

How FreeBSD Boots: a soft-core MIPS perspective

Brooks Davis, Robert Norton, Jonathan Woodruff, Robert N. M. Watson

Abstract

We have implemented an FPGA soft-core, multithreaded, 64-bit MIPS R4000-style CPU called BERI to support research on the hardware/software interface. We have ported FreeBSD to this platform including support for multithreaded and soon multicore CPUs. This paper describes the process by which a BERI system boots from CPU startup through the boot loaders, hand off to the kernel, and enabling secondary CPU threads. Historically, the process of booting FreeBSD has been documented from a user perspective or at a fairly high level. This paper aims to improve the documentation of the low level boot process for developers aiming to port FreeBSD to new targets.

1. Introduction

From its modest origins as a fork of 386BSD targeting Intel i386 class CPUs, FreeBSD has been ported to a range of architectures including DEC Alpha¹, AMD x86_64 (aka amd64), ARM, Intel IA64, MIPS, PC98, PowerPC, and Sparc64. While the x86 and Alpha are fairly homogeneous targets with mechanics for detecting and adapting to specific board and peripheral configurations, embedded systems platforms like ARM, MIPS, and PowerPC are much more diverse. Porting to a new MIPS board often requires adding support for a new System on Chip (SoC) or CPU type with different interrupt controllers, buses, and peripherals. Even if the CPU is supported, boot loaders and associated kernel calling conventions differ significantly between boards.

We have ported FreeBSD/MIPS to BERI, an open-source MIPS R4000-style[1] FPGA-based soft-core processor that we have developed. This required a range of work including boot loader support, platform startup code, a suite of device drivers (including the PIC), but also adapting FreeBSD's existing FDT support to FreeBSD/MIPS. We currently run FreeBSD/BERI under simulation, on an Altera Stratix IV FPGA on a Terasic DE4 FPGA board, and on an Xilinx Virtex-5 FPGA on the NetFPGA-10G platform. The majority of our peripheral work has been on simulation and the DE4 platform. FreeBSD BERI CPU support is derived from the MALTA port with some inspiration from the sbyte port.

Based on our experiences bringing up FreeBSD on BERI we have documented the way we boot FreeBSD from the firmware embedded in the CPU to userspace to provide a new view on the boot process. FreeBSD is generally very well documented between project documentation and books like the Design and Implementation of the FreeBSD Operating System [3], but detailed documentation of the boot process has remained a gap. We believe this paper will help porters gain a high level understanding of the boot process and go allow interested users to understand the overall process without the need to create a new port.

¹Removed in 2006.



Figure 1: BERIpad with application launcher

The rest of this paper narrates the boot process with a special focus on the places customization was required for BERI. We begin by describing the BERI platform (Section 2), and then in detail documents the kernel architecture-specific boot process for FreeBSD on BERI: boot loader (Section 3) and kernel boot process (Section 4). In the interest of brevity many aspects of boot are skipped and most that are not platform or port-specific are ignored. Some platform-specific components such as the MIPS pmap are not covered. The goal is to provide a guide to those pieces someone porting to a new, but relatively conventional MIPS CPU would need to fill in. Porters interested in less conventional CPUs will probably want to examine the NLM and RMI ports in `mips/nlm` and `mips/rmi` for examples requiring more extensive modifications.

2. The BERIpad platform

We have developed BERI as a platform to enable experiments on the hardware-software interface such as our ongoing work on hardware supported capabilities in the CHERI CPU[5]. Our primary hardware target has been a tablet based on the Terasic DE4 FPGA board with a Terasic MTL touch screen and integrated battery pack. The design for the tablet has been released as open source at <http://beri-cpu.org/>. The CPU design will be released in the near future. The modifications to FreeBSD—except for MP—support have been merged to FreeBSD 10.0. The tablet and the internal architecture of BERI are described in detail in the paper *The BERIpad Tablet* [2] The following excerpt provides a brief overview of BERI and the drivers we have developed.

The Bluespec Extensible RISC Implementation (BERI) is currently an in-order core with a 6-stage pipeline which implements the 64-bit MIPS

instruction set used in the classic MIPS R4000. Some 32-bit compatibility features are missing and floating point support is experimental. Achievable clock speed is above 100MHz on the Altera Stratix IV and average cycles per instruction is close to 1.2 when booting the FreeBSD operating system. In summary, the high-level design and performance of BERI is comparable to the MIPS R4000 design of 1991, though the design tends toward extensibility and clarity over efficiency in the micro-architecture.

...

We developed device drivers for three Altera IP cores: the JTAG UART (altera jtag uart), triple-speed MAC (atse), and SD Card (altera sdcard), which implement low-level console/tty, Ethernet interface, and block storage classes. In addition, we have implemented a generic driver for Avalon-attached devices (avgen), which allows memory mapping of arbitrary bus-attached devices without interrupt sources, such as the DE4 LED block, BERI configuration ROM, and DE4 fan and temperature control block.

Finally, we have developed a device driver for the Terasic multitouch display (terasic mtl), which implements a memory-mapped pixel buffer, system console interface for the text frame buffer, and memory-mapped touchscreen input FIFO. Using this driver, UNIX can present a terminal interface, but applications can also overlay graphics and accept touch input.

In addition to the drivers described above, made extensive modifications to the exiting `cfi(4)` (Common Flash Interface) driver to fully support Intel NOR flash and improve write performance.

2.1. Flat Device Tree

Most aspects of BERI board configuration is described in a Flat Device Trees (FDT) which are commonly used on PowerPC and ARM-based systems [4]. Currently a Device Tree Blob (DTB) is built into each FreeBSD kernel and describes a specific hardware configuration. Each DTB is built from a device tree syntax (DTS) file by the device tree compiler² before being embedded in the kernel. Figure 2 excerpts the DTS file `boot/fdt/dts/beripad-de4.dts` and includes the BERI CPU, 1GB DRAM, programmable interrupt controller (PIC), hardware serial port, JTAG UART, SD card reader, flash partition table, gigabit Ethernet, and touchscreen.

3. The early boot sequence

The common FreeBSD boot sequence begins with CPU firmware arranging to run the FreeBSD `boot2` second-stage boot loader which in turn loads `/boot/loader` which loads

```

model = "SRI/Cambridge BeriPad (DE4)";
compatible = "sri-cambridge,beripad-de4";
cpus {
    cpu@0 {
        device-type = "cpu";
        compatible = "sri-cambridge,beri";
    };
};
soc {
    memory {
        device_type = "memory";
        reg = <0x0 0x40000000>;
    };
    beripic: beripic@7f804000 {
        compatible = "sri-cambridge,beri-pic";
        interrupt-controller;
        reg = <0x7f804000 0x400 0x7f806000 0x10
            0x7f806080 0x10 0x7f806100 0x10>;
    }
    serial@7f002100 {
        compatible = "ns16550";
        reg = <0x7f002100 0x20>;
    };
    serial@7f000000 {
        compatible = "altera,jtag_uart-11_0";
        reg = <0x7f000000 0x40>;
    };
    sdcard@7f008000 {
        compatible = "altera,sdcard_11_2011";
        reg = <0x7f008000 0x400>;
    };
    flash@74000000 {
        partition@20000 {
            reg = <0x20000 0xc0000>;
            label = "fpga0";
        };
        partition@1820000 {
            reg = <0x1820000 0x027c0000>;
            label = "os";
        };
    };
    ethernet@7f007000 {
        compatible = "altera,atse";
        reg = <0x7f007000 0x400 0x7f007500 0x8
            0x7f007520 0x20 0x7f007400 0x8
            0x7f007420 0x20>;
    };
    touchscreen@70400000 {
        compatible = "sri-cambridge,mtl";
        reg = <0x70400000 0x1000
            0x70000000 0x177000 0x70177000 0x2000>;
    };
};

```

Figure 2: Excerpt from Flat Device Tree (FDT) description of the DE4-based BERI tablet.

²`dtc(1)`

the kernel and kernel modules. Finally the kernel boots which is described in Section 4.

3.1. Miniboot

At power on or after reset, the CPU sets the program counter of at least one thread to the address of a valid program. From the programmer perspective the process by which this occurs is essentially magic and of no particular importance. Typically the start address is some form of read-only or flash upgradable firmware that allows for early CPU setup and may handle details such as resetting cache state or pausing threads other than the primary thread until the operating system is ready to handle them. In many systems, this firmware is responsible for working around CPU bugs.

On BERI this code is known as miniboot for physical hardware and simboot for simulation. Miniboot is compiled with the CPU as a read-only BRAM. It is responsible for settings registers to initial values, setting up an initial stack, initializing the cache by invalidating the contents, setting up a spin table for MP boot, running code to initialize the HDMI output port on the DE4 tablet, and loading a kernel from flash or waiting for the next bit of code to be loaded by the debug unit and executing that. With BERI we are fortunate to not need to work around CPU bugs in firmware since we can simply fix the hardware.

Miniboot’s kernel loading and boot behavior is controlled by two DIP switches on the DE4. If DIP0 is off or miniboot with compiled with `-DALWAYS_WAIT` then we spin in a loop waiting for the general-purpose register `t1` to be set to 0 using JTAG. This allows the user to control when the board starts and given them an opportunity to load a kernel directly to DRAM before boot proceeds. DIP1 controls the relocation of a kernel from flash. If the DIP switch is set, the kernel is loaded from a flash at offset of `0x2000000` to `0x100000` in DRAM. Otherwise, the user is responsible for loading a kernel to DRAM by some other method. Currently supported mechanisms are described in the BERI Software Reference [7].

The kernel loading functionality occurs only on hardware thread 0. In other hardware threads, miniboot skips this step and enter a loop waiting for the operating system to send them a kernel entry point via the spin-table. Multithread and multicore boot is discussed in more detail in section 4.3.

Before miniboot enters the kernel it clears most registers and sets `a0` to `argc`, `a1` to `argv`, `a2` to `env`, and `3` to the size of system memory. In practice `argc` is 0 and `argv` and `env` are `NULL`. It then assumes that an ELF64 object is located at `0x100000`, loads the entry point from the ELF header, and jumps to it.

We intend that miniboot be minimal, but sufficiently flexible support debugging of various boot layouts as well as loading alternative code such as self contained binaries. This allows maximum flexibility for software developers who may not be equipped to generate new hardware images.

3.2. boot2

On most FreeBSD systems two more boot stages are interposed between the architecture dependent boot code and the

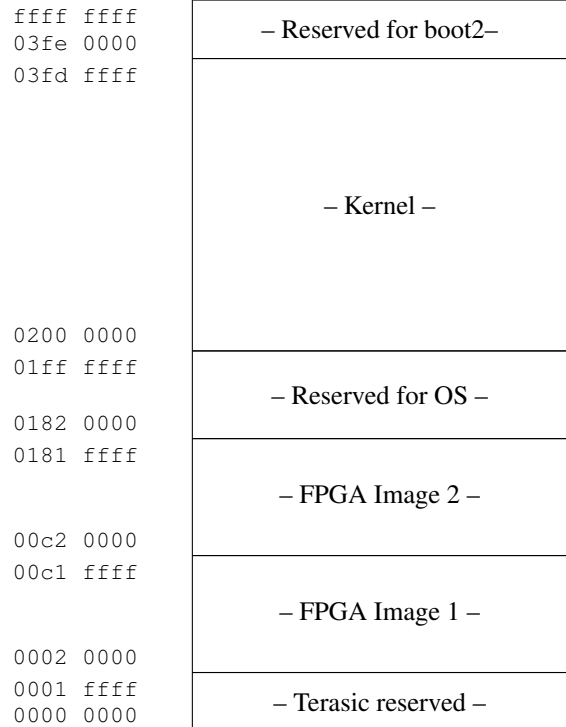


Figure 3: Layout of the DE4 flash

kernel. The first of these is `boot23`, the second stage bootstrap, which has a mechanism for accessing local storage and has code for read-only access to a limited set of file systems (usually one of UFS or ZFS). Its primary job is to load the loader and to pass arguments to it. By default it loads `/boot/loader`, but the user can specify an alternative disk, partition, and path.

We have ported `boot2` to BERI, creating three ‘microdrivers’ allowing JTAG UART console access, and use of CFI or the SD card to load `/boot/loader` or the kernel. These microdrivers substitute for boot device drivers provided by the BIOS on x86 or OpenFirmware on SPARC. It also supports jumping to an instance of `/boot/loader` loaded via JTAG. In our current implementation, `boot2` is linked to execute at `0x100000` and loaded from CFI flash as the kernel currently is allowing it to be used with an unmodified miniboot. In the future, we plan to place a similar version of `boot2` at `0x03fe0000`, a 128K area reserved for its use. This will allow a normal filesystem to be placed in CFI flash from `0x1820000`, which might contain the full boot loader, a kernel, etc. Currently, we use `boot2` to load `/boot/loader` from the SD card, which offers an experience more like conventional desktop/server platforms than a conventional embedded target.

Many versions of `boot2` exist, targeted at different architectures. The version of `boot2` in BERI is derived from the x86 `boot2`, and is hence (marginally) more feature-rich than ones targeted at more space-constrained embedded architectures.

³`boot(8)`



Figure 4: FreeBSD loader boot menu

3.3. loader

The third common boot stage is the `loader(8)`. The loader is in effect a small kernel whose main job is to set up the environment for the kernel and load the kernel and any configured modules from the disk or network. The loader contains a Forth interpreter based on FICL⁴. This interpreter is used to provide the boot menu shown in Figure 4, parses configuration files like `/boot/loader.conf`, and implements functionality like `nextboot(8)`. In order to do this, the loader also contains drivers to access platform-specific devices and contains implementations of UFS and ZFS with read and limited write support. On x86 systems that means BIOS disk access and with the `pxeloder` network access via PXE. On BERI this currently includes a basic driver for access to the CFI flash found on the DE4.

We have ported the loader to FreeBSD/MIPS and share the SD card and CFI microdrivers with `boot2` to allow kernels to be loaded from CFI flash or SD card. We currently load the kernel from the SD card. We hope to eventually add a driver for the onboard Ethernet device to allow us to load kernels from the network.

The loader's transition to the kernel is much the same as `miniboot`. The kernel is loaded to the expected location in the memory, the ELF header is parsed, arguments are loaded into registers, and the loader jumps into the kernel.

3.4. The bootinfo structure

In order to facilitate passing information between `boot2`, `/boot/loader`, and the kernel a pointer to a `bootinfo` structure is between them allowing information such as memory size, boot media type, and the locations of preloaded modules to be shared. In the future we will add support for passing a pointer to the FDT device database that will be embedded in the CPU or stored separately in flash.

4. The path to usermode

This section narrates the interesting parts of the FreeBSD boot process from a MIPS porter's perspective. In the electronic version of this document most of the paths, function names, and symbols are links to appropriate parts of <http://fxr.watson.org> to enable further exploration.

⁴<http://ficl.sourceforge.net>

4.1. Early kernel boot

The FreeBSD MIPS kernel enters at `_start` in the `_locore` function defined in `mips/mips/locore.S`. `_locore` performs some early initialization of the CPO registers, sets up an initial stack and calls the platform-specific startup code in `platform_start`.

On BERI `platform_start` saves the argument list, environment, and pointer to `struct bootinfo` passed by the loader. BERI kernels also support an older boot interface, in which memory size is passed as the fourth argument (direct from `miniboot`). It then calls the common mips function `mips_postboot_fixup` which provides kernel module information for manually loaded kernels and corrects `kernel_kseg0_end` (the first usable address in kernel space) if required. Per CPU storage is then initialized for the boot CPU by `mips_pcpu0_init`. Since BERI uses Flat Device Tree (FDT) to allow us to configure otherwise non-discoverable devices `platform_start` then locates the DTB and initializes FDT. This is the norm for ARM and PowerPC ports, but is currently uncommon on MIPS ports. We expect it to become more popular over time. The `platform_start` function then calls `mips_timer_early_init` to set system timer constants, currently to a hardcoded 100MHz, eventually this will come from FDT. The console is set up by `cninit` and some debugging information is printed. The number of pages of real memory is stored in the global variable `realmem`⁵. The BERI-specific `mips_init`⁶ function is then called to do the bulk of remaining early setup.

BERI's `mips_init` is fairly typical. First, memory related parameters are configured including laying out the physical memory range and setting a number of automatically tuned parameters in the general functions `init_param1` and `init_param2`. The MIPS function `mips_cpu_init` performs some optional per-platform setup (nothing on BERI), identifies the CPU, configures the cache, and clears the TLB. The MIPS version of `pmmap_bootstrap` is called to initialize the `pmmap`. Thread 0 is instantiated by `mips_proc0_init` which also allocates space for dynamic per CPU variables. Early mutexes including the legacy Giant lock are initialized in `mutex_init` and the debugger is initialized in `kdb_init`. If so configured the kernel may now drop into the debugger or, much more commonly, return and continue booting.

Finally `mips_timer_init_params` is called to finish setting up the timer infrastructure before `platform_start` returns to `_locore`. `_locore` switches to the now configured `thread0` stack and calls `mi_startup` never to return.

4.2. Calling all SYSINITs

The job of `mi_startup` is to initialize all the kernel's subsystems in the right order. Historically `mi_startup` was called `main` and the order of initialization was hard coded.

⁵The `btoc` macro converts bytes to clicks which in FreeBSD are single pages. Mach allowed multiple pages to be managed as a virtual page.

⁶Most ports have one of these, but it seems to be misnamed as it is not MIPS generic code.

```

static void
print_caddr_t(void *data)
{
    printf("%s", (char *)data);
}
SYSINIT(announce, SI_SUB_COPYRIGHT,
        SI_ORDER_FIRST, print_caddr_t,
        copyright);

```

Figure 5: Implementation of copyright message printing on FreeBSD boot.

This was obviously not scalable so a more dynamic registration mechanism called `SYSINIT(9)` was created. Any code that needs to be run which at startup can use the `SYSINIT` macro to cause a function to be called in a sorted order to boot or on module load. The `sysinit` implementation relies on the ‘linker set’ feature, in which constructors/destructors for kernel subsystems and modules are tagged in the ELF binary so that the kernel linker can find them during boot, module load, module unload, and kernel shutdown.

The implementation of `mi_startup` is simple. It sorts the set of `sysinits` and then runs each in turn marking each done when it is complete. If any modules are loaded by a `sysinit`, it resorts the set and starts from the beginning skipping previous run entries. The end of `mi_startup` contains code to call `swapper`, this code is never reached as the last `sysinit` never return. One implementation detail of note in `mi_startup` is the use of bubble sort to sort the `sysinits` due to the fact that allocators are initialized via `sysinits` and thus not yet available.

Figure 5 shows a simple example of a `sysinit`. In this example `announce` is the name of the individual `sysinit`, `SI_SUB_COPYRIGHT` is the subsystem, `SI_ORDER_FIRST` is the order within the subsystem, `print_caddr_t` is the function to call, and `copyright` is an argument to be passed to the function. A complete list of subsystems and orders within subsystems can be found in `sys/kernel.h`. As of this writing there are more than 80 of them. Most are have little or no port-specific function and thus are beyond the scope of this paper. We will highlight `sysinits` with significant port-specific content.

The first `sysinit` of interest is `SI_SUB_COPYRIGHT`. It does not require porting specifically, but reaching it and seeing the output is a sign of a port nearing completion since it means low level consoles work and the initial boot described above is complete. The MIPS port has some debugging output earlier in boot, but on mature platforms the copyright message is the first output from the kernel. Figure 6 shows the three messages printed at `SI_SUB_COPYRIGHT`.

The next `sysinit` of interest to porters is `SI_SUB_VM`. The MIPS `bus_dma(9)` implementation starts with a set of statically allocated maps to allow it to be used early in boot. The function `mips_dmamap_freelist_init` adds the static maps to the free list at `SI_SUB_VM`. The ARM platform does similar work, but does require `malloc` and thus runs `busdma_init` at `SI_SUB_KMEM` instead.

Further `bus_dma(9)` initialization takes place at

`SI_SUB_LOCK` in the platform-specific, but often identical, `init_bounce_pages` function. It initializes some counters, lists, and the bounce page lock.

All ports call a platform-specific `cpu_startup` function at `SI_SUB_CPU` set up kernel address space and perform some initial buffer setup. Many ports also perform board, SoC, or CPU-specific setup such as initializing integrated USB controllers. Ports typically print details of physical and virtual memory, initialize the kernel virtual address space with `vm_ksubmap_init`, the VFS buffer system with `bufinit`, and the swap buffer list with `vm_pager_bufferinit`. On MIPS the platform-specific `cpu_init_interrupts` is also called to initialize interrupt counters.

Most platforms have their own `sf_buf_init` routine to allocate `sendfile(2)` buffers and initialize related locks. Most of these implementations are identical.

The bus hierarchy is established and device probing is performed at the `SI_SUB_CONFIGURE` stage (aka autoconfiguration). The platform-specific portions of this stage are the `configure_first` function called at `SI_ORDER_FIRST` which attaches the nexus bus to the root of the device tree, `configure` which runs at `SI_ORDER_THIRD` and calls `root_bus_configure` to probe and attach all devices, and `configure_final` which runs at `SI_ORDER_ANY` `cninit_finish` to finish setting up the console with `cninit_finish`, and clear the `cold` flag. On MIPS and some other platforms `configure` also calls `intr_enable` to enable interrupts, A number of console drivers complete their setup with explicit `sysinits` at `SI_SUB_CONFIGURE` and many subsystems like CAM and `acpi(4)` perform their initialization there.

Each platform registers the binary types it supports at `SI_SUB_EXEC`. The primarily consists of registering the expected ELF header values. On a uniprocessor MIPS this is the last platform-specific `sysinit`.

The final `sysinit` is an invocation of the scheduler function at `SI_SUB_RUN_SCHEDULER` which attempts to swap in processes. Since `init(8)` was previously created by `create_init` at `SI_SUB_CREATE_INIT` and made runnable by `kick_init` at `SI_SUB_KTHREAD_INIT` starting the scheduler results in entering userland.

4.3. Multiprocessor Support

Multiprocessor systems follow the same boot process as uniprocessor systems with a few added `sysinits` to enable and start scheduling the other hardware threads. These threads are known as application processors (APs).

The first MP-specific `sysinit` is a call to `mp_setmaxid` at `SI_SUB_TUNABLES` to initialize the `mp_ncpus` and `mp_maxid` variables. The generic `mp_setmaxid` function calls the platform-specific `cpu_mp_setmaxid`. On MIPS `cpu_mp_setmaxid` calls the port-specific `platform_cpu_mask` to fill a `cpuset_t` with a mask of all available cores or threads. BERI’s implementation extracts a list of cores from the DTB and verifies that they support the spin-table enable method. It further verifies that the spin-table entry is properly initialized or the thread is

Copyright (c) 1992-2013 The FreeBSD Project.

Copyright (c) 1979, 1980, 1983, 1986, 1988, 1989, 1991, 1992, 1993, 1994

The Regents of the University of California. All rights reserved.

FreeBSD is a registered trademark of The FreeBSD Foundation.

Figure 6: Copyright and trademark messages in FreeBSD 10

```
struct spin_entry {
    uint64_t    entry_addr;
    uint64_t    a0;
    uint32_t    rsvd1;
    uint32_t    pir;
    uint64_t    rsvd2;
};
```

Figure 7: Definition of a `spin_entry` with explicit padding and the argument variables renamed to match MIPS conventions.

ignored.

The initialization of APs is accomplished by the `mp_start` function called at `SI_SUB_CPU` after `cpu_startup`. If there are multiple CPUs it calls the platform-specific `cpu_mp_start` and upon return prints some information about the CPUs. The MIPS implementation of `cpu_mp_start` iterates through the list of valid CPU IDs as reported by `platform_cpu_mask` and attempts to start each one except it self as determined by `platform_processor_id`⁷ with the platform-specific `start_ap`. The port-specific `platform_start_ap`'s job is to cause the AP to run the platform-specific `mpentry`. When runs successfully, it increments the `mp_naps` variable and `start_ap` waits up to five seconds per AP for this to happen before giving up on it.

A number of mechanisms has been implemented to instruct a CPU to start running a particular piece of code. On BERI we have chosen to implement the spin-table method described in the ePAPR 1.0 specification[4] because it is extremely simple. The spin-table method requires that each AP have an associated `spin_entry` structure located somewhere in the address space and for that address to be recorded in the DTB. The BERI specific definition of `struct spin_entry` can be found in Figure 7. At boot the `entry_addr` member of each AP is initialized to 1 and the AP waits for the LSB to be set to 0 at which time it jumps to the address loaded in `entry_addr` passing `a0` in register `a0`. We implement waiting for `entry_addr` to change with a loop in `miniboot`. In BERI's `platform_cpu_mask` we look up the `spin_entry` associated with the requested AP, set the `pir` member to the CPU id and then assign the address of `mpentry` to the `entry_addr` member.

The MIPS implementation of `mpentry` is assembly in `mips/mips/mpboot.S`. It disables interrupts, sets up a stack, and calls the port-specific `platform_init_ap` to set up the AP before entering the MIPS-specific `smp_init_secondary` to complete per-CPU setup and await the end of the boot process. A typical MIPS implementation of `platform_init_ap` sets up interrupts on

⁷Implemented in `mips/beri/beri_asm.S` on BERI.

the AP and enables the clock and IPI interrupts. On BERI we defer IPI setup until after device probe because our programmable interrupt controller (PIC) is configured as an ordinary device and thus can not be configured until after `SI_SUB_CONFIGURE`.

The MIPS-specific `smp_init_secondary` function initializes the TLB, setups up the cache, and initializes per-CPU areas before incrementing `mp_naps` to let `start_ap` know that it has finished initialization. It then spins waiting for the flag `aps_ready` to be incremented indicating that the boot CPU has reached `SI_SUB_SMP` as described below. On BERI it then calls `platform_init_secondary` to route IPIs to the AP and set up the IPI handler. The AP then sets its thread to the per-CPU idle thread, increment's `smp_cpus`, announces it self on the console, and if it is the last AP to boot, sets `smp_started` to inform `release_aps` that all APs have booted and the `smp_active` flag to inform a few subsystems that we are running with multiple CPUs. Unless it was the last AP to boot it spins waiting for `smp_started` before starting per-CPU event timers and entering the scheduler.

The final platform-specific `sysinit` subsystem is `SI_SUB_SMP` which platform-specific `release_aps` functions are called to enable IPIs on the boot CPU, inform previously initialized APs that they can start operating, and spin until they do so as described above. In the MIPS case this means atomically setting the `aps_ready` flag to 1 and spinning until `smp_started` is non-zero.

4.4. A word on IPIs

In multiprocessor (MP) systems CPUs communicate with each other via Inter-Processor Interrupts (IPIs). A number of IPI mechanisms exist, with FreeBSD MIPS using the simplest model, a per-CPU integer bitmask of pending IPIs and a port-specific mechanism for sending an interrupt, almost always to hardware interrupt 4. This is implemented by the `ipi_send` which is used by the public `ips_all_but_self`, `ipi_selected`, and `ipi_cpu` functions. MIPS IPIs are handled by `mips_ipi_handler` which clears the interrupt with a call to `platform_ipi_clear`, reads the set of pending IPIs, and handles each of them.

On BERI IPIs are implemented using the BERI PIC's soft interrupt sources. IPIs are routed by `beripic_setup_ipi`, sent by `beripic_send_ipi`, and cleared by `beripic_clear_ipi`. These functions are accessed via `kobj(9)` through the `FDT_IC` interface defined in `dev/fdt/fdt_ic_if.m`. The internals of BERI PIC are described in the BERI Hardware Reference[6].

5. Conclusion

Porting FreeBSD to a new CPU, even within a previously supported family, is a significant undertaking. We hope this paper will help prospective porters orient themselves before they begin the process. While we have focused on a MIPS ports, the code structure in other platforms—especially ARM—is quite similar.

5.1. Acknowledgments

We would like to thank our colleagues - especially Jonathan Anderson, David Chisnall, Nirav Dave, Wojciech Koszek, Ben Laurie, A Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Peter G. Neumann, Philip Paeps, Michael Roe, and Bjoern Zeeb.

This work is part of the CTSRD Project that is sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.

References

- [1] J. Heinrich, *MIPS R4000 Microprocessor User's Manual*, 1994, second Edition.
- [2] A. T. Markettos, J. Woodruff, R. N. M. Watson, B. A. Zeeb, B. Davis, and S. W. Moore, "The BERIPad tablet: Open-source construction, CPU, OS and applications," 2013.
- [3] M. K. McKusick and G. V. Neville-Neil, *The Design and Implementation of the FreeBSD Operating System*. Pearson Education, 2004.
- [4] Power.org, *Power.org Standard for Embedded Power Architecture Platform Requirements (ePAPR)*, 2008.
- [5] R. Watson, P. Neumann, J. Woodruff, J. Anderson, R. Anderson, N. Dave, B. Laurie, S. Moore, S. Murdoch, P. Paeps *et al.*, "CHERI: A Research Platform Deconflating Hardware Virtualization and Protection," in *Workshop paper, Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE 2012)*, 2012.
- [6] R. N. M. Watson, D. Chisnall, B. Davis, W. Koszek, S. W. Moore, S. J. Murdoch, R. Norton, and J. Woodruff, "BERI bluespec extensible RISC implementation: Hardware reference," 2014, forthcoming publication.
- [7] R. N. M. Watson, D. Chisnall, B. Davis, W. Koszek, S. W. Moore, S. J. Murdoch, and J. Woodruff, "BERI bluespec extensible RISC implementation: Software reference," 2014, forthcoming publication.