Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems

José Martins
Centro Algoritmi, Universidade do Minho, Portugal
jose.martins@dei.uminho.pt

Adriano Tavares
Centro Algoritmi, Universidade do Minho, Portugal
atavares@dei.uminho.pt

Marco Solieri
Università di Modena e Reggio Emilia, Italy
marco.solieri@unimore.it

Marko Bertogna
Università di Modena e Reggio Emilia, Italy
marko.bertogna@unimore.it

Sandro Pinto
Centro Algoritmi, Universidade do Minho, Portugal
sandro.pinto@dei.uminho.pt

Abstract

Given the increasingly complex and mixed-criticality nature of modern embedded systems, virtualization emerges as a natural solution to achieve strong spatial and temporal isolation. Widely used hypervisors such as KVM and Xen were not designed having embedded constraints and requirements in mind. The static partitioning architecture pioneered by Jailhouse seems to address embedded concerns. However, Jailhouse still depends on Linux to boot and manage its VMs. In this paper, we present the Bao hypervisor, a minimal, standalone and clean-slate implementation of the static partitioning architecture for Armv8 and RISC-V platforms. Preliminary results regarding size, boot, performance, and interrupt latency, show this approach incurs only minimal virtualization overhead. Bao will soon be publicly available, in hopes of engaging both industry and academia on improving Bao's safety, security, and real-time guarantees.

2012 ACM Subject Classification Security and privacy → Virtualization and security; Software and its engineering → Real-time systems software

Keywords and phrases Virtualization, hypervisor, static partitioning, safety, security, real-time, embedded systems, Arm, RISC-V

Digital Object Identifier 10.4230/OASIcs.NG-RES.2020.3

Funding This work is supported by European Structural and Investment Funds in the FEDER component, through the Operational Competitiveness and Internationalization Programme (COMPETE 2020) [Project nº 037902; Funding Reference: POCI-01-0247-FEDER-037902].
José Martins: Supported by FCT grant SFRH/BD/138660/2018.

1 Introduction

In domains such as automotive and industrial control, the number of functional requirements has been steadily increasing for the past few years [8, 42]. As the number of the resulting increasingly complex and computing power-hungry applications grows, the demand for high-performance embedded systems has followed the same trend. This has led to a paradigm shift from the use of small single-core microcontrollers running simple bare-metal applications or real-time operating systems (RTOSs), to powerful multi-core platforms, endowed with complex memory hierarchies, and capable of hosting rich, general-purpose operating systems (GPOSs).

At the same time, the market pressure to minimize size, weight, power, and cost, has pushed for the consolidation of several subsystems onto the same hardware platform. Furthermore, these typically take the form of mixed-criticality systems (MCSs) by integrating components with distinct criticality levels. For example, in automotive systems, network-connected infotainment is often deployed alongside safety-critical control systems [8]. As such, great care must be taken when consolidating mixed-criticality systems to balance the conflicting requirements of isolation for security and safety, and efficient resource sharing.

Virtualization, an already well-established technology in desktop and servers, emerges as a natural solution to achieve consolidation and integration. It requires minimal engineering efforts to support legacy software while guaranteeing separation and fault containment between virtual machines (VMs). Several efforts were made to adapt server-oriented hypervisors, such as Xen [19, 47] or KVM [26, 12], to embedded architectures (mainly Arm) with considerable 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 [1]. Moreover, these embedded hypervisors often depend on a large GPOS (typically Linux) either to boot, manage virtual machines, or provide a myriad of services, such as device emulation or virtual networks [4, 41]. From a security and safety perspective, this dependence bloats the system trusted computing base (TCB) and intercepts the chain of trust in secure boot mechanisms, overall widening the system's attack surface [32]. More, due to the size and monolithic architecture of such OSs, this tight coupling also hampers the safety certification process of systems deploying such a hypervisor.

The static partitioning hypervisor architecture, pioneered by Siemens' Jailhouse [41], has been recently experiencing increasing adoption in MCSs from both academia and industry. This architecture leverages hardware-assisted virtualization technology to employ a minimal software layer that statically partitions all platforms resources and assigns each one exclusively to a single VM instance. It assumes no hardware resources need to be shared across guests. As each virtual core is statically pinned to a single physical CPU, there is no need for a scheduler, and no complex semantic services are provided, further decreasing size and complexity. Although possibly hampering the efficient resource usage requirement, static partitioning allows for stronger guarantees concerning isolation and real-time. Despite its design philosophy, Jailhouse falls short by still depending on Linux to boot the system and manage its "cells", suffering from the same aforementioned security ills of other hypervisors.

Despite the strong CPU and memory isolation provided by the static partitioning approach, this is still not enough as many micro-architectural resources such as last-level caches, interconnects, and memory controllers remained shared among partitions. The resulting contention leads to a lack of temporal isolation, hurting performance and determinism [3, 2]. Furthermore, this can be exploited by a malicious VM to implement DoS attacks by increasing their consumption of a shared resource [6], or to indirectly access other VM's data through the implicit timing side-channels [13]. To tackle this issue, techniques such as cache partitioning (either via locking or coloring) or memory bandwidth reservations were already proposed and implemented at both the operating system and hypervisor level [48, 27, 30, 22].

In this paper, we present Bao, a minimal, from-scratch implementation of the partitioning hypervisor architecture. Despite following the same architecture as Jailhouse, Bao does not rely on any external dependence (except the firmware to perform low-level platform management). Also, given the simplicity of the mechanism, it provides baked in support for cache coloring. Bao originally targets the Armv8 architecture, and experimental support for the RISC-V architecture is also available. As we strongly believe that security through obscurity, the approach followed by a majority of industry players, has been proven time and time again to be ineffective, Bao will be available open-source by the end of 2019.

2 Bao Hypervisor

Bao (from Mandarin Chinese "bǎohù", meaning "to protect") is a security and safety-oriented, lightweight bare-metal hypervisor. Designed for MCSs, it strongly focuses on isolation for fault-containment and real-time behavior. Its implementation comprises only a minimal, thin-layer of privileged software leveraging ISA virtualization support to implement the static partitioning hypervisor architecture (Figure 1): resources are statically partitioned and assigned at VM instantiation time; memory is statically assigned using 2-stage translation; IO is pass-through only; virtual interrupts are directly mapped to physical ones; and it implements a 1-1 mapping of virtual to physical CPUs, with no need for a scheduler. The hypervisor also provides a basic mechanism for inter-VM communication based on a static shared memory mechanism and asynchronous notifications in the form of inter-VM interrupts triggered through a hypercall. Besides standard platform management firmware, Bao has no external dependencies, such as on privileged VMs running untrustable, large monolithic GPOSs, and, as such, encompasses a much smaller TCB.

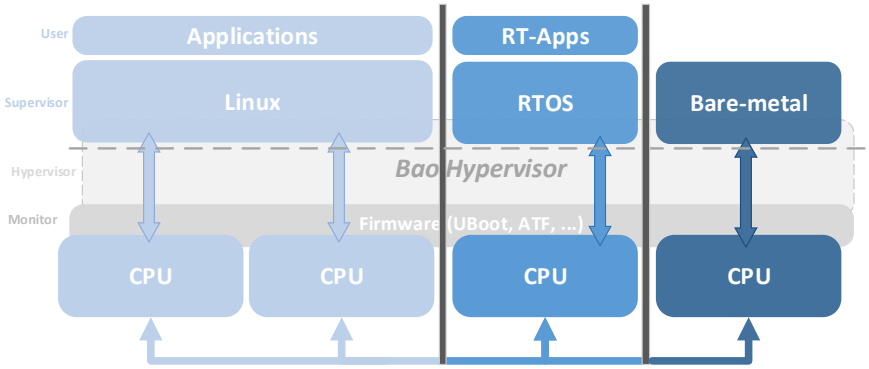

Figure 1 Bao's static partitioning architecture.

2.1 Platform Support

Bao currently supports the Armv8 architecture. RISC-V experimental support is also available but, since it depends on the hypervisor extensions, which are not yet ratified, no silicon is available that can run the hypervisor. Consequently, the RISC-V port was only deployed on the QEMU emulator, which implements the latest version of the draft specification (at the time of this writing, version 0.4). For this reason, for the remaining of the paper, we will only focus on the Arm implementation. As of the time of this writing, Bao was ported to two Armv8 platforms: Xilinx's Zynq-US+ on the ZCU102/4 development board and HiSilicon's Kirin 960 on the Hikey 960. So far, Bao was able to host several bare-metal applications, the FreeRTOS and Erikav3 RTOSs, and vanilla Linux and Android.

Except for simple serial drivers to perform basic console output, Bao has no reliance on platform-specific device drivers and requires only a minimal platform description (e.g., number of CPUs, available memory, and its location) to be ported to a new platform. For this reason, Bao relies on vendor-provided firmware and/or a generic bootloader to perform baseline hardware initialization, low-level management, and to load the hypervisor and guest images to main memory. This significantly reduces porting efforts.

On the supported Arm-based platforms, Bao relies on an implementation of the standard Power State Coordination Interface (PSCI) to perform low-level power control operations, further avoiding the need for platform-dependent drivers. On Arm-based devices, this has

been provided by Arm Trusted Firmware (ATF). On such platforms, Linux itself depends on PSCI for CPU hot-plugging. When such guests invoke PSCI services, Bao merely acts as a shim and sanitizer for the call arguments, to guarantee the VM abstraction and isolation, deferring the actual operation to ATF. Although we've been able to boot directly from ATF, we've been also using the well-known U-boot bootloader to load hypervisor and guest images.

2.2 Spatial and Temporal Isolation

Following the main requirement of isolation, Bao starts by setting up private mappings for each core. Using the recursive page table mapping technique, it avoids the need for a complete contiguous mapping of physical memory, which would otherwise be essential to perform software page table walks. This approach is usually not suitable when managing multiple address spaces and typically incurs a higher TLB footprint for page table look-ups. However, given that only a single address space is managed per CPU, and page tables are completely set-up at initialization, this is not necessarily true for our static architecture and design philosophy. Nevertheless, all cores share mappings for a per-CPU region for inter-core communication, and the hypervisor's image itself. Furthermore, only cores hosting the same VM will map its global control structure. These design decisions follow the principle of least privilege, where each core, and privilege level within it, only has (at least, direct) access to what it absolutely must. This hardens data integrity and confidentially by minimizing the available data possibly accessed by exploiting read/write gadgets available in the hypervisor. Furthermore, hypervisor code pages are marked as read-only and a X⊕W policy is enforced on hypervisor data pages by configuring them as non-executable.

Guest isolation itself starts, of course, with the logical address space isolation provided by 2-stage translation hardware virtualization support. To minimize translation overhead, page table, and TLB pressure, Bao uses superpages (in Arm terminology, blocks) whenever possible, which also possibly improves guest performance by facilitating speculative fetches. Regarding time, given exclusive CPU assignment, no scheduler is needed, which coupled with the availability of per-CPU architectural timers directly managed by the guests, allows for complete logical temporal isolation.

Despite the strong partitioning inherent to this architecture and the efforts taken to minimize the existent virtualization overheads, this is not enough to guarantee deterministic execution and meet the deadlines of critical guests' tasks. Micro-architectural contention at shared last-level caches (LLCs) and other structures still allows for interference between guest partitions. As such, given its simplicity, Bao implements a page coloring mechanism from the get-go, enabling LLC cache partitioning. Coloring, however, has several drawbacks. Firstly, it forces the use of the finest-grained page size available, precluding the benefits of using superpages. Secondly, as it also partitions the actual physical address space, leading to memory waste and fragmentation. Another problem regarding coloring is that, as Bao relies on a bootloader to load guest images, which are continuously laid out in memory, it needs to recolor them, i.e., copy the non-color compliant pages from the original loaded image to pages agreeing with the colors assigned to that specific VM, which will increase the VM's boot time. Coloring can be enabled and each color selected, independently for each VM.

2.3 IO and Interrupts

Bao directly assigns peripherals to guests in a pass-through only IO configuration. As in the supported architectures, specifically Arm, all IO is memory-mapped, this is implemented for free by using the existing memory mapping mechanisms and 2-stage translation provided by virtualization support. The hypervisor does not verify the exclusive assignment of a given peripheral, which allows for several guests to share it, albeit in a non-supervised manner.

The Generic Interrupt Controller (GIC) is the interrupt router and arbiter in the Arm architecture. Although it provides some interrupt virtualization facilities, the majority of the available hardware platforms feature either GICv2 or GICv3, which do not support direct interrupt delivery to guest partitions. All interrupts are forward to the hypervisor, which must re-inject the interrupt in the VM using a limited set of pending registers. Besides the privileged mode crossing overheads leading to an unavoidable increase in interrupt latency, this significantly increases interrupt management code complexity, especially if features such as interrupt priority are to be emulated. Bao's implementation does follow this path, as many RTOSs make use of interrupt priorities, sometimes even as a task scheduling mechanism [33, 40]. This problem was solved in the newest version of the spec, GICv4, which bypasses the hypervisor for guest interrupt delivery [12]. Furthermore, the limited virtualization support dictates that guest access to the central distributor must be achieved using trap and emulation. Depending on the frequency and access patterns of a guest to the distributor, this might significantly decrease performance. As of now, Bao only supports GICv2.

3 Evaluation

In this section, we present Bao's initial evaluation. First, we will focus on code size and memory footprint. Then we evaluate the boot time, performance, and interrupt latency. We compare guest native execution (bare) with hosted execution (solo) and hosted execution under contention (interf) to evaluate the arising interference when running multiple guests. We repeat the hosted scenarios with cache partitioning enabled (solo-col and interf-col), to understand the degree to which this first level of micro-architectural partitioning impacts the target partitions and helps to mitigate interference.

Our test platform is the Xilinx ZCU104, featuring a Zynq-US+ SoC with a quad-core Cortex-A53 running at 1.2 GHz, per-core 32K L1 data and instruction caches, and a shared unified 1MB L2/LLC cache. We execute the target test VM in one core while, when adding interference, we execute two additional bare-metal applications, each in a separate VM, which continuously write and read a 512KiB array with a stride equal to the cache line size (64 bytes). When enabling coloring, we assign half the LLC (512 KiB) to the VM running the benchmarks and one fourth (256 KiB) to each of the interfering bare-metal apps. Both the hypervisor code and benchmark applications were compiled using the Arm GNU Toolchain version 8.2.1 with -O2 optimizations.

3.1 Code Size and Memory Footprint

Bao is a complete from-scratch implementation with no external dependencies. In this section, we evaluate (i) code complexity using source lines of code (SLoC), and (ii) memory footprint by looking at the size of the final binary and then analyzing run-time consumption.

The code is divided into four main sections: the arch and platform directories contain target-specific functionality while the core and lib directories feature the main hypervisor logic and utilities (e.g., string manipulation, formatted print code), respectively. The total SLoC and final binary sizes for each directory are presented in Table 1.

Table 1 shows that, for the target platform, the implementation comprises a total of 5.6 KSLoC. This small code base reflects the overall low degree of complexity of the system. Most of the code is written in C, although functionalities such as low-level initialization and context save/restore (exception entry and exit) must be implemented in assembly. We can also see that the architecture-specific code contributes the most of the total SLoC. The largest culprit is the GIC virtualization support that amounts to almost 1/3 of the total Armv8

Table 1 Source lines of code (SLoC) and binary size (bytes) by directory.

	SLoC			size (bytes)				
	C	asm	total	.text	.data	.bss	.rodata	total
arch/armv8	2659	447	3106	22376	888	16388	482	40134
platform/xilinx/zcu104	281	0	281	464	136	0	0	600
core	1697	0	1697	14492	168	656	835	16151
lib	517	0	517	2624	0	0	24	2648
total	5154	447	5601	39956	1192	17045	1341	59535

code with about 750 SLoC. In core functionality, the code memory subsystem which includes physical page allocation and page-table management encompasses the bulk of the complexity comprising 540 SLoC. The resulting binary size is detailed in the rightmost section of Table 1. The total size of statically allocated memory is about 59 KiB. Note that the large .bss section size is mainly due to the static allocation of the root page tables. Ignoring it, this brings the total size of the final binary to be loaded to about 43 KiB.

Next, we assess the memory allocated at run-time. At boot time, each CPU allocates a private structure of 28 KiB. This structure includes the private CPU stack and page tables as well as a public page used for inter-CPU communication. For this quad-core platform, it amounts to a total of 112 KiB allocated at boot time. During initialization, Bao further allocates 4 pages (16 KiB) to use for an internal minimal allocation mechanism based on object pools. Furthermore, for each VM, the hypervisor will allocate a fixed 40 KiB for the VM global control structure plus 8 KiB for each virtual CPU. The largest memory cost for each VM will be the number of page tables which will depend first on the size of the assigned memory and memory-mapped peripherals, and second on if cache coloring is enabled or not. Table 2 shows the number of page tables used for different sizes of assigned memory. It highlights the large overhead introduced by the cache coloring mechanism on page table size. After all VMs are initialized, with the small exception of inter-CPU message allocation using the aforementioned object pools, no more memory allocation takes place.

Table 2 Page table size by VM memory size.

	no coloring		coloring	
size (MiB)	num. pages	size (KiB)	num. pages	size (KiB)
32	4	16	20	80
128	5	20	68	272
512	5	20	260	1040
1024	5	20	516	2064

3.2 Boot Overhead

In this section, we evaluate Bao's overhead on boot time (not the system's overall boot time). As such, no optimizations were carried out in any of the system's or the VMs' boot stages. In this platform, the complete boot flow includes several platform-specific boot stages: (i) a BootRom performs low-level initializations and loads the First-Stage Bootloader (FSBL) to on-chip memory, which then (ii) loads the ATF, Bao, and guest images to main memory. Next, (iii) the FSBL jumps to the ATF which then (iv) handles control to the hypervisor.

For our measurements, we use Arm's free-running architectural timer which is enabled in the early stages of ATF. Therefore, these are only approximate values to the platform's total boot time, as they do not take into account previous boot stages. We consider two cases: a

small VM (117 KiB image size and 32 MiB of memory) running FreeRTOS, and a large one (39 MiB image size and 512 MiB of memory) running Linux. For each VM, we consider the native execution (bare) scenario, and hosted execution with coloring disabled and enabled (solo and solo-col, respectively). We measure (i) hypervisor initialization as the time taken from the first instruction executed by the hypervisor to the moment it handles control to the VM, and (ii) the total boot time to the beginning of the first application inside the guest. We stress the fact that Bao does not perform guest image loading, as is the case for other embedded hypervisors. For this, it depends on a bootloader. As such, the image loading overhead is only reflected in the total time.

Table 3 Hypervisor initialization time and total VM boot time (ms).

	hyp. init. time		total boot time	
	avg	std-dev	avg	std-dev
freertos bare	n/a	n/a	2707.13	0.124
freertos solo	6.48	0.003	2720.84	0.118
freertos solo-col	9.21	0.004	2723.49	0.150
linux bare	n/a	n/a	11069.48	0.545
linux solo	9.59	0.004	11152.87	0.305
linux solo-col	155.39	1.202	11337.71	2.236

Table 3 shows the average results of 100 samples for each case. In the small VM case, the hypervisor initialization overhead is minimal (6.5 and 9.2 ms for the solo and sol-col scenarios, respectively). The total boot time increases by approximately 13 (0.5%) and 16 (0.6 %) ms, respectively, when compared with the bare scenario. In the case of the large VM running a Linux guest, Bao takes about 9.6 and a 156.2 ms to initialize itself and the VMs in the solo and solo-col case, respectively. Comparing with the native execution, the total boot time increases by about 83 (0.7 %) ms and 184 (2.4 %) ms with coloring disabled and enabled, respectively. The first point to highlight is the large increase in hypervisor initialization time with coloring enabled. This is mainly because Bao needs to color the flat image laid out by the bootloader, copying several segments of the image to color-compliant pages in the process. This is aggravated in the case of large guest images. Second, the increase in total boot time is always larger than the hypervisor initialization time. We believe this is the result of the virtualization overhead during guest initialization (e.g. 2-stage translation and GIC distributor trap and emulation).

3.3 Performance Overhead and Interference

To assess virtualization performance overhead and inter-VM interference, we employ the widely-used MiBench Embedded Benchmark Suite [14]. MiBench is a set of 35 benchmarks split into six subsets, each targeting a specific area of the embedded market: automotive (and industrial control), consumer devices, office automation, networking, security, and telecommunications. For each benchmark, MiBench provides two input data sets (small and large). We focus our evaluation on the automotive subset as this is one of the main application domains targeted by Bao. It includes three of the more memory-intensive benchmarks and therefore more susceptible to interference due to cache and memory contention [7] (qsort, susan corners, and susan edges).

Figure 2 shows the results for 1000 runs of the automotive MiBench subset. For each benchmark, we present the results as performance normalized to the bare-metal execution case, so higher values reflect poorer performance. To further investigate and understand the

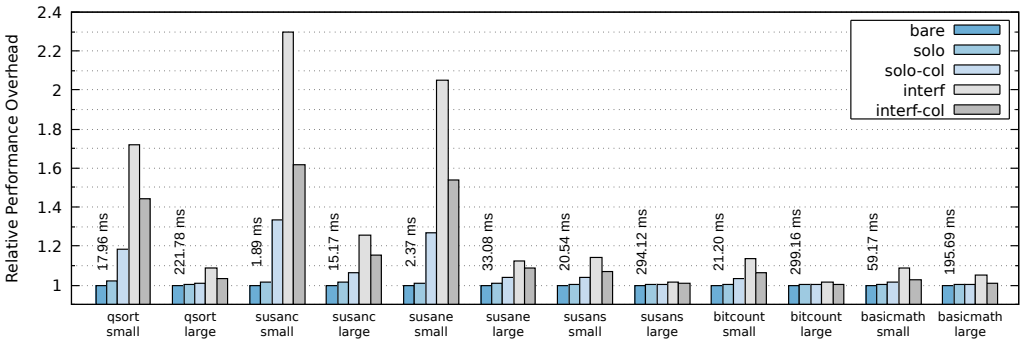

Figure 2 Performance overheads of Mibench automotive benchmark relative to bare-metal execution.

behavior of the benchmark, we collected information on L2 cache miss rate, data TLB miss rate, and stall cycle rate for memory access instructions for the qsort benchmarks. Table 4 shows the results for the small and large qsort benchmarks for each scenario.

Analyzing Figure 2, the same trend can be observed across all benchmarks to a higher or lower degree. First, observe that hosted execution causes a marginal decrease in performance. This is reflected in Table 4 by a small increase in both L2 cache and data TLB miss rates, which in turn explain the increase in memory access stall rate. As expected, this stems from the virtualization overheads of 2-stage address translation. Second, when coloring is enabled, the performance overhead is further increased. This is supported by the results in Table 4 that show an already noticeable increase across all metrics. Again, as expected, this can be explained by the fact that only half of L2 is available, and that coloring precludes the use of superpages, significantly increasing TLB pressure. In the interference scenario, there is significant performance degradation. The results in Table 4 confirm that this is due to the foreseen explosion of L2 caches misses. Finally, we can see that cache partitioning through coloring can significantly reduce interference. Table 4 shows that coloring can completely reduce L2 miss rate back to the levels of the solo colored scenario. However, looking back at Figure 2, we can see that this cutback is not mirrored in the observed performance degradation, which 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) reflected in the difference in memory stall cycle rate. As expected, basicmath and bitcount were significantly less impacted by coloring and interference, given that these are much less memory-intensive.

Another visible trend in Figure 2 is that performance degradation is always more evident in the small data set variation of the benchmark. When comparing the small and large input data set variants, we see that, despite the increase in L2 cache miss rate in Table 4 being similar, the small variant experiences greater performance degradation. We believe this might be due to the fact that, given that the small input data set benchmarks has smaller total execution times, the cache miss penalty will more heavily impact them. This idea is supported by the observed memory access stall cycle rate in Table 4, which incurs in a much higher percentage increase for the small input data set case.

3.4 Interrupt Latency

To measure interrupt latency and minimize overheads unrelated to virtualization, we crafted a minimal bare-metal benchmark application. This application continuously sets up the architectural timer to trigger an interrupt each 10 ms. As the instant the interrupt is triggered

Table 4 Average L2 miss rate, data TLB miss rate and stall cycle on memory access rate for the small and large variants of MiBench's qsort benchmark.

		bare	solo	solo-col	interf	interf-col
small	L2 miss %	15.5	15.7	22.6	38.1	22.7
	DTLB miss %	0.021	0.023	0.058	0.023	0.059
	Mem. stall cyc. %	28.6	28.7	37.4	52.6	46.6
large	L2 miss %	10.1	10.1	13.4	31.7	13.4
	DTLB miss %	0.002	0.002	0.007	0.002	0.007
	Mem. stall cyc. %	4.9	5.0	5.6	8.5	7.2

is known, we calculate the latency as the difference between the expected wall-clock time and the actual instant it starts handling the interrupt. The timer has a 10 ns resolution. Results obtained from 1000 samples for each scenario are summarized in Table 5.

Table 5 Interrupt Latency (ns).

	avg	std-dev	min	max
native	140.4	11.1	140.0	490.0
solo	571.64	50.63	560.0	2170.0
solo-col	571.75	54.74	570.0	2300.0
interf	583.95	91.64	560.0	3460.0
interf-col	583.11	99.70	570.0	3620.0

When comparing native with the standalone hosted execution, we see a significant increase in both average latency and standard deviation, of approximately 430 ns and 40 ns, respectively, and of the worst-case latency by 1680 ns. This reflects the already anticipated GIC virtualization overhead due to the trap and mode crossing costs, as well as the interrupt management and re-injection. It is also visible that coloring, by itself, does not significantly impact average interrupt latency, but slightly increases the worst-case latency. The results in Table 5 also confirm the expected adverse effects of interference by cache and memory contention in interrupt latency, especially in the worst-case. Average latency grows ≈12 ns with an increase in the standard deviation of ≈41 ns and in worst-case of 1160 ns. Enabling coloring has no expressive benefits in average latency, and actually increases standard deviation and worst-case latency. We believe this was because, in this case, the relevant interference is not actually between VMs, but between the interfering guests and the hypervisor itself, which is not itself colored.

4 Related Work

Virtualization technology was introduced in the 1970's [38]. Nowadays, virtualization is a well-established technology, with a rich body of hypervisor solutions, mainly due to the large number of use cases ranging from servers, desktops, and mobiles [4, 29, 5, 44], to high- and low-end embedded systems [16, 46, 12, 28, 41, 21, 35].

Xen [4] and KVM [26] stand as the best representative open-source hypervisors for a large spectrum of applications. Xen [4] is a bare-metal (a.k.a. type-1) hypervisor that relies on a privileged VM, called Dom0, to manage non-privileged VMs (DomUs) and interface with peripherals. KVM [26] follows a different design philosophy; it was designed as a hosted hypervisor and integrated into Linux's mainline as of 2.6.20. Although initially developed

for desktop and server-oriented applications, both hypervisors have found their place into the embedded space. Xen on Arm [19] has presented the first implementation of Xen for Arm platforms and RT-Xen [47] has extended it with a real-time scheduling framework. KVM/ARM [12], in turn, has brought to light the concept of split-mode virtualization and pushed forward the hardware virtualization specification for Arm platforms.

From a different perspective, and to cope with the strict timing requirements of embedded real-time applications, a different class of systems proposes the extension of widely-used commercial RTOSes with virtualization capabilities. Green Hills INTEGRITY Multivisor, SysGo PikeOS [20], and OKL4 MicroVisor [17] are great examples of systems that take advantage of the already developed and certified RTOS infrastructure to provide the foundation to implement virtualization capabilities as services or modules atop. Also, there is another class of systems that makes use of security-oriented technologies, e.g. Arm TrustZone [37], for virtualization. TrustZone-assisted hypervisors such as SafeG [43] and LTZVisor [36] are typically dual-OS solutions which allow the consolidation of two different execution environments, i.e. an RTOS and a GPOS. In spite of both design philosophies striving for low-performance overhead and minimal interrupt latency, they typically present some limitations and fall short while supporting multiple VMs and scaling for multi-core configurations [36, 37].

Small-sized type-1 embedded hypervisors, such as Xtratum [11], XVisor [34], Hellfire/prpl-Hypervisor [31], ACRN [23], and Minos [39] provide a good trade-off between fully-featured hypervisors and virtualization-enhanced RTOSes. Xtratum [11] was designed for safety-critical aerospace applications targeting LEON processors; nowadays, it is also available for the x86, PowerPC, and Armv7 instruction sets. Hellfire/prplHypervisor [31] was specially designed for real-time embedded systems targeting the MIPS architecture (with Virtualization Module support). XVisor [34] was designed as a tool for engaging both academia and hobbyist with embedded virtualization for Arm platforms. Intel researchers have developed ACRN [23], a lightweight hypervisor for the IoT segment and currently targeting the x86 platform. Minos [39] is an embryonic solution targeting mobile and embedded applications. Similarly to these hypervisors, Bao is also a type-1 hypervisor targeting Arm and RISC-V processors (and open to future support for MIPS or other embedded platforms); however, it distinguishes from the aforementioned solutions by following a static partition architecture which has an even reduced TCB and improved real-time guarantees.

Siemens's Jailhouse [41] pioneered the static partitioning architecture adopted by Bao. Jailhouse leverages the Linux kernel to start the system and uses a kernel module to install the hypervisor underneath the already running Linux. It then relies on this root cell to manage other VMs. Due to the proven advantages of static partitioning in embedded domains such as the automotive, other hypervisors are striving to support it. Xen has recently introduced Dom0-less execution [45], allowing DomUs to boot and execute without a Dom0, which also eliminates the Linux dependency. We strongly believe that Bao will still be able to distinguish itself from Xen Dom0-less by providing the same static partitioning benefits with a much smaller TCB and by implementing clean security features (see Section 5).

Recently, Google open-sourced Hafnium [15], a security-focused, type-1 hypevisor. It aims to provide memory isolation between a set of security domains, to better separate untrusted code from security-critical code, where each security domain is a VM.

5 On the Road

Bao's development is still at an embryonic stage. As of this writing, we are expanding support for the Arm architecture including SMMU (Arm's IOMMU) and the latest GIC versions (v3 and v4). We are also porting the system to a range of different platforms including NVIDIA's

Jetson TX2 and NXP's i.MX 8. Also, given the small size codebase, we are planning an overall refactoring to adhere to the MISRA C coding guidelines.

Bao implements cache coloring from the get-go, as a first-line of micro-architectural partitioning and isolation. We aim at implementing other state-of-the-art partitioning mechanisms (e.g. memory throttling), and color the hypervisor image itself, since we have verified that there are still contention issues between VMs or between VMs and the hypervisor. However, we believe that these issues should be supported by dedicated hardware mechanisms, to not increase code complexity and size as well as minimize overheads. Indeed, Arm has proposed the Memory System Resource Partitioning and Monitoring (MPAM) [25] extensions on Armv8.4. MPAM provides hardware support for shared cache, interconnect, and memory bandwidth partitioning. Unfortunately, no hardware featuring these extensions is available to date. We plan to implement support for MPAM using Arm Foundation Platform models, so we can test it on real hardware as soon as it is available.

Finally, since Bao is also a security-oriented hypervisor, Trusted Execution Environment (TEE) support is also on the roadmap. Typically, Arm TEEs are anchored in TrustZone technology, a set of secure hardware extensions that splits the platform into a secure and normal world [37]. TEE kernels and applications run on the secure side, while everything else (including the hypervisor) executes in the normal world. Currently, TrustZone does not support multiple isolated TEEs. Several secure world virtualization approaches have been proposed [18, 10, 24] and, recently, Arm has added secure world hypervisor support on Armv8.4. However, the dual-world approach of TrustZone-based TEEs has been shown to be fundamentally flawed [9]. Furthermore, we believe running an additional secure hypervisor would unnecessarily increase complexity, and that the secure world should only be used to encapsulate absolute security primitives (e.g. secure boot, attestation, authentication, key management). Bao's approach will take this into account, and using the already existing virtualization mechanisms, with no additional scheduling logic, will allow for multiple VMs inside a single hardware partition in the normal world. TEEs will be deployed on auxiliary VMs and only executed per request of the main guest. Another advantage of this approach is that it is portable and scalable across architectures and not specific to Arm.

6 Conclusion

In this paper, we presented the Bao hypervisor, a minimal, standalone and clean-slate implementation of the static partitioning architecture as a lightweight alternative to existing embedded hypervisors. Although development is still at an embryonic stage, preliminary evaluation shows it incurs only minimal virtualization overhead. We outline Bao's development roadmap which includes extended platform support and per-partition TEE support. Bao will be open-sourced by the end of 2019 in hopes of engaging both academia and industry in tackling the challenges of VM isolation and security.

References

1 L. Abeni and D. Faggioli. An Experimental Analysis of the Xen and KVM Latencies. In *2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)*, pages 18–26, May 2019. doi:10.1109/ISORC.2019.00014.

2 P. Axer, R. Ernst, He. Falk, A. Girault, D. Grund, N. Guan, B. Jonsson, P. Marwedel, J. Reineke, C. Rochange, M. Sebastian, Reinhard Von Hanxleden, R. Wilhelm, and W. Yi. Building Timing Predictable Embedded Systems. *ACM Trans. Embed. Comput. Syst.*, 13(4):82:1–82:37, March 2014. doi:10.1145/2560033.

3 A. Bansal, R. Tabish, G. Gracioli, R. Mancuso, R. Pellizzoni, and M. Caccamo. Evaluating the Memory Subsystem of a Configurable Heterogeneous MPSoC. In *Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, page 55, 2018.

4 P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM. doi:10.1145/945445.945462.

5 K. Barr, P. Bungale, S. Deasy, V. Gyuris, P. Hung, C. Newell, H. Tuch, and B. Zoppis. The VMware Mobile Virtualization Platform: Is That a Hypervisor in Your Pocket? *SIGOPS Oper. Syst. Rev.*, 44(4):124–135, December 2010. doi:10.1145/1899928.1899945.

6 M. Bechtel and H. Yun. Denial-of-Service Attacks on Shared Cache in Multicore: Analysis and Prevention. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 357–367, April 2019. doi:10.1109/RTAS.2019.00037.

7 A. Blin, C. Courtaud, J. Sopena, J. Lawall, and G. Muller. Maximizing Parallelism without Exploding Deadlines in a Mixed Criticality Embedded System. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 109–119, July 2016. doi:10.1109/ECRTS.2016.18.

8 P. Burgio, M. Bertogna, I. S. Olmedo, P. Gai, A. Marongiu, and M. Sojka. A Software Stack for Next-Generation Automotive Systems on Many-Core Heterogeneous Platforms. In *2016 Euromicro Conference on Digital System Design (DSD)*, pages 55–59, August 2016. doi:10.1109/DSD.2016.84.

9 D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto. SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems. In *IEEE Symposium on Security and Privacy (S&P)*, Los Alamitos, CA, USA, 2020.

10 G. Cicero, A. Biondi, G. Buttazzo, and A. Patel. Reconciling security with virtualization: A dual-hypervisor design for ARM TrustZone. In *2018 IEEE International Conference on Industrial Technology (ICIT)*, pages 1628–1633, February 2018. doi:10.1109/ICIT.2018.8352425.

11 A. Crespo, I. Ripoll, and M. Masmano. Partitioned Embedded Architecture Based on Hypervisor: The XtratuM Approach. In *2010 European Dependable Computing Conference*, pages 67–72, April 2010. doi:10.1109/EDCC.2010.18.

12 C. Dall. *The Design, Implementation, and Evaluation of Software and Architectural Support for ARM Virtualization.* PhD thesis, Columbia University, 2018.

13 Q. Ge, Y. Yarom, D. Cock, and G. Heiser. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *Journal of Cryptographic Engineering*, 8:1–27, April 2018. doi:10.1007/s13389-016-0141-6.

14 M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, pages 3–14, December 2001. doi:10.1109/WWC.2001.990739.

15 Hafnium. Hafnium, 2019. URL: https://hafnium.googlesource.com/hafnium/.

16 G. Heiser. The Role of Virtualization in Embedded Systems. In *Workshop on Isolation and Integration in Embedded Systems*, 2008. doi:10.1145/1435458.1435461.

17 G. Heiser and B. Leslie. The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors. In *Proceedings of the First ACM Asia-pacific Workshop on Workshop on Systems*, APSys '10, pages 19–24, New York, NY, USA, 2010. ACM. doi:10.1145/1851276.1851282.

18 Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan. vTZ: Virtualizing ARM TrustZone. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 541–556, Vancouver, BC, August 2017. USENIX Association.

19 J. Hwang, S. Suh, S. Heo, C. Park, J. Ryu, S. Park, and C. Kim. Xen on ARM: System Virtualization Using Xen Hypervisor for ARM-Based Secure Mobile Phones. In *IEEE Consumer*

Communications and Networking Conference, pages 257–261, 2008. doi:10.1109/ccnc08.2007.64.

20 R. Kaiser and S. Wagner. Evolution of the PikeOS microkernel. In *First International Workshop on Microkernels for Embedded Systems*, volume 50, 2007.

21 N. Klingensmith and S. Banerjee. Hermes: A Real Time Hypervisor for Mobile and IoT Systems. In *Proceedings of the 19th International Workshop on Mobile Computing Systems and Applications*, HotMobile '18, pages 101–106, New York, NY, USA, 2018. ACM. doi:10.1145/3177102.3177103.

22 T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna. Deterministic Memory Hierarchy and Virtualization for Modern Multi-Core Embedded Systems. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–14, April 2019. doi:10.1109/RTAS.2019.00009.

23 H. Li, X. Xu, J. Ren, and Y. Dong. ACRN: A Big Little Hypervisor for IoT Development. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2019, pages 31–44, New York, NY, USA, 2019. ACM. doi:10.1145/3313808.3313816.

24 W. Li, Y. Xia, L. Lu, H. Chen, and B. Zang. TEEv: Virtualizing Trusted Execution Environments on Mobile Platforms. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2019, pages 2–16, New York, NY, USA, 2019. ACM. doi:10.1145/3313808.3313810.

25 Arm Ltd. Arm Architecture Reference Manual Supplement - Memory System Resource Partitioning and Monitoring (MPAM), for Armv8-A, 2018. URL: https://developer.arm.com/docs/ddi0598/latest.

26 U. Lublin, Y. Kamay, D. Laor, and A. Liguori. KVM: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium*, 2007.

27 R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 45–54, April 2013. doi:10.1109/RTAS.2013.6531078.

28 J. Martins, J. Alves, J. Cabral, A. Tavares, and S. Pinto. uRTZVisor: A Secure and Safe Real-Time Hypervisor. *Electronics*, 6(4), 2017. doi:10.3390/electronics6040093.

29 Mark F. Mergen, Volkmar Uhlig, Orran Krieger, and Jimi Xenidis. Virtualization for High-performance Computing. *SIGOPS Oper. Syst. Rev.*, 40(2):8–11, April 2006. doi:10.1145/1131322.1131328.

30 P. Modica, A. Biondi, G. Buttazzo, and A. Patel. Supporting temporal and spatial isolation in a hypervisor for ARM multicore platforms. In *2018 IEEE International Conference on Industrial Technology (ICIT)*, pages 1651–1657, February 2018. doi:10.1109/ICIT.2018.8352429.

31 C. Moratelli, S. Zampiva, and F. Hessel. Full-Virtualization on MIPS-based MPSOCs Embedded Platforms with Real-time Support. In *Proceedings of the 27th Symposium on Integrated Circuits and Systems Design*, SBCCI '14, pages 44:1–44:7, New York, NY, USA, 2014. ACM. doi:10.1145/2660540.2661012.

32 D. G. Murray, G. Milos, and S. Hand. Improving Xen Security Through Disaggregation. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, pages 151–160, New York, NY, USA, 2008. ACM. doi:10.1145/1346256.1346278.

33 R. Müller, D. Danner, W. S. Preikschat, and D. Lohmann. Multi Sloth: An Efficient Multi-core RTOS Using Hardware-Based Scheduling. In *26th Euromicro Conference on Real-Time Systems*, pages 189–198, July 2014. doi:10.1109/ECRTS.2014.30.

34 A. Patel, M. Daftedar, M. Shalan, and M. W. El-Kharashi. Embedded Hypervisor Xvisor: A Comparative Analysis. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 682–691, March 2015. doi:10.1109/PDP.2015.108.

35 S. Pinto, H. Araujo, D. Oliveira, J. Martins, and A. Tavares. Virtualization on TrustZone-Enabled Microcontrollers? Voilà! In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 293–304, April 2019. doi:10.1109/RTAS.2019.00032.

36 S. Pinto, J. Pereira, T. Gomes, A. Tavares, and J. Cabral. LTZVisor: TrustZone is the Key. In *29th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 4:1–4:22, 2017. doi:10.4230/LIPIcs.ECRTS.2017.4.

37 S. Pinto and N. Santos. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Comput. Surv.*, 51(6):130:1–130:36, January 2019. doi:10.1145/3291047.

38 Gerald J. Popek and Robert P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Commun. ACM*, 17(7):412–421, July 1974. doi:10.1145/361011.361073.

39 Minos Project. Minos - Type 1 Hypervisor for ARMv8-A, 2019. URL: https://github.com/minos-project/minos-hypervisor.

40 E. Qaralleh, D. Lima, T. Gomes, A. Tavares, and S. Pinto. HcM-FreeRTOS: Hardware-centric FreeRTOS for ARM multicore. In *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*, pages 1–4, September 2015. doi:10.1109/ETFA.2015.7301570.

41 R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer. Look Mum, no VM Exits!(Almost). In *Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2017.

42 A. Sadeghi, C. Wachsmann, and M. Waidner. Security and privacy challenges in industrial Internet of Things. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2015. doi:10.1145/2744769.2747942.

43 D. Sangorrín, S. Honda, and H. Takada. Dual Operating System Architecture for Real-Time Embedded Systems. In *International Workshop on Operating Systems Platforms for Embedded Real-Time Applications, Brussels, Belgium*, pages 6–15, 2010.

44 J. Shuja, A. Gani, K. Bilal, A. Khan, S. Madani, S. Khan, and A. Zomaya. A Survey of Mobile Device Virtualization: Taxonomy and State of the Art. *ACM Computing Surveys*, 49(1):1:1–1:36, April 2016. doi:10.1145/2897164.

45 S. Stabellini. Static Partitioning Made Simple. In *Embedded Linux Conference (Noth America)*, 2019. URL: https://www.youtube.com/watch?v=UfiP9eAV0WA.

46 P. Varanasi and G. Heiser. Hardware-supported Virtualization on ARM. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 11:1–11:5, New York, NY, USA, 2011. ACM. doi:10.1145/2103799.2103813.

47 S. Xi, J. Wilson, C. Lu, and C. Gill. RT-Xen: Towards Real-time Hypervisor Scheduling in Xen. In *Proceedings of the Ninth ACM International Conference on Embedded Software*, EMSOFT '11, pages 39–48, New York, NY, USA, 2011. ACM. doi:10.1145/2038642.2038651.

48 H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, April 2013. doi:10.1109/RTAS.2013.6531079.