embeddedworld2023
Exhibition&Conference
...it's a smarter world

Virtualization today, Virtualization tomorrow:

Problems, Challenges, and Opportunities for Mixed-Criticality Systems

Sandro Pinto
Centro ALGORITMI / LASI, Universidade do Minho
Guimarães, Portugal
sandro.pinto@dei.uminho.pt

***Abstract*— There is an ongoing trend in several embedded industries to consolidate multiple subsystems onto the same hardware platform. For example, in the automotive industry, network-connected infotainment starts to be deployed alongside safety-critical control systems (e.g., steering, brake, ABS). To guarantee the temporal and spatial isolation of these components with different criticalities/safety integrity levels (ASIL), it is nowadays common to rely on virtualization technology. However, there is an erroneous belief that hypervisors are the new magic bullet, working as transparent layers of software that provide perfect guarantees and have no impact on the overall system. In this paper, we provide a comprehensive picture of the state of affairs concerning the use of virtualization in the context of the so-called mixed-criticality systems. While sharing our experience on the development of an open-source static partitioning hypervisor (Bao) and implementing the hardware virtualization support of a novel computer architecture (RISC-V), we will (i) cover the main problems and limitations currently affecting existing hypervisor solutions and (ii) discuss the (research) challenges and opportunities lying ahead of us.**

Keywords—Virtualization; hypervisor; mixed-criticality; static partitioning; Arm; RISC-V; MCS.

I. INTRODUCTION

Virtualization is a technological enabler used on a broad set of applications, i.e., from cloud computing and servers to mobiles and embedded and mixed-criticality systems (MCS). For cloud computing, virtualization provides advantages for workload management, data protection, and cost and power effectiveness [1]. For embedded and mixed-critical systems, the industry has been leveraging virtualization to minimize size, weight, power, and cost (SWaP-C), while guaranteeing temporal and spatial isolation for certification (e.g., ISO26262) [2-4]. Due to the proliferation of virtualization across multiple industries and use cases, mainstream computing architectures such as Intel, Arm, and more recently RISC-V, have introduced hardware virtualization primitives in the ISA, i.e., Intel Virtualization Technology, Arm Virtualization Extensions, and RISC-V Hypervisor extension, respectively [5-8].

There are a plethora of hypervisor solutions leveraging hardware virtualization support for MCS. Initially, there were efforts to adapt well-established server-oriented hypervisors, such as KVM [6] or Xen [9], to embedded architectures (mainly Arm) with some degree of success. However, given the mixed-criticality nature of the target systems, the straightforward logical isolation has proven to be insufficient for the tight embedded constraints and real-time requirements. Moreover, these embedded hypervisors often depend on a large GPOS (typically Linux) to boot and manage virtual machines (VMs), or for services such as device emulation or virtual networks [6],[9-10]. From a safety (and security) perspective, this dependence bloats the system trusted computing base (TCB), thus enlarging the overall attack surface and hampering certification [3],[10-11].

The static partitioning hypervisor architecture is the zeitgeist for MCSs [3],[11-12]. This architecture leverages hardware-assisted virtualization technology to build a minimal software layer that statically partitions platform resources and assigns each one exclusively to a single VM instance. It assumes no hardware resources need to be shared across guests. Each virtual CPU is pinned to a single physical CPU; thus, there is no need for a scheduler (and other complex semantic services), resulting in a considerable decrease in size and complexity. Although possibly hampering the efficient resource usage requirement, static partitioning allows for stronger guarantees concerning isolation, freedom from interference, and real-time. Quest-V [12], Jailhouse [10], and Bao [3] are examples of hypervisors implementing the pure static partitioning architecture.

Despite the strong CPU and memory isolation provided by the static partitioning approach, this has proven to be not enough, as micro-architectural resources such as last-level caches, interconnects, and memory controllers remain shared among partitions. The resulting contention leads to weak temporal isolation, hurting performance, interrupt latency, and predictability/determinism [3],[11],[13-14]. From another perspective, the absence of strong temporal isolation guarantees can also be leveraged by a malicious VM to implement DoS attacks or to launch timing side-channel attacks [11],[15-16].

In this paper, we provide a comprehensive picture of the state of affairs concerning the use of virtualization in the context of the so-called MCS. Based on the experience consolidated through the years, in particular with the development of an open-source static partitioning hypervisor (Bao) and the implementation of the hardware virtualization support for two RISC-V cores, we cover the main problems and limitations currently affecting existing hypervisor solutions and discuss the (research) challenges and opportunities.

II. MIXED-CRITICALITY SYSTEMS AND VIRTUALIZATION

A. Mixed-Criticality System (MCS)

A mixed-criticality system (MCS) is an embedded and/or real-time system that consolidates workloads with two or more distinct criticality levels (e.g., safety-critical and non-safety-critical). There are two conflicting trends in the design of such systems. One is related to the safety guarantees in terms of real-time, predictability, and freedom-from-interference. The other is related to feature-richness and functionalities, which are increasingly consolidated onto the same platform due to size, weight, power, and cost (SWaP-C) constraints. For example, in automotive systems, the network-connected infotainment is often deployed alongside safety-critical control systems [2,3].

B. Virtualization Technology

Virtualization is a well-established technology that enables the consolidation of multiple, unrelated software stacks onto the same physical machine by partitioning and multiplexing hardware resources (e.g., CPUs, memory, etc) between multiple virtual machines (VMs). The software layer that implements virtualization is called virtual machine monitor (VMM) or hypervisor. The software executing in the VM is referred to as guest, typically an operating system, thus guest OS.

C. Hardware-assisted Virtualization

Due to the proliferation of virtualization across multiple industries and use cases, prominent players in the silicon industry started to introduce hardware virtualization support in mainstream computing architectures, i.e., Intel Virtualization Technology, Arm Virtualization Extensions, and RISC-V Hypervisor extension. Bellow, we overview the virtualization support in Arm and RISC-V.

Arm Virtualization Extensions (VE). Arm has had hardware virtualization support since version 7 of the ISA. The most recent version of the architecture (Armv8/9-A) extends the privileged architecture with a dedicated hypervisor privilege mode (EL2) which sits between the secure firmware mode (EL3) and the kernel/user modes (EL1/EL0). A hypervisor running at EL2 has fine-grained control over which CPU resources are directly accessible by guests (e.g., control registers). EL1/EL0 memory accesses are subject to a second stage of translation, in control of the hypervisor. Arm VE support different page sizes: 4 KiB, 16 KiB, and 64 KiB. Arm also defines the System Memory-Management Unit (SMMU), which extends memory virtualization mechanisms from the CPU to the bus to restrict VM-originated direct-memory accesses (DMAs). The virtualization acceleration also spans to the interrupt controller, i.e., the Generic Interrupt Controller (GIC). The GICv2 standard has two main components: a central distributor and a per-core interface. GICv2 provides virtualization support only on the interface; the distributor, however, must be fully emulated. The GICv3 and GICv4 provide support for the direct delivery of hardware interrupts to VMs; however, this feature is only available for inter-processor interrupts (IPIs) and message-signaled interrupts (MSIs).

RISC-V Hypervisor Extension. The RISC-V privileged architecture specification introduced hardware support for virtualization through the Hypervisor extension. This extension execution model follows an orthogonal design where the supervisor mode (S-mode) is modified to a hypervisor-extended supervisor mode (HS-mode). There are two new privileged modes, i.e., virtual supervisor mode (VS-mode) and virtual user mode (VU-mode). The Hypervisor extension also defines a second stage of translation (G-stage in RISC-V lingo) to virtualize the guest memory by translating guest-physical addresses (GPA) into host-physical addresses (HPA). The HS-mode operates like S-mode but with additional hypervisor registers and instructions to control the VM execution and G-stage translation. We have implemented the Hypervisor extension in two RISC-V CPUs: Rocket [7] and CVA6 [8]. In tandem with this extension, there are two specifications key to virtualization: the Input/Output Memory-Management Unit (IOMMU) and the Advanced Interrupt Architecture (AIA). The IOMMU provides hardware virtualization support for peripherals at the platform level, while the AIA (consisting of APLIC + IMSIC) provides virtualization support for interrupts. Notwithstanding, the AIA virtualization support is only available through the IMSIC; it is still a topic of ongoing study whether an APLIC should be allowed to directly delivery of interrupts to VM.

D. Hypervisors for MCS

Over the past decade, many hypervisors have been designed or retrofitted for MCS. Quest-V essentially pioneered the static partitioning hypervisor architecture, but only has support for x86. Other x86-only solutions include ACRN. ACRN was built for MCS but still allows for more flexible configuration and device sharing. Jailhouse is an open-source static partitioning hypervisor developed by Siemens that relies on Linux to boot and initialize the system. Xen is a widely-used hypervisor that relies on a privileged VM, called Dom0, to manage non-privileged VMs (DomUs) and interface with peripherals. Xen Dom0-less is a novel approach that enables the deployment without any Dom0, booting all guests directly from the hypervisor and statically partitioning the system. seL4 is a formally verified microkernel that can be used as a hypervisor in combination with an user-level VMM (e.g., CAmkES VMM). Xtratum is a hypervisor of particular interest in the aerospace but lacks support for Armv8-A. Xvisor is an embedded hypervisor that targets mainly soft real-time applications. The NOVA microhypervisor follows a similar design to seL4, but is tailored for virtualization. LTZVisor and VOSYS have leveraged TrustZone to build mainly dual-guest systems; uRTZVisor provides multi-guest support. There are also commercial offerings such as LynxSecure, PikeOs, VXworks, Integrity, Coqos or newcomers such as CLARE. Despite it tight independence with Linux, KVM has also been considered for embedded real-time use cases. We have developed Bao.

Bao Hypervisor. Bao [3] is an open-source static partitioning hypervisor that implements the pure static partitioning architecture, i.e., a minimal, thin layer of privileged software that leverages the existing ISA virtualization primitives to partition the hardware. Bao has no scheduler and does not rely on external libraries or privileged VM (e.g., Linux), consisting of a standalone component that depends only on standard firmware to initialize the system and perform platform-specific

tasks. Bao initially targeted Armv8-A. The mainline now includes support for RISC-V, Armv7-A, and Armv8-R.

III. Virtualization Today: Problems and Limitations

Ideally, the "hypervisor" would just be a thin configuration layer with no runtime overheads; however, this utopia is far from reality. In this section, we discuss a few problems and limitations of existing virtualization technologies and hypervisor solutions, mainly driven by MCS requirements and use cases.

A. *Performance Overhead*

Virtualization has been connotated, for quite long time, to a significant impact in terms of performance. This was mainly due to the fact that mainstream computer architectures a decade ago lacked hardware virtualization acceleration. Thus, back then, existing hypervisor solutions (mainly for the cloud) resorted to software-based techniques such as (i) dynamic binary translation, (ii) shadow page tables, (iii) or paravirtualization, to virtualize the system. With such software-centric techniques, reported performance impact achieved up to ~350% in case of paravirtualization (LMbench) [7].

With the introducing of the hardware virtualization support in COTS processors, the performance impact reduced significantly (depending also on the target microarchitecture). Of course, this performance impact still depends on the design of the hypervisor and on the different features. For example, embedded hypervisors with scheduler to time multiplex virtual CPUs in the same physical CPU have an extra overhead comparing to static partitioning hypervisors. To support our argument and corroborate what we have already reported in previous works [3], we ran the MiBench (Automotive and Industrial Control Suite) applicational benchmark, to collect the execution time of Linux running directly atop the hardware and Linux running in a VM atop Bao (Xilinx ZCU104). We ran the experiments for the virtualization setup with different page sizes, i.e., 2MiB and 4KiB.

For each benchmark, we collected the average absolute execution time for the native execution. The first observation is that Bao incurs a negligible performance penalty, i.e., less than 1% across all benchmarks when the page size is 2MiB. When Bao is configured to use the 4KiB page size, the overhead can reach up to 6%. Thus, for a virtualized system configured with a single guest VM, there are two main possible sources of overhead. The first source is the increase in TLB miss penalty due to the second stage of translation. Second, the overhead of trapping to the hypervisor and performing interrupt injection, e.g., timer tick interrupt.

As a final takeaway, we can highlight that modern static partitioning hypervisors such as Bao, do not incur meaningful performance impacts due to (i) modern hardware virtualization support, (ii) one-to-one mapping between virtual and physical CPUs (i.e., no scheduler), and (iii) minimal traps. However, we highlight the need for support for / make use of superpages (e.g., 2 MiB) to minimize TLB misses and page-table walks.

B. *Boot time*

System boot time is key to several MCSs, in particular for automotive. This is, however, an overlooked property in many hypervisors, in particular the ones that are open-source.

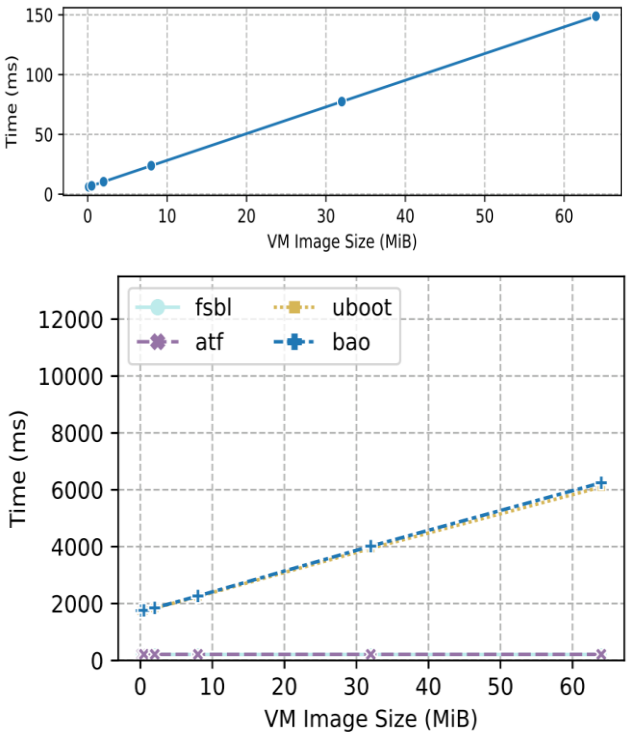

Figure 2. Hypervisor boot time (top) and System boot time (bottom) for a Bao-based system by VM image region size.

To provide empirical evidence about the boot time, we measured the hypervisor boot time and the full system boot time with Bao, targeting a Xilinx ZCU104 platform (Zynq UltraScale+). The platform boot flow starts by executing ROM code in two co-processors, which load the first-stage bootloader (FSBL) and start the main Cortex-A CPUs. These boot stages setup the raw platform infrastructure (e.g., clocks, DRAM) and load the Arm Trusted Firmware (TF-A) and U-boot. U-boot will then load the hypervisor and the guest images. Bao boot guests right after the initialization.

Figure 2 depicts the hypervisor boot time (top) and the system cumulative time for each boot stage (down), i.e., FSBL (fsbl), TF-A (atf), U-boot (uboot), and Bao (bao). The hypervisor boot time is dependent on the VM and how it is configured. We observed that the VM image size impacts the hypervisor boot time. However, there are other factors at stake, such as specific hypervisor features/drivers/modules (e.g., health monitoring), communication channels, scheduling schemes, and others [17]. We measure boot time by varying the size of the VM image and keeping other configuration parameters (e.g., size of VM memory) intact. Thus, to understand the overhead of the hypervisor in the context of the complete boot flow, we also measure the full system boot time. Here, we can confirm that the bulk of boot time is spent by U-boot. From a macro perspective, the hypervisor adds an almost constant offset to U-boot's boot time. This results are in line on what was previously disclosed for other experts in the field, in the context of commercial hypervisors [17].

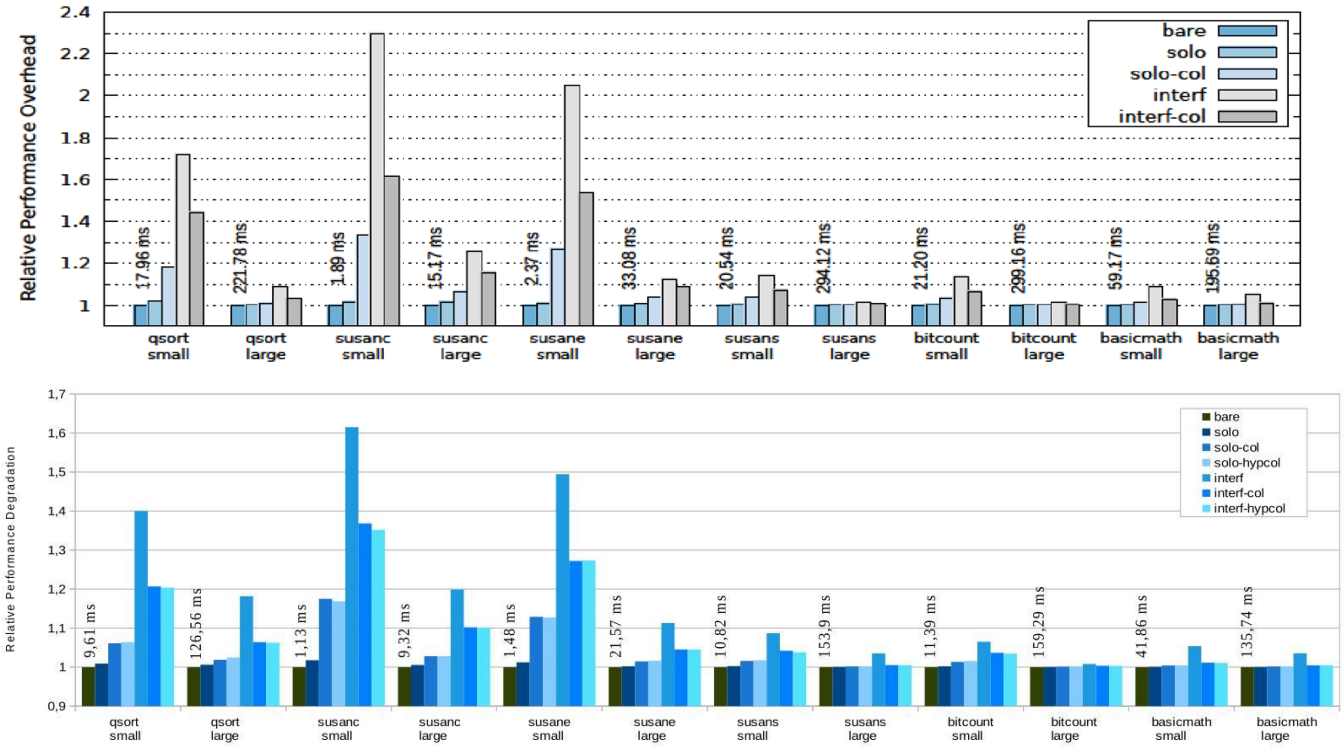

Figure 3. Bao hypervisor relative performance degradation/overhead for the MiBench Automotive and Industrial Control Suite for different configuration scenarios: Armv8-A (top) [3] and RISC-V (bottom) [7].

All in all, the major bottleneck for the VM boot time is caused by the bootloader, not the hypervisors. We stress the need for new boot mechanisms that speed up the boot process, in particular for lightweight critical VMs.

C. *Temporal Interference*

Embedded virtualization has several proven benefits but still some concerns while providing real-time and freedom-from-interference guarantees. It is true that virtualization already provides a reasonably high degree of time and space encapsulation and isolation of VMs by time-multiplexing resources such as the CPU, partitioning memory, and assigning or emulating devices. Notwithstanding, partitioning and multiplexing of micro-architectural shared system resources were, until recently, neglected by most hypervisors. This led to contention and a lack of truly temporal isolation, hurting determinism by increasing jitter. Although AMP hypervisors with VMs pinned to dedicated cores already remove part of this contention when compared to single-core or symmetric multiprocessing (SMP) implementations, systemwide resources such as last-level caches (LLCs), memory controllers, and interconnects remain shared and subject to contention. This is aggravated as mechanisms such as cache replacement, cache coherency, hardware prefetching, or memory controller scheduling focus mainly on performance and bandwidth maximization. There are existing techniques that address this issue, which we highlight cache coloring and memory bandwidth reservation [11],[13-14].

Figure 3 provides evidence about the impact on performance due to interference on Arm (top) and RISC-V (bottom). These graphs were borrowed from our previous works [3],[7]. The system configuration and all the replicability information can be found in [3] and [7]. Analyzing the figures, it is possible to draw a few observations. First, when coloring is enabled (solo-col), the performance overhead is aggravated. This is explained by the reduced L2 cache size (due to colors), and that coloring precludes the use of superpages, significantly increasing TLB pressure (in line with results from Section III.A 4KiB page size, but slightly aggravated due to reduced L2 cache size). Second, in the interference scenario (interf), there is significant performance degradation. Also, we can see that cache partitioning through coloring can reduce interference; however, it is not optimal, since it is still higher in the interf-col than the solo-col scenario. This can be explained by the still not address contention introduced downstream from LLC (e.g. write-back buffer, MSHRs, interconnect, memory controller).

As conclusion, we can argue that multicore memory hierarchy interference significantly affects guest performance. Cache partitioning via page coloring helps, but is not the perfect solution; despite fully eliminating inter-core conflict misses, it does not fully mitigate interference. This is related to the fact that there is also significant contention downstream in the memory hierarchy, e.g., interconnects, memory controller, or even in the internal LLC structures.

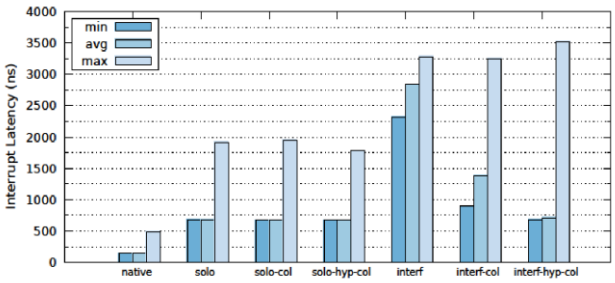

Figure 4. Bao Hypervisor interrupt latency for different configuration scenarios [3].

D. Interrupt Latency

In MCS, interrupt latency is a key requirement. Notwithstanding, as aforementioned, the existing GIC (and PLIC as well) virtualization support is not ideal for MCS: hypervisors have to handle and inject all interrupts and must actively manage list registers (GIC) when the number of pending interrupts is larger than the physical list registers. To provide empirical evidence about the interrupt latency in embedded / static partitioning hypervisors, we resorted to the results from our previous work on Bao [3]. Results translate the interrupt latency for a bare-metal custom application that continuously sets up the architectural timer to trigger an interrupt every ten milliseconds. Since the instant at which the interrupt is triggered is well-known, the interrupt latency is calculated as the difference between the expected wall-clock time and the actual instant it starts handling the interrupt.

Figure 4 depicts the Bao interrupt latency for different configuration scenarios: (i) native execution of the custom application (native); (ii) standalone hosted execution of the custom application atop Bao (solo); (iii) standalone hosted execution of the custom application with cache coloring enabled (solo-col); (iv) standalone hosted execution of the custom application with cache coloring enabled for VMs and hypervisor (solo-hyp-col); (v) hosted execution of the custom application atop Bao under the interference of other "stress" application (interf); (vi) hosted execution of the custom application atop Bao under the interference of other "stress" application with cache coloring enabled (interf-col); and (vii) hosted execution of the custom application atop Bao under the interference of other "stress" application with cache coloring enabled for VMs and hypervisor (interf-hyp-col).

When comparing native with the standalone hosted execution, we observe a significant increase in both average latency and standard deviation. This is due to the already anticipated GIC virtualization overhead, i.e., the trap and mode crossing costs and interrupt management/re-injection. It is also visible that coloring, per se, does not significantly impact average interrupt latency, but slightly increases the worst-case latency. The results also confirm the expected adverse effects of interference (interf) by cache and memory contention in interrupt latency, especially in the worst case. Enabling coloring under interference (interf-col) does not reduce the latency to the values reported in the standalone hosted execution setup (solo). This is due to the interference between guests and the hypervisor itself, i.e., corroborated by the lastest bars, i.e. the interf-hyp-col scenario.

Direct Interrupt Injection. Direct interrupt injection is a novel technique implemented in Arm-based hypervisors to eliminate the need for the hypervisor mediating interrupt injection. With this technique, the hypervisor passes through the physical GIC CPU interface and routes all interrupts directly to the VM while still emulating the (shared) distributor. This allows physical interrupts to be directly delivered to the VM with no hypervisor intervention. This is not a major issue as hypervisors do not directly manage devices. To communicate internally the hypervisor leverages the SDEI to register an event with EL3 firmware during initialization, which will then map it to its own dedicated secure IPIs. It can then trigger the event by issuing an SMC, making the firmware will divert execution to a predefined hypervisor handler. In one of our most recent works [11], we provide empirical evidence about the effectiveness of the technique in reducing interrupt latency to near to native values.

IV. Virtualization Tomorrow: Challenges and Opportunities

In this section, we discuss a set of challenges and opportunities for hypervisors targeting MCS.

A. Stronger and Holistic Isolation

Cache coloring does not fully mitigate the effects of interference. Furthermore, coloring has inherent inefficiencies such as (i) precluding the use of superpages and (ii) internal memory fragmentation. To further improve determinism and minimize coloring bottlenecks, there are other COTS-applicable contention mitigation mechanisms, e.g., memory bandwidth regulation (PMU-based CPU throttling) [14] and DRAM bank coloring [13]. We also stress the importance of including support for novel hardware extensions such as MPAM and call for platform designers to include such facilities in their upcoming designs targeting MCS. We also advocate for the RISC-V community to specify similar hardware facilities.

Another overlooked topic related to interference is the general lack of support in embedded and static partitioning hypervisors for managing traffic from peripheral DMAs. We advocate that SPH must provide contention mitigation mechanisms at the platform level, e.g., (i) leveraging QoS hardware available on the bus and (ii) controlling interference from DMA-capable devices or accelerators. This is more problematic as platforms become increasingly heterogenous (e.g., GPUs, TPUs, FPGA). Furthermore, since IOMMU structures (e.g., TLBs) are still shared between multiple DMA masters controlled by different VMs, we hypothesize that bandwidth regulation techniques may fall short while mitigating interference originating from these structures.

B. "Near-native" Interrupt Latency

The impact on interrupt latency in Arm-based (MCS) hypervisors is primarily due to inadequate hardware support in GICv2/3. The GICv4 will introduce support for direct interrupt injection to VMs; however, this feature is only implemented for inter-processor interrupts (IPIs) and message-signaled interrupts (MSIs). We stress the attention of Arm silicon makers and designers to provide additional hardware support at the GIC

level for direct interrupt injection. The same holds for RISC-V. The PLIC does not provide any sort of support for interrupt virtualization and the new AIA specification seems to be mainly driven by high-performance computing (HPC) requirements (where interrupt latency is not so critical such as in MCSs).

Besides the need for additional hardware support, we argue system designers need to revisit the implementation of the interrupt injection paths in (open-source) hypervisors for MCSs [11]. Finally, Bao and Jailhouse implement the direct interrupt injection technique; however, we must stress that using this technique severely hinders the ability of the hypervisors to manage devices or implement any functionality dependent on interrupts. A plausible research direction would be a hybrid approach, i.e., selectively enabling specific cores for critical guests while providing the more complex functionality in cores running non-critical guests.

C. Novel Architectures

Embedded hypervisors for MCS (in particular, static partitioning hypervisors) should be as small and simple as possible while providing the necessary partitioning mechanisms. Ideally, it would be only a thin layer of software for configuring hardware virtualization partitioning mechanisms at boot time. However, several factors hinder this goal in Arm (and RISC-V) processors, including the need to inject interrupts or emulate firmware services such as PSCI. From a more realistic perspective, pure static partitioning hypervisors such as Bao and Jailhouse are often too strict, especially from a resource utilization perspective. Also, there might be the need to share devices or securely multiplex access to indivisible resources (e.g., clocks).

A hybrid, more flexible approach is to have the system resources statically partitioned while dynamically multiplexing or emulating the resources in one of the partitions (e.g., scheduling multiple vCPUs in a single CPU). Guaranteeing isolation and freedom from interference between these two domains would still fit MCS requirements, while providing a more adaptable interface. This is possible in a hypervisor such as Xen with the use of privileged Dom0, but there is no clear separation between partitioning and virtualization functions. In this vein, we argue that these two concepts are often conflated and implemented in a monolithic approach. Naturally, microkernels such as the seL4 virtualization design, but with the expected resulting impact on latency for VMs. All in all, we advocate for novel architectures that combine both the flexibility and robust fault encapsulation of microkernels with the simplicity and minimalist latencies of static partitioning hypervisors.

D. CHERI & Virtualization

The use of software capabilities to implement fine-grain control access is not a new endeavor. Microkernels have been using software capabilities as a mechanism to enforce security for decades [16]. Notwithstanding, hardware capabilities just recently started to raise some interest with the Capability Hardware Enhanced RISC Instructions (CHERI) architecture. CHERI extends conventional ISAs with a new type of hardware-supported data, the architectural capability [18]. CHERI was first implemented for the MIPS architecture but then spanned to Arm (CHERI-ARM) and RISC-V (CHERI-RISC-V). Significant effort has been made to understand and evaluate the implications of CHERI on compatibility, performance, and security for off-the-shelf software stacks. In particular, CheriRTOS have demonstrated the applicability of the CHERI architecture to deeply embedded systems. In 2019, the UK government, partnering with key industry players (i.e., Arm, Microsoft, and Google), decided to invest in the Digital Security by Design (DSbD) program aiming at bringing CHERI from prototype to mainstream (e.g., Arm Morello).

Despite the ongoing academic and industry-related efforts built atop the CHERI technology, very little has been done on the use of CHERI for virtualization. Recently, CAP-VMs [19] introduced a new VM-like abstraction that leverages hardware support for CHERI memory capabilities for secure isolation; however, this targets cloud applications. From a research point of view, it is still an open challenge to understand how CHERI can be used and interact with embedded virtualization environments targeting mixed-criticality systems. An interesting path would be to design or extend an open-source embedded / static partitioning hypervisor (e.g., Bao) to (i) compile as hybrid-capability or pure-capability code and (ii) host CHERI-aware VMs. This would provide a compelling common ground to start investigating, for example, the trade-offs between the hybrid and pure approaches in the context of MCSs.

E. Other Directions

The list discussed so far is not meant to be exhaustive. The discussed topics are representative of a set of challenges and opportunities existing in the hypervisor for MCSs, and we argue that may be representative of the ones with more impact in the overall field. Notwithstanding, there are other topics of relevant importance, including (but not limited to): (i) support for emerging virtualization processors based on dual-stage MPU such as the new Armv8-R Cortex-R real-time processors (e.g., Cortex-R52) [20]; the need for strong security guarantees (e.g., timing side-channels, TEE support, control flow integrity, VM introspection); (ii) the compliance of functional safety standards (mostly in open-source embedded hypervisors); (iii) the implementation of communication primitives, in particular leveraging open standards such as VirtIO and OpenAMP; and (iv) even the use of novel cost-effective formal verification techniques [21-22] to very hypervisor and firmware (e.g., Arm TF-A, RISC-V OpenSBI).

V. Conclusion

There is a common erroneous belief that hypervisors, per se, are magic bullets that guarantee the safety certification of MCSs. In this paper, we provided a comprehensive picture of the use of virtualization in the context of mixed-criticality systems (MCSs). While sharing evidence collected in more than a decade of experience in the field, we highlighted the main problems and limitations currently affecting existing hypervisor solutions and (ii) discussed a set of open research and industrial challenges and opportunities.

Acknowledgment

This work is supported by (i) FCT – Fundação para a Ciência e Tecnologia within the R&D Units Project Scope UIDB/00319/2020 and (ii) European Union's Horizon Europe research and innovation program under grant agreement No

101070537, project CROSSCON (Cross-platform Open Security Stack for Connected Devices).

REFERENCES

[1] Y. Xing and Y. Zhan, "Virtualization and Cloud Computing," Future Wireless Networks and Information Systems, Lecture Notes in Electrical Engineering, 2012.

[2] J. Cerrolaza et al., “Multi-Core Devices for Safety-Critical Systems: A Survey,” ACM Computing Surveys, 2020.

[3] J. Martins et al., “Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems,” in Workshop on NG-RES, 2020.

[4] José Martins and Sandro Pinto, “Shedding Light on Static Partitioning Hypervisors for Arm-based Mixed-Criticality Systems,” in Proc. Of RTAS, 2023.

[5] R. Uhlig et al., "Intel virtualization technology," in Computer, 2005.

[6] C. Dall and J. Nieh, "KVM/ARM: the design and implementation of the linux ARM hypervisor," in Proc. of ASPLOS, 2014.

[7] B. Sa et al., “A First Look at RISC-V Virtualization from an Embedded Systems Perspective,” IEEE Transactions on Computers, 2021.

[8] B. Sa et al., “CVA6 RISC-V Virtualization: Architecture, Microarchitecture, and Design Space Exploration,” arXiv:2302.02969, 2023.

[9] J. Hwang et al., "Xen on ARM: System Virtualization Using Xen Hypervisor for ARM-Based Secure Mobile Phones," in IEEE Consumer Communications and Networking Conference, 2008.

[10] R. Ramsauer et al., “Look Mum, no VM Exits!(Almost),” in Proc. of Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), 2017.

[11] José Martins and Sandro Pinto, “Shedding Light on Static Partitioning Hypervisors for Arm-based Mixed-Criticality Systems,” in Proc. Of RTAS, 2023.

[12] R. West et al., "A Virtualized Separation Kernel for Mixed-Criticality Systems," ACM Transactions on Computing Systems, 34, 3, 2016.

[13] T. Kloda et al., “Deterministic Memory Hierarchy and Virtualization for Modern Multi-Core Embedded Systems,” in Proc. of Real-Time and Embedded Technology and Applications Symposium (RTAS), 2019.

[14] H. Yun et al., “MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms,” in Proc. of Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013.

[15] T. Moscibroda and O. Mutlu, “Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems,” in Proc. of USENIX Security Symposium, 2007.

[16] Q. Ge et al., “Time Protection: The Missing OS Abstraction,” in Proc. of European Conference on Computer Systems (EuroSys), 2019.

[17] E. Hamelin et al., “Selection and evaluation of an embedded hypervisor: application to an automotive platform”, in Proc. of European Congress on Embedded Real Time Systems, 2020.

[18] R. Watson et al., "CHERI: A hybrid capability-system architecture for scalable software compartmentalization," IEEE Symposium on Security and Privacy, 2015.

[19] V. Sartakov et al., "CAP-VMs: Capability-Based Isolation and Sharing in the Cloud," in Proc. of OSDI, 2022.

[20] P. Austin et al., “Best Practices for Armv8-R Cortex-R52+ Software Consolidation,” White Paper, 2022.

[21] R. Gu et al., "CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels", in Proc. of OSDI, 2016.

[22] S. Li at al., "A Secure and Formally Verified Linux KVM Hypervisor," in Proc. of IEEE Symposium on Security and Privacy (SP), 2021