bhyvearm64: Generic Interrupt Controller Version 3 Virtualization

Alexandru Elisei
University Politehnica of Bucharest
Bucharest, Romania
alexandru.elisei@gmail.com

Mihai Carabas
University Politehnica of Bucharest
Bucharest, Romania
mihai.carabas@upb.ro

***Abstract*—Traditionally associated with low-power, mobile computing, Arm is now seeking to enter the PC and the server markets. Virtualization is especially used in these areas, and current hypervisors rely on various hardware features to achieve efficient virtualization. To this end, the Armv8 architecture introduces a series of hardware mechanisms to reduce or eliminate some of the overhead associated with running virtual machines. Modern computers rely on hardware interrupts to communicate with peripherals, and this aspect of virtualization has seen a series of architectural optimizations from Arm. We will present our experience emulating the Generic Interrupt Controller version 3, the interrupt controller designed by Arm. We have used a mix of virtualization techniques: trap-and-emulate for the memory-mapped regions of the controller, which are accessed less frequently, and hardware accelerated virtualization where possible. To validate our approach, we have created a virtualized timer which is used to deliver timer interrupts to the virtual machines. Timers are essential for modern operating systems, and the virtualized timer is an abstraction over the Arm architectural timer, the Generic Timer. As with the interrupt controller, we have taken special care to take advantage of the available hardware mechanisms to reduce the cost of virtualization. The end result is a fully functioning hypervisor which is able to create, run and destroy virtual machines on Armv8.0-A and later processors.**

***Index Terms*—Arm, Armv8, virtualization, hypervisor, interrupts, timer**

I. Introduction

Arm is the dominant architecture in the mobile space and banking on its expertise in efficient computing, Arm is now looking to enter the PC [1], [2], [3] and the server markets [4], [5]. Virtualization is popular in these areas, especially in the server room, where more than 75% of the servers use this technology [6]. As a result, efficient virtualization solutions are necessary in order for Arm's CPU ambitions to come to fruition.

Virtualization makes it possible for an operating system to run in an environment that is indistinguishable from the real hardware [7]. One of the characteristics that makes virtual machines so appealing is resource control: the virtual machine manager is always in control of the underlying hardware resources: CPU, memory and input/output (I/O) devices. [8]. Early CPUs were slow, with speeds matching the I/O devices of the day, and a simple method of communication was used, called programmed I/O [9]. As CPUs became faster than the devices with which they interacted, a new method of communication was developed, which uses interrupts. Interrupts are electrical signals sent directly to the processor. Today, all processors use interrupts, and virtually every operating has support for such a communication mechanism. As a consequence, a virtualization solution must also provide a way for delivering interrupts to a virtual machine, while still retaining control over the hardware.

bhyvearm64 [10] is a type 2 hypervisor for the FreeBSD operating system and it implements the virtual interrupt controller on top of the Arm interrupt controller. Arm calls its implementation of the interrupt controller the Generic Interrupt Controller (GIC), and we have focused on version 3 (GICv3) of the controller. As with other components of the architecture, Arm has taken care to design the controller with features that make virtualization more efficient.

GICv3 has three separate components: the Distributor, the Redistributor and the CPU interface [11]. The Distributor and Redistributor are memory-mapped and are accessed usually only during the kernel's boot process. Accessing these two components is not performance critical and we have opted for a trap-and-emulate approach to virtualization, where the I/O registers are emulated and stored in memory as part of the virtual machine context. On the other hand, the CPU interface is used frequently, each time an interrupt is handled, and has been designed as part of the processor, accessible via fast hardware registers. For the CPU interface we have taken advantage of various hardware features designed to achieve fast virtualization.

Interrupt controllers are used by input/output devices to communicate with the processor, and we have emulated the system timer as the first device to use our virtual interrupt controller. Timers are essential to the operation of any system for a variety of reasons [9]. Among other things, they are a key part of process scheduling. Arm's implementation of the system timer is called the Generic Timer, and it actually consists of several timers [12]: the physical and the virtual timers, which are always present; the secure physical timer, present when the CPU implements the secure mode of operation; and the physical Exception Level 2 (EL2) timer, part of the virtualization extensions. An operating system is free to choose between the physical and the virtual timer, and we

This work has been sponsored by the FreeBSD Foundation.

have chosen to emulate both of them in order to not restrict a guest operating system to one of the timers. While emulating the timers we also had to take into account the fact that the host operating system must have exclusive access to one of the timers for its own uses.

With the interrupt controller and the timer both emulated, bhyvearm64 is capable of successfully booting and running a FreeBSD guest, albeit with some limitations due to the project being in its early stages: a virtual machine can only run on one virtual CPU, and besides virtio support there is little user space device emulation support.

The structure of the paper is as follows: the next section will cover present hardware and software approaches to interrupt controller and timer virtualization. In section 3 we will describe the Armv8.0 virtualization model that bhyvearm64 implements. Section 4 is dedicated to the Arm interrupt controller: first the GICv3 architecture will be explained, followed by our approach to virtualization. Then we will proceed to presenting the emulated timer in Section 6. Section 7 will cover what we believe the immediate goals for bhyvearm64 should be in order to reach a level of functionality similar to bhyve on x86. Finally, we will present our conclusions regarding virtualization on the Arm platform.

II. Related Work

Virtualization is not new. It appeared in the 1960s [9], and gained momentum in the late 1990s when VMware launched the first virtualization solution for the x86 platform [13], [9]. At that time, the x86 CPU architecture was unvirtualizable according to the Popek and Goldberg definition and interrupt injection and handling was done entirely in software. The current version of the Intel interrupt controller, called the x2APIC, has advanced support for virtualization and virtual interrupts can now be asserted by the hardware, without hypervisor intervention [14].

On the Arm side, the first hypervisor to take advantage of the Arm virtualization extensions was KVM [15], [16], which is part of the Linux kernel. KVM uses a similar approach to bhyvearm64 for virtualizing the interrupt controller, taking full advantage of hardware virtualization where available, and resorting to a trap-and-emulate technique when that is not possible.

Operating systems require timers to perform basic operations like scheduling or measuring the passing of time, but when running inside a virtual environment it is impossible to have the same precision as the physical machine because of the inherent virtualization costs.

For historical reasons, the x86 architecture implements several timers with various and sometimes overlapping uses: some of them are used by software to measure the frequency of another timer; and several are capable of generating a periodic interrupt suitable for timekeeping. This adds to the software complexity of offering a virtualized view of the passage of time, as multiple timers need to be emulated and kept in sync with each other [13].

By contrast, timer virtualization is relatively simple on the Arm platform, as the General Timer is mandated by the architecture and is adequate for timekeeping usage by an operating system. KVM uses an approach similar to bhyvearm64: the virtual timer is made available to the virtual machine, with the caveat that the interrupts generated by the timer still have to injected by the hypervisor, The physical timer is emulated in software because it is used by the host [15].

Arm has created a new virtualization model in version 8.1 of the architecture called Virtual Host Extensions (VHE) [12]. When this feature is enabled, the host operating system executes in a different CPU execution mode than the virtual machine, with access to its own separate hardware timer. The host can then assign the physical timer as well as the virtual timer to the virtual machine, eliminating the need for software emulation. KVM is working towards removing timer emulatation in this scenario [17]. Currently bhyvearm64 doesn't take advantage of VHE, but support is planned in the near future.

III. Background

bhyvearm64 is a type 2 hypervisor and virtual machine manager for the FreeBSD operating system. It is based on the existing bhyve virtualization solution for the x86 architecture. Fig. 1 shows the main components of bhyvearm64, which can be broadly categorized into user space programs and kernel code. The user space programs are bhyveload, bhyve and bhyvectl which an user employs to create, run and destroy a virtual machine. Communication with the kernel is facilitated by the library libvmmapi, which serves as a wrapper over ioctl calls to a special device which uniquely identifies the virtual machine. On the kernel side, the hypervisor is implemented as a loadable kernel module name vmm.ko. The virtual interrupt controller is abstracted as the software component named VGIC and the virtualized timer as vtimer. Perhaps contrary to its name, the virtual timer is a physical, hardware timer and not a software abstraction, as is our virtualized timer. Both the VGIC and the vtimer are emulated in kernel space to achieve better performance and less overhead.

Arm introduced the virtualization extensions with version 7 of the Arm architecture, Armv7, and Armv8.0 follows the same virtualization model. For CPUs that support virtualization, the necessary hardware support is implemented as a distinct processor execution mode, called Exception Level 2 (EL2). The hypervisor architecture is similar to KVM [15], the Linux hypervisor. EL2 was created with a type 1 hypervisor in mind. A type 1 hypervisor [9] runs directly on the hardware and its functionality is centered around managing virtual machines, as opposed to both virtual machines and user space programs as is the case with a type 2 hypervisor plus host kernel. This design decision unfortunately makes it impractical to run the host operating system in EL2, and instead has forced us to split the hypervisor code to run across two different CPU execution modes.

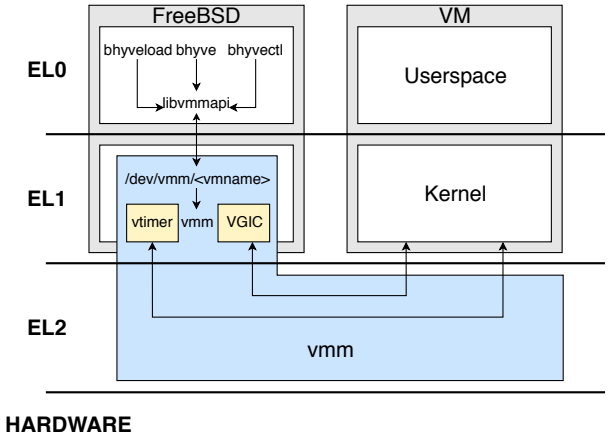

Fig. 1. bhyvearm64 architecture

There are several versions of the Arm Generic Interrupt Controller. The most common are GICv2 and GICv3. GICv2 has two major drawbacks:

- Supports at most eight processors, which is limiting for today's platforms, especially when it comes to server hardware.
- It is entirely memory-mapped, which makes accessing the registers expensive.

Version 3 of the controller addresses these drawbacks by not putting a strict limit on the number of CPUs and implementing hardware registers for the most frequently used operations, registers which have virtualization support. Version 4 of the interrupt controller is identical to version 3 from a software perspective, the only difference being added support for virtual message-based interrupts. We expect adding support for GICv4 in the future will be relatively painless.

IV. GICv3 Architecture

Before describing the process of emulating interrupts, it is worth getting familiar with the inner workings of the Arm Generic Interrupt Controller version 3 (GICv3). An interrupt is an asynchronous, external electrical signal delivered to the processor [9]. The GICv3 controller implements four different types of interrupts:

- Software Generated Interrupts (SGI). These are generated by the operating system and used for inter-processor communication. On other architectures, they are known as Inter-Processor Interrupts (IPIs).
- Private Peripheral Interupts (PPI). This type of interrupts are generated by devices that communicate with only one CPU core, which is always the target for the interrupt. Timer interrupts are PPIs.
- Shared Peripheral Interrupts (SPI). These are interrupts that originate from I/O devices and can target any core in the system.
- Locality-specific Peripheral Interupts (LPI). These are message based interrupts and can be used by PCI Express devices or other devices. The PCI Express specification calls them Message-Signaled Interrupts (MSI) [18].

Besides their type, interrupts have other attributes that can play a major role in deciding when and how they are delivered to the processor: interrupt group, which can be group 0, non-secure group 1 or secure group 1, and interrupt priority. Interrupts can be delivered to the CPU as either an IRQ or a FIQ (distinguished by their offset in the interrupt vector). The security state (secure or non-secure) and their group are the deciding factors in asserting an interrupt [19]. Group 0 interrupts are always delivered as FIQ interrupts. FreeBSD configures all interrupts as non-secure group 1 interrupts, which are always delivered as IRQs.

Interrupts can be masked based on their priority. Interrupt priority also serves to arbitrate between multiple interrupts: the interrupt with the highest priority will be asserted first. There are attempts to use this priority mechanism to allow for pseudo Non-Maskable Interrupts (NMI) on the Arm architecture: interrupts designated as NMI will have a higher priority associated with them, and when disabling interrupts, instead of setting the PSTATE.I bit, a priority mask is used that will block all "regular" interrupts, while NMIs can still be asserted [20].

Fig. 2 is an overview of the GICv3 architecture. There are three main components: a single Distributor per system, one Redistributor and one CPU interface per core. The Distributor is responsible for the configuration of the global interrupts (SPIs) and the Redistributor is used to configure various properties of the interrupts that are private to the core (PPIs and SGIs). The CPU interface is responsible for advancing the state machine associated with handling an interrupt. The Distributor and the Redistributor are expected to be used sporadically, typically at boot to configure the interrupts, as opposed to the CPU interface, which is used each time an interrupt is handled. Their implementation mirrors this usage pattern: the Distributor and the Redistributor are memory-mapped, and accessing them is slower, but that is acceptable because it is rarely done; the CPU interface is implemented as hardware registers and that means faster accesses.

There is one optional component that is missing from the figure. That component is the Interrupt Translation Service (ITS) and it is responsible for message-based interrupts. It is optional because system integrators can choose to implement equivalent functionality in the Redistributor. bhyvearm64 doesn't support LPI virtualization and for this reason it has been omitted.

V. GICv3 Virtualization

The virtual interrupt controller has been implemented taking into account the specifics of the GIC components and how FreeBSD uses the interrupt controller. Interrupts can be configured as group 0, non-secure group 1 or secure group 1 interrupts. Secure group 1 interrupts are always delivered to the firmware running in the secure world, and those type of interrupts haven't been implemented in bhyvearm64, as the hypervisor runs in the non-secure world.

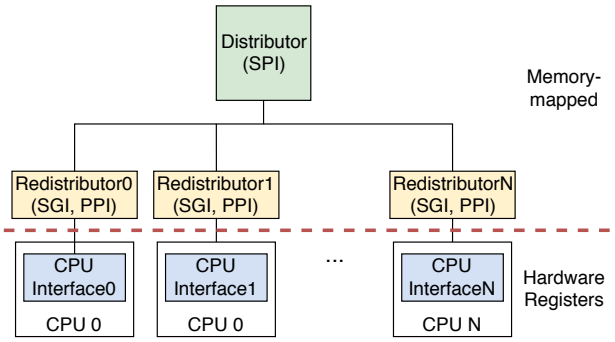

Fig. 2. GICv3 architecture

Group 0 interrupts are always delivered as FIQ interrupts and the firmware can configure the hardware to deliver those interrupts to the secure world, similar to how secure group 1 interrupts work. For this reason, FreeBSD and Linux choose to configure all interrupts as non-secure group 1 interrupts and bhyvearm64 has support for only this use case.

The interrupt controller is emulated entirely in kernel space. Interrupts are time sensitive events, and switching execution to user space to handle the emulation, and then switching again to the kernel would have proven too costly. However, in order to make it possible for the virtual machine manager to emulate various devices, we have provided an API for asserting and retiring interrupts.

A. Distributor and Redistributor emulation

Because the Distributor and the Redistributors are memory-mapped and are seldom accessed, we have chosen a trap-and-emulate approach for virtualization. This approach takes advantage of how memory virtualization works: the guest physical addresses that correspond to the Distributor and Redistributor registers aren't mapped in the Stage 2 translation tables. The Stage 2 tables are responsible for translating a guest physical address generated by the virtual machine into a real address in physical memory. When the guest address isn't present in the tables, an exception occurs. With the information associated with the exception, the hypervisor is able to reconstruct the guest instruction and emulate it accordingly without the need to propagate the fault to user space.

The virtual Distributor and Redistributors are purely software constructs that exist in the host's memory as part of the virtual machine context. Each time the guest tries to access these virtual registers, the hypervisor is able to extract the address from the exception syndrome. For each such memory region we maintain an array sorted by the start address, and using binary search we are able to quickly determine which virtual register the virtual machine is accessing. Emulation consists mainly in saving the value written by the guest and returning that value on a read. The register values are also used for determing which interrupt can be presented to the virtual machine, in a manner similar to how the hardware works.

A complete list of interrupt controller registers that are part of the virtual machine context can be found in Table I.

B. CPU Interface virtualization

The CPU Interface is used every time an interrupt is handled, therefore it makes sense to make read and writes fast. The CPU Interface is implemented as registers that are part of the CPU and has support for hardware virtualization. Virtualization is activated when the hypervisor configures EL2 to route all physical group 0 and group 1 IRQs to EL2 by setting the HCR_EL2.IMO and HCR_EL2.FMO bits. The purpose of these settings is twofold:

- All physical interrupts will be routed to the host, therefore enforcing the separation between the hardware and the guest.
- All accesses to the CPU Interface registers are transparently redirected to a separate set of registers with identical functionality, but which control the handling of virtual interrupts instead of physical interrupts.

Because the virtual CPU Interface registers are used when advancing the state machine of a virtual interrupt in exactly the same way that the non-virtual registers are used for physical interrupts, they are not writable by the hypervisor and are not considered part of the virtual CPU context. However, additional registers are used for asserting a virtual interrupt, and these registers are only accessible at EL2.

C. Virtual interrupt injection

Virtual interrupt injection and handling is done mostly in hardware. The hypervisor is responsible for choosing which interrupt to inject in the guest. After the interrupt is injected, its state becomes pending. When the guest is resumed, the interrupt is asserted to the guest. The rest of the state transitions are handled by the virtual CPU interface and no intervention from the hypervisor is necessary.

TABLE I
VIRTUAL GIC REGISTERS

Component	Type	Register	Description
Distributor	uint32_t	GICD_CTLR	Distributor Control
	uint32_t	GICD_TYPER	Distributor Type
	uint32_t	GICD_PIDR2	Peripheral ID2
	uint32_t *	GICD_ICFGR	Interrupt Config
	uint32_t *	GICD_IPRIORITYR	Interrupt Priority
	uint32_t *	GICD_IXENABLER[a]	Interrupt Enable
	uint64_t *	GICD_IROUTER	Interrupt Routing
Redistributor	uint32_t	GICR_CTLR	Redistributor Control
	uint32_t	GICR_TYPER	Redistributor Type
	uint32_t	GICR_IXENABLER0[b]	Interrupt Enable
	uint32_t	GICR_ICFGR0	Interrupt Config 0
	uint32_t	GICR_ICFGR1	Interrupt Config 1
	uint32_t[]	GICR_IPRIORITYR	Interrupt Priority
System Registers	uint32_t	ICH_EISR_EL2	EOI Status
	uint32_t	ICH_ELRSR_EL2	Empty LRs
	uint32_t	ICH_HCR_EL2	Hypervisor Control
	uint32_t	ICH_MISR_EL2	Maintenance Status
	uint32_t	ICH_VMCR_EL2	VM Status
	uint64_t[]	ICH_LR_EL2	List Registers

[a] Combination of GICD_ICENABLER and GICD_ISENABLER.
[b] Combination of GICR_ICENABLER0 and GICR_ISENABLER0.

To inject an interrupt, the CPU provides the hypervisor with a series of registers, called List Registers. Each List Register contains information about one virtual interrupt that will be handled by the guest: the interrupt group, state, priority, interrupt number and if the virtual interrupt maps directly to a physical interrupt. A virtual interrupt can shadow a physical interrupt, and in this case, when the guest deactivates the virtual interrupt, the corresponding physical interrupt is also deactivated.

The number of List Registers is limited and hardware-dependent. The maximum number is 16 and it is possible to have more pending interrupts for the virtual machine than the number of List Registers. To get around this limitation we keep our own buffer for the pending interrupts. Each time the guest is resumed, we check this buffer and select the highest pending interrupts to be injected in the guest.

The interrupts that will be asserted are selected based on the guest interrupt configuration and it takes into account:

- The group, type and interrupt number: the interrupt must be enabled in the Distributor and the Redistributor.
- If the target CPU for the interrupt is the current CPU.
- The priority of the interrupt relative to the other pending interrupts.
- If two interrupts are equal in terms of priority, the hypervisor keeps an extra field for each interrupt for additional information. For example, a clock interrupt should always have higher priority because this is how the guest operating system does its timekeeping.

Another hardware features that is designed to help with virtualization is the presence of a special interrupt, called the maintenance interrupt. The purpose of this interrupt is to address scenarios where the hypervisor wants to inject more interrupts than available list registers or when a special action needs to be performed when a certain virtual interrupt is handled. In such situations, the hypervisor enables the maintenance interrupt which when asserted will trigger a world switch to the host. The hypervisor is then free to execute the action it deems appropriate.

VI. Timer Virtualization

The timer is essential for any operating system: without it, there could be no process scheduling in the context of preemptive scheduling. Operating systems also use a timer for periodic tasks, either as a functionality offered to user space processes, or for internal purposes. It is necessary for a virtual machine to have access to a virtualized timer in order for the guest operating system to function properly.

A. The Generic Timer

The timer provided by the Armv8 architecture is called the Generic Timer. The implementation actually consists of at least two different timers, up to seven [12]. A system can have a secure physical timer, a non-secure physical timer, which we will call simply the physical timer, a virtual timer, physical and virtual non-secure EL2 timers, and physical and virtual secure EL2 timers. For the purpose of virtualization, we will focus our attention on the timers that a regular operating system uses, the physical timer, which counts the passing of real time, and the virtual timer, which counts the passing of time from a fixed offset.

The host operating system needs to use a timer exclusively; it is not desirable for a virtual machine to slow down the host. bhyvearm64 assigns the physical timer to the host and the virtual timer to the virtual machine currently running on the CPU core for the following reasons:

- Because the virtual timer counts time from a fixed offset, a guest running inside a virtual machine can be tricked into thinking that the timer started at the same time as the virtual machine.
- FreeBSD [21] and Linux [22] prefer choosing the virtual timer over the physical timer when they are both present and virtualization is not active, which is always the case in a virtual machine with no nested virtualization support.
- The Armv8.0 architecture provides a mechanism to emulate the physical timer by trapping reads and writes; there is no such mechanism for the virtual timer.

B. Virtual Timer Virtualization

Timer interrupts are extremely time sensitive. Timer interrupts come at regular intervals (the FreeBSD kernel configures the timer to fire once every 1 millisecond) and because they are so frequent it is extremely undesirable to spend too much time servicing the interrupt. That time can be used instead to execute other tasks. The same is true for the virtualized timer: the less time the hypervisor spends emulating a timer, the more CPU time a virtual machine has at its disposal before the next interrupt.

To achieve minimal overhead for injecting timer interrupts, bhyvearm64 assigns the virtual timer component of the Generic Timer directly to the virtual machine. The guest operating system is free to configure the timer as it sees fit, without any intervention from the hypervisor. However, virtual timer interrupts still need to be managed by the hypervisor. This is necessary because according to Popek and Goldberg's control property [8], the host must always be in control of the hardware, and this also means controlling the delivery of interrupts. There is no hardware mechanism for selecting which interrupts get redirected to the virtual machine. When a guest is running, all interrupts are routed to the host, which will choose which of them will be presented to the virtual machine.

By their nature, interrupts are asynchronous; they can come at any point in time regardless of the program that the processor is executing. This also applies to the virtual timer: a virtual timer interrupt can fire when another host program is running on the CPU instead of the virtual machine that programmed the timer. The virtual timer requires a mechanism for identifying the virtual machine that programmed it before it fired. To achieve this, we have modified the machine dependent part of `struct pcpu` to save a pointer to the last virtual CPU that ran on the core, as shown in Listing 1. The virtual CPU is changed each time a different virtual processor is run by

the machine-dependant part of bhyvearm64 and set to `NULL` when that machine is destroyed.

The correct usage of the virtual timer also requires considering the case when two distinct virtual machines are sharing the same physical core and thus using the same virtual timer. It is important to note that in this context, "sharing" means the CPU execution is alternating between the two virtual machine. The virtual machines most likely started at different times, and their virtual timer offsets will reflect that; most importantly each will set the timer to fire at different moments in the future. To account for this scenario, when switching virtual machines, it is necessary to save the virtual machine timer state and restore the state of the virtual machine that is replacing it.

Listing 1. struct pcpu

```
#define PCPU_MD_FIELDS                  \
    u_int   pc_acpi_id;                 \
    u_int   pc_midr;                    \
    uint64_t pc_clock;                  \
    void    *pc_vcpu;                   \
    pcpu_bp_harden pc_bp_harden;        \
char __pad[225]
```

There is one other important aspect of timer virtualization that needs to be addressed: what happens when the virtual machine is running behind timer interrupts? We have experienced this situation when running bhyvearm64 on the Foundation Platform simulator [23] with multiple virtual machines on the same (simulated) Armv8.0 CPU. For bhyvearm64 we have chosen a conservative approach in order to prevent the guest kernel from spending too much of its CPU time handling timer interrupts. When a virtual timer interrupt is asserted, we don't inject the interrupt in the guest unconditionally, but instead we check if another timer interrupt is active. This can happen in the interrupt handler, after the guest enables the timer and before it signals the end of interrupt, events that are shown in Fig. 3. In this case, we save the new interrupt in the interrupt buffer and we inject it next time we perform a world switch. Because world switches occur at least once every host tick, the guest will have lost at most one full host tick.

C. Physical Timer Emulation

We have discovered that FreeBSD and Linux prefer using the virtual timer when it is available. However, there is nothing stopping an operating system from choosing the physical timer over the virtual timer. Because bhyvearm64 lets the host have control over the physical timer, for physical timer virtualization we have chosen a trap-and-emulate approach. This is achieved by setting the CNTHCTL_EL2.EL1PCEN and CNTHCTL_EL2.EL1PCTEN bits, which cause all accesses to the physical timer to be trapped to the hypervisor.

Fig. 3 shows the steps the FreeBSD kernel executes when handling a timer interrupt. To get the interrupt number, the Interrupt Acknowledge Register (IAR) is read. The interrupt number is the number programmed in the List Register. This changes the interrupt state from pending to active. The kernel disables the timer by writing to the CNTP_CTL_EL0 register, which causes a trap to the hypervisor where the hypervisor does in-kernel emulation. As a result of the write, the hypervisor disables all pending timer alarms for the guest. The guest programs the timer for the next alarm, and we save this value. We don't program any alarms to inject an interrupt because the timer is still disabled.

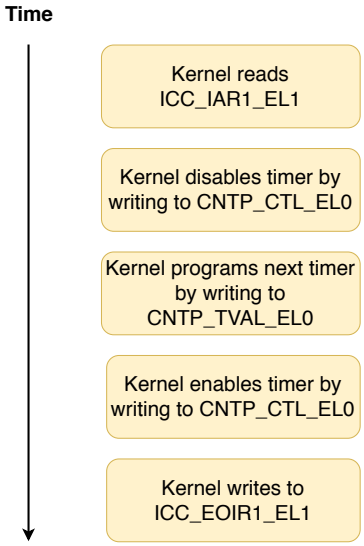

Fig. 3. Timer Usage

Only after the guest enables the timer with another write to CNTP_CTL_EL0 we trap to the hypervisor and program an alarm at the time specified by the guest by using the FreeBSD's callout API. To end the handling of this interrupt, the kernel writes to the End Of Interrupt Register (EOIR), which marks the interrupt as inactive in the List Register. The List Register that held the interrupt is now available to be used for injecting another interrupt.

VII. Future Work

bhyvearm64 is in the early stages and our main goal moving forward is to integrate it with the FreeBSD operating system. To this end, we are pursuing three different approaches: splitting the existing bhyve implementation into machine independent (MI) and machine dependent (MD) code, improving user space device emulation support and improving the hypervisor.

The arm64 vmm module duplicates code from the x86 bhyve implementation. It is obvious that, at the very least, the vmm device code should be very similar between the two architectures. This also applies to the user space components of bhyvearm64, because much of the libvmmapi ioctl wrappers and device emulation code should be shared. This issue was also raised during the review process for the Armv7 version of bhyve [24].

We are currently working on separating the machine independent from machine independent code and we have started with libvmmapi [25]. We will continue with the rest of the

user space utilities, before turning our attention to the kernel module.

At the moment, bhyve for arm64 has support for virtio devices and bvmconsole, which is a development console. This is inadequate for proper virtual machine management. We plan to emulate the Intel 16650 UART and a CD-ROM device. The UART will make interacting with the virtual machine faster, and the emulated CD-ROM will make it possible for the user to install a FreeBSD operating system inside the virtual machine.

The Armv8.0 virtualization model was intended for type 1 hypervisors, and this has the unfortunate effect of making type 2 hypervisors not only more complicated from a software perspective, but also slower. Better support for type 2 hypervisors was added in Armv8.1 under the name of Virtual Host Extensions (VHE) [12]. KVM on Linux implements VHE and this approach has led to better performance compared to Armv8.0 virtualization in all scenarios [16]. Implementing VHE brings the added advantage of noticeably reduced software complexity for the hypervisor. This reduced complexity will make our next goal easier to achieve: adding Symmetric Multiprocessor Support (SMP) support to the virtual machine. Machines with one CPU are rarely seen today, and we plan to make it possible for a virtual machine to use multiple virtual processors.

VIII. Conclusions

Operating systems rely on interrupts to communicate with input/output devices. It is therefore necessary for modern hypervisors to implement interrupt virtualization. bhyvearm64 abstracts Arm's Generic Interrupt Controller version 3 into a virtual interrupt controller by using a multifaceted approach to virtualization. bhyvearm64 takes advantage of hardware features to achieve minimum overhead by enabling the virtual CPU interface. For the seldom accessed, memory-mapped components of the GIC we make use of translation faults to emulate the corresponding reads and writes.

Timers are essential for modern computers because, among other things, they make multiprogramming possible. The first device that uses the virtual interrupt controller is the virtualized Generic Timer, which provides the virtual machine with timer interrupts. bhyvearm64 virtualizes both hardware timers that are part of the Generic Timer. A guest running in a virtual environment is allowed to take full control over the virtual timer. Physical timer accesses are emulated using a trap-and-emulate approach, where the timer state is a software construct part of the in-memory virtual machine state.

Acknowledgment

The current version of bhyvearm64 was inspired by bhyve for Armv7 by Mihai Carabas. bhyvearm64 uses a virtio MMIO implementation by Mihai Darius.

References

[1] Qualcomm Incorporated, "Always On Always Connected PCs are here." https://www.qualcomm.com/snapdragon/always-connected-pc. Last accessed: 21 January 2019.

[2] AnandTech, "Arm's Cortex-A76 CPU Unveiled: Taking Aim at the Top for 7nm." https://www.anandtech.com/show/12785/arm-cortex-a76-cpu-unveiled-7nm-powerhouse. Last accessed: 21 January 2019.

[3] Arm Ltd, "Accelerating mobile and laptop performance: Arm announces Client CPU roadmap." https://www.arm.com/company/news/2018/08/accelerating-mobile-and-laptop-performance. Last accessed: 21 January 2019.

[4] S. McIntosh-Smith, J. Price, T. Deakin, and A. Poenaru, "Comparative Benchmarking of the First Generation of HPC-optimised ARM Processors on Isambard." https://uob-hpc.github.io/assets/cug-2018.pdf. Last accessed: 28 January 2019.

[5] Amazon.com, Inc, "Introducing Amazon EC2 A1 Instances Powered By New Arm-based AWS Graviton Processors." https://aws.amazon.com/about-aws/whats-new/2018/11/introducing-amazon-ec2-a1-instances/. Last accessed: 21 January 2019.

[6] Gartner, Inc, "Gartner says worldwide server virtualization market is reaching its peak." https://www.gartner.com/en/newsroom/press-releases/2016-05-12-gartner-says-worldwide-server-virtualization-market-is-reaching-its-peak. Last accessed: 21 January 2019.

[7] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*. Hoboken, New Jersey: John Wiley & Sons, Inc., 2013.

[8] G. J. Popek and R. P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures," *Commun. ACM*, vol. 17, pp. 412–421, July 1974.

[9] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*. Upper Saddle River, New Jersey: Pearson Education, Inc., 2015.

[10] A. Elisei, "bhyvearm64." https://github.com/FreeBSD-UPB/freebsd/tree/projects/bhyvearm64. Last accessed: 27 January 2019.

[11] ARM Holdings, "ARM Generic Interrupt Controller Architecture Specification GIC architecture version 3.0 and version 4.0." https://static.docs.arm.com/ihi0069/c/IHI0069C_gic_architecture_specification.pdf. Last accessed: 28 January 2019.

[12] ARM Holdings, "ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile." https://static.docs.arm.com/ddi0487/da/DDI0487D_a_armv8_arm.pdf. Last accessed: 23 January 2019.

[13] E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, and E. Y. Wang, "Bringing virtualization to the x86 architecture with the original vmware workstation," *ACM Trans. Comput. Syst.*, vol. 30, pp. 12:1–12:51, Nov. 2012.

[14] Intel Corporation, "Intel 64 and IA-32 Architectures Software Developer's Manual." https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf. Last accessed: 20 February 2019.

[15] C. Dall and J. Nieh, "KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor." http://www.cs.columbia.edu/~cdall//pubs/asplos019-dall.pdf. Last accessed: 28 January 2019.

[16] C. Dall, S.-W. Li, and J. Nieh, "Optimizing the Design and Implementation of the Linux ARM Hypervisor," in *2017 USENIX Annual Technical Conference (USENIX ATC '17)*, 2017.

[17] M. Zyngier and C. Dall, "KVM: arm/arm64: arch_timer: Assign the phys timer on VHE systems." https://www.spinics.net/lists/arm-kernel/msg702086.html. Last accessed: 21 February 2019.

[18] PCI-SIG, "Specification." http://pcisig.com/specifications (login required). Last accessed: 26 Junuary 2019.

[19] ARM Holdings, "GICv3 and GICv4 Software Overview." http://infocenter.arm.com/help/topic/com.arm.doc.dai0492b/GICv3_Software_Overview_Official_Release_B.pdf. Last accessed: 26 January 2018.

[20] J. Thierry, "arm64: provide pseudo NMI with GICv3." https://lkml.org/lkml/2019/1/21/1060. Last accessed: 28 January 2019.

[21] The FreeBSD Project, "Generic Timer driver." https://github.com/freebsd/freebsd/blob/master/sys/arm/arm/generic_timer.c. Last accessed: 26 January 2019.

[22] L. Torvalds, "The Linux Kernel." https://github.com/torvalds/linux/blob/master/drivers/clocksource/arm_arch_timer.c. Last accessed: 27 January 2019.

[23] ARM Holdings, "Fixed Virtual Platforms." https://developer.arm.com/products/system-design/fixed-virtual-platforms. Last accessed: 27 January 2019.

[24] M. Carabas, "Adding virtualization support for ARMv7 platforms." https://reviews.freebsd.org/D10213. Last accessed: 28 January 2018.

[25] A. Elisei, "libvmmapi: Separate MI from MD code." https://reviews.freebsd.org/D17874. Last accessed: 28 January 2018.