BRING-UP BHYVE ON ESPRESSOBIN BOARD

Andrei-Costin Martin[1], Darius Mihai[2], Maria-Elena Mihailescu[3],
Mihai Carabas[4], Sergiu Weisz[5]

In order to be sustainable, the cloud architectures must have good scalability and throughput. On the processors market, ARM started to show interest for servers and desktop processors. x86 and x86_64 CPU market have a strong opponent, with a lower power consumption and comparable performances. A solution to the scalability and throughput requirements could be a virtualized environment. One example is the FreeBSD's hypervisor, bhyve*, mainly because FreeBSD is often used in server environments, has a strong TCP/IP stack implementation.* bhyve's *implementation for ARM was not accepted in the mainline, due the lack of testing on real hardware. This paper presents the steps taken in order to bring-up the hypervisor on a development board, called ESPRESSObin.*

Keywords: virtualization, ARMv8, bhyve, FreeBSD, EspressoBin, bring-up

1. Introduction

Nowadays, there is a great inclination towards Internet, cloud and streaming. These topics raise the issue of scalability in order to obtain a high network traffic from all over the world. One solution can be virtualization. Using virtual machines, all the servers can be duplicated and deployed everywhere, in a safe environment to accept connections. To manage these virtual machines, a hypervisor is needed. The hypervisor has the role of starting, running and managing the execution of virtual machines. These cloud and streaming markets saw continuous growth over the last decade, alongside the virtualization solutions in order to increase the exchange speed of information. Besides that, virtualization aims to the best hardware resources usage and to increase the throughput. At the same time, ARM started to take an interest for the server and desktop processors market. That means that x86 and x86_64 CPUs will have a strong rival, one with a better energy efficiency and

[1]Master Student, Faculty of Automatic Control and Computer Science, University Politehnica of Bucharest, Romania, e-mail: andrei.martin@stud.acs.upb.ro

[2]Teaching Assistant, Faculty of Automatic Control and Computer Science, University Politehnica of Bucharest, Romania, email: darius.mihai@ubp.ro

[3]Teaching Assistant, Faculty of Automatic Control and Computer Science, University Politehnica of Bucharest, Romania, email: maria.mihailescu@ubp.ro

[4]Associate Professor, Faculty of Automatic Control and Computer Science, University Politehnica of Bucharest, Romania, email: mihai.carabas@ubp.ro

[5]PhD Candidate, Faculty of Automatic Control and Computer Science, University Politehnica of Bucharest, Romania, email: sergiu.weisz@ubp.ro

comparable performances.
This paper presents the steps taken in the ARMv8 bring-up process in order to run a virtual machine using FreeBSD's bhyve, both in the ARMs Foundation Platform emulator[3] and ESPRESSObin[4], an off-the-shelf board developed by Marvell. The reason for choosing FreeBSD's hypervisor is because FreeBSD is very often used as operating system on serves, due to its TCP/IP stack implementation and its stability and, with virtualization, would solve the scalability issue. Regarding the virtualization, an ARM implementation for bhyve [2] is not yet available in the FreeBSD's mainline. There are implementations for ARMv7 and ARMv8 but not yet accepted by the maintainers.
The main goal is to have a fully operational hypervisor, that can run virtual machines in the emulator and on the board. After being able to run the hypervisor, it need to be tested, using FreeBSD, Linux and Windows as guests operating systems. The last step is the hypervisor code redesign, both in user-space and kernel-space, requested by the maintainers, in order to accept the code in the FreeBSD's mainline.

2. State of the art

In this section, we will present information regarding the virtualization, FreeBSD and its hypervisor, bhyve, we present a comparison between ARMv7 and ARMv8 and the components used for deploying FreeBSD on a board.

2.1. FreeBSD and bhyve

FreeBSD [1] is an operating system designed to run on desktops, servers and embedded platforms. Despite the wide range of compatibility, *FreeBSD*'s main target is the server market (e.g. web servers or email servers). BSD operating systems have the advantage of stability, offered by a small number of updates that can damage the system. Besides that, their implementation of the *TCP/IP* [7] stack makes this operation system the best for servers that have a huge number of concurrent connections.

bhyve is a type-two hypervisor, FreeBSD's solution regarding the virtualization problem. It supports a wide range of guest operating systems, such as FreeBSD, Linux or Windows and is available only for FreeBSD on x86 [8] CPU architecture. Its implementation contains three user space utilities, *bhyveload*, *bhyve* and *bhyvectl*, which communicate with the kernel driver, *vmm*, through *libvmmapi* library. These three utilities have the following roles: *bhyveload* that has the role of loading a new guest virtual machine, *bhyve* that runs a virtual machine with a guest operating system, and *bhyvectl* that has the role of controlling *bhyve* virtual machines, by creating, destroying or getting and changing statistics/preferences.

2.2. ARMv7 vs ARMv8

A brief comparison between ARMv8 and ARMv7 can be found in Table 1. We can see that one of the improvements is the increased address space, from 32 bits to 64 bits. There is only one exception, the instruction set is 32 bits in both

ARMv7 and ARMv8. That means that the encoding of instructions is not changed. Based on the registers size, ARMv7 and ARMv8 are also called arm32 (or arm) and arm64.

TABLE 1. Comparison between ARMv7 and ARMv8

	ARMv7	**ARMv8**
General-purpose registers	13 x 32bit	31 x 64bit
Program Counter	32bit	64bit
Stack Pointer	32bit	64bit
Exception Link Register	32bit	64bit
Virtual addressing	Support for 32bit	Support for 64bit
Instruction Set	16/32bit	32bit

This new version of the ARM processor, ARMv8 [5], defines two 'Execution States' [6]: AArch64 and AArch32. AArch64 state uses all registers as 64bit registers, compared to AArch32, which uses them as 32bit registers, to provide backwards compatibility. This is the reason why an ARMv7 application can also work well on ARMv8 CPUs.

Besides Execution States, presented in Subsection 2.**2**, ARMv8 [5] has a total of 4 Execution levels [6]. While the first two levels, **EL0** and **EL1**, represent where the user program and kernel are usually running on any system, the third level, **EL2**, will be the place where the virtual machine is running. As a requirement, Kernel and Virtualization spaces must communicate with each other. The last level is divided in **Secure State** where the processor can access the entire memory address space and can access all system control resources, and **Non-Secure State** that specifies that the CPU can access only the Non-Secure address space and cannot access the system control resources.

2.**3**. **ESPRESSObin, Device Trees and U-Boot**

In the bring-up phase, this project used an off-the-shelf board, called ESPRESSObin v5. We chose this board because it has an ARMv8 Cortex-A53 CPU, the same CPU as one version of the ARMs Foundation Platform emulator. The following lists presents the board components **SATA HDD data cable** – the data transfer port for a SATA peripheral; **SATA HDD power** – the power supply for a SATA connected peripheral; textbfMicro USB port (J5) – is a serial (UART) connection; **10-pin ARM JTAG** – 10 JTAG pins, dedicated to debugging; **12V DC Jack** – is the power supply port; **GPIO** – General Purpose Input/Output pins, **USB 2.0 and 3.0 ports** – Two USB ports for peripherals; **GbE LAN ports** – Two LAN Ethernet ports; **WAN port** – A WAN port; **J3 connector** – Three jumpers representing the ESPRESSObin boot mode; **Reset switch** – A button which resets the board.

Device Trees[9], in the embedded systems, are one of the most important things, describing the hardware specification of a board. Due to the fact that the components (e.g. memory, CPU) are glued to the motherboard, a board specification is static. It is represented by a human-readable file, containing all the specifications of a board, that will go through a compilation process. The compiled binary, called *Device Tree Blob* (DTB), will be loaded by the associated board, via a bootloader.

U-Boot[1] is an open source bootloader, designed for embedded systems. It is versatile, being able to accept *FAT32*, *ext4*, even *UFS* (in newer versions) filesystem formats for different types of storage, such as *usb* or *mmc* for MicroSD cards. It also lets the user configure environment variables and command, for example, a variable that contains the path to the *Device Tree Blob*.

For this project, we used a MicroSD card, formatted as *FAT32*, to store the operating system. This MicroSD card contained the FreeBSD operating system, alongside the *DTB*. In the compile process, the Marvell's U-Boot repository was used, found on the wiki page [4], in the *Build from source - Bootloader* section. When the U-Boot binary has to be made as a valid payload for ARM's Trusted Firmware (ATF), some values are used, presented in Table 2 . The values are used to set three important variables at the *make* command, describing the board specification (CPU and memory frequencies, memory topology and the boot device).

TABLE 2. CPU and DDR presets

Value	**CPU frequency (MHz)**	**DDR frequency (MHz)**
CPU_600_DDR_600	600	600
CPU_800_DDR_800	800	800
CPU_1000_DDR_800	1000	800

3. **Related work**

bhyve for ARM started as a Summer of Code [10] project, for the ARMv7 architecture. The final goal was to run a FreeBSD guest operating system, on a development board, called CubieBoard2. The kernel-space virtualization driver was ported from x86_64 to arm64, and the context switch between the host and the hypervisor mode, the page-tables and the virtual generic interrupt controller (vGIC) were also implemented. Regarding the guest operating system, this implementation was able to start a virtual machine on the ARM Foundation Platform emulator, but not on the development board. When the virtual machine booted, but ended up with "Spurious interrupts", as described on the project page [10].

After the summer, the project moved in the University POLITEHNICA of Bucharest with the implementation of VirtIO [11]. The implementation relied on the one for *amd64*, but with two major differences: ARM does not have support

[1] https://www.denx.de/wiki/U-Boot

for the PCI bus, so the memory mapped I/O (MMIO) support had to be added, and that for *arm64*, the interrupt mechanism was Generic Interrupt Controller (GIC), not Local Advanced Programmable Interrupt Controller (LAPIC).

bhyve for ARMv8 project also started and implemented the kernel-space and user-space virtualization model, having the x86 and ARMv7 implementations as starting points and models. When the project stopped, a guest operating system was able to boot in an emulated environment, using the ARM Foundation Platform, and was tested on CubieBoard2, but did not work.

4. ARMv8 bring-up Infrastructure

A full description of the hardware modules is presented in Figure 1. In the left side is the computer, which contains all the source code and the utilities needed and in the right side is the ESPRESSObin board with its most relevant ports, *J5* serial port, the *LAN* port and the *MicroSD* slot.

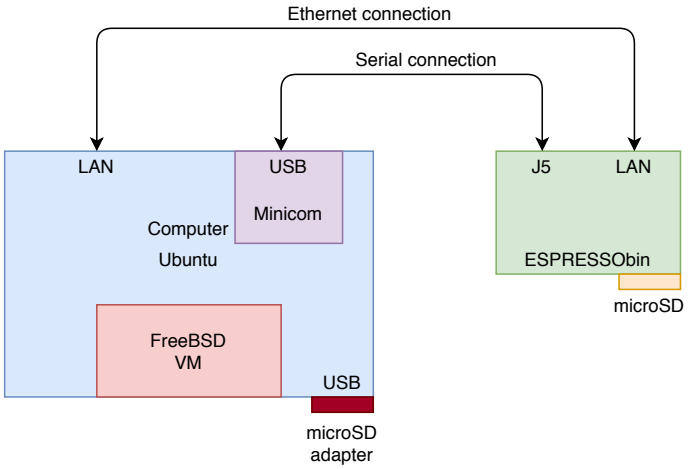

FIG. 1. Hardware modules

The communication between the computer and the port is done in three ways: through a *Serial connection* to send commands to the board, such as setting variables in the bootloader or run instructions, through an *Ethernet connection* (a TFTP server is used so that the board can fetch certain files), and though a *microSD card* used for the FreeBSD root filesystem that is copied into the card from the computer and used to start the operating system on the board.

To cross-compile the ARM implementation, we need a FreeBSD environment, mainly because of the Makefile system, which differs from the Linux Makefile. A Linux envirnoment is needed as well to start the ARM Foundation Platform emulator [3] with the FreeBSD disk image, generated in the FreeBSD environment.

A configuration file must be present when compiling the hypervisor's kernel to specify various parameters. One of the most important setting is `kernconf`,

which dictates what kernel to compile. *FOUNDATION* means that is used to compile a custom kernel, especially designed for the host operating system in the ARM Foundation Platform emulator. For the guest operating system on the emulator the `kernconf` variable should be set as *FOUNDATION_GUEST* and for an ARM board it should be *GENERIC*. In particular for *FOUNDATION* and *FOUNDATION_GUEST* kernels, the device tree is not loaded by the bootloader but is compiled inside the kernel. The `FDT` and `FDT_DTB_STATIC` options enable the *Device Tree* and set it as static and the `FDT_DTS_FILE` command tells the compiler which *Device Tree* to choose. The *FOUNDATION*, *FOUNDATION_GUEST* and *GENERIC* kernel configuration can be found in the FreeBSD repository, *projects-bhyvearm64* branch, in the *sys/arm64/conf* directory.

5. FreeBSD hypervisor bring-up on real hardware

5.1. FOUNDATION and GENERIC build

The FOUNDATION build was used to build the FreeBSD operating system with the ARM Foundation Platform emulator. There are four steps needed to build the operating system and generate the disk image for the emulator. The first step is to compile the host user-space binaries - build almost everything, except the kernel, which will be built in the next two steps. It is necessary to build all the tools before the kernel because this step also generates the *arm64_obj* directory. The second step uses a different configuration file, dedicated for the guest kernel. The *kernconf* variable must be set to *FOUNDATION_GUEST*. The third step is similar with the second, but uses another *kernconf*. Because it builds the host operating system kernel, the custom kernel must be *FOUDNATION*. The fourth step is to generate the disk image, based on the object directory and on the *FOUNDATION* and *FOUNDATION_GUEST* kernels.

The *GENERIC* build dedicated for the ESPRESSObin board. To build the operating system, the first and third steps are the same as the ones used to configure the FOUNDATION build, with the mention that the *kernconf* variable for the host kernel must be set to *GENERIC*. To build the root filesystem two more steps are required: the first one is to generate the root filesystem, filled with all the files needed by the operating system, except the kernel, and the second one is to copy the specified kernel from the object directory into the root filesystem.

5.2. FreeBSD started in ARM Foundation Platform

Once the *FOUNDATION build* was made and the disk image was generated, the next steps must be done on the local computer. All the scripts used for these steps can be found in the ARMv8 utils repository [12]. After creating the disk images, the ARM Foundation Platform should be started using the following parameters:`cores` – to specify the number of cores to emulate, `use-real-time` – to make the emulator to take the real time from the host, `armv8.0` and `gicv3` – to tell the emulator the processor version and the GIC version, textttblock-device – for the

FreeBSD disk image, `data` – to send the *BL1* and *FIP* files for UEFI boot. A Foundation Platform window opens and the host operating system starts to boot. To start a virtual machine, the following commands must be given: `kldload` to load the virtualization driver in the kernel, `bhyveload` to load a virtual machine, textttmount to make the filesystem writeable, `bhyve` to start the virtual machine using VirtIo Block, Network, Console and Entropy drivers.

5.3. Compiling and flashing U-Boot for ESPRESSObin

The guides used in the compiling, flashing and recovering the U-Boot bootloader can be found on the official guide [4], *Guides/Software HowTo* section. From this section, we used *Update the Bootloader*, *Bootloader recovery via UART* and *Build from source - Bootloader* guides.

There are multiple steps needed to be taken to compile u-boot and generate a bootloader image. The first step is to clone the Marvell's version U-Boot. To add DDR4-support for ESPRESSObin, one need to jump at a commit dedicated for ESPRESSObin and apply a set of patches in the repository. The following step is to set the *CROSS_COMPILER* environment variable with the cross-compiler dedicated for ARMv8, *aarch64-linux-gnu*. Then, we need to generate the *.config* file, based on the *make* parameter. Finally, we need to generate the *u-boot.bin* binary based on the ESPRESSObin Device Tree.

The *u-boot.bin* file was not ready to be flashed on ESPRESSObin, it had to be made a valid payload for the ARM Trusted Firmware (ATF). The first thing step was to set a variable, called *BL33*, with the path to the *u-boot.bin* file. Also, two repositories had to be cloned: the first repository represents the Marvell ATF source code and the second one is an utils repository, containing additional sources for the generation of the U-Boot image. These repositories had to been patched as well to work for the ARM Emulation Platform. Then, we must build all the files needed, to flash the bootloader.

Then, on the computer a TFTP server must be started by installing the *tftp-hpa* package. The next steps should be setting *tftp* as a shared director, over TFTP, and to copy there the *flash-image.bin*. A private IPv4 address must be set on the local computer interface to be used by the TFTP client. On the ESPRESSObin, we need to set the *serverip* environment variable with the same IPv4 address set in the previous step, set an IP address for the local interface, using the *ipaddr* variable and test the connection between the computer and the board. The *bubt* command can be used to download the bootloader image and flash it.

The lack of board specifications, especially for the values for *CLOCKPRESET* and *DDR_TOPOLOGY* variables, produced errors in the U-Boot build that put the board in an undefined state and we needed to recover the bootloader on ESPRESSObin using the steps form the board wiki page [4], the *Bootloader recovery via UART* guide. The first steps were to change to a stable branch on the Marvell utils repository. Secondly, on ESPRESSObin, the *wtp* command must be used to signal the board that the computer is starting an UART transfer. Finally, on the computer,

the *WtpDownload_linux* command is used to transfer three files to ESPRESSObin and recover the bootloader.

5.4. FreeBSD on ESPRESSObin

To assure that the ESPRESSObin was suitable for the project, we compiled and then run successfully the OpenWrt operating system on board using U-boot. This shows that the bootloader works and an operating system can boot. The steps presented can be found on the board wiki page [4], in the *Booting initramfs image via TFTP* guide. OpenWrt was chosen, instead of Ubuntu, Yocto, because the DTB and root filesystem were separated from the root filesystem, making the booting process more intuitive. To boot FreeBSD on the ESPRESSObin board, we had to face and solve different kind of issues that will be described further.

```
1 setenv fdt_name "boot/fdt/marvell/armada-3720-espressobin.dtb"
2 setenv image_name "boot/loader.efi"
3 setenv bootcmd 'mmc dev 0; fatload mmc 0:1 $kernel_addr
    $image_name;fatload mmc 0:1 $fdt_addr $fdt_name; bootefi
    $kernel_addr $fdt_addr'
4 [...]
5 run bootcmd
6 set currdev=disk0p1
7 boot
```

LISTING 1. U-Boot variables for FreeBSD

Listing 1 presents the variables set to make U-Boot understand the DTB path, *fdt_name*, know the FreeBSD image path, *image_name*, and load them and boot the operating system, *bootcmd*. The *bootcmd* variable contains a set of individual commands: `mmc dev 0`, which accesses the microSD slot, `fatload ...$image_name`, which loads the FreeBSD's loader at the kernel address, `fatload ... $fdt_name`, which loads the DTB at its address, `bootefi`, which starts a UEFI application, using the kernel and DTB addresses. These variables were chosen to automate the boot task. Running the `run bootcmd`, the loader started and wanted to boot the kernel, but did not find the needed devices. Furthermore, the kernel was not able to find the root filesystem.

The issues mentioned above are related to the fact that the MicroSD card support was missing from our project[12], but it was implemented in the mainline. However, even with the mainline FreeBSD implementation for ARM platforms, the loader could not mount the filesystem as a FAT32 partition. To solve the latter, we used two partitions, both containing the entire root filesystem, the FAT32 partition for U-Boot and a UFS partition for FreeBSD, because the compiled U-Boot version had no support for UFS. Having two partitions, the *currdev* variable seemed logical to be set in the UFS partition (the second one). This time, the GEOM database could be interogated and the UFS partition could be selected, using *ufs:mmcsd0s2*. The FreeBSD host was able to start, proving that the *master* branch had the MicroSD support.

After inspecting the differences between our project and the mainline FreeBSD project, our project was before main project with over 400 commits behind on the ARMv8 system directory. In these commits we could find the MicroSD support patch. Since the bhyve implementation has to be tested using the latest FreeBSD to be committed in the upstream, we decided that the best approach is to rebase the entire code. However, due to the the large amount of changes, the rebase process introduced issues that we had to solve. In kernel-space, some function names were changed. The changes needed to be made by hand to compile the operating system. To call the correct functions, the code from the ARMv8 bhyve branch needed to be compared with the one on the master branch. In user-space, in *usr.sbin/bhyve* the *amd64* and *pci* code was moved in the *bhyve* root directory. Also, the *devemu* generic name was removed, the variables and source code files being renamed in *pci* (for amd64) or *mmio* (for arm64). The separation of amd64/arm64 dependent code and *devemu* architecture independent code was done in an older version and had to be removed and is preffered to let the *amd64* code as it is in the mainline (in the root of the *bhyve* directory). The FOUNDATION, FOUNDATION_GUEST and GENERIC kernel configs had to be changed, due to the change of drivers or driver names. The *iflib* and *virtio-rnd* devices needed to be added, the *random* device removed and the *if_tap* device name was changed into *if_tuntap*. The device name modification had to be updated also in the *host.json* config file, found in the ARMv8 utils repository. In the kernel-space, a register was misspelled, *ICH_ELSR_EL2*. The actual name of the register, found in the ARMv8 documentation [5], is *ICH_EL**R**SR_EL2*. Another change in kernel-space is the use of macros that describe the *ID_AA64MMFR0_EL1* register. The operating system found in the mainline has changed some macro definitions and these updates needed to be also made in the ARMv8 bhyve source code.

After the build could complete successfully, the emulation process started again, but the host operating system blocked at the *kldload vmm* command, which inserts the virtualization module into the kernel. The origin of the problem is in the `vmm_call_hyp` ENTRY, where is used the hypervisor call (`hvc` assembly instruction).

The reason for choosing emulation first, over the ESPRESSObin board, is the fact that before the rebase it was possible to start a virtual machine with less effort. Copying the root filesystem from the FreeBSD virtual machine on the MicroSD card is time consuming (15 minutes) and a small change in the operating system kernel would have to be tested. It takes about 1 minute to build the disk image, copy it on the local computer and start the emulator.

This is the current state of the project. Knowing that there over 12000 commits added in the rebase process, a lot of operating system changes could appear, separate from virtualization, which could affect the *bhyve* functionality.

6. Emulation and bring-up results

For testing the virtualization implementation using the ARM Foundation Platform emulator, we need to build the kernel for the emulated host using the FOUNDATION configuration settings and to build the kernel for the emulated guest using the FOUDNATION_GUEST configuration settings. To run the emulator, we need to generate the disk image on the FreeBSD virtual machine and download it on the computer, then to run the ARM Foundation Platform (with GIC version, disk image and UEFI as parameters). An xTerm terminal will show up, where the virtual machine can be started by loading the virtualization module into the kernel using `kldload`, loading the virtual machine using `bhyveload` and starting the virtual machine using `bhyve`.

For testing the virtualization implementation using the ESPRESSObin board, we need to build the kernel for host using the GENERIC configuration settings. To start the host operating system on ESPRESSObin, we need to install the world and the kernel, to copy the root filesystem on the MicroSD card (both on the FAT32 partition and the UFS partition), set the *fdt_name* with the DTB file, *image_name* with the loader binary and *bootcmd*, then execute the bootcmd. When the loader asks for a a disk where it can find the kernel, we have to set the *currdev* variable with the second partition (*disk0p2*) and run the `boot` command to start the kernel. When the kernel asks for a root filesystem, we must give it the UFS partition, by running `ufs:mmcsd0s2`.

We rebased the ARMv8 branch and resolved all the compilation conflicts and errors and some of the newly appeared issues. At this point, we tested the virtualization implementation using the ARM Foundation Platform. However, in this state, the *vmm* driver does not load and block the host operating system. After the *START* message, there had to be twelve messages and then *STOP* message, but the module remain in the initialization function after four messages. We also tried to bring-up the host FreeBSD on ESPRESSObin but it blocks in the booting process. In this case, the operating system starts, but it waits for *usbus0*,*usbus1* and *CAM*.

7. Conclusions and Further Work

This paper presents a recipe for building and deploying a FreeBSD operating system on ARM hardware. This recipe is based on compiling, testing and solving the issues occurred by debugging (the trial and error method). Moreover, the thesis describes the working environment from a bhyve on ARMv8 developer point of view. To work on a type-2 hypervisor the host should work fine and all the issues to be found in the guest operating system, in the connection between the host and the guest or in the context switch. In this particular case (with an old FreeBSD version), some of the problems (the MicroSD missing support) make the hypervisor development and bring-up process more difficult.

Due to the issues encountered with the lack of the MicroSD support and the tremendous number of changes in the ARMv8 base code in FreeBSD, the project had to be rebased with a newer version of FreeBSD. Since the code base added on

the main FreeBSD project contains a large number of code lines, the rebase procedure introduced issues that needed to be solved before continuing with the bring-up procedure. Some of the issues encountered were related to the misspelling of a ARMv8 registry (ICH_ELRSR_EL2), the renaming of some definitions (ID_AA64MMFR0_EL1 registry related) and the removal some devices (e.g. random).

However, even if some of the issues were resolved, others are still work in progress. At this moment, the virtual machine monitor kernel module (vmm) does not load properly and an investigation of the problem is made.

We plan to further develop the bhyve implementation for ARM architectures. Our current implementation is facing some issues (*vmm* module load is blocked in the ARM Foundation Platform emulator, and the root mount waiting for *usbus0*, *usbus1* and *CAM* issue) that must be solved to bring-up the bhyve code on real hardware. The next steps we want to take are to fix the virtualization driver load issue for the ARM Foundation Platform emulator to start an virtual machine in an emulated system, start the FreeBSD host on the ESPRESSObin v5 by solving the root mount waiting problem, start a virtual machine on the board, with the FreeBSD operating system and solve all the issues in this process, test the implementation with virtual machines that have other operating systems, such as Linux or Windows, and open a review ticket to add the ARMv8 bhyve in the FreeBSD's upstream.

Acknoledgments

The work has been funded by the Operational Programme Human Capital of the Ministry of European Funds through the Financial Agreement 51675/09.07.2019, SMIS code 125125.

REFERENCES

[1] "FreeBSDs official site." Online link: `https://www.freebsd.org`. Last accessed: 24th of April, 2020.

[2] "Bhyves wiki page." Online link: `https://wiki.freebsd.org/bhyve`. Last accessed: 18th of April, 2020.

[3] "ARMv8 Foundation Platform." Online link: `https://developer.arm.com/docs/100961/1190/armv8a-foundation-platform-introduction/platform-overview`. Last accessed: 24th of April, 2020.

[4] "ESPRESSObin wiki page." Online link: `http://wiki.espressobin.net/tiki-index.php`. Last accessed: 28th of April, 2020.

[5] "ARMv8 architecture." online link: `https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile`. Last accessed: 28th of April, 2020.

[6] "ARMv8 wiki page." online link: `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0488c/CHDHJIJG.html`. Last accessed: 28th of April, 2020.

[7] "RFC 793: Transmission Control Protocol." Online link: `https://tools.ietf.org/html/rfc793`. Last accessed: 28th of April, 2020.

[8] "Intel 64 and IA-32 Architectures Software Developer's Manual." Online link: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf. Last accessed: 28th of April, 2020.

[9] "Flattened Device Tree." Online link: https://wiki.freebsd.org/FlattenedDeviceTree. Last accessed: 15th of February, 2020.

[10] "Summer of Code FreeBSD wiki." Online link: https://wiki.freebsd.org/SummerOfCode2015/PortingBhyveToArm. Last accessed: 18th of February, 2020.

[11] Christoffer Dall, Jason Nieh, "KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor." Online link: http://systems.cs.columbia.edu/files/wpid-asplos2014-kvm.pdf. Last accessed: 15th of February, 2020.

[12] "FreeBSD-UPB Github organisation." Online link: https://github.com/FreeBSD-UPB. Last accessed: 2nd of May, 2020.