2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE 10 YEAR ANNIVERSARY

Reverse Hypervisor

Hypervisor as fast SoC simulator.

François-Frédéric Ozog, Shokubai, Paris, France (*ff@shokubai.tech*)

Mark Burton, Qualcomm France S.A.R.L., Chalagnac, France (*mburton@qti.qualcomm.com*)

***Abstract*—Use hypervisor technology on processors with similar architectures but different number of cores and specific architectural features to simulate embedded platforms and allow execution of production software on the resulting virtual system.**

Keywords—hypervisor; QEMU;HVF; KVM; Emula4;simulation

i. Introduction

Software development on virtual platforms is becoming ubiquitous but requires very high performance. Today's Just In Time [8,12] simulation technologies are struggling to keep up with the demands on performance. For instance, the development of autonomous driving is currently expected to consume a massive amount of simulation, both 'desktop' use cases (developers using simulated environments to develop their code), and massive 'cloud' deployment for continuous integration and test.

At the same time, high performance hosts that have (specifically) Arm® architecture[1] based hardware are now commonplace. This raises the possibility of using the host to accelerate the virtual platform: execute the guest software directly on that host. But this must work from (secure) boot code through to user applications.

Hypervisor technologies such as KVM [1] are often proposed to provide better emulator performance, but they are primarily designed to run, in guaranteed isolation, "neutral" virtual platforms on a single physical system. The cloud providers are leveraging this capability to sell more virtual core count than they have physically. In effect this is the opposite of what's required for simulation of a hardware model. The virtualization environment should expose a specific machine to its guest, with no isolation, while running on a "neutral" host, effectively leveraging a hypervisor in a "reverse way". But current hypervisors are not designed to addressing all the simulation needs:

- Not made for "bare metal" execution – for secure firmware [2] running at "Exception Level 3" on Arm microprocessors which limits the ability to model large software stacks that themselves include a hypervisor (particularly for Software Defined Vehicles).
- Incomplete – Not allowing to simulate Cortex-A and Cortex-M that share memory and a device bus; or implement architectural extensions such as the "auxiliary registers" that are implementation dependent.

In this paper we provide details on using a simulation oriented Virtual Machine Monitor that reverses the role of the supporting hypervisor to provide game changer performance enhancements to virtual hardware models.

ii. Tools & technologies landscape

A. *Technologies preamble*

The word "hypervisor" can be used to designate a variety of functional blocks. In this paper, the hypervisor is the code that controls processor facilities at the lowest level to create virtual cores and deal with memory translations. A Virtual Machine Monitor (VMM) complements the hypervisor with virtual devices and can be

[1] Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere

inserted in the memory map to deal with specific devices or to it can be used to handle situations such as unknown instructions.

Note: The VMM should not be confused with Virtual Machine Management tools such as virt-manager [3] which are administrator tools to deal with a single VM configuration or many VMs.

Hence, in this context, Linux KVM privileged code is composed of a hypervisor (running at Exception Level 2 on Arm architecture) and in-kernel partial VMM (providing virtual GIC for instance). This in-kernel VMM is usually controlled by a user-land VMM such as kvmtool [4] or Firecracker [5] that provide virtual devices (virtio devices, emulated devices such as PL011 UART). The MacOS Hypervisor Framework (HVF) [6] is another example of a hypervisor supported on Apple silicon. Contrary to KVM, HVF does not offer any VMM service and limits itself to hypervisor functional control. The Apple provided VMM, which could be considered as the equivalent of KVM in-kernel VMM and Kvmtool is called the MacOS Virtualization Framework [7].

B. Existing tools

1) Fast Models & Fixed Virtual Platforms

Fast Models [8] are Arm technology that represents accurate, flexible programmer's view of Arm microprocessors, allowing users to develop software such as drivers, firmware, OS, and applications prior to silicon availability. They allow full control over the simulation, including profiling, debug, and trace. Fast Models can be exported to SystemC and TLM 2.0, allowing integration into the wider SoC design process. They are based on a Just In Time simulation technology, which suffers from performance limitations.

Fixed Virtual Platforms [9] are built on top of Fast Models and provide developers ready-to-use model of a complete “neutral” Arm-based architecture system.

Performance of these models depends on the guest and host, but it can be that a minute of CPU time may take an hour of wall clock time to complete in a complex virtual machine.

1) Corellium Virtual Hardware Platform

Corellium VHP [10] is a VMM built on top of a proprietary hypervisor: Charm. The main focus of the platform is to simulate mobile devices and provide very rich tooling to exercise devices like GPS and many other sensors.

The public information is unclear about Charm’s capabilities with respect to EL3 firmware support and unimplemented instructions for instance. So, it is not possible to assess whether the solution is suited to simulate the heterogeneous hardware models needed for complex industrial and automotive applications.

2) Arm Virtual Hardware

According to Fierce Electronics [11], Arm Virtual Hardware service is derived from Corellium. Several boards are emulated but Emulation may be partial. For instance, the RPI4 model does not offer GPU emulation. At this moment it is not possible to assess whether Virtual Hardware allows development of end-to-end stacks, from secure firmware up to application, on heterogeneous hardware models and the accuracy of the modeling.

3) Qemu

Qemu [12] is an emulator that can run binary programs, including operating systems, made for a processor architecture on an entirely different architecture. It does so by translating target instructions into native instructions on the fly (A just-in-time compiler). Qemu can also leverage KVM and HVF but, doing so, Qemu loses its architectural emulation capabilities ant thus it is not possible to:

- select a processor and its features (it is limited to the host processor)
- choose a Arm Generic Interrupt Controller implementation
- run secure code in Arm TrustZone

Qemu refers to KVM and HVF as “accelerators”, but this is not quite correct as they bring with them specific restrictions on the way in which Qemu behaves. Others are working on ways to combine accelerators and the JIT

engine itself to overcome some of these restrictions, specifically to allow e.g., EL3 code to execute in the JIT while an accelerator such as HVF or KVM can be used for the EL2/EL1 code. This 'hybrid' approach is interesting but may suffer from some performance penalties when switching occurs because of state management costs between JIT and non-JIT. In addition, the number of switches may become overwhelmingly high as, in some situations, application related IRQs must be routed to EL3. Furthermore, simulation of MPAM features on non MPAM host processors would trigger the JIT for the whole solution and thus the scope of JIT may not be just contained to the secure world.

In the approach we are taking, while topologically very similar, rather than use a full JIT engine and 'switching' to perform EL3, we are proposing a very light weight emulation of processor features and the secure switch itself and then using the host core to directly execute the rest of the processor behavior.

iii. EMULA4

A. *Overview*

Emula4 is a new type of VMM that allows the creation of virtual hardware models based on existing or future Arm architecture based heterogeneous platforms. Those virtual hardware models can run software stacks at wall clock speed. It is using the "reverse hypervisor" concept to support simulation of architectural extensions, platform micro-controllers, accelerators, and interconnects.

Emula4 intends to be a generic framework to allow hardware providers (silicon and board makers) to provide their customers with virtual models of their own hardware. Emula4 ambition is also to empower those customers to assemble and test virtual hardware models of their choice without changes of the main Emula4 software.

Emula4 allows running firmware/software at any Exception Levels and thus allows execution of Trusted Firmware and OP-TEE in the VM at close to native speeds.

Emulation is done such that silicon vendor extensions are usable by the virtual hardware model software stack. In a similar way, a program made for Arm architecture version 8.6 can run on an Arm architecture version 8.1 as close as possible to the intended behavior. There are still limitations, for instance, it will not be possible to run a debug program that makes use of 32 hardware debug registers on a processor that has only 6 (it should be possible though to run it if it makes use of 6 out of 32). Equally, while it is possible to simulate MPAM access control on cores that are not MPAM capable, it is not a goal to try to simulate memory bandwidth measurement and control of MPAM (yet performance monitors may allow a close enough implementation).

To enable user-level composition, Emula4 makes use of System Device Trees [13] concepts to describe the whole system so that individual subsystems can be given a relevant device tree thanks to the lopper tool [14]. This methodology is in line with the Arm specification System Ready [15] standard that intends to decouple device trees from software stacks and associate them more closely with hardware.

B. *Reverse mode capable hypervisors*

To be used according to the reverse hypervisor concepts, a hypervisor needs to offer sufficient control on what the VMM is capable of intercepting, or, in other words, what exits are possible from the VM. In that respect, MacOS HVF is offering an effective interface to leverage processor trapping capabilities as HVF is only a hypervisor. However, it does not support all Arm architecture fine grain control of instruction trapping such as trapping on read of memory management registers. Linux KVM is both a hypervisor and in-kernel VMM and, as such, it deals directly with elements such as the GIC for instance. In doing so, it prevents the simulation framework (in user space) from providing its own "GIC" (or, indeed, other features). While providing a secure GIC implementation is a valuable capability in the context of cloud virtualization, it prevents a simulation oriented VMM from fully controlling the details of the virtual hardware model. Because of this, upstream KVM cannot be used as a reverse hypervisor.

C. *Interception: an extension of virtualization "exit"*

Processor virtualization implements "exit mechanisms" to inform the VMM that it needs to handle a situation. Currently, the Arm architecture does not allow the VMM to "exit" on everything that a emulator would like. For instance, there are no means to trigger "exits" when the current Exception Level register is red. Some "exits" such as reading VBAR register (at any level) is only defined on newer processor specifications but are not available on silicon. And should a vendor have a particular behavior on a "standard" instruction, there is no way to trigger an "exit".

In this context, implementing the reverse hypervisor concepts requires forcing exits when needed: that is VMM guided interception. To that end, two interception strategies were identified:

- Metadata based execution.
- Runtime patching.

1) Metadata based interception

With the metadata strategy, the code is first analyzed once at slow speed to produce a metadata, then the code can be executed at full speed with the metadata. Programs run unmodified which allows natural support of intellectual property protection through self-modifying code or memory coherency protection. The metadata consists of information used to set hardware breakpoints at desired locations to complement the processor exit capabilities.

There are a few downsides about this approach:

- Even with complex logic associated with the metadata to create smart "chains" of breakpoints (less than 300 points in a code base comprising Trusted Firmware A, U-Boot and the Linux kernel), the limited number of effective breakpoints in available hardware (typically 6 hardware breakpoints) make the implementation not scalable or fragile to different code paths (suspend/resume…)
- There is no "exit" to VMM on EL3 instructions: a synchronous exception inside the VM and thus the VMM needs to hijack the exception handling framework. This is a complex endeavor as the exception handling may be in many different states (not configured, partially configured, with/without MMU) at different Exception Levels.
- Though runtime code analysis proved to be able to execute up to 2.5 million instructions per second, the overall observable speed of the running program is only a fraction of the expected time. Speed up strategies can be implemented by smartly excluding analysis of some parts of the code and this allows a full system to boot in a few minutes.

2) Runtime patching based interception

With this strategy, the code is described by its "handover" mechanisms: Trusted Firmware to U-Boot to Shim to grub to Linux. At each handover, code analysis is performed, and patching occurs (mechanism similar to debugger instrumentation). Additional steps of code analysis can be introduced through LUA [16] logic. For instance, after applying Linux alternatives, loading a module… The current code analysis is based on having the symbols of each phase but that may be changed.

3) Comparison with Qemu/TCG

As has been mentioned above, topologically this approach is similar to splitting the exception levels between e.g. a QEMU TCG and an 'accelerator' such as KVM or HVF. However, rather than using QEMU TCG to execute the EL2/3 code, we are using the accelerator itself. Emula4 then emulates specific individual instructions, rather than using the TCG to emulate all instructions at EL2/3. This has the advantage of requiring less support from QEMU TCG, no need to move state from the accelerator into TCG and back. It means that the 'switch time' is reduced, and it will improve the performance of executing 'normal' EL2/3 code itself.

That said, while simple code patching occurs, a smart code injection may be implemented. The VM exception handling hijacking showed that it is possible to divert normal code to VMM produced code while confirming to MMU and other processor states. So rather than provoking exits to VMM, the static analysis could generate those in-VM diversions to provide very high-speed execution.

D. Current state of proof of concept and roadmap

1) Interception strategy and no-exit strategy

The two interception strategies (metadata and patching as described in the previous sections) were implemented and compared:

- the last metadata strategy iteration consumed all breakpoints for the emulation, and none were available for debugging.
- Associated metadata logic complexity kept increasing at each new corner case.
- running the same image on two hardware platforms requires some level of abstraction in the metadata format itself.

- The code patching resulted in a smaller code base with a lot of refactoring that removed corner cases handling needed by the hardware breakpoint strategy. The static code analysis is not noticeable from a tester's perspective which is quite a difference from the runtime analysis of the hardware breakpoint strategy. This static code analysis does not depend on the host hardware platform eliminating metadata need.

- The teachings from code injection required for hijacking the VM exception in the metadata approach proved it would be possible to implement no-exit custom behaviors (switch read physical timer with read virtual timer instruction, read CurrentEL recoded as load constant for the simplest cases) with proper MMU and other processor state aspects.

So, the patching interception strategy was selected as the strategy of choice despite the complexities of handling self- modifying code (Linux alternatives) and the need of symbols for code coverage assurance (should the symbols be a problem between suppliers, a form of meta-data file with "points" of interests in the code can be envisaged).

2) Processor emulation details

Emula4 can run EL3, S-EL1 and EL1 code.

EL3 has 41 registers accessed thru MRS/MSR instructions but implementation of only a subset is required to run Trusted Firmware A: CPTR_EL3, ELR_EL3, ESR_EL3, MAIR_EL3, MDCR_EL3, SCR_EL3, SCTLR_EL3, SPSR_EL3, TCR_EL3, TTBR0_EL3, VBAR_EL3. As stated earlier, CurrentEL access needs to be trapped "artificially" to obtain proper behavior while running EL3 code. Note that even though the number of registers is limited, the functionalities controlled by the register fields is quite important. In addition to registers simulation, some instructions such as TLBI have some EL3 reserved flavors that also need to be implemented.

For EL1 and S-EL1, additional registers need to be controlled: PAN, SP_EL1, CNTFRQ_EL0, ELR_EL1, SCTLR_EL1, SPSR_EL1, TCR_EL1, TTBR0_EL1, TTBR1_EL1. Other registers may be dealt with opportunistically such as MAIR_EL1 that deals with memory attributes, but they could be intercepted should it become necessary. Debug, trace, and performance counters are reserved for Emula4 at this stage.

Some stats about numbers of times Emula4 handles those instructions:

- 259 for Trusted Firmware A
- 7,357 for U-Boot
- Around 380,000 for booting Ubuntu 23.04 with a 6.2 kernel

Out of the 7,357 interceptions in U-Boot, 6,929 are caused by read CurrentEL and 411 by read CNTFRQ_EL0: those could be smartly inlined, resulting in only 17 exits.

The 380,000 interceptions can be significantly reduced because Emula4 simply should not care about them:

Percentage	Instruction	Location
19%	mrs x22, elr_el1	el0t_64_sync+016c
19%	mrs x23, spsr_el1	el0t_64_sync+0170
19%	msr elr_el1, x21	ret_to_user+00ac
19%	msr spsr_el1, x22	ret_to_user+00b0
19%	mrs x1, esr_el1	el0t_64_sync_handler+0014
5%	Many forms	Rest of code

Out of the remaining 5% of interceptions many are dealing with MMU (TTBR[01]_EL1…) updates in cpu_do_switch_mm which also should not be of much interest for Emula4 in the general case. Bottom line, it is expected that only a few thousands of interceptions are needed for a Linux boot (MMIOs are not counted).

3) Supported hypervisor

Currently, Emula4 supports MacOS HVF and is planned to support additional hypervisors such as KVM and other commercial hypervisors in the future. KVM support will require support for a "raw mode" that removes the in-kernel handling of devices such as the GIC and allows them to be simulated externally (e.g. by providing an exit from KVM). A patch which negotiates this functionality at runtime through standard KVM API has been developed as part of the Emula4 POC.

4) Simulated hardware, board assembly and device tree

Qemu allows its users to augment a machine with some devices such as disks. In this context the machine is defined by Qemu programmers and changing even simple aspects of the machine may prove quite complex.

In contrast, Emula4 aims to allow its users to assemble a machine and its devices with its devices based on "libraries" of components maintained by vendors or communities.

To validate this principle, the 'SolidRun Macchiatobin board' [17] has been simulated from components like memory controller, clock hierarchy, devices such as GIC v2M and v3, PL011 UART, NS16550 UART to the point it is possible to boot the SolidRun provided binary image on the simulated board. The software stack in the image comprises of the Trusted Firmware, U-Boot, and Ubuntu 23.04 with the Macchiatobin BSP.

Assembling the board is done through configuration (excerpt):

```
-vobj "RAM#address=0x4000000||hostmem#size=4"
-vobj "SECRAM#address=0x4400000||hostmem#size=12"
-vobj "RAM#address=0x05000000||hostmem#size=2048"
-vobj AP806@MARVELL#address=0xf0000000
-vobj CP110@MARVELL#address=0xf2000000
-vobj CP110@MARVELL#address=0xf4000000
```

The device tree for the board is generated from the configuration. For instance, the following configuration snippet will generate all phandle references to make the connections between device tree elements:

```
-vobj "GIC@QEMU#name=main_gic;root=true"
-vobj "PL011#uartclk=main_clock;apb_pclk=main_clock;irq=spi:1@main_gic…"
-vobj "PL011#uartclk=main_clock;apb_pclk=main_clock;irq=spi:2@main_gic…"
```

This device tree generation capability is just a first step towards building a complete System Device Tree. In the future, Emula4 should be capable of taking input from System Device Tree or other formats to assemble the virtual hardware model.

5) Performance elements

Benchmarking is not that simple, is it hard to compare apples to apples. For instance, the Macchiatobin memory controller needs around 5 seconds to train the DRAM connections while it is instantaneous on the simulated hardware. The approach to benchmarking has been to create a simulated version of the Qemu "virt" machine on Emula4 so that Emula4/virt and Qemu/virt can be compared running the same VMs on the same Apple M1 host.

U-Boot loading of the Linux kernel image and initrd from a virtio block disk is measured at around 6GB/s which is like Qemu with KVM acceleration. The comparison is not entirely fair at this stage as Emula4 virtio implementation is just embryonic and uses lower performance legacy mode.

The Linux kernel provides a CPU microbenchmark during startup when selecting the software RAID implementation. Here is the comparison between JITed code from QEMU and Emula4 on an Apple M1:

	raid6:int64x8	raid6:neonx4	xor:32reg	xor:neon
Qemu + TCG	2,007 MB/s	3,006 MB/s	4,595 MB/sec	2,899 MB/sec
Emula4	14,438 MB/s	35,871 MB/s	43,763 MB/sec	54,311 MB/sec

This just shows that with Emula4, CPU intensive calculations run at native speed and allows execution of complex software stacks at close to real hardware speed, while Qemu, when executing the full software stack (all Exception levels) is of course much slower. It is expected that running Arm architecture version 9 with realms on Arm architecture version 8 will perform at the same close to real speed with Emula4.

From a boot time perspective, following are three kernel consecutive message excerpts from Ubuntu with kernel 6.2 with Emula4, Qemu+TCG, Qemu+HVF:

#Emula4:

```
[    0.330107] Run /init as init process
Loading, please wait...
Starting systemd-udevd version 252.5-2ubuntu3
[    1.222881] virtio_blk virtio0: 1/0/0 default/read/poll queues
```

#Qemu+TCG:

```
[    1.703274] Run /init as init process
Loading, please wait...
Starting systemd-udevd version 252.5-2ubuntu3
[    8.031730] virtio_blk virtio0: 1/0/0 default/read/poll queues
```

#Qemu+HVF:

```
[    0.232395] Run /init as init process
Loading, please wait...
Starting systemd-udevd version 252.5-2ubuntu3
[    0.318886] virtio_blk virtio0: 1/0/0 default/read/poll queues
```

Emul4 is thus way more performant than TCG and close to Qemu+HVF. The delta (0.10s) to /init between Qemu+HVF and Emula4 is the static analysis plus patching. The delta (0.90s) to virtio_blk queues is due to some device + GIC emulation issues (not solved as of August 20).

6) Debugging

Emula4 provides an embryonic debug framework to debug both itself and the payloads it runs. For instance, it is possible to launch a LUA script when a entering a Linux function:

bp2 = breakpoints.acquire(vmm, "$linux:folio_wait_bit_common", on_folio_wait_bit_common)

Those debugging facilities are entirely independent from Linux debugging capabilities such as kprobes. It is possible to use Emula4 debug framework on any part of the payload (EL3, transitions from EL1 to EL3…).

Single stepping allows Emula4 user to follow an SMC call from Linux to OP-TEE.

7) Source code

Emula4 code source is not available to the public for now. The following table presents a cloc analysis of the code:

Language	files	blank	comment	code
C	55	3297	2006	18475
C/C++ Header	34	1602	1079	7366
Assembly	3	49	28	189
XML	1	0	0	10
Markdown	1	1	0	1
SUM:	94	4949	3113	26041

Out of the 18K lines of code, 7.4K are the “real meat” (1K are generated by a script that handles sysreg XML specification), 6.8K are devices and components, the rest is symbol management, debugging, scripting, and faceplates for OS and hypervisor support.

iv. Future work

Many features and functionalities remain to be added. The most salient milestones will be addition of:

- a Cortex-M core with address space aliasing with the main Cortex-A environment to allow execution of SCP firmware on the SolidRun Macchiatobin board.
- Emulation of MPAM memory protection (not bandwidth measurement and enforcement) on processors that are not capable of MPAM.

v. References

- Kernel Virtual Machine
  https://www.linux-kvm.org/page/Main_Page
- TrustedFirmware
  https://www.trustedfirmware.org
- The virt-manager application is a desktop user interface for managing virtual machines through libvirt.
  https://virt-manager.org/
- kvmtool is a lightweight tool for hosting KVM guests
  https://github.com/kvmtool/kvmtool
- Firecracker is a virtual machine monitor (VMM) that uses the Linux Kernel-based Virtual Machine (KVM) to create and manage microVMs.
  https://firecracker-microvm.github.io
- MacOS Hypervisor Framework - Build virtualization solutions on top of a lightweight hypervisor, without third-party kernel extensions.
  https://developer.apple.com/documentation/hypervisor

- MacOS Virtualization Framework - Create virtual machines and run macOS and Linux-based operating systems.
  https://developer.apple.com/documentation/virtualization
- Fast Models – accurate models of Arm IP
  https://developer.arm.com/Tools%20and%20Software/Fast%20Models
- Fixed Virtual Platforms
- Corellium Virtual Hardware Platform
  https://www.corellium.com/platform
- Arm Virtual Hardware
  https://www.fierceelectronics.com/iot-wireless/arm-teams-corellium-speed-iot-product-development
- Qemu - A generic and open source machine emulator and virtualizer
  https://www.qemu.org
- System Device Tree
  https://static.linaro.org/connect/san19/presentations/san19-115.pdf
- Lopper
  https://static.linaro.org/connect/lvc20/presentations/LVC20-314-0.pdf
- Arm specification System Ready
  https://www.arm.com/architecture/system-architectures/systemready-certification-program
- LUA - programming language
  https://www.lua.org/
- SolidRun Macchiatobin
  https://staging3.solid-run.com/arm-servers-networking-platforms/macchiatobin/