Shedding Light on Static Partitioning Hypervisors for Arm-based Mixed-Criticality Systems

José Martins
Centro ALGORITMI/LASI, Universidade do Minho
jose.martins@dei.uminho.pt

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

***Abstract*—In this paper, we aim to understand the properties and guarantees of static partitioning hypervisors (SPH) for Arm-based mixed-criticality systems (MCS). To this end, we performed a comprehensive empirical evaluation of popular open-source SPH, i.e., Jailhouse, Xen (Dom0-less), Bao, and seL4 CAmkES VMM, focusing on two key requirements of modern MCS: real-time and safety. The goal of this study is twofold. Firstly, to empower industrial practitioners with hard data to reason about the different trade-offs of SPH. Secondly, we aim to raise awareness of the research and open-source communities to the still open problems in SPH by unveiling new insights regarding lingering weaknesses. All artifacts will be open-sourced to enable independent validation of results and encourage further exploration on SPH.**

***Index Terms*—Virtualization, Static Partitioning, Hypervisor, Mixed-Criticality, Arm.**

I. Introduction

The explosion in the number of functional requirements in industries such as automotive has led to a trend for centralized architectures that consolidate heterogeneous software stacks in high-performance platforms [1], [2]. These typically take the form of mixed-criticality systems (MCSs) [3], [4] as they often integrate safety- or mission-critical workloads with real-time requirements, alongside Unix-like operating systems (OSs) providing rich functionality. Virtualization technology is the *de facto* enabler for these architectures as, by definition, it allows for consolidation with strong fault encapsulation. In this context, hypervisor design must balance, on one side, minimality for safety and security, and feature-richness and efficient sharing of resources on the other. While traditional hypervisors were optimized for the latter [5], [6], on the opposite end of the spectrum we have static partitioning hypervisors (SPH) specifically designed for MCS [7], [8]. Besides statically assigning system resources (e.g., CPUs, memory, or devices) to virtual machines (VMs), SPH must provide latency and isolation guarantees at the microarchitectural level to comply with the freedom from interference requirements of industry safety standards such as ISO 26262 [1], [9]–[11].

In this paper, we shed light on open-source SPH for Arm-based MCS. Despite the existence of multiple reports, research papers, and public artifacts, information on these systems tends to be scattered or focus on a single hypervisor or metric, while, in some cases, empirical evidence is non-existent. Thus, it is difficult to obtain a comprehensive understanding of the overall properties and guarantees of these systems in the context of MCS. To fill this gap, we conduct a leveled playing-field evaluation of four open-source static partitioning virtualization solutions, i.e., Jailhouse, Xen (Dom0-less), Bao, and the seL4 CAmkES virtual machine monitor (VMM). We drive our study based on two key requirements of modern MCS, i.e., real-time and safety, focusing on (i) performance, (ii) interrupt latency, (iii) inter-VM communication, (iv) boot time, and (v) code size. For each metric, we assess the effectiveness of the cache coloring technique, pervasive in SPH, for inter-VM interference mitigation.

The goal of this study is twofold. Firstly, we aim at empowering industrial practitioners with hard data to understand the trade-offs and limits of modern Arm-based SPH as well as how best configure these systems for their use case and requirements. For example, the use of superpages significantly decreases the number of TLB misses, resulting in negligible performance overhead (<1% without interference), but it is precluded by enabling page coloring, a widely adopted cache partitioning technique in these hypervisors. Also, experiments demonstrated that coloring, per se, can impact the performance up to 20% and that it cannot fully mitigate interference, where overhead can still reach up to 60%. Regarding inter-VM communication, we show that for bulk data transfers, buffer size choice is crucial to maximize throughput while avoiding degradation due to inter-VM interference.

Secondly, with the collected empirical data, we aim at raising awareness of the research and open-source communities to the still open problems in SPH, by highlighting both new and previously known weaknesses for these SPH, which seem to be mostly due to interrupt virtualization issues. Prominent examples include: (i) the need for implementing state-of-the-art mechanisms to fully mitigate inter-VM interference (e.g., memory throttling) in mainstream SPH; (ii) the extent of the impact of interference on interrupt latency, which can increase by several orders of magnitude; (iii) the lack of support for correctly handling and delivering interrupts in priority order; (iv) the absence of mechanisms that prioritize the boot of a critical VM; and (v) the lack of plasticity of the SPH architecture which might hinder achieving its own goal of allowing full IO passthrough. To address observed shortcomings, we discuss potential solutions and research directions.

We made all artifacts openly available [12] to enable practitioners and researchers with the methods and materials to (i) independently replicate all experiments and corroborate assessed results, as well as (ii) encourage and facilitate additional experiments and further exploration of SPH.

In summary, this paper makes the following contributions: (1) presents the most comprehensive empirical study to date on popular open-source SPH focusing on a set of key metrics for modern MCS; (2) provides hard empirical data to empower industrial practitioners with the knowledge to understand the limits and trade-offs of SPH; (3) raises awareness of the research and open-source communities to the open problems of SPH by shedding light on their shortcomings; and finally, (4) opens all artifacts to enable independent validation of results and facilitate further research.

II. Background

In this section, we start by overviewing Armv8-A virtualization support. We then explain the concept of static partitioning virtualization, including key techniques implemented in SPH. Finally, we describe Xen, Jailhouse, Bao, and seL4 CAmkES.

A. Arm Virtualization

CPU & Memory. Given the widespread proliferation of virtualization in the last decades, Arm implemented hardware support since version 7 of the ISA. The most recent version of the architecture, i.e., 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) [13] where guests execute. A hypervisor running at EL2 has fine-grained control over which CPU resources are directly accessible by guests (e.g., control registers). Access to a denied functionality by a guest OS results in a trap to the hypervisor. It is possible to route specific guest exceptions and system interrupts to EL2. Other resources that can be managed by the hypervisor include the CPU-private generic timer and the performance monitor unit (PMU). EL1/EL0 memory accesses are subject to a second stage of translation which is in full control of the hypervisor [13]. Any guest access to a memory region not mapped in the second stage of translation will result in a precise trap to EL2. Arm provides multiple "translation granules", resulting in pages of different sizes: 4 KiB, 16 KiB, and 64 KiB. For each page size it is also possible to map large contiguous memory regions. These are known as superpages (or hugepages), which reduces TLB pressure. The more commonly used 4KiB granule allows for 1GiB and 2MiB superpages. Arm also defines the System Memory-Management Unit (SMMU), that extends memory virtualization mechanisms from the CPU to the bus, to restrict VM-originated direct-memory accesses (DMAs).

Interrupts. Arm virtualization acceleration spans the full platform, including the Generic Interrupt Controller (GIC). The GICv2 standard has two main components: a central distributor and a per-core interface. All interrupts are routed first to the distributor, which then forwards them to the interfaces. The distributor allows the configuration of interrupt parameters (e.g., priority, target CPU) and the monitoring of interrupt state, while the interface enables the core management of interrupts. GICv2 provides virtualization support only on the interface; there is a fully virtual interface with which the guests can directly interact without VM exits. The distributor, however, must be fully emulated. Furthermore, all interrupts must first be handled by the hypervisor, which can then inject them in the VM, by writing to GIC list registers (LRs). These registers essentially take the place of the distributor for the virtual interface: when a given interrupt (along with metadata such as priority or state) is present on a register, it is forwarded to the virtual interface. The GICv2 spec limits the number of LRs to a maximum of 16. GICv3 and GICv4 provide support for direct delivery of hardware interrupts to VMs; however, this feature is only implemented for inter-processor interrupts (IPIs) and message-signaled interrupts (MSIs), i.e., interrupts implemented as write operations to special interrupt controller registers and propagated via the system interconnect. Standard wired interrupts, propagated by dedicated side-band signals, are still subject to the mentioned limitation, i.e., hypervisor interrupt injection through the list register.

B. Static Partitioning Virtualization (SPV)

Static partitioning is the practice of, either at build or initialization time, distributing all platform resources to different subsystems. This can be materialized in many shapes and forms, depending on the hardware primitives. Virtualization is a natural enabler for the static partitioning architecture, due to the strong encapsulation guarantees and flexible resource assignment. Hypervisors designed for the static partitioning use case (or providing such a configuration) have three fundamental properties: (i) exclusive assignment of virtual CPUs to physical CPUs (i.e., no scheduler); (ii) static allocation, assignment, and mapping of all hypervisor and VM memory at build or initialization time; and (iii) direct assignment of devices to VMs (passthrough) and exclusive allocation of their interrupts to the same VM. To implement this efficiently, these hypervisors are highly dependent on virtualization hardware support both at the CPU and platform level (e.g., SMMU). SPH also have non-functional requirements centered around minimizing interrupt latency and inter-VM interference. Thus, over the past few years, there have been efforts to enhance SPH with mechanisms to address these requirements. These include cache coloring and, analogously to what has been done for x86 [14], direct injection in Arm processors. Furthermore, it is important for the code base to be minimal and follow industry coding standards (e.g., MISRA); this eases functional safety (FuSa) certification efforts.

Cache Coloring. In SPH, VMs still share microarchitectural resources such as the last-level cache (LLC). The behavior and memory access pattern of one VM might result in the eviction of another VM's cache lines, impacting the latter's hit rate and consequently its execution time. Thus, there is the need to partition shared caches assigning each partition to a different VM. While in the past Armv7 processors provided hardware means to apply this partitioning by way of per-master cache-locking, modern-day Arm CPUs do not provide those facilities. A solution is cache coloring, a software technique for index-based cache partitioning [15]. Cache coloring explores the intersection of the virtual addresses' cache index and the page number when creating virtual-to-physical memory mappings. Each color is a specific bit pattern in this intersection that maps only to specific cache sets. Thus, hypervisors can control which cache sets are assigned to a given VM by selecting which physical pages are mapped to it. By exclusively assigning a cache partition (i.e., group of cache sets or colors) to a given VM, cache coloring fully eliminates the conflict misses resulting from inter-VM contention. Cache coloring can also be implemented at the hypervisor level by assigning the hypervisor one or more color(s).

Direct Interrupt Injection. Direct interrupt injection is a new technique implemented in Arm-based SPH to eliminate the need of 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 by configuring the CPU to trigger interrupt traps directly at EL1, i.e., kernel mode. The hypervisor must still emulate the shared distributor to ensure isolation between VMs, i.e., prevent misconfiguration of a given VM interrupts by another VM. This allows physical interrupts to be directly delivered to the VM with no hypervisor intervention, reducing latency to native execution levels. The forfeiting of interrupts should not be a major issue as SPH do not directly manage devices. However, SPH still need to communicate internally using IPIs. Direct interrupt injection implementations address this issue by leveraging standard software-delegated exception interface (SDEI) [16] events instead of directly using IPIs. SDEI is implemented by firmware, allowing the hypervisor to register an event during initialization. The hypervisor can then trigger the event by issuing a system call to firmware (via a secure monitor call instruction, SMC), which will result in diverting execution to a predefined hypervisor handler, similarly to Unix signals. In reality, firmware maps these events to its own secure reserved IPIs since, as part of TrustZone [16], the GIC provides further facilities to reserve interrupts to EL3.

C. Static Partitioning Hypervisors (SPH)

Jailhouse Hypervisor. Jailhouse [7], [17] is an open-source hypervisor developed by Siemens. Unlike traditional baremetal hypervisors, Jailhouse leverages the Linux kernel to boot and initialize the system and uses a kernel module to install the hypervisor. Once Jailhouse is activated, it runs as a baremetal component, taking full control over the hardware. Jailhouse has no scheduler and only leverages the ISA virtualization primitives to partition hardware resources across multiple isolated domains, a.k.a. "cells". Guest OSes or baremetal applications running inside cells are called "inmates". The mainline includes support for x86 and Armv7/8-A, and a work-in-progress RISC-V port [18]. The research community has been actively contributing with mechanisms to enhance predictability, namely: cache coloring, DRAM bank partitioning [19], memory throttling, and device quality of service (QoS) regulation [20]. An unofficial fork including these features is available [21]. Direct injection [22] was also implemented.

Xen (Dom0-less) Hypervisor. Xen [5] is an open-source hypervisor widely used in a broad range of application domains. A key distinct feature of Xen is its dependency on a privileged VM (Dom0) that typically runs Linux, to manage non-privileged VMs (DomUs) and interface with peripherals. Xen was initially designed for servers and desktops, but has found also adoption on embedded applications. For embedded and automotive applications, Xilinx has led the implementation of Xen Dom0-less. With this novel approach, it is possible to have a Xen deployment without any Dom0, booting all guests directly from the hypervisor and statically partitioning the system. A patch for guest and hypervisor cache coloring support [23] is available. There is also a SIG working towards facilitating downstream FuSa certifications by fostering multiple initiatives within the community including MISRA refactoring, or providing the option of running Zephyr [24] as Dom0. Besides Armv8-A, Xen also supports x86, and Armv8-R and RISC-V ports are underway.

Bao Hypervisor. Bao [8] is an open-source static partitioning hypervisor that was made publicly available in 2020. It implements the pure static partitioning architecture, i.e., a minimal, thin-layer of privileged software which leverages the existing ISA virtualization primitives to partition the hardware. Bao has no scheduler and does not rely on any external libraries or privileged VM (e.g., Linux), consisting on a standalone component which depends only on standard firmware to initialize the system and perform platform-specific tasks such as power management. Bao originally targeted Armv8-A [8]. The mainline now includes support for RISC-V [25], Armv7-A, and Armv8-R ports are in the making. Bao was specifically designed to provide strong real-time and safety guarantees. It implements hardware partitioning mechanisms to guarantee true freedom from interference, i.e., cache coloring (VM and hypervisor), and direct interrupt injection. There are ongoing efforts to implement memory throttling.

seL4 CAmkES VMM. seL4 is a formally verified microkernel [26]. Its design model revolves around the use of capabilities. When used as a hypervisor, seL4 executes in hypervisor mode (e.g, EL2) and exposes extra capabilities and APIs to manage virtualization functionality [27]. A user-level VMM uses its resource capabilities to create VMs. As of this writing, only the seL4 CAmkES VMM [28], [29] code is open-source. Each CAmkES VMM manages a single VM. One current issue of the CAmkES VMM is that, although it supports multicore VMs, each VMM runs as a single thread pinned to a single CPU. seL4 supports x86, Armv7/8-A and RISC-V, but the latter is not supported by CAmkES VMM. In CAmkES, resources are statically allocated to each component using capabilities. Originally, seL4 provided only a priority-based preemptive scheduler. The newest MCS kernel extends it with scheduling context capabilities, allowing time management policies to be defined in user space [30]. Cache coloring has also been implemented in seL4 [31], not only at the user/VM level, but also for the kernel, but it was not publicly available at the time of writing. seL4 has formal proofs for its specification, implementation from C to binary, and security properties [32], [33]. There are also ongoing efforts to extend the formal verification to prove the absence of covert timing channels [34]. Finally, CAmkES is being deprecated in the near future in favor of the seL4 Core Platform (seL4CP) [35]. seL4CP

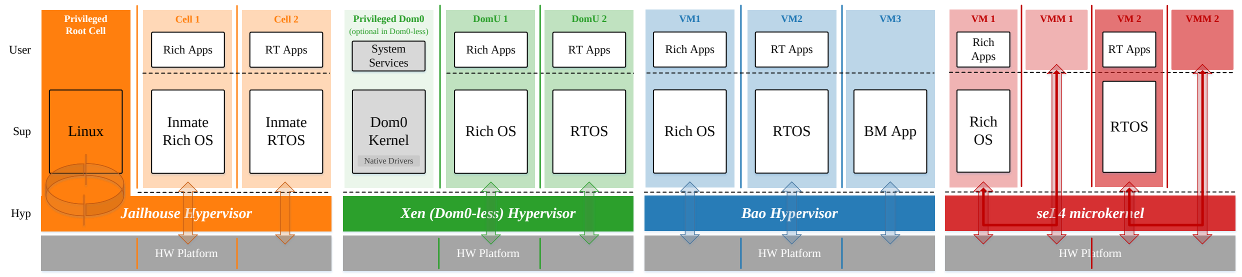


Fig. 1: Architectural overview of the assessed hypervisors: Jailhouse, Xen (Dom0-less), Bao and seL4 CAmkES VMM

will also provide support for per-VM user-mode VMMs[1] while promising to alleviate the performance overhead of CAmkES.

III. Methodology and Experimental Setup

A. Methodology

Selected Hypervisors. We have selected four open-source SPH (Fig.1). Jailhouse and Bao were designed for the static partitioning use case; both are open-source and target Arm platforms. Xen Dom0-less is a novel deployment that allows directly booting multiple VMs (bypassing Dom0) and passthrough of peripherals to VMs. Finally, seL4 is a well-established open-source microkernel, which can be used as a hypervisor in combination with a user-level VMM. The seL4 CAmkES VMM is an open-source reference VMM implementation with static allocation of resources. These systems are actively maintained, adopted for commercial purposes, and there is a fair amount of information about them. We have excluded other open-source SPH that do not support Armv8-A (e.g., Quest-V, ACRN), and other popular open-source hypervisors that don't explicitly target static partitioning (e.g., KVM, Xvisor). We have excluded microkernels such as NOVA [36] due to the lack of availability of an open-source reference user-space VMM, and because we believe seL4 serves as a faithful representative of the microkernel architecture. TrustZone-assisted hypervisors [37]–[39] were left out due to multicore scalability issues and lack of active maintenance. Finally, we have excluded commercial products (e.g., PikeOS, LynxSecure) as these often require licenses the authors did not have access to (hindering the sharing of artifacts, which we believe is crucial for the viability of this study).

Empirical Evaluation. The evaluation focuses on performance, interrupt latency, inter-VM communication latency and bandwidth, boot time, and code size. We also assess the effect of interference and of the available mitigation mechanism (i.e., cache coloring). Although we consider virtual device performance, IO interference, and applied security techniques such as stack canaries or guards, data execution prevention or control-flow integrity very relevant, these are out of scope of this work. We advocate for a follow up study as future work.

B. Experimental setup

Hardware Platform. Experiments were carried out on a Xilinx ZCU104, featuring a Zynq Ultrascale+ SoC. It includes a quad-core Cortex-A53 running at 1.2 GHz, a GIC-400 (GICv2) featuring 4 list registers, and an MMU-500 (SMMUv2). Cores have private 32KiB separate L1 instruction and data caches, and share a L2 1MiB unified cache. It also includes a programmable logic (PL) component (i.e., FPGA).

Hypervisors configuration. We made an effort to use the latest versions of each SPH. Still, we applied a few patches to Jailhouse, Xen, and Bao to include features such as coloring or direct injection, which are not yet fully merged. Further, we had to make small adjustments to all SPH to enable homogeneous configurations (e.g., uniforming VM memory map), allow direct guest access to PMUs, or instrumenting hypervisors for specific experiments. For each SPH, we leveraged the default configuration for the target SoC, with some tweaked options such as disabling debug and logging features. There were, however, specific adjustments that were made on a per-hypervisor basis. For example, to remove or minimize the invocation of a scheduler in Xen, we used the null scheduler and disabled trapping of wait-for-interrupt (WFI) instructions; in seL4, since it was not possible to disable the timer tick, we configured the tick with a period of about 5 seconds. We compiled all hypervisors with GCC 11.2, with the default optimization level defined by each hypervisor's build system. All these SPH configurations and modifications are available and clearly discernible in the provided artifact [12].

VM configuration. VM configurations are as similar as possible, mainly w.r.t. number of vCPUs and memory. For Jailhouse and seL4-VMM, where memory must be manually allocated, we set memory regions aligned to 2 MiB. The only device assigned to each VM is a UART. We evaluated two different classes of VMs: (i) large VMs running Linux (v5.14), as representative of rich, Unix-like OSs; and (ii) small VMs running baremetal applications or FreeRTOS (v10.4), as representative of critical workloads. When cache coloring is enabled, we assign half of the colors (four out of eight[2]) to the VM executing the benchmark, three colors to the interference application, and one color to the hypervisor (just supported in Bao and Xen). Note that color assignment configuration can significantly impact the final measurements for all metrics. In real deployments, the color assignment should be carefully defined based on the profile of the final system.

[1]Only after the bulk of this work was carried out, virtualization support in seL4CP was made openly available. At the time of writing, it still appears to be in a beta stage and not as mature as CAmkES.

[2]We consider only eight cache colors while, in truth, the target platform allows for 16. We do this to avoid color assignment configurations that would partition the L1 cache.

Interference Workload. When evaluating memory hierarchy interference, we use a custom baremetal guest which continuously writes a buffer with the size of the LLC (1MiB). Unless noted otherwise, this interference guest runs on a VM with two vCPUs. We stress that although parameterized to cause a significant level of interference, the observed effects cause by the interference workload do not necessarily reflect the worst case that could be achieved if further fine-tuned.

Measurement tools. We use the Arm PMU to collect microarchitectural events on benchmark execution. The selected events include instruction count, TLB accesses and refills, cache access and refills, number of exceptions taken, and number of interrupts triggered; we register the exception level on which these events occur. For the Linux VMs, we use the perf tool [40] to measure the time and to collect microarchitectural events. For baremetal or RTOS VMs, we use the Arm Generic Timer, with a resolution of 10 ns, and a custom PMU driver.

C. Threats to validity

Experiments were independently conducted by two researchers. Each used a different ZCU104 platform and pre-agreed VM configurations (cross-checked). We have contacted key individuals and/or maintainers as representatives of each SPH community. We have received replies from all of them, which led to a few iterations and repetition of some experiments. Overall, the comments and issues raised by these individuals are reflected in the presented ideas and results. Despite all efforts, these experiments may still be subject to latent inaccuracies. We will open source all artifacts to enable independent validation of the results. This study may also include limitations on the generalization to other platforms. For the hardware platform, we argue both the SoC (Zynq Ultrascale+) and the Cortex-A53 are representative of others used in automotive and industrial settings (e.g., NXP i.MX8 or Renesas R-Car M3). To corroborate this, we have also carried out the performance and interrupt latency experiments for the Bao hypervisor in an NXP i.MX8QM, which features the GIC-500 (GICv3). The obtained results are fully consistent with those presented in Sections IV and V. Furthermore, we argue next generation platforms, such as i.MX9 featuring Cortex-A55 CPUs, implement very similar microarchitectures.

IV. SPH: Performance

We start by assessing the performance degradation[3] of a single-core Linux VM atop each SPH. The main results are depicted in Figures 2, 3, and 4. We then evaluate the system under interference to understand the effectiveness of microarchitectural isolation mechanisms available in each SPH.

Selected Benchmark. We use the MiBench Embedded Benchmarks' Automotive and Industrial Control Suite (AICS) [41]. These benchmarks are intended to emulate the environment of embedded applications such as airbag controllers and sensor systems. Each test has two variants: *small* operates in a reduced input data set representing a lightweight use of the benchmark, while *large* operates over a considerable input data set, emulating a real-world application scenario.

[3]Performance degradation is the ratio between the total execution time of the benchmark running atop the hypervisors and native execution.

Base Performance Overhead. Fig. 2 presents the relative performance degradation for the MiBench AICS. For each benchmark, below the plotted bars, we present the average absolute execution time for the native execution. The first observation is that, independently of the hypervisor, different benchmarks are affected to different degrees. Secondly, Jailhouse, Xen, and Bao incur a negligible performance penalty, i.e., less than 1% across all benchmarks. Although seL4 CAmkES-VMM also presents a small overhead for most benchmarks, the overhead can reach up to 7%.

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, since it can, in the worst case, increase the number of memory accesses in a page-walk by a factor of four. Second, the overhead of trapping to the hypervisor and performing interrupt injection, e.g., timer tick interrupt. Additionally, the pollution of caches and TLBs by the hypervisor might also affect guest performance. To further understand the behavior of the benchmarks, in particular the larger overhead of the CAmkES-VMM, we have collected a number of microarchitectural events. Fig. 3 shows them normalized to the number of executed instructions. We highlight two events whose increase is highly correlated with the degradation observed: hypervisor L2 cache refills (Fig. 3a) and guest TLB misses (Fig. 3b), with Pearson correlation coefficients of up to 0.94 and 0.96, respectively.

An important hypervisor feature to minimize the impact of two-stage translation is to leverage superpages. By inspecting hypervisor code, we concluded that only CAmkES-VMM does not have support for 2MiB superpages. This justifies the higher number of TLB misses. Notwithstanding, to corroborate this argument, we have configured the other SPH to preclude the use of superpages. As expected, we observed an increase in the performance degradation (and TLB misses) similar to CAmkES-VMM (Fig. 4). We still observed a gap of up to 2% between CAmkES-VMM and the other SPH; this is related to the aforementioned interrupt handling and injection overheads, i.e., a consequence of the microkernel design: more costly switches between VM and VMM and a high number of VMM to microkernel calls for managing and inject the interrupts. This is confirmed by Figures 3c and 3d, which show the hypervisor-to-guest executed instruction ratio and the number of exceptions taken by the hypervisor, respectively. For these events, seL4 has a higher ratio when compared to the other SPH. We further investigate interrupt injection in Section V.

Takeaway 1. SPH do not incur in meaningful performance impacts due to: (i) modern hardware virtualization support; (ii) 1-to-1 mapping between virtual and physical CPUs; and (iii) minimal traps. However, one key aspect is that SPH must have support for / make use of superpages to minimize TLB misses and page-table walk overheads.

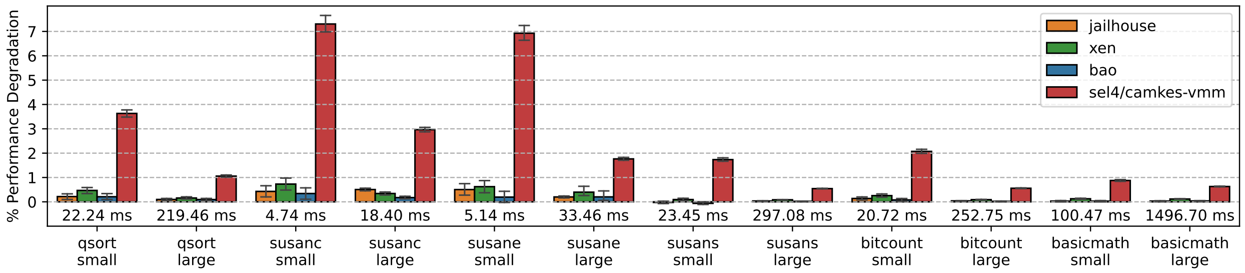


Fig. 2: Relative performance degradation for the MiBench Automotive and Industrial Control Suite.

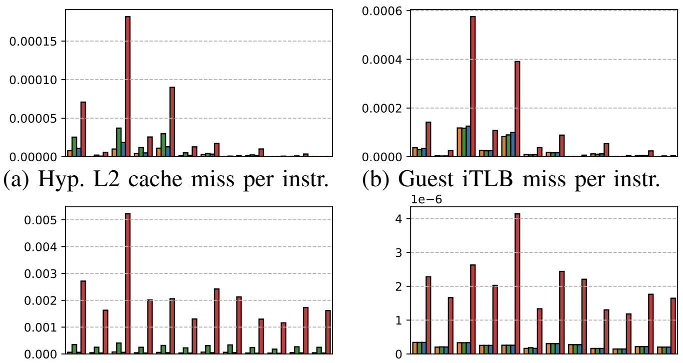


Fig. 3: MiBench AICS microarchitectural events.

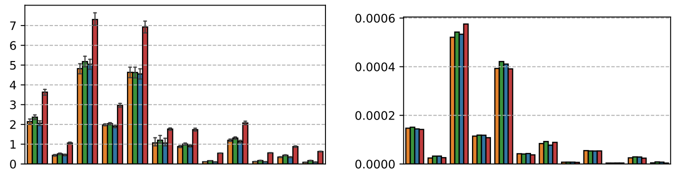


Fig. 4: MiBench AICS without the use of superpages on second-stage translation.

Performance under interference. We also evaluate inter-VM interference and the effectiveness of cache coloring at both guest and hypervisor levels. Fig. 5 plots the results under interference (+*interf*), with coloring enabled (+*col*), and with interference and coloring enabled (+*interf*+*col*). seL4 CAmkES VMM shows no results for coloring enabled as this feature is not openly available yet.

There are four conclusions to be drawn. Firstly, interference significantly affects the benchmark execution over all hypervisors. As expected, this is explained by a significant increase in L2 cache misses. On Jailhouse, Xen, and Bao performance is degraded by a similar factor, i.e., to a maximum of about 105%; seL4-VMM is more susceptible to interference, reaching up to 125% in the worst case. This pertains to the fact that, given that seL4-VMM executes a much higher number of instructions, the interference also impacts the execution of the hypervisor. Secondly, coloring, per se, significantly impacts performance (up to about 20%). This seems logical given that coloring (i) forces the hypervisor to use 4KiB pages, reducing TLB reach, and (ii) reduces the available cache space, which for working sets larger than LLC increases memory system pressure (i.e., L2 cache misses). Thirdly, coloring can only reduce interference but not completely mitigate it. In these experiments, the interference workload runs continuously. However, in a more realistic scenario, it might be intermittent. The improvement in predictability achieved by coloring is reflected in the difference between the base experiment results (bars in Fig. 2 and +*interf* in Fig. 5) and respective variants with coloring enabled (+*col* in Fig. 5). The lower the difference, the higher the predictability. For example, in the case of *susanc-small*, we observed that without coloring, the variation can go up to 105 percentage points (pp), while when coloring is enabled, the observed overhead is around 58%, which corresponds to a variation of 38 pp compared to the configuration with coloring enabled but without interference. Nevertheless, we observed that cache misses are essentially reduced to the same level as when coloring is enabled but without interference. Clearly, the observed interference is not only due to cache-line contention. There are points of contention at deeper levels of the memory hierarchy, e.g., buses and memory controller [42] or even in internal LLC structures [43]. Finally, results on Xen and Bao demonstrate that hypervisor coloring has no substantial benefit as it only reduces performance degradation due to interference by at most 1% (omitted due to lack of space).

Takeaway 2. Multicore memory hierarchy interference significantly affects guests' performance. Cache partitioning via page coloring is not a silver bullet as despite fully eliminating inter-core conflict misses, it does not fully mitigate interference (up to 38 pp increase in relative overhead).

V. SPH: Interrupt Latency

As discussed in Section II-A, the existing GIC virtualization support is not ideal for MCS: hypervisors have to handle and inject all interrupts and must actively manage list registers when the number of pending interrupts is larger than the physical list registers. This is of particular importance to guarantee the correct interrupt priority order which might be critical for an RTOS [44]. In this section, we investigate the overhead of each SPH in the interrupt latency, their susceptibility to interference, and the effectiveness of cache coloring. Then, we evaluate the direct injection technique and analyze interrupt priority support as well as virtual IPI latencies.

Methodology. To measure interrupt latency, we used a custom lightweight baremetal benchmark, which measures the latency of a periodic interrupt triggered by the Arm Generic Timer. The timer is programmed in auto-reload mode, to continuously

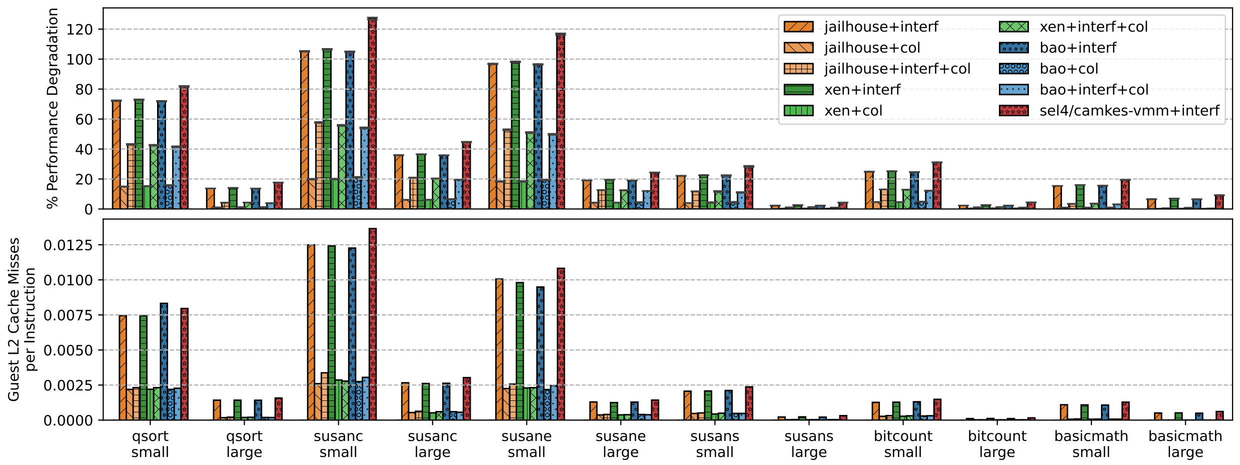


Fig. 5: Performance degradation and L2 cache misses per instruction for the Mibench AICS under interference and coloring.

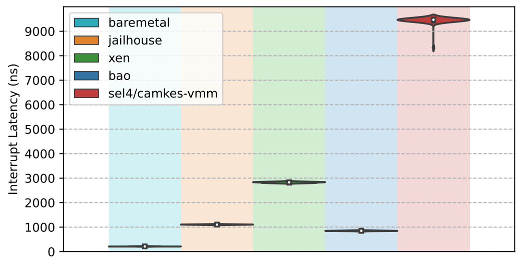


Fig. 6: Base interrupt latency.

trigger an interrupt at each 10 ms. The interrupt handler reads the value of the timer, i.e., it measures the time elapsed since the interrupt was triggered. Each measurement is carried out with cold L1 caches. To achieve this, after each measurement, we flush the instruction cache. During the 10 ms, we also prime the L1 data cache with useless data.

Base Latency. Fig. 6 depicts the violin plots for the custom benchmark running atop each SPH. From the baseline of about 200 ns, Bao and Jailhouse incur the smallest increase, albeit significant, to an interrupt latency of about 4x (840 ns) and 5x (1090ns), respectively. Xen shows an increase of about 14x (2800 ns). The variance observed in these three systems is negligible. The difference observed between Jailhouse/Bao and Xen is justified by the interrupt injection path being highly optimized in the former, while more generic in Xen. We confirmed this by studying the source code and assessing the number of instructions executed by each hypervisor on the interrupt handling and injection path: while Jailhouse and Bao execute around 200 instructions, Xen executes about 1050.

seL4-VMM presents the largest interrupt latency (47x, 9400 ns), an order of magnitude higher than Jailhouse and Bao. The variance of the latency is also affected. This can be explained by the interrupt handling and injection mechanism of a microkernel architecture. In the other SPH, each interrupt results in a single exception taken at EL2, where the interrupt is handled and injected in the VM; virtualization support is leveraged such that no further traps occur. In CAmkES VMM it results in four traps to the microkernel: (i) the first due to the interrupt that results in forwarding it as a message to the VMM; (ii) a system call from the VMM to inject the interrupt in the VM (i.e., write the list register); (iii) another to "reply" to the exception, resuming the VM; and (iv) a final one where the VMM waits for a message signaling a new VM event or interrupt, resulting in a final context-switch back to the VM. We have also concluded that seL4 does not use a GIC feature that would allow guests to directly deactivate[4] the physical interrupt, resulting in an extra trap.

Takeaway 3. Due to the lack of efficient hardware support for directly delivering interrupts to guests in Arm platforms, all SPH increase the interrupt latency by at least one order of magnitude. However, by-design, SPH such as Jailhouse and Bao are able to achieve the lowest latencies as they provide an optimized path for hardware interrupt injection.

Latency Under Interference. Fig. 7 shows the results for interrupt latency under interference, including the baseline results of Fig. 6 for relative comparison as *solo*. Analyzing the effects of VM interference on interrupt latency (*interf*), we observed that Bao latency increases to an average of 7260 ns, Jailhouse to 7730 ns, Xen to 23000 ns, and seL4-VMM to 85940 ns. It corresponds to an increase of 36x, 38x, 115x, and 430x, respectively, compared to the base latency. It is also worth noting that the variance also increases. When enabling coloring (*col*), we measured no significant difference in interrupt latency compared to the base case. However, when enabling cache coloring in the presence of inter-VM interference (*interf+col*), there is a visible improvement in average latency and variance. However, note that the observed variance does not constitute a measure of predictability. As explained in Section IV, predictability is reflected in the difference between the *interf* and *interf+col* results and respective baselines, i.e., *solo* and *col*. Finally, by applying coloring also to the hypervisor (*interf+col+hypcol*), Bao latency is reduced to almost no-interference levels with negligible variance. Xen latency also drops considerably to an average of 6300 ns.

The observed interrupt latency under interference can be

[4]Deactivating an interrupt in the GIC means marking it as handled, enabling the distributor to forward it to the CPU when it occurs again.

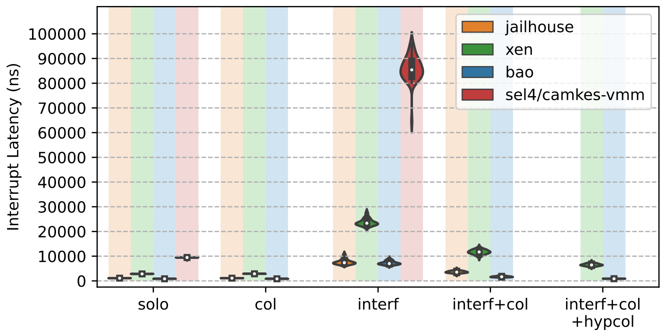
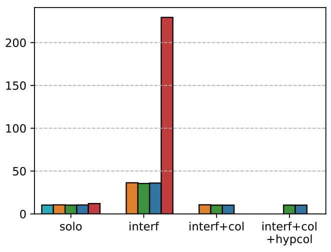
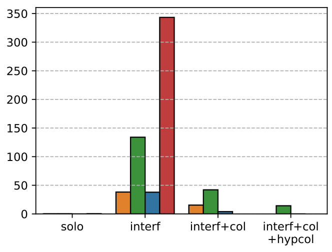


Fig. 7: Interrupt latency under interference and cache coloring.



(a) Guest L2 cache misses.



(b) Hypervisor L2 cache misses.

Fig. 8: L2 cache misses for the interrupt latency benchmark.

mostly explained by L2 cache misses. Fig. 8 shows the L2 cache misses for both guest and hypervisor during interrupt latency measurement. We can see that interference increases guest L2 cache misses, but that cache coloring can lower them back to the base case values. However, this is not the case for hypervisor L2 cache misses. For the base case, there are no cache misses for the hypervisor, which increases substantially under interference. Despite VM coloring contributing to reduce hypervisor L2 cache misses, only by coloring the hypervisor level, it is possible to minimize L2 cache misses for the hypervisor. On Bao, L2 cache misses are fully eliminated, but not on Xen[5], which might explain why latency does not reduce to non-interference levels.

Takeaway 4. Interrupt latency increases tenfold under the interference workload. Applying cache coloring to VMs proves very beneficial, but for it to be fully effective, it is imperative to reserve a color for the hypervisor itself.

Direct Injection. We evaluate the effectiveness of the direct injection technique, implemented only in Jailhouse and Bao. Fig. 9 depicts the results. The first conclusion is that for the base case, i.e., no interference, the interrupt latency is near to native (about 210 ns). Indeed, we have confirmed that during the execution of the benchmark, there are no traps to the hypervisor. Next, we observed that interference somewhat increases latency and its variance, but much less than in the previous experiments. Finally, we concluded that by enabling coloring, it is possible to lower the average latency to near native (243 and 232 ns for Bao and Jailhouse, respectively), however, there is still some variance due to the interference.

[5]At the time of writing, Xen's coloring patch was still under review. Thus, the assessed implementation may contain some imprecisions that are likely to be fixed by the time the patch is merged.

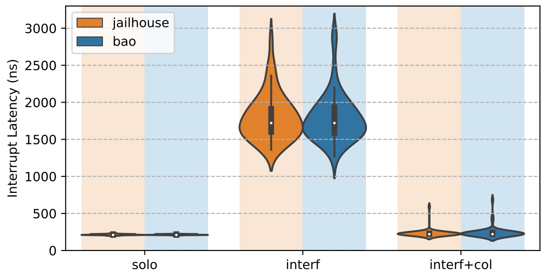


Fig. 9: Interrupt latency with direct injection enabled.

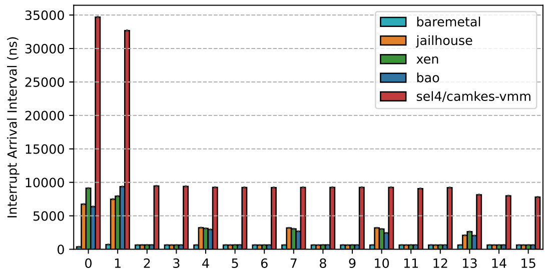


Fig. 10: Time between the handling of different priority interrupts triggered simultaneously, i.e., for interrupt N in the X-axis, the time between the arrival of interrupts N-1 and N.

Takeaway 5. The direct injection technique is effective in addressing the shortcomings of GIC interrupt virtualization, as results clearly demonstrated interrupt latency overhead is reduced to near native latencies.

Priority Handling. For studying the support of SPH delivering interrupts in the correct priority order, we have implemented a PL device which can be used to trigger up to 16 simultaneous interrupts, and a custom benchmark that assigns each a different priority. It starts by triggering the eight lowest priority interrupts. When handling the first, it triggers the eight highest priority interrupts. This would force the hypervisor to refill the four available LRs with the new higher priority interrupts, and refill them in priority order as LRs become available. The benchmark verifies if the priority order was kept and measures the arrival interval between each interrupt. We verified that only Xen and Bao guarantee the delivery of interrupts in the correct priority order. By inspecting the code, we have confirmed that both seL4-VMM and Jailhouse fill the GIC LRs following a FIFO policy. Furthermore, the seL4-VMM does not even commit the interrupt priorities configured in the virtual GIC distributor to hardware, precluding the arrival of physical interrupts in the correct priority order. Fig. 10 shows that across all hypervisors if multiple interrupts are delivered simultaneously, there is an increase by several orders of magnitude in the arrival time of the first and second interrupts, which is less than 700 ns for the baremetal case. This larger increase is justified by the fact that the hypervisors must handle all interrupts before the guest starts handling the first interrupt. Another observation is that there is a periodic increase in the interval of arrival. We have concluded this is the point at which there are no pending interrupts left in the LRs, which triggers the hypervisor to refill these registers with previously spilled pending interrupts.

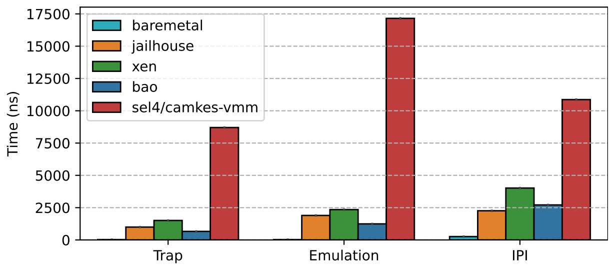


Fig. 11: Average cost for each send IPI operation component.

Takeaway 6. Only Xen and Bao respect interrupt priority order. Additionally, we observe that for all SPH, if multiple interrupts are triggered simultaneously, there is a partial priority inversion as lower priority interrupts take precedence due to the need for the hypervisor to handle and inject them.

Inter-Processor Interrupts. IPIs (SGIs) are critical for multi-core VM performance. For a vCPU to send an SGI, the guest must write a virtual GIC distributor register. This will trap to the hypervisor that must emulate the access and forward the event to the target core, where the SGI is injected via list registers. We use a custom baremetal benchmark to measure IPI latency. It works by measuring the time between when the source vCPU writes the distributor register and when the final IPI handler starts executing. It also measures the overhead of the trap. We instrument the SPH to sample the time the IPI is forwarded internally; this signals the end of the emulation and translates the overhead of injecting the interrupt in the target.

Figure 11 shows that IPI latency increases significantly for all SPH. While the baremetal IPI latency is around 260 ns, it reaches 2258 ns for Jailhouse, 4157 ns for Xen, 2711 ns for Bao, and 10868 ns for the CAmkES VMM. However, the costs of the register access emulation and interrupt injection are not proportional across all SPH. For example, Bao has the lowest emulation and event forwarding times, but the overall IPI latency is higher than Jailhouse's. This means that the interrupt injection path on Bao is slower than on Jailhouse. By inspecting the source of both hypervisors, we have observed that Bao immediately forwards the SGI event to the target core, performing all interrupt injection operations in the target core. Jailhouse, in turn, manages the interrupt injection structures at the source core and only then signals the target vCPU by writing the list register. Xen follows the same approach as Jailhouse, but presents higher overhead. The CAmkES VMM has the highest overhead due to the large number of system calls the VMM issues to the microkernel (in total, 7). Four are issued before the event forwarding, and the rest only after the SGI is forward to the target core. All in all, the access to the virtual distributor is more expensive than the IPI itself.

Takeaway 7. IPI latency reflects the same overheads of external interrupts. Future Arm platforms might reduce them with GICv4.1 [45]. In the short term, direct injection might alleviate this issue. However, both approaches fall short of achieving native latency as they still pay the price of emulating the write to the "IPI send" register.

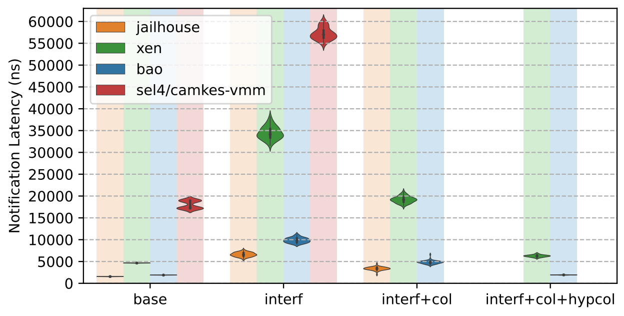


Fig. 12: Inter-VM notification latencies.

VI. SPH: Inter-VM communication

For inter-VM communication, SPH typically only provide statically allocated shared memory. This is usually coupled with an asynchronous notification mechanism signaled as an interrupt. All four SPH provide such mechanisms. Next, we analyze inter-VM notification latency and transfer throughput.

Inter-VM latency. Fig. 12 shows the inter-VM notification latency, reflecting the time since the notification is issued until the execution of the handler in the destination VM. The relative differences between the latencies for each SPH are similar to those observed for passthrough interrupts and IPIs. Jailhouse achieves the lower latency (1500 ns), followed by Bao (1900 ns). Xen shows an intermediate value of 4600 ns, while seL4 CAmkES VMM is significantly larger than others (average 18000 ns). Studying the internals of the implementations, we note that while most hypervisors synthesize and inject the virtual interrupts, Jailhouse uses non-allocated physical interrupts for these notifications. Thus, to send one, Jailhouse only sets the interrupt pending in the GIC distributor. This is significantly advantageous when combined with direct injection. Note that enabling direct injection in Bao would preclude the use of this mechanism. For seL4, we highlight the impact of the microkernel architecture since atop VM/VMM context switches, we observe additional overheads due to inter-VMM communication. Lastly, we see that interference increases all latencies accordingly and that coloring can mitigate it.

Inter-VM throughput. In Fig. 13, we evaluate the throughput of bulk data transfers via a shared memory buffer. The benchmark transmits 16 MiB of random data through a shared buffer with varying sizes. When the source VM finishes writing the buffer, it either signals the destination VM via a shared memory flag or via an asynchronous notification, and waits for a signal back to start writing the next chunk. For the polling scenario, the obtained throughput is very similar across all hypervisors; this confirms that are no significant differences in how they allocate and map memory or configure memory attributes. Throughput is stable (1500 MiB/s) until the buffer size surpasses the LLC size (1 MiB), dropping to about 1300 MiB/s. For the asynchronous scenario, throughput is significantly impacted when using smaller buffer sizes, given the high number of synchronization points that reflects the observed interrupt overheads. Finally, we note that interference has no significant effect as long as the buffer size is kept

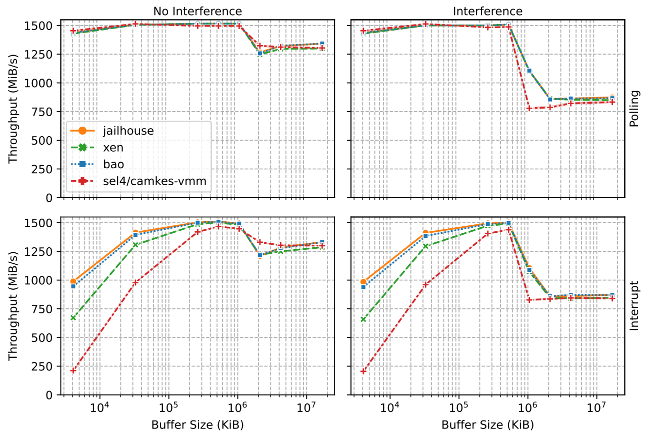


Fig. 13: Inter-VM communication throughput.

below about half the size of LLC. Beyond that, throughput is reduced from 1300 to 850 MiB/s. Although not shown due to lack of space, using coloring does not prove beneficial, as the throughput illustrated in Fig. 13 remains virtually unchanged.

Takeaway 8. Inter-VM notification latencies are significant and, as is the case for hardware interrupts, very susceptible to the effects of interference. However, for bulk data transfers it does not seem to significantly affect throughput if the shared buffer size is chosen on a range of about one-fourth to half the LLC size (i.e., 256 KiB to 512 KiB).

VII. SPH: Boot time

System boot time is a crucial metric in industries such as automotive [46], [47] as critical components have strict timing requirements for becoming fully operational.

Platform's Boot Flow. The platform's boot flow [48] starts by executing ROM code which loads the first-stage bootloader (FSBL) and enables the main cores. These initial boot stages setup the platform basic infrastructure (e.g., clocks, DRAM) and load the TF-A and U-boot. U-boot will load the hypervisor and, except for Jailhouse, the guest images. Bao and Xen directly boot guests after initialization. Jailhouse starts with the boot of the Linux root cell, that installs the hypervisor which then loads the guests. seL4's execution starts with an ELF loader which loads the all images, initializes secondary cores, and sets up an initial set of page tables for the microkernel. The microkernel initializes and hands control to user space.

Total VM Boot Time. The hypervisor boot time is heavily dependent on the VM and how it is configured. We observed that the VM image size is one of the parameters that has the higher impact in the hypervisor boot time. We measure boot time as a function of VM image size. Thus, to understand the overhead of the hypervisor in the context of the complete boot flow, in Fig. 14, we plot the cumulative time for each boot stage. Here, we can confirm that in all hypervisors but Jailhouse, the bulk of boot time is spent by U-boot. For Jailhouse, U-boot run time is constant, albeit large, as it always only loads the root cell's image. Jailhouse execution time increases steeply while loading the VM image. From

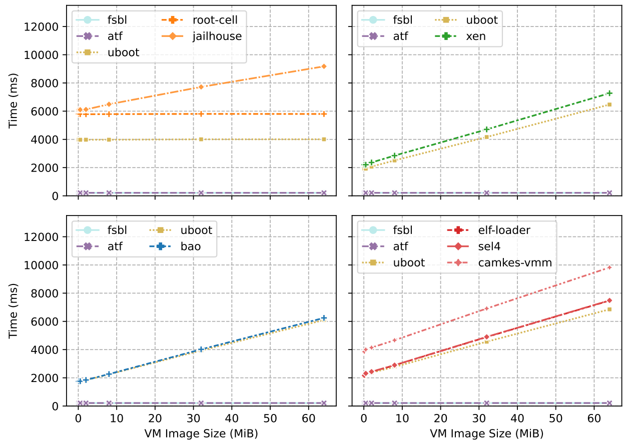


Fig. 14: Boot time for each stage by VM image region size.

this macro perspective, the other hypervisors add an almost constant offset to U-boot's boot time, the largest being seL4-VMM's. We observe this overhead is not on the microkernel, but at user level, which nevertheless heavily interacts with the microkernel to setup capabilities and kernel objects. We can conclude that VM boot time has its bottleneck by the loading of guest images to memory, not the hypervisor logic.

FreeRTOS and Linux Boot Times. We also measure the boot time of (i) a small VM running FreeRTOS with a 90 KiB image and (ii) a large VM with a Linux guest (built-in ramfs) totaling 59 MiB of image size. For Jailhouse, the Linux VM is a non-root cell. In Table I, we present results for a single-guest and a dual-guest system. For the latter, both VMs boot simultaneously; thus, we did not run experiments for dual-guest with Jailhouse, because it launches VMs sequentially. Table I presents the absolute boot time for the guest's native and virtualized execution, highlighting the relative percentage increase compared to native execution. For the single-guest FreeRTOS VM, all hypervisors but Bao cause a non-negligible increase in boot time. The same happens with the single-guest Linux VM. For the dual-guest configuration, we concluded that the small VM is heavily affected for all hypervisors. Surprisingly, we observe that although the cost of booting a single FreeRTOS in Bao is negligible, this is not true for a dual-guest configuration. Booting it alongside a Linux VM significantly increases its boot time, reaching similar overheads to those observed in Jailhouse's sequential boot.

Takeaway 9. The major bottleneck for the VM boot time is caused by the bootloader, not the hypervisors. Notwithstanding, the hypervisor can significantly increase the boot time of a critical VM (small RTOS) when booting it alongside a larger VM (e.g., in dual-OS Linux+RTOS configuration).

VIII. SPH: Code Size and TCB

In MCS, the size of the hypervisor code, measured in source lines of code (SLoC), is critical. It should be minimal as it is part of the trusted computing base (TCB) of all VMs. In this paper, we consider that a VM TCB encompasses any component with sufficient privileges that if it is compromised or malfunctions, might be able to affect the safety and/or

		Baremetal	Jailhouse	Xen	Bao	seL4-VMM
FreeRTOS	Single	1670.89	6242.18 / 173.58%	2338.24 / 39.94%	1716.23 / 2.71 %	3496.19 / 109.24%
	Dual		N/A	6887.88 / 312.23%	5734.04 / 143.17%	9291.02 / 456.05%
Linux	Single	7665.14	12284.92 / 60.27%	8533.88 / 11.33%	7805.54 / 1.83 %	12629.79 / 64.77%
	Dual		N/A	8707.15 / 13.59%	7895.95 / 3.01 %	13086.86 / 70.73%

TABLE I: Total boot time (ms) and relative increase compared to the baremetal case, for FreeRTOS and Linux VMs.

		C		Asm	Total (SLoC)	*.text* (KiB)
		.c	.h			
jailhouse	hypervisor	7308	2279	342	9929	79.3
	driver	2041	139	N/A	2180	20.1
xen		57360	8127	1765	67342	451.5
bao		5046	2840	537	8423	57.9
seL4 CAmkES VMM	microkernel	14569	N/A	189	14758	224.7
	VMM	20932	19291	N/A	40223	724.3

TABLE II: Hypervisor SLoC count and binary code size.

security properties of the VM. As well understood in the literature, a larger TCB typically has a higher number of bugs and wider attack surface [49], resulting in a higher probability of vulnerabilities. It is important to understand that each VM has its own TCB. Thus, CAmkES VMM is only considered for the managed VM's TCB, not the others. Further, large code bases are impractical for certification, both from a technical and economic perspective. To qualify a component assigned a safety integrity level (SIL), all components on which it depends must also be qualified to the same or higher SIL [4].

Methodology. We measured SLoC for the target configurations using *cloc* [50]. Xen build system offers a make target to assess the SLoC for a specific configuration. However, it does not count header files, which we believe must be accounted for since they provide function-like macros and inline functions. We have modified the Xen makefile to measure headers. We have also extended Jailhouse and Bao build systems with the same functionality. For seL4, we used the fully unified and pre-processed kernel source file to assess the microkernel code base. For the CAmkES VMM, given that its source code is scattered throughout multiple seL4 project libraries, we were not able to list its source code files from the build system. Instead, we used debug information from the final executable and inspected each source to assess the included header files.

Code Size. Looking at Table II we see Bao and Jailhouse have the smallest code base of about 8400 and 9900 SLoC, respectively. Bao is implemented as a standalone component with no external dependencies. However, since part of Jailhouse functionality is implemented as a Linux kernel module, we also account that for the code base. It adds about 2180 SLoC, bringing Jailhouse total code base to 12 KSLoC. For Xen we use a custom config with almost all features disabled, except a few ones such as coloring and static shared memory. It features the largest code base with around 67 KSLoC. Finally, seL4 microkernel has 14.5 KSLoC, while the CAmkES VMM can go up to 40K, i.e., almost 55 KSLoC in total. The visible difference between Bao and Jailhouse, and seL4 microkernel and, especially, Xen, lies in the fact that the former were designed specifically for the static partitioning use case, while the latter aim at being more generic and adaptable. These differences are reflected in the binary size of each hypervisor.

TCB. The hypervisor SLoC does not directly reflect the VM TCB. Although by design SPH such as Bao has a smaller SLoC count, the seL4-VMM is vastly superior from a security perspective: shared TCB is limited only to the formally verified microkernel, because each VM is managed by a fully isolated VMM. From a FuSa certification standpoint, however, the VMM would still need to be considered. Moreover, seL4 formal proofs are limited to a set of kernel configurations, currently not including multicore. Regarding Jailhouse, despite its small size, the root cell is a privileged component of the system. It executes part of all VM management logic, being in the critical path for booting all other VMs. It is arguably part of all VM's TCB, increasing it significantly [49]. Analogously, Xen must depart from true Dom0-less to leverage richer features (e.g., PV drivers, dynamic VM creation). Recently, the Xen community has ignited efforts to use a smaller OS, such as Zephyr [24], as Dom0, refactor Xen to MISRA C, and provide extensive requirements and test documentation [51].

Takeaway 10. Hypervisors specifically targeting static partitioning have the smallest code bases. Despite facilitating certification, none of the evaluated SPH provide other artifacts (e.g., requirements specification, coding standards). Xen is the first to take steps in this direction; nevertheless, seL4's formal proofs provide the most comprehensive guarantees.

IX. Discussion and Future Directions

In this section, we discuss some of the open issues and potential research directions to improve the guarantees of SPH.

Interference Mitigation Techniques. Cache coloring does not fully mitigate the effects of inter-core interference. Furthermore, coloring has inherent inefficiencies such as (i) precluding the use of superpages and (ii) increasing memory pressure which affects performance and predictability, as well as (iii) internal fragmentation (exclusively assigning 1 out of N colors, implicitly allocates 1/Nth of physical memory, a portion of which may remain unused for small RTOSs or the SPH). While the latter could be solved by employing cache bleaching [52] in heterogeneous platforms, to further minimize coloring bottlenecks, we advocate for SPH to adopt other proven, widely applicable contention mitigation mechanisms, e.g., bandwidth regulation mechanisms implemented via PMU-based CPU throttling [53], [54]. We also stress the importance of including support for hardware extensions such as Arm's Memory Partitioning and Monitoring (MPAM) [11], [55], which provide flexible hardware means for partitioning cache space and memory bandwidth and call for platform designers to include such facilities in their upcoming designs targeting MCS. Finally, we stress the need for instrumentation, analysis, and profiling tools [20], [56] that integrate with these hypervisors to help system designers understand the trade-offs and fine-tune these mechanisms (e.g., through automation).

Platform-Level Contention and Mitigation. None of the studied SPH manages traffic from peripheral DMAs. We advocate that SPH must provide contention mitigation mechanisms at the platform level, e.g., (i) leveraging QoS hardware [20], [57] available on the bus and (ii) controlling interference from DMA-capable devices or accelerators. Furthermore, since DMA masters still share SMMU structures (e.g., TLBs [58]), we hypothesize that bandwidth regulation techniques may fall short of efficiently mitigating interference at this level.

Interrupt Injection Optimization. Arm-based SPH's interrupt latency is mainly due to inadequate support in GICv2/3. GICv4 will provide direct interrupt injection support, but only for IPIs and MSIs. We want to raise awareness of Arm silicon makers and designers of the need for additional hardware support at the GIC level for direct injection of wired interrupts. The same holds for RISC-V [25]. Besides hardware support, we observed that simple SPH provide optimized interrupt injection paths. It is also possible to optimize this path in larger SPH (e.g., Xen) and in microkernels (e.g., moving injection logic to the microkernel). Finally, Bao and Jailhouse implement direct interrupt injection; however, we must stress that using this technique severely hinders the ability of the SPH to manage devices or implement any functionality dependent on interrupts. A plausible research direction would be a hybrid approach, i.e., selectively enabling direct injection only in specific cores for critical guests while providing the more complex functionality in cores running non-critical guests.

Interrupt Priority Inversion Fix. As discussed in Section V, the studied SPH suffer from partial interrupt priority inversion because all currently pending interrupts are handled by the hypervisor and injected in the guest before it can service the highest-priority one. We advocate for implementing a lightweight solution by dynamically setting the interrupt priority mask based on the priority of the last injected interrupt. This approach ensures the hypervisor only receives the next interrupt once the guest has handled the highest priority one.

Critical VM Boot Priority. Section VII highlights the issue of critical VM boot time overhead when booted under a dual-OS configuration. We advocate for the development of boot mechanisms that prioritize the boot of small critical VMs. However, as noted in Jumpstart [47], it must encompass the full boot flow and be optimized across stages and components since the bottleneck of the boot time is in the image loading process performed by the bootloader, not the hypervisor.

Per-Partition Hypervisor Replica. Memory contention highly affects interrupt latency but can be minimized by assigning different colors for VMs and the hypervisor. Notwithstanding, coloring the hypervisor may prove wasteful and insufficient to address other interference channels internal to the hypervisor. We advocate for *à la* multikernel [59] implementations such as the one implemented in seL4, where the hypervisor image is replicated per cache partition [31], fully closing internal channels. For SPH with a small enough footprint, memory consumption or boot time costs should not be prohibitive.

Architecture Flexibility. Purely monolithic SPH (e.g., Jailhouse or Bao) have smaller code bases at the cost of feature richness and flexibility. The same holds for Xen, i.e., many widely-used rich features are absent when configured as an SPH (to minimize code size). On the other hand, the seL4 microkernel architecture is much more flexible as it allows for an isolated user space VMM per guest, providing more robust isolation and customization; however, it comes at the cost of non-negligible latencies. We advocate for novel architectures that combine microkernels' flexibility and strong fault encapsulation with SPH's simplicity and minimalist latencies by hosting per-partition VMMs directly at the hypervisor privilege level. Such a design could arguably be achieved by combining multikernel-like architectures [59] and per-core memory protection mechanisms (e.g., Armv9 RME's GPT [60], or RISC-V PMP [61]) statically configured by firmware.

Full IO Passthrough. Pure static partitioning supports only passthrough IO. However, as highlighted by [7], there is a critical problem in providing full IO passthrough when controls over IO resources such as clock, reset, power, or pin-muxes cannot be securely partitioned or shared, e.g., if their MMIO registers reside on the same frame or they are configured via platform management co-processors oblivious of SPH's VMs. Thus, SPH should provide controlled guest access to these resources by emulation or through standard interfaces such as SCMI [62]. Nevertheless, this would require including drivers in the hypervisor, increasing its code base. Again, we urge hardware designers to provide hardware primitives that enable SPH to pass through IO resource controls.

X. Related Work

There are several hypervisor analyses in the context of embedded and MCSs, but none provide a cross-section analysis and comparison on SPH. Some works focus on a single hypervisor while others evaluate a single metric or feature. In [63], authors compare the performance of Xvisor with Xen and KVM. Others have evaluated the effectiveness of cache coloring and bandwidth reservations in Xvisor [54]. Similarly, in [19], authors evaluate cache and DRAM bank coloring in Jailhouse. Other works have evaluated Jailhouse interrupt latency [64] or VM interference [65]. There are also studies about the feasibility of using Xen and KVM as real-time hypervisors [66], but mainly for x86. Little has been published regarding the new Xen Dom0-less and cache coloring features, but results can be found in [67]. Evaluation of the seL4 CAmkES VMM has also been done for performance and interrupt latency [29]. There have been works providing a qualitative analysis for MCS hypervisors, contrasting architectural approaches and highlighting future trends [68] while others layout guidelines on how to choose such a hypervisor in industrial settings [46].

XI. Conclusion

We have conducted the most comprehensive empirical evaluation of open-source SPH to date, focusing on key metrics for MCS. With that, we drew a set of observations that (i) will help industrial practitioners understand the trade-offs of SPH and (ii) raise awareness of the research and open-source communities to the still open problems in SPH. We are opening all artifacts to enable independent validation of results and encourage further exploration on SPH.

XII. Acknowledgments

We would like to express our gratitude to the reviewers for their valuable feedback and suggestions, as well as to our friendly shepherd for guiding us in making final improvements. Additionally, we appreciate the time and thoughtful input from all the representatives of SPH, namely Ralf Ramsauer (Jailhouse), Stefano Stabellini (Xen), and Gernot Heiser (seL4/CAmkES-VMM). José Martins was supported by FCT grant SFRH/BD/138660/2018. This work is supported by FCT – Fundação para a Ciência e Tecnologia within the RD Units Project Scope UIDB/00319/2020, and 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] J. Cerrolaza et al., "Multi-Core Devices for Safety-Critical Systems: A Survey," *ACM Computing Surveys*, 2020.

[2] M. Staron, *Contemporary Software Architectures: Federated and Centralized*. Springer International Publishing, 2021.

[3] A. Burns and R. Davis, "A Survey of Research into Mixed Criticality Systems," *ACM Computing Surveys*, 2017.

[4] A. Esper et al., "An industrial view on the common academic understanding of mixed-criticality systems," *Real-Time Systems*, 2018.

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

[6] C. Dall and J. Nieh, "KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor," *ACM SIGARCH Computer Architecture News*, 2014.

[7] R. Ramsauer et al., "A Novel Software Architecture for Mixed Criticality Systems," in *Digital Transformation in Semiconductor Manufacturing*, 2020.

[8] J. Martins et al., "Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems," in *Proc. of Workshop on Next Generation Real-Time Embedded Systems (NG-RES)*, 2020.

[9] S. VanderLeest and D. White, "MPSoC hypervisor: The safe & secure future of avionics," in *Proc. of Digital Avionics Systems Conference (DASC)*, 2015.

[10] P. Burgio et al., "A software stack for next-generation automotive systems on many-core heterogeneous platforms," *Microprocessors and Microsystems*, 2017.

[11] F. Rehm et al, "The Road towards Predictable Automotive High - Performance Platforms," in *Proc. of Design, Automation and Test in Europe Conference (DATE)*, 2021.

[12] J. Martins, "ESRGv3/shedding-light-static-partitioning-hypervisors: v0.1.0," 2023. [Online]. Available: https://doi.org/10.5281/zenodo.7696937

[13] Arm, "Learn the architecture: AArch64 Virtualization," https://developer.arm.com/documentation/den0125/latest, 2022.

[14] A. Gordon et al., "ELI: Bare-Metal Performance for I/O Virtualization," *SIGPLAN Notices*, 2012.

[15] G. Gracioli et al., "A Survey on Cache Management Mechanisms for Real-Time Embedded Systems," *ACM Computing Surveys*, 2015.

[16] Arm, "Software Delegated Exception Interface (SDEI)," https://developer.arm.com/documentation/den0054/latest, 2021.

[17] 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.

[18] ——, "Static Hardware Partitioning on RISC-V – Shortcomings, Limitations, and Prospects," in *Proc. of IEEE World Forum on Internet of Things*, 2022.

[19] 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.

[20] P. Sohal et al., "E-WarP: A System-wide Framework for Memory Bandwidth Profiling and Management," in *Proc. of Real-Time Systems Symposium (RTSS)*, 2020.

[21] "jailhouse-rt: Bu-maintained version of the jailhouse partitioning hypervisor with real-time features." [Online]. Available: https://github.com/rntmancuso/jailhouse-rt

[22] A. Biondi et al., "SPHERE: A Multi-SoC Architecture for Next-Generation Cyber-Physical Systems Based on Heterogeneous Platforms," *IEEE Access*, 2021.

[23] G. Corradi, "Xen on Arm: Real-Time Virtualization with Cache Coloring," in *Proc. of Embedded World Conference*, 2020.

[24] "Zephyr project," Feb 2023. [Online]. Available: https://www.zephyrproject.org/

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

[26] G. Klein et al., "SeL4: Formal Verification of an OS Kernel," in *Proc. of ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[27] G. Heiser, "The seL4 Microkernel: An Introduction," The seL4 Foundation, 2020.

[28] G. Klein et al., "Formally Verified Software in the Real World," *Communications of the ACM*, 2018.

[29] J. Millwood et al., "Performance Impacts from the seL4 Hypervisor," in *Proc. of the Ground Vehicle Systems Engineering and Technology Symposium*, 2020.

[30] A. Lyons et al., "Scheduling-Context Capabilities: A Principled, Light-Weight Operating-System Mechanism for Managing Time," in *Proc. of European Conference on Computer Systems (EuroSys)*, 2018.

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

[32] T. Murray et al., "seL4: From General Purpose to a Proof of Information Flow Enforcement," in *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 2013.

[33] G. Klein et al., "Comprehensive Formal Verification of an OS Microkernel," *ACM Transactions on Computer Systems*, 2014.

[34] G. Heiser et al., "Towards Provable Timing-Channel Prevention," *ACM SIGOPS Operating Systems Review*, 2020.

[35] ——, "Can We Put the "S" Into IoT?" in *Proc. of IEEE World Forum on Internet of Things*, 2022.

[36] U. Steinberg and B. Kauer, "NOVA: A Microhypervisor-Based Secure Virtualization Architecture," in *Proc. of European Conference on Computer Systems*, 2010.

[37] S. Pinto et al., "LTZVisor: TrustZone is the Key," in *Proc. of Euromicro Conference on Real-Time Systems (ECRTS)*, 2017.

[38] J. Martins et al., "μRTZVisor: A Secure and Safe Real-Time Hypervisor," *Electronics*, 2017.

[39] S. Pinto and N. Santos, "Demystifying Arm TrustZone: A Comprehensive Survey," *ACM Computing Surveys*, 2019.

[40] "perf: Linux profiling with performance counters." [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page

[41] M. Guthaus et al., "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. of International Workshop on Workload Characterization (WWC)*, 2001.

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

[43] P. Valsan et al., "Taming Non-Blocking Caches to Improve Isolation in Multicore Real-Time Systems," in *Proc. of Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

[44] W. Hofer et al., "Sloth: Threads as Interrupts," in *Proc. of Real-Time Systems Symposium (RTSS)*, 2009.

[45] Arm Ltd., "Arm Generic Interrupt Controller v3 and v4 - Virtualization," 2022.

[46] E. Hamelin et al., "Selection and evaluation of an embedded hypervisor: Application to an automotive platform," in *Proc. of European Congress of Embedded Real Time Software and Systems*, 2020.

[47] A. Golchin and R. West, "Jumpstart: Fast Critical Service Resumption for a Partitioning Hypervisor in Embedded Systems," in *Proc. of Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2022.

[48] Xilinx, "Zynq UltraScale+ Device: Technical Reference Manual," https://docs.xilinx.com/v/u/en-US/ug1085-zynq-ultrascale-trm, 2020.

[49] S. Biggs et al., "The Jury Is In: Monolithic OS Design Is Flawed: Microkernel-Based Designs Improve Security," in *Proc. of Asia-Pacific Workshop on Systems*, 2018.

[50] Al Danial, "cloc - count lines of code," https://github.com/AlDanial/cloc.

[51] A. Mygaiev and S. Stabellini, “Xen FuSa SIG update,” in *Xen Project Developer and Design Summit*, 2021. [Online]. Available: https://www.youtube.com/watch?v=XMNaIWZ-2sU

[52] S. Roozkhosh and R. Mancuso, “The potential of programmable logic in the middle: Cache bleaching,” in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020.

[53] 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.

[54] P. Modica at al., “Supporting temporal and spatial isolation in a hypervisor for ARM multicore platforms,” in *Proc. of International Conference on Industrial Technology (ICIT)*, 2018.

[55] Arm Ltd., “Arm Architecture Reference Manual Supplement - Memory System Resource Partitioning and Monitoring (MPAM), for A-profile architecture,” 2022.

[56] G. Ghaemi et al., “Governing with Insights: Towards Profile-Driven Cache Management of Black-Box Applications,” in *Proc. of Euromicro Conference on Real-Time Systems (ECRTS)*, 2021.

[57] M. Zini et al., “Profiling and controlling I/O-related memory contention in COTS heterogeneous platforms,” *Software: Practice and Experience*, 2022.

[58] A. Panchamukhi and F. Mueller, “Providing task isolation via tlb coloring,” in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015.

[59] A. Baumann et al., “The multikernel: A new os architecture for scalable multicore systems,” in *Proc. of ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[60] X. Li et al., “Design and verification of the arm confidential compute architecture,” in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[61] D. Lee et al., “Keystone: An open framework for architecting trusted execution environments,” in *Proc. of European Conference on Computer Systems (EuroSys)*, 2020.

[62] Arm Ltd., “Arm System Control and Management Interface - Platform Design Document, Version 3.1,” 2022.

[63] A. Patel et al., “Embedded Hypervisor Xvisor: A Comparative Analysis,” in *Proc. of Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2015.

[64] I. Pavic and H. Dzapo, “Virtualization in multicore real-time embedded systems for improvement of interrupt latency,” in *Proc. of International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2018.

[65] J. Danielsson et al., “Testing Performance-Isolation in Multi-core Systems,” in *Proc. of Annual Computer Software and Applications Conference (COMPSAC)*, 2019.

[66] L. Abeni and D. Faggioli, “Using Xen and KVM as real-time hypervisors,” *Journal of Systems Architecture*, 2020.

[67] S. Stabellini, “Xen Cache-Coloring: Interference Free Real-Time Systems,” in *Open Source Summit (Noth America)*, 2020. [Online]. Available: https://www.youtube.com/watch?v=9cA0QK2CdwQ

[68] M. Cinque et al., “Virtualizing mixed-criticality systems: A survey on industrial trends and issues,” *Future Generation Computer Systems*, 2022.