8 is a significantly different platform to ARMv7, and that the ARMv7 codebase carries a significant legacy, reaching right back to ARMv5, we are pursuing substantial redesign of the CPU driver. Taking advantage of the standardisation of the hardware platform mandated by the SBSA (Section 2.1.1), and the facilities provided by UEFI (Section 2.1.2), in addition to a relatively unrestricted virtual address space, we are able to significantly reduce the complexity of the CPU driver. In this section we describe the updated design, and our progress on its implementation, while the UEFI interface (Hagfish) is described separately, in Section 4.1.

Terminology In the interest of clarity, in the discussion that follows, we use a few terms with precise intent:

shall indicates features or characteristics of the design to which the Barrelfish implementation must conform.

should indicates features which should be supported if at all possible.

initially indicates features which will be provided from the outset in the Barrelfish implementation.

eventually indicates features which will be provided later in the Barrelfish implementation, and which the initial design will aim to facilitate.

3.1.1 Goals

Our goal is to provide a reference design for the CPU driver and user-space execution environment for Barrelfish on an ARMv8 core, in order to understand both positive and negative implications of the architecture for a multikernel system. The design **should** be applicable to any ARMv8 with virtualisation (EL2) support.

Initially, our hardware development platform is the APM X-Gene 1, using the Mustang Development Board. We are using the Mustang principally as it was relatively easily available, as well as being a comparatively complex and powerful CPU. The ThunderX platform from Cavium is very interesting for Barrelfish, as it ties a large number (48) of less-powerful (2-issue) cores. We do not have the resources to develop for two platforms simultaneously, but we hope to **eventually** add support for the ThunderX.

Our target simulation environment is the ARM Fixed Virtual Platform, and the Foundation Platform. These models are supplied by ARM. The Foundation Platform is freely available, and will be the default supported simulation platform for the public Barrelfish tree, while we will use the FVP internally to

allow bare-metal debugging. Future support for QEMU is desirable, to the extent that it models a compatible system — GEM5, which the ARMv7 port targets, currently does not.

Initially, the design will support running both the CPU driver and user-space processes in AArch64 mode without support for virtualisation. **Eventually** the design will support running the CPU driver in AArch64 mode, and user-space processes in both AArch64 and AArch32 modes without virtualisation, and virtual machines in AArch64 mode. We will only support virtualisation on ARMv8.1 or later platforms, that support the VHE extensions, as described in Section 3.1.3.

3.1.2 Processor Modes and Virtualisation

Where possible, we will keep the virtualisation model similar to that on Barrelfish/x86. In particular, it **should** be possible to implement native applications, fully virtualised (e.g. Linux) VMs, and VM-level applications e.g. Arrakis [Peter et al., 2014].

ARMv8 has a somewhat different virtualisation model to x86, and different again from the ARMv7 virtualisation extensions. Rather than having exception levels (rings) duplicated between guest and host, ARMv8 provides 4 exception levels (ELs):

- `EL0` is unprivileged — user applications.
- `EL1` is privileged — OS kernel.
- `EL2` is hypervisor state.
- `EL3` is for switching between secure and non-secure (TrustZone) modes. The X-Gene 1 does not implement `EL3`, and it is currently not of interest for Barrelfish.

Explicit traps (syscalls/hypercalls) target only the next level up: `EL0` can call `EL1` using `svc` (syscall), and `EL1` can call `EL2` using `hvc` (hypercall), but `EL0` cannot directly call `EL2`, unless `EL1` is completely disabled. Exceptions return to the caller's exception level.

ELs **shall** be distributed as follows: The CPU driver **shall** exist at both `EL1` and `EL2`, and take both syscalls (`svc`, from `EL0` applications) and hypercalls (`hvc`, from `EL1` applications). The system **shall** support applications both at `EL0`, and at `EL1` (e.g. Arrakis, VMs). Most code paths **should** be identical, as most CPU driver operations do not depend on `EL2` privileges. Hypercalls from `EL0` **shall** be chained via `EL1` (with appropriate permission checks).

`EL1` apps such as Arrakis, and paravirtualised VMs using hypercalls know that they are being virtualised, and will use `hvc` explicitly. Fully-virtualised `EL1` VMs do not make hypercalls.

ARMv8 implements two-level address translation: VA (virtual address) to IPA (intermediate physical address), and IPA to PA (physical address). `EL1` guests **shall** be isolated at the L1 translation layer, and by trapping all accesses to system control registers.

3.1.3 Virtual Address Space Layout

ARMv8 has an effective 48-bit virtual address space. At the lowest execution levels (0 — BF user & 1 — BF CPU driver), the hardware supports two (up to) 48-bit (256TB) 'windows' in a 64-bit space: one at the bottom, and one at the top. Each region has its own translation table base register (`TTBR0` & `TTBR1`). `TTBR0` is used at `EL0`, and `TTBR1` at `EL1`.

In the initial ARMv8 specification, this split address space was not implemented at `EL2`, which would require a separate CPU driver instance for virtualisation, and hypercalls (e.g. for Arrakis). ARMv8.1 introduced the virtualisation host extensions (VHE) which, among other things, extends the split address space to `EL2`. As this provides a cleaner implementation model, and to avoid having to support a now-deprecated interface, virtualisation will **only** be supported on ARMv8.1 and later. This means that we will not support virtualisation on the X-Gene 1. Both the simulation environment (FVP/FP) and, seemingly, the ThunderX chips, support VHE.

tpidrro_el0	EL0 Read-Only Software Thread ID Register
tpidr_el0	EL0 Read/Write Software Thread ID Register
tpidr_el1	EL1 Read/Write Software Thread ID Register
tpidr_el2	EL2 Read/Write Software Thread ID Register
tpidr_el3	EL3 Read/Write Software Thread ID Register

Table 3.1: Thread ID registers in ARMv8

The CPU driver **shall** use TTBR1 to provide a complete physical window. The ARMv8 CPU driver **shall not** dynamically map device memory into its own window (as the ARMv7 CPU driver does) — the few memory-mapped devices required will be statically mapped on boot, with appropriate memory attributes. All physical addresses, RAM and device, **shall** be accessible at a static, standard offset (the base of the TTBR1 region).

User-level page tables will **initially** be limited to a 4k translation granularity. **Eventually** user-level page tables **should** have access to all page-table formats and page sizes, as is the case in the current Barrelfish x86 implementation.

3.1.4 Address Space, Context, and Thread Identifiers

ARMv8 also provides address-space identifiers (ASIDs) in the TLB to avoid flushing the translation cache on a context switch.

ARMv8 ASIDs (referred to in ARM documentation as context IDs) are architecturally allowed to be either 8 or 16 bits, although the SBSA specifies that they must be at least 16. Relying on the SBSA platform will allow us to avoid multiplexing IDs among active processes, on any reasonably-sized system. Managing the reuse of context IDs can be left to user-level code, and does not need to be on the critical path of a context switch. The CPU driver need only ensure that every allocated dispatcher has a unique ASID, which is loaded into the ContextID register on dispatch.

The value in the ContextID register is also checked against the hardware breakpoint and watchpoint registers, in generating debug exceptions. Therefore, it shall be possible for authorised user-level code to load the Context ID for a given dispatcher into a breakpoint register — this may be an invocation on the dispatcher capability.

In addition to the ContextID register, used to tag TLB entries, ARMv8 also provides a set of thread ID registers with no architecturally-defined semantics, as listed in Table 3.1. The client-writeable tpidr_el0 and tpidr_el1 **shall** have no CPU driver-defined purpose, but **shall** be saved and restored in a dispatcher's trap frame, to allow their use as thread-local storage (TLS). Recall that the Barrelfish CPU driver has no awareness of threads, which are implemented purely at user level.

To implement the upcall/dispatch mechanism of Barrelfish, the CPU driver and the user-level dispatcher need to share a certain amount of state — the user-visible portion of the dispatcher control block, which contains the trap frames, and the disabled flag (used to achieve atomic dispatch). The address of this structure needs to be known to both the CPU driver, and to user-level code, and moreover be efficiently-accessible, as the CPU driver needs to find the trap frame on the critical path of system calls and exceptions. This pointer also needs to be trustworthy, from the CPU driver's perspective, and thus cannot be directly modifiable by user-level code.

The x86-32, x86-64, and ARMv7 CPU drivers all store the address of the running dispatcher's shared segment at a fixed known address, dcb_current, which is loaded by the trap handler. At user level, on x86 this address is held in a *segment register* (fs on x86-64, and gs on x86-32), while on ARMv7 we sacrifice a general-purpose register (r9) for this purpose. Using the tpidrro_el0 register to hold the address of the current dispatcher structure will allow us to avoid both a memory load on the fast path, and sacrificing a register in user-level code, thus tpidrro_el0 **shall** hold the address of the currently-running dispatcher.

3.1.5 Instruction Sets

ARMv8 supports both AArch64, and legacy ARM/Thumb (renamed AArch32). Switching execution mode is only possible when switching execution level i.e. on a trap or return, and can only be changed while at the higher execution level. Thus, `EL2` can set execution mode for `EL1`, and `EL1` for `EL0`. There is no way for a program to change its own execution mode. If `ELn` is in AArch64, then `EL(n-1)` can be in either AArch64 or AArch32. If `ELn` is in AArch32, all lower ELs must also be AArch32.

The CPU driver **shall** execute in AArch64.

Initially, the CPU driver will enforce that all directly-scheduled threads also use AArch64, by controlling all downward EL transitions. An `EL1` client (such as Arrakis or a full virtual machine) may execute its own `EL0` clients in AArch32 (and there is no way to prevent this). However, all transitions into the CPU driver (`svc`, `hvc` or exception) must come from a direct client of the CPU driver, and thus from AArch64. The syscall ABI **shall** be AArch64.

Eventually, Barrelfish **should** also support the execution of AArch32 dispatcher processes, by marking each dispatcher with a flag indicating the instruction set to be used (much as is already done with VM/non-VM mode in the Arrakis CPU driver).

3.1.6 User-Space Access to Architectural Functions

Generally, anything that can be safely exported, **should** be made available outside of the CPU driver, preferable as a memory-mapped interface, at 4kiB granularity. The SBSA mandates that devices be present at addresses that can be individually mapped, thus this should not be a problem.

3.1.7 Cache Management

ARMv8 has moved most cache and TLB management from the system control coprocessor (cp15), into the core ISA. Several cache operations (invalidate/clean by VA) are executable at `EL0`, and thus no kernel interface is required. The system must take into account that user-directed flushes may have occurred, or may occur concurrently with any memory operation.

3.1.8 Performance Monitors

Performance monitors **should** be exposed, if it can be done safely.

3.1.9 Debugging

Self-hosted debug **should** be exposed, if it can be done safely. This is under active development.

3.1.10 Booting

Platform support i.e. a standard set of peripherals, and a defined boot process, has improved dramatically on ARM, as it has been repositioned as a server platform. UEFI and ACPI support are widespread, including on the Mustang development board. We will assume support for UEFI booting, make use of ACPI data, where available.

The Barrelfish CPU driver and initial image **shall** be loaded and executed by a UEFI shim, which will pass through all UEFI-supplied information, such as ACPI tables, and be able to interpret a Barrelfish Multiboot image. This shim, or second-stage bootloader, is called Hagfish, and is described in Section 4.1.

3.1.11 Interrupts

ARMv8 interrupt handling is not substantially different from the existing architectures and platforms supported by Barrelfish. While a redesign of the Barrelfish interrupt system is under way (to use capabilities to grant access to receive interrupts), we do not anticipate ARMv8 to impose any particular challenges.

The ARMv8 systems we **initially** target all use minor variations on the ARM Generic Interrupt Controller (GIC) design, already supported in Barrelfish. We currently have support for version 2 of the GIC, with which later implementations are backward-compatible. We will **eventually** support GICv3, the current specification at time of writing.

3.1.12 Inter-Domain Communication

User-level communication between cache-coherent cores in Barrelfish for ARMv8 is likely to the same as with ARMv7 and x86, and we expect the existing User-level Message-Passing over Cache-Coherence (UMP-CC) interconnect driver to work unmodified.

Between dispatchers on the same core, however, the different register set on the ARMv8 is likely to result in a very different Local Message Passing (LMP) interconnect driver—this is always an architecture-specific part of the CPU driver. In practice, its design will be closely tied to the context switch and upcall dispatch code.

Chapter 4

Booting

Booting ARM systems has always been difficult to do in a standard way, and ARMv8 systems are no exception. Barrelfish uses one of two methods of booting an initial ARMv8 core, depending on whether the hardware platform supports UEFI UEFI [2016b] or U-Boot. If a platform supports neither, more work will be required to boot the board.

If a board has full support for UEFI (such as TianoCore), you can use Hagfish 4.1 to individually load the modules needed to boot Barrelfish and set up the initial CPU/MMU environment before entering the CPU driver proper.

Note that U-Boot also claims to support UEFI. However, in practice it supports a small subset of UEFI functionality sufficient to boot `grub` or the Linux kernel as an EFI binary. If your board boots via U-Boot, you should use the minimal EFI bootloader 4.2 which loads a single multiboot image into memory and sets up the environment similar to Hagfish.

4.1 Hagfish

The Barrelfish/ARMv8 UEFI loader prototype is called Hagfish[1]. Hagfish is a second-stage bootloader for Barrelfish on UEFI platforms, initially the ARMv8 server platform. Hagfish is loaded as a UEFI application, and uses the large set of supplied services to do as much of the one-time (boot core) setup that the CPU driver needs as is reasonably possible. More specifically, Hagfish:

- Is loaded over BOOTP/PXE.
- Reuses the PXE environment to load a menu.lst-style configuration.
- Loads the kernel image and the initial applications, as directed, and builds a Multiboot image.
- Allocates and builds the CPU driver's page tables.
- Activates the initial page table, and allocates a stack.

4.1.1 Why Another Bootloader?

The ARMv8 machines that we're porting to are different to both existing ARM boards, and to x86. They have a full pre-boot environment, unlike most embedded boards, but it's not a PC-style BIOS. The ARM Server Base Boot Requirements specify UEFI. Moreover, there is no mainline support from GNU GRUB for the ARMv8 architecture, so no matter what, we need some amount of fresh code.

[1] A hagfish is a basal chordate i.e. something like the ancestor of all fishes.

Given that we had to write at least a shim loader, and keeping in mind that UEFI is multi-platform (and becoming more and more common in the x86 world), we're taking the opportunity to simplify the initial boot process within the CPU driver by moving the once-only initialisation into the bootloader. In particular, while running under UEFI boot services, we have memory allocation available for free, e.g. for the initial page tables. By moving ELF loading and relocation code into the bootloader, we can eliminate the need to relocate running code, and can cut down (hopefully eliminate) special-case code for booting the initial core. Subsequent cores can rely on user-level Coreboot code to relocate them, and to construct their page tables.

4.1.2 Assumptions and Requirements

Hagfish is (initially at least) intended to support development work on AArch64 server-style hardware and, as such, makes the following assumptions:

- 64-bit architecture, using ELF binaries. Porting to 32-bit architectures wouldn't be hard, if it were ever necessary (probably not).
- PXE/BOOTP/TFTP available for booting. Hagfish expects to load its configuration, and any binaries needed, using the same PXE context with which it was booted. Changing this to boot from a local device (e.g. HDD) wouldn't be hard, as the UEFI `LoadFile` interface abstracts from the hardware.

4.1.3 Boot Process

In detail, Hagfish currently boots as follows:

1. `Hagfish.efi` is loaded over PXE by UEFI, and is executed at a runtime-allocated address, with translation (MMU) and caching enabled.
2. Hagfish queries EFI for the PXE protocol instance used to load it, and squirrels away the current network configuration.
3. Hagfish loads the file `hagfish.A.B.C.D.cfg` from the TFTP server root (where `A.B.C.D` is the IP address on the interface that ran PXE).
4. Hagfish parses its configuration, which is essentially a GRUB menu.lst, and loads the kernel image and any additional modules specified therein. All ELF images are loaded into page-aligned regions of type `EfiBarrelfishELFData`.
5. Hagfish queries UEFI for the system memory map, then allocates and initialises the inital page tables for the CPU driver (mapping all occupied physical addresses, within the `TTBR1` window, see Section 3.1.3). The frames holding these tables are marked with the EFI memory type `EfiBarrelfishBootPagetable`, allocated from the OS-specific range (`0x80000000`–`0x8fffffff`). All memory allocated by Hagfish on behalf of the CPU driver is page-aligned, and tagged with an OS-specific type, to allow EFI and Hagfish regions to be safely reclaimed.
6. Hagfish builds a Multiboot 2 information structure, containing as much information as it can get from EFI, including:
    - ACPI 1.0 and 2.0 tables.
    - The EFI memory map (including Hagfish's custom-tagged regions).
    - Network configuration (the saved DHCP ack packet).
    - The kernel command line.
    - All loaded modules.
    - The kernel's ELF section headers.

7. Hagfish allocates a page-aligned kernel stack (type `EfiBarrelfishCPUDriverStack`), of the size specified in the configuration.
8. Hagfish terminates EFI boot services (calls `ExitBootServices`), activates the CPU driver page table, switches to the kernel stack, and jumps into the relocated CPU driver image.

4.1.4 Post-Boot state

When the CPU driver on the boot core begins executing, it can assume the following:

- The MMU is configured with all RAM and I/O regions mapped via `TTBR1`.
- The CPU driver's code and data are both fully relocated into one or more distinct 4kiB-aligned regions.
- The stack pointer is at the top of a distinct 4kiB-aligned region of at least the requested size.
- The first argument register holds the Multiboot 2 magic value.
- The second holds a pointer to a Multiboot 2 information structure, in its own distinct 4kiB-aligned region.
- The console device is configured.
- Only one core is enabled.
- The Multiboot structure contains at least:
  - The final EFI memory map, with all areas allocated by Hagfish to hold data passed to the CPU driver marked with OS-specific types, all of which refer to non-overlapping 4k-aligned regions:

    `EfiBarrelfishCPUDriver` The currently-executing CPU driver's text and data segments.

    `EfiBarrelfishCPUDriverStack` The CPU driver's stack.

    `EfiBarrelfishMultibootData` The Multiboot structure.

    `EfiBarrelfishELFData` The unrelocated ELF image for a boot-time module (including that for the CPU driver itself), as loaded over TFTP.

    `EfiBarrelfishBootPageTable` The currently-active page tables.
  - The CPU driver (kernel) command line.
  - A copy of the last DHCP Ack packet.
  - A copy of the section headers from the CPU driver's ELF image.
  - Module descriptions for the CPU driver and all other boot modules.
  - If UEFI provided an ACPI root table, the Multiboot structure contains a pointer to it.

4.1.5 Configuration

Hagfish configures itself by loading a file whose path is generated from its assigned IP address. Thus if your development machine receives the address 192.168.1.100, Hagfish will load the file `hagfish.192.168.1.100.cfg` from the same TFTP server used to load it. The format is intended to be as close as practical to that of an old-style GRUB menu.lst file. The example configuration in Figure 4.1 loads `/armv8/sbin/cpu_apm88xxxx` as the CPU driver, with arguments `loglevel=3`, and an 8192B (2-page) stack.

```
kernel /armv8/sbin/cpu_apm88xxxx loglevel=3
stack 8192
module /armv8/sbin/cpu_apm88xxxx
module /armv8/sbin/init

# Domains spawned by init
module /armv8/sbin/mem_serv
module /armv8/sbin/monitor

# Special boot time domains spawned by monitor
module /armv8/sbin/chips boot
module /armv8/sbin/ramfsd boot
module /armv8/sbin/skb boot
module /armv8/sbin/kaluga boot
module /armv8/sbin/spawnd boot bootarm=0
module /armv8/sbin/startd boot

# General user domains
module /armv8/sbin/serial auto portbase=2
module /armv8/sbin/fish nospawn
module /armv8/sbin/angler serial0.terminal xterm

module /armv8/sbin/memtest

module /armv8/sbin/corectrl auto
module /armv8/sbin/usb_manager auto
module /armv8/sbin/usb_keyboard auto
module /armv8/sbin/sdma auto
```

Figure 4.1: Hagfish configuration file

4.1.6 Booting with Hagfish in QEMU

When booting a QEMU image for 64-bit ARM, a number of options are available (see `make help-boot`). Building a boot image for QEMU with ARMv8 will typically result in a file in the build directory called armv8$_{<}core_{t}ype >_{q}$ $emu_{i}mage.Thisisadiskimagewhichcanbereadby Hagfish through EFI calls.$

Booting this with a boot target from make will run the following:

```
  srcdir/tools/qemu-wrapper.sh \\
     --image armv8_<core_type>_qemu_image \\
     --arch armv8 \\
     --bios ../git/barrelfish/tools/hagfish/QEMU_EFI.fd
```

This wrapper script is complex, but reasonably well documented (use '--help'). It will invoke QEMU as follows:

```
  qemu-system-aarch64 \\
  -m 1024 \\
  -cpu cortex-a57 \\
  -M virt \\
  -d guest_errors \\
  -M gic_version=3 \\
  -smp 1 \\
  -bios ../git/barrelfish/tools/hagfish/QEMU_EFI.fd \\
  -device virtio-blk-device,drive=image \\
  -drive if=none,id=image,file=armv8_<core_type>_qemu_image,format=raw \\
  -nographic
```

Note that for this script to work, you need to have mtools (the MS-DOS file system manipulation tools) installed, since they are used to prepare the armv8$_{<}core_type >_q$ $emu_image file$.

More specifically, the armv8$_{<}core_type >_q$ $emu_image file is generated by$tools/harness/efiimage.py.$This creates a$

4.2 Booting from U-Boot

Where a full UEFI environment is not available, it is possible to boot Barrelfish from U-Boot DENX Software Engineering [2017]. We boot Barrelfish from U-Boot using U-Boot's limited EFI support: a build-time tool (armv8_bootimage builds a single binary which only requires the minimal EFI environment provided by U-Boot. This binary contains a loader (efi_loader) which sets up the rest of the image as a multiboot image in memory before starting the CPU driver.

4.2.1 Booting in QEMU with U-Boot

A ''platform'' target like QEMU_UBoot which build such an image for QEMU, and the qemu-wrapper.sh script can be invoked to use U-Boot instead of Hagfish:

```
1  srcdir/tools/qemu-wrapper.sh \\
2  --image armv8_a57_qemu_image.efi \\
3  --arch armv8 \\
4  --uboot-img srcdir/tools/qemu-armv8-uboot.bin
```

This invoked QEMU as follows:

```
1  qemu-system-aarch64 \\
2  -m 1024 \\
3  -cpu cortex-a57 \\
4  -M virt \\
5  -d guest_errors \\
6  -M gic_version=3 \\
7  -smp 1 \\
8  -bios srcdir/tools/qemu-armv8-uboot.bin \\
9  -device loader,addr=0x50000000,file=armv8_a57_qemu_image.efi \\
10 -nographic
```

As you can see, the UBoot binary is given as the BIOS, and the minimal EFI image with the complete set of multiboot modules compiled in is pre-loaded into memory when QEMU starts.

Chapter 5

Technical Observations

5.1 User-Space Threading

```
clrex
/* Restore CPSR */
ldr r0, [r1], #4
msr cpsr, r0
/* Restore registers */
ldmia r1, {r0-r15}
```

```
/* Restore PSTATE, load resume
 * address into x18 */
ldp x18, x2, [x1, #(PC_REG * 8)]
/* Set only NZCV. */
and x2, x2, #0xf0000000
msr nzcv, x2
/* Restore the stack pointer and x30. */
ldp x30, x2, [x1, #(30 * 8)]
mov sp, x2
/* Restore everything else. */
ldp x28, x29, [x1, #(28 * 8)]
ldp x26, x27, [x1, #(26 * 8)]
ldp x24, x25, [x1, #(24 * 8)]
ldp x22, x23, [x1, #(22 * 8)]
ldp x20, x21, [x1, #(20 * 8)]
/* n.b. don't reload x18 */
ldr      x19, [x1, #(19 * 8)]
ldp x16, x17, [x1, #(16 * 8)]
ldp x14, x15, [x1, #(14 * 8)]
ldp x12, x13, [x1, #(12 * 8)]
ldp x10, x11, [x1, #(10 * 8)]
ldp  x8,  x9, [x1, #( 8 * 8)]
ldp  x6,  x7, [x1, #( 6 * 8)]
ldp  x4,  x5, [x1, #( 4 * 8)]
ldp  x2,  x3, [x1, #( 2 * 8)]
/* n.b. this clobbers x0&x1 */
ldp  x0,  x1, [x1, #( 0 * 8)]
/* Return to the thread. */
br x18
```

Figure 5.1: `disp_resume_context` on ARMv7 (left) and ARMv8 (right)

The ARMv8 architecture is in some ways an improvement, and in other ways problematic, for the sort of user-level threading implemented in Barrelfish, via *scheduler activations*. Under this scheme, the kernel (in Barrelfish terms, the *CPU driver*), does not schedule threads directly, but instead exposes all scheduling-relevant events via *upcalls* to predefined user-level handlers (in Barrelfish, the *dispatcher*), which then implements thread scheduling (or something else entirely), as it sees fit. This differs from the behaviour of a system

such as UNIX, which only ever restores a user-level execution context simultaneously with dropping from a privileged to an unprivileged execution level.

Processor architectures are, understandably, designed with common software in mind. Thus, the primitives available for restoring an execution context i.e. register state are often tied closely to those for changing privilege level. A common design (which ARMv8 also implements) is the *exception return*, where privileged code can atomically drop its privilege, and jump to a user-level execution address. In ARMv8, the eret instruction atomically updates the program state (PSTATE, most importantly the privilege level bits), and branches to the address held in the *exception link register*, elr.

In implementing user-level threading, we're not concerned with privilege levels, but the lack of some equivalent of elr is frustrating. Not only does eret provide an atomic update of the program counter and the program state, it does so without modifying any general-purpose register. Replicating this behaviour at EL0, where eret is unavailable is problematic. ARMv8 differs from ARMv7, in that the program counter can no longer be the target of a load instruction, but can only be loaded via a general-purpose register.

Specifically, the only PC-modifying instructions (other than eret) are PC-relative branches (which are useless in this scenario) and branch-to-register (of which br, blr and ret are all special encodings). Since ARMv8 has also removed the ldm (load multiple) instruction, there is no way to load the program counter with an arbitrary value (the thread's restart address), without overwriting one of the general-purpose registers. We cannot restore the thread's register value *before* we branch to it, as we'd overwrite the return address, and we obviously can't do so afterwards, as the thread likely has no idea that it's been interrupted. The only alternative is to trampoline through kernel mode in order to use eret (which would eliminate the speed benefit of user-level threading), or to reserve a general-purpose register for use by the dispatcher. Neither option is appealing, but we went with the second option, reserving x18, reasoning that with 31 general-purpose registers available, the loss of one isn't a huge penalty. Register x18 is explicitly marked as the *platform register* in the AArch64 ABI [ARM, 2013], for such a purpose.

Future revisions of the ARM architecture could prevent this issue in a number of ways: allowing the use of eret at EL0 or providing an equivalent functionality (specifically a non-general-purpose register such as elr, that doesn't need to be restored); or alternatively, adding indirect jumps (load to PC) back to the instruction set.

Figure 5.1 compares the user-level thread resume code for the Barrelfish dispatcher (function disp_resume) for ARMv7 and ARMv8 side-by-side. The effect of removing the load-multiple instructions, and direct-to-SP loads, on code density is clearly visible: everything on lines 8--29 for ARMv8 corresponds to the single ldmia instruction on lines 9 for ARMv7 --- one instruction is now 18, on the thread-switch critical path! Note also, on line 17, that the ARMv8 code does not restore the thread's r18, but instead uses it to hold the branch address for use on line 29. The only improvement on ARMv8 is that the clrex (clear exclusive monitor) instruction is no longer required, as the monitor is cleared on returning from the kernel. Note also that the usual method to efficiently load multiple registers, using 16-word SIMD (NEON) loads, isn't available, as there's no guarantee that the SIMD extensions are enabled on this dispatcher, and we cannot handle a fault in this code.

5.2 Trap Handling

Figure 5.2 shows the CPU driver exception stub, for a synchronous abort from EL0. This exception class includes system calls, breakpoints, and page faults on both code and data. The effect of the loss of store multiple instructions is again visible, for example on lines 27--32. Although not as severe as in the case of the user-level thread restore in Section 5.1,

```
el0_aarch64_sync:
    msr daifset, #3 /* IRQ and FIQ masked, Debug and Abort enabled. */

    stp x11, x12, [sp, #-(2 * 8)]!
    stp x9,  x10, [sp, #-(2 * 8)]!

    mrs x10, tpidr_el1
    mrs x9, elr_el1

    ldp x11, x12, [x10, #OFFSETOF_DISP_CRIT_PC_LOW]
    cmp x11, x9
    ccmp x12, x9, #0, ls
    ldr w11, [x10, #OFFSETOF_DISP_DISABLED]
    ccmp x11, xzr, #0, ls
    /* NE <-> (low <= PC && PC < high) || disabled != 0 */

    mrs x11, esr_el1  /* Exception Syndrome Register */
    lsr x11, x11, #26 /* Exception Class field is bits [31:26] */

    b.ne el0_sync_disabled

    add x10, x10, #OFFSETOF_DISP_ENABLED_AREA

save_syscall_context:
    str x7,       [x10, #(7 * 8)]

    stp x19, x20, [x10, #(19 * 8)]
    stp x21, x22, [x10, #(21 * 8)]
    stp x23, x24, [x10, #(23 * 8)]
    stp x25, x26, [x10, #(25 * 8)]
    stp x27, x28, [x10, #(27 * 8)]
    stp x29, x30, [x10, #(29 * 8)] /* FP & LR */

    mrs x20, sp_el0
    stp x20, x9, [x10, #(31 * 8)]

    mrs x19, spsr_el1
    str x19, [x10, #(33 * 8)]

    cmp x11, #0x15 /* SVC or HVC from AArch64 EL0 */
    b.ne el0_abort_enabled

    add sp, sp, #(4 * 8)

    mov x7, x10

    b sys_syscall
```

Figure 5.2: BF/ARMv8 synchronous exception handler

the extra instructions required do constrain us somewhat, as each trap handler is constrained to 128 bytes, or 32 instructions, before branching to another code block.

We were able to squeeze the necessary code into the space available, including the optimised test for a disabled dispatcher at lines 10--14, but only by splitting the page fault handler (el0_abort_enabled) into a separate subroutine, incurring an unnecessary branch. A more significant annoyance is that system calls (svc and hvc) are routed to the same exception vector as page faults (aborts). The effect of this is that we are forced to spill registers to the stack (x9--x12 on lines 4--5), even on the system call fast path, as we need at least one register to check the exception syndrome (esr_el1) to distinguish aborts (where we must preserve all registers) from system calls (where we could immediately begin using the caller-saved registers). Note that the code on lines 27--32 only needs to stack the callee-saved registers, and leaves the system call arguments in x0--x7, to be read as required by sys_syscall (in C).

This sort of mismatch between the exception-handling interface of the CPU architecture, and what is required for really high-performance systems code is unfortunately extremely common. Unnecessary overheads, such as the additional stacked registers here hurt the performance of highly-componentised systems, such as Barrelfish, which rely on frequently crossing protection domains.

The relatively well-compressed boolean arithmetic on lines 10--14 demonstrates that, even with the loss of ARM's fully-conditional instructions, the conditional compares which remain are still relatively powerful.

5.3 Cache Coherence

One aspect of the ARM architecture that is of particular interest for the Barrelfish project, but which we have not yet explored in depth, is the configurable cache coherency and fine-grained cache management operations available. Any virtual mapping on a recent ARM architecture, including both ARMv7 and ARMv8, can be tagged with various cacheability properties: inner (L1), outer (L2+, usually), write-back or write-through. Combined with the explicit flush operations at cache-line granularity, able to target either PoU (point of unification, where data and instruction caches merge) or PoC (point of coherency, typically RAM), a multi-core, multi-socket ARMv8 system would make a very interesting testbed for investigating efficient cache management and communication primitives for future partially-coherent architectures. Indeed, the latest revision of the ARMv8 specification, ARMv8.2, introduced flush to PoP, or *point of persistence* --- perhaps in response to interest from well-known systems integration firms investigating large persistent memories.

The design presented in this report is intended to expose as much control over the caching hierarchy as possible to user-level code, to provide a platform for future research.

References

ARM. Procedure call standard for the ARM 64-bit architecture (AArch64). Technical Report ARM-IHI-0055B, May 2013. URL http://infocenter.arm.com/help/topic/com.arm.doc.ihi0055b/IHI0055B_aapcs64.pdf.

ARM. Server base system architecture. Technical Report ARM-DEN-0029, February 2016. URL http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0029/index.html.

DENX Software Engineering. Das U-Boot -- the Universal Boot Loader. https://www.denx.de/wiki/U-Boot/, April 2017.

Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, Broomfield, Colorado, USA, October 2014.

Barrelfish Project. Barrelfish on ARMv7. Barrelfish Technical Note 017, Systems Group, ETH Zurich, December 2013.

UEFI. Advanced configuration and power interface specification. Technical report, January 2016a. URL http://www.uefi.org/sites/default/files/resources/ACPI_6_1.pdf.

UEFI. Unified extensible firmware interface specification. Technical report, January 2016b. URL http://www.uefi.org/sites/default/files/resources/UEFI%20Spec%202_6.pdf.