Bao Hypervisor: Temporal and Fault Isolation

Riccardo Tammariello
University of Naples Federico II
r.tammariello@studenti.unina.it

Simone Vallifuoco
University of Naples Federico II
s.vallifuoco@studenti.unina.it

Abstract

Virtualization is emerging as a practical solution to manage Mixed Criticality Systems, as it can represent a natural enabler for the isolation that components with different criticality require when running on the same hardware platform. Static partitioning hypervisors statically distribute physical resources to different subsystems, thus preventing, when possible, latencies due to contentions and runtime overhead; this class of hypervisors represents a very beneficial solution for embedded real-time domains, since they can provide strong isolation, reduce performance overhead, achieve safety and security, all while having very small size. The aim of this study is to delve into Bao, a safety-oriented and lightweight static partitioning hypervisor. First, the general characteristics of Bao Hypervisor have been described. Then, the mechanisms through which it ensures temporal, spatial and fault isolation have been explored. Finally, these isolation mechanisms have been directly tested on a Xilinx ZCU104 board, in order to better evaluate their effectivness.

Keywords: Virtualization, Hypervisor, Mixed Criticality, Bao, Isolation, Partitioning, ARMv8

1 Introduction

Nowadays, there is a trend of interest in mixed criticality systems (MCSs), and virtualization emerges as a natural solution to achieve strong spatial, temporal and fault isolation; indeed, VMs can be seen as isolated environments on which we can perform different applications where some of those are safety-critical, while others are not.

However, solutions like the most used hypervisors, such as KVM and XEN, were not designed having embedded constraints and requirements in mind, neither they are an optimal solution; for example, they use virtualized I/O mechanisms that add a lot of overhead [4].

On the other hand, static partitioning hypervisors seem to address embedded concerns: under systems like these, there is a one-to-one mapping between virtual CPUs and physical CPUs, so there is no contention for CPU time; devices are mapped directly into the guests, avoiding any added I/O overhead. To implement these features efficiently, these hypervisors are highly dependent on virtualization hardware support, such as Armv8-A virtualization support.
The best-known hypervisor designed for this purpose is Jailhouse, but it still depends on Linux to boot and manage its VMs, which is bad from a security and safety perspective and also complicates the certification process. Also, due to the proven advantages of static partitioning in embedded domains, other hypervisors have begun to support it; one of the main example is Xen with the Dom0-less configuration, which also eliminates the Linux dependency. However Bao Hypervisor aims to providing the same static partitioning benefits with a much smaller TCB.
Therefore, Bao Hypervisor, an open source project born in 2020, is emerging as a good alternative, since it is a minimal implementation of the static partitioning architecture. In fact, results regarding size, boot, performance, and interrupt latency, show this approach incurs only minimal virtualization overhead [4].
This study is structured as follows: firstly we introduced Bao and its main characteristics, focusing on its isolation mechanisms; secondly we directly tested temporal and fault isolation on a Xilinx ZCU104 board with a Linux + FreeRTOS configuration.
The written or adapted code can be found in the github repository related to this project (RTIS: Bao Hypervisor, RXM).

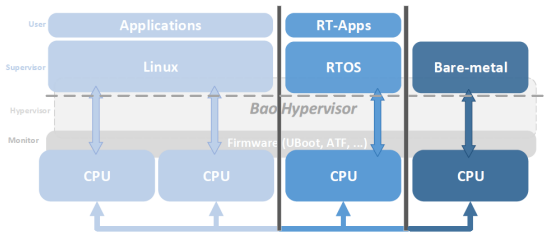

Figure 1: Bao's architecture [8]

2 Bao Overview

Bao (from Mandarin Chinese "b˘aohù", meaning "to protect") is a security and safety-oriented, lightweight type-1 hypervisor. Designed for MCSs, its main focuses are isolation for fault-containment and real-time behavior. In order to achieve the static partitioning hypervisor architecture (Figure 1), it simply uses a thin layer of privileged software that uses ISA virtualization support: resources are statically partitioned and assigned at VM instantiation time; memory is statically assigned; IO is pass-through only; virtual interrupts are directly mapped to physical ones; it implements a 1-1 mapping of virtual to physical CPUs, with no need for a scheduler. As in such systems VMs often have the need to interact with each other, the hypervisor also provides simple primitives for inter-VM communication, based on a static shared memory and asynchronous notifications in the form of inter-VM interrupts triggered through an hypercall [7].

Despite following an architecture similiar to Jailhouse's one, Bao does not rely on any external dependence (except the firmware to perform low-level platform management), such as on privileged VMs running untrustable, large monolithic GPOSs, and, as such, includes a much smaller TCB.

Bao is built around a set of key principles that guide its implementation [7]:

- **Minimality and Simplicity**: The code base is as minimal and simple as possible; it is implemented in only about 8000 LOC [9] and requires about 50KB of memory on the target system; run-time memory requirements add up to about 250KB [4]. This is possible since Bao is implemented only in architectures which provide hardware-assisted virtualization: taking advantage of such mechanisms significantly reduces virtualization overheads and the system's TCB by minimizing code size and complexity.

- **Least Privilege**: The system is designed to limit each component's access to only what is necessary; each core can only access information related to its own partition; the hypervisor itself cannot directly access the memory of virtual machines, which mandates that all hypercall arguments passed by value in processor registers and never by reference [7]; only essential virtualization mechanisms are performed at the hypervisor's privileged level, with all other functionalities being moved to the VMs.

- **Strong Isolation**: a great effort is done to achieve isolation in all its forms: temporal, spatial and fault isolation, by applying different strategies at different levels. Temporal isolation, is the ability to isolate or limit the impact of resource consumption (e.g. CPU, network, disk) of a virtual machine on the performance degradation of other virtual machines [3]. this principle becomes of the utmost importance in mixed criticality environments, where a main goal is to preclude non-critical subsystems to interfere with critical ones. The other crucial property is spatial isolation (also known as memory isolation). This characteristic represents the capability of isolating code and data between virtual domains as well as between virtual domains and hosts. This indicates that a task should not be allowed to change the private data of other tasks, including devices assigned to a specific task. Spatial isolation is implemented using hardware memory protection mechanisms, such as the Memory Management Unit (MMU). Considering the case of shared physical devices, also I/O isolation becomes important. Often, the IOMMU is used to properly resolve the isolation of memory-mapped devices [3]. Finally, fault isolation, or fault/error containment, provides the containement of failures in a VM, preventing the propagation to the external environments.

We focused more deeply on Bao Hypervisor's mechanisms of isolation in the dedicated section.

3 Supported Platforms

Bao originally targeted Armv8-A architectures (e.g. Xilinx Zynq UltraScale+ MPSoC ZCU104, Ultra96 Zynq UltraScale+ ZU3EG, NXP MCIMX8QM-CPU, NVIDIA Jetson TX2, 96Boards HiKey 960, Raspberry Pi 4). The mainline also includes support for RISC-V, while Armv7-A, and Armv8-R ports are in the making, at the time of this writing [1].

As already mentioned, as the majority of static partitioning hypervisors, Bao strongly relies on hardware virtualization support. Since these hardware technologies are the real enabler of this kind of virtualization techniques and hypervisors, and given the fact that the main target architectures of Bao and most of the supported hardware platforms are Armv8-A architectures, it's important to recall some of the main Armv8-A Arch64 hardware virtualization support mechanisms. The system's architecture is shown in Figure 2.

The ARMv8 exception model defines Exception levels EL0-EL3, where: EL0 has the lowest software execution privilege; increased values of n, from 1 to 3, indicate increased software execution privilege. Software running at EL2 or higher has access to several controls for virtualization:

- Stage 2 translation
- EL1/0 instruction and register access trapping
- Virtual exception generation

Stage 2 translation allows a hypervisor to control a view of memory in a Virtual Machine (VM). Specifically, it allows the hypervisor to control which memory-mapped system resources a VM can access, and where those resources appear in the address space of the VM. Stage 2 translation can be used to ensure that a VM can only see the resources that are allocated to it, and not the resources that are allocated to other VMs or the hypervisor. For memory address translation, stage 2 translation is a second stage of translation. To support this, a new set of translation tables known as Stage 2 tables, are required [2]. Arm also defines the SMMU, that extends stage 2 regime to also cover other masters, like DMA controllers. The SMMU, sometimes also called IOMMU, is a key technology to enable device pass-through I/O, which is explored in the section dedicated to isolation mechanisms.

Arm also provides superpages (blocks) support, which enables to map large contiguous memory

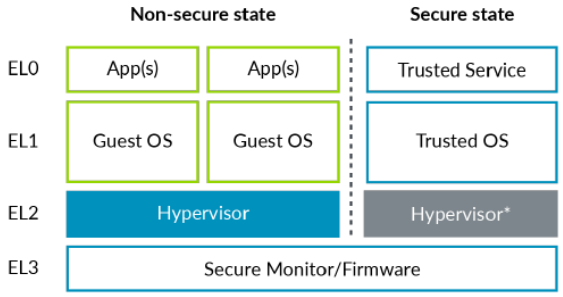

Figure 2: AArch64 virtualization [2]

regions; can improve guest performance by minimazing translation overhead and TLB pressure.

Concerning interrupts, the Generic Interrupt Controller (GIC) is the interrupt router and arbiter in the Arm architectures; while in the newest versions (i.e., GICv4), the interrupts are directly delivered to guest partitions, bypassing the hypervisor, in the older GIC versions (i.e., GICv2), all interrupts are forwarded to the hypervisor which must re-inject the interrupt in the VMs.The advantage of new approach is that the hypervisor only needs to set up the virtual interface, and does not need to emulate it. This approach reduces the number of times that the execution needs to be trapped to EL2, and therefore reduces the overhead of virtualizing interrupts.

4 Isolation Mechanisms

In order to guarantee isolation, the main requirement of MCSs, Bao, as a static partitioning hypervisor, creates strong isolated environments by setting up private mappings for each core.

First of all, Bao implements a 1-1 mapping of virtual CPUs to physical ones, so there is no contention for the system's CPUs and no scheduler is needed, thus allowing temporal isolation. It's worth mentioning that an experimental Bao branch implements a policy-free mechanism that relaxes the static 1:1 mapping of vCPUs to pCPUs, allowing multiple vCPUs belonging to different VMs to execute on the same pCPU, in order to achieve better utilization of physical CPUs, which is the main drawback of the static 1-1 mapping of CPUs; this mechanism, called VM Stacking, permits at configuration time to define a tree of vCPUs for each pCPU; at runtime, the executing vCPU can schedule any of its child vCPUs triggering a hypercall. A vCPU can also issue a hypercall to give back execution to its parent vCPU. If during a child vCPU

execution, an interrupt arrives targeting a parent vCPU, the hypervisor will immediately schedule it [9].

Memory is statically assigned to each VM, using 2-stage translation hardware virtualization support, which allows the hypervisor to control which memory-mapped system resources a VM can access, and where those resources appear in the address space of the VM; this ensures that a VM can only see and access the resources that are allocated to it, and not the resources that are allocated to other VMs or the hypervisor, following the principle of least privilege. Only a single address space is managed per CPU, and page tables are completely set-up at initialization. Furthermore, in order to minimize translation overhead and TLB pressure, Bao uses superpages (in Arm terminology, blocks) whenever possible. However, all cores do share mappings for a per-CPU area for inter-core communication, and the hypervisor's image.

In order to avoid interference due to contention at shared last-level-caches (LLCs), Bao adopts a page coloring solution, enabling LLC partitioning, so that each VM gets its own allocation of cache entries and there are no cache entries shared between VMs; this partitioning can also mitigate cache-based timing side-channels used in a myriad of modern attacks. Anyway cache coloring also forces the partitioning of actual physical memory, leading to fragmentation and memory waste. Hypervisor coloring is also allowed: this permits to reserve a color to the hypervisor image, in order to avoid or reduce hypervisor induced LLC cache misses; anyway, this has the drawbacks of wasting an entire color for the hypervisor image, which is, in the case of SPHs, very small, and reducing the number of colors available to the VMs.

Regarding I/O virtualization, Bao implements a pass-through only I/O configuration, directly assigning peripherals to VMs in an exclusive way. As in the supported architectures, specifically Arm, all I/O is memory-mapped, this is implemented for free by using the existing memory mapping mechanisms and 2-stage translation provided by virtualization support. It's worth noting that the hypervisor does not verify the exclusive assignment of a given peripheral, which allows for several guests to share it, even though in a non-supervised manner [8].

Regarding interrupts, while in the latest versions of the GIC (i.e., GICv4), the interrupts are directly delivered to guest partitions, bypassing the hypervisor, in the older GIC versions (i.e., GICv2), all interrupts are forwarded to the hypervisor which must re-inject the interrupt in the VMs, leading to an increase in interrupt latency [8].

Thanks to the hardware resource partitioning, fault isolation is also improved by design; the limited sharing of resources should prevent the fault propagation between VMs.

5 Configuration

In order to implement the static partitioning features, a configuration file *config.c* is used to statically assign resources to the guests at instantiation time.

The main configuration structure used for this purpose is the struct *config* (defined in *config.h* file), whose specification is reported below:

```
extern struct config {
    struct {
        bool relocate;
        paddr_t base_addr;
        colormap_t colors;
    } hyp;
    size_t shmemlist_size;
    struct shmem *shmemlist;
    size_t vmlist_size;
    struct vm_config vmlist[];
} config;
```

where the struct *hyp* sets the location (*base_addr*) of the hypervisor if *relocate* is set to true (this is only meaningful to MPU-based platforms, and default hypervisor base address is platform's base address); *colors* is a bitmap for the assigned colors of the hypervisor; *shmemlist_size* and *shmemlist* are used for the definition of shared memory regions to be used by VMs; *vmlist_size* is the number of VMs specified by the configuration; *vmlist* is the array list with VMs configuration structures, *vm_config* (defined in *config.h* file), whose specification is reported below:

```
struct vm_config {
    struct {
        vaddr_t base_addr;
        paddr_t load_addr;
        size_t size;
        bool separately_loaded;
        bool inplace;
    } image;
    vaddr_t entry;
```

```
    cpumap_t cpu_affinity;
    colormap_t colors;
    struct vm_platform platform;
};
```

where the struct *image* gives information about the VM's image size and location, both in the virtual machine's address space (*base_addr*) and the hypervisor address space (*load_addr*); *cpu_affinity* is a bitmap signaling the preferred physical CPUs assigned to the VM: if this value is each mutual exclusive for all the VMs, this field allows to direcly assign specific physical CPUs to the VMs; *colors* is a bitmap for the assigned colors of the VM; *platform* is a description of the virtual platform available to the guest; struct *vm_platform* is defined in *vm.h*:

```
struct vm_platform {
    size_t cpu_num;
    size_t region_num;
    struct vm_mem_region *regions;
    size_t ipc_num;
    struct ipc *ipcs;
    size_t dev_num;
    struct vm_dev_region *devs;
    bool mmu;
    struct arch_vm_platform arch;
};
```

where *cpu_num* is the number of CPUs assigned to the VM; *region_num* and *regions* are used to specify the memory region(s) reserved to the image of the VM in terms of base address and size (a boolean attribute *place_phys* can be set to *true* inside the struct to specify also the physical base address with the *phys* attribute); *ipc_num* and *ipcs* are used to specify the shared memory area(s)'s location, size and interrupt line: this is a shared memory area between the VMs that allows communication between the guests; in MPU-based platforms which might also support virtual memory the hypervisor sets up the VM using an MPU by default; if the user wants this VM to use the MMU they must set the *mmu* parameter to true [1]; *arch* is used in armv8 platforms to specify the GIC (Generic Interrupt Controller) addresses.

6 Experimental Setup

All the tests have been executed on a Xilinx Zynq UltraScale+ MPSoC ZCU104 board, with Bao Hypervisor running on top of it and hosting two virtual machines: a non-critical one, with oeprating system Linux v6.1, and a critical one, with FreeRTOS 10.5.1. The non-critical VM has been assigned 3 cpus, while the critical one has been assigned 1 cpu.

In order to build, configure and run Bao in dual-guest (linux+freertos) mode, this guide has been followed: https://github.com/bao-project/bao-demos.

The setup flow, however, can be summarized as follows:

1. Install the aarch64-none-elf- cross-compile toolchain (version 11.2) and other dependencies;
2. Build the guests, i.e. Linux and FreeRTOS;
3. Build Bao Hypervisor;
4. Get pre-built Zynq UltraScale+ MPSoC Firmware for ZCU104;
5. Prepare a U-boot image of the final system;
6. Copy the firmware and bao's final image to a SD-card;
7. Setup the board through the SD-card.

7 Isolation tests

In order to test temporal and fault isolation of the virtual machines running on top of Bao, two variants of the FreeRTOS guest main program have been coded, one executing a product of two matrix of size 100x100 of *uint32_t*, and the other one executing a quicksort algorithm on an array of 10000 *uint32_t* elements; these two algorithms have been chosen considering their intensive use of the cache and the memory. When the FreeRTOS guest's task executes these operations, it measures the time that has been required to complete a single repetition of them, using the *xTaskGetTickCount* function, which counts the number of ticks since *vTaskStartScheduler* function was called (known the frequency of the tick, set through the constant *configTICK_RATE_HZ* in FreeRTOSConfig.h file, the time in microseconds has been obtained). Note that the accuracy of the measurement is thus limited by the resolution of the clock, which depends on the tick rate: in our case, the tick rate has been set to 1kHz, as suggested in FreeRTOS official documentation [5], since a higher rate would cause overhead and inefficiency for tick interrupts handling. In order to execute user programs on linux guest, the source code has been first cross-compiled

with AArch64 GNU/Linux target cross-compiler, and then copied to the guest via *ssh*, using the *scp* command.

7.1 Temporal Isolation tests

The operations (matrix product or quicksort) have been repeated 100 times and the mean of the calculation time has been obtained, for 100 times for each configuration, thus having 100 samples of average calculation time (calculated on the 100 repetitions of the product of matrix or quicksort) for each configuration.

First of all, the measurement has been conducted in standard conditions, i.e. with no stress of the memory by the linux guest. This has been done with both coloring enabled and disabled. In order to enable the coloring, the *colors* parameter in the *vm_config* structure has been set to 0xF0F0F0F0 for the linux guest and to 0x0F0F0F0F for the freeRTOS guest. When the *colors* parameter is set to null or 0x00000000, the coloring is disabled instead. For the temporal isolation tests, the calculation time of the freeRTOS task has been measured in conditions of memory stress made by the linux guest in order to compare it to the calculation time measured in the standard conditions described above. The memory stress has been realized through the stress-ng tool. In particular, the following commands have been executed from the linux guest to generate stress[10]:

```
# stress-ng --matrix 3 --matrix-size 64
# stress-ng --class memory --all 3
# stress-ng --class cpu-cache --all 3
# stress-ng --brk 3 --stack 3 --bigheap 3
```

Stress-ng with the *–matrix* option starts N (3) workers that perform various matrix operations on floating point values. the *–class* option specifies the class of stressors to run; the *–brk* option starts N workers that grow the data segment by one page at a time using multiple *brk()* calls; the *–stack* option starts N workers that rapidly cause and catch stack overflows by use of *alloca()*.

The calculation time of the freeRTOS task has been measured in each of these condition (i.e. for each of the above stress-ng commands), with both coloring enabled and disabled.

7.1.1 Hypervisor coloring tests

In order to enable hypverisor coloring, the *colors* parameter in the *config* structure has been set to 0x0000000F: in this way, a cache (and memory) partition has been reserved to the hypervisor image. In this configuration, the *colors* parameter in FreeRTOS's *vm_config* structure has been set to 0xF0F0F0F0, while Linux's *colors* parameter has been set to 0x0F0F0F00, so one less color has been assigned to the Linux guest with respect to the previous configuration.

The calculation time under this configuration was calculated as explained before, running the same *stress-ng* commands.

7.2 Fault Injection tests

Fault injection is a technique for evaluating robustness by artificially inducing a fault and observing the system's response [6].

7.2.1 Configuration faults

The first kind of faults that have been injected in the system are configuration faults: intentionally, incorrect Bao configuration files have been used; in particular, the following configuration faults have been introduced to observe the system's behavior:

- Assign a partition more cpus then available;
- Assign a partition a memory area already assigned to the other partition;
- Assign a partition a memory area already assigned to the hypervisor image (0x50000000).

By introducing these faults at the configuration level, Bao's robustness to incorrect configurations can be evaluated, and the fault containment to the incorrectly configured partition can be observed.

7.2.2 Cache coloring fault injection

In order to evaluate Bao's cache coloring mechanism's robustness, the fault injection has been applied at memory initialization source code level too. The core function called during the memory allocation steps, at VMs initialization phase, is:

```
bool pp_alloc_clr(struct page_pool *pool,
size_t n, colormap_t colors, struct ppages
*ppages)
```

This function is defined in *src/core/mmu/mem.h* and called from *src/core/mem.c* to assign the specified colors to the VMs, only if the *colors* bitmap is different from NULL (therefore if colors have been assigned to that VM during configuration). In particular, the *pp_alloc_clr* function tries to find n contiguous pages that fit the given *colors* bitmap, and if so, mark them as allocated and assign them

	avg (ms)	std (ms)	min (ms)	max (ms)
no stress, no col	7.991	0.004	7.990	8.000
matrix, no col	8.015	0.005	8.000	8.020
memory, no col	8.161	0.029	8.060	8.400
cpu-cache, no col	8.427	0.014	8.360	8.490
brk, no col	8.221	0.122	8.040	8.510
no stress, col	7.992	0.004	7.990	8.000
matrix, col	8.001	0.005	7.980	8.010
memory, col	8.078	0.106	8.040	8.130
cpu-cache, col	8.258	0.019	8.100	8.310
brk, col	8.169	0.094	8.020	8.430
no stress, hyp col	7.992	0.003	7.990	8.000
matrix, hyp col	8.005	0.005	7.990	8.010
memory, hyp col	8.118	0.015	8.080	8.180
cpu-cache, hyp col	8.425	0.028	8.170	8.470
brk, hyp col	8.195	0.070	8.070	8.410

Table 1: Results of the matrix product execution

	avg (ms)	std (ms)	min (ms)	max (ms)
no stress, no col	1.756	0.009	1.740	1.780
matrix, no col	1.757	0.008	1.730	1.780
memory, no col	1.761	0.014	1.720	1.820
cpu-cache, no col	1.791	0.017	1.750	1.820
brk, no col	1.801	0.083	1.710	2.230
no stress, col	1.757	0.008	1.740	1.780
matrix, col	1.757	0.008	1.740	1.780
memory, col	1.758	0.013	1.730	1.790
cpu-cache, col	1.788	0.015	1.750	1.820
brk, col	1.779	0.055	1.730	2.030
no stress, hyp col	1.755	0.008	1.740	1.770
matrix, hyp col	1.756	0.008	1.740	1.780
memory, hyp col	1.758	0.013	1.730	1.790
cpu-cache, hyp col	1.789	0.019	1.750	1.830
brk, hyp col	1.777	0.044	1.730	1.950

Table 2: Results of the quicksort execution

to that VM. Following the Ballista approach [6], we have performed fault injection at the API level using combinations of valid and exceptional inputs, in particular 0 or -1 for integers and NULL for pointers.

7.2.3 Fault isolation between guests

In order to evaluate Bao's fault isolation properties, different faults have been directly introduced in the guest user programs, and their propagation to the other guest and to the rest of the system has been observed:

- Run *echo c > /proc/sysrq-trigger* in linux guest to cause the crash of the VM;
- Try to access a memory location not reserved to the VM;
- Saturate one guest's heap area.

We have also tried to reproduce these tests activating the aforementioned faults during the communication between the guests via the shared memory, in order to observe the consequences and the possible propagation of these faults due to the inter-VMs communication mechanism.

8 Results

The isolation and fault injection tests conducted have produced the following results.

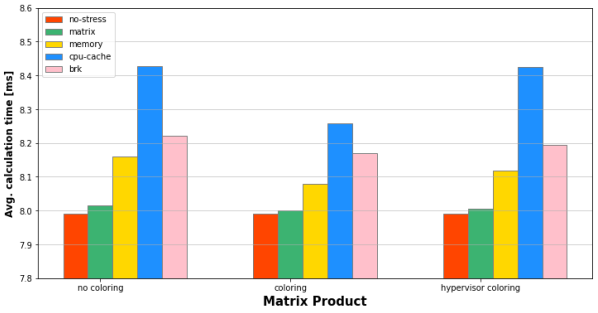

Figure 3: Results of the matrix product execution

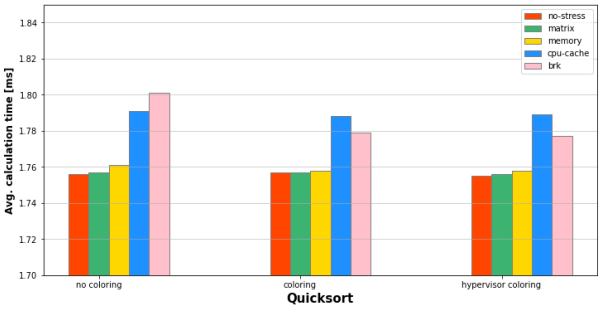

Figure 4: Results of the quicksort execution

8.1 Temporal Isolation results

The results have been summarized in the tables 1 and 2: no-stress/matrix/memory/cpu-cache/brk refers to the workload executed by the co-running linux guest, col/no-col/hyp-col stands for coloring enabled (col), coloring disabled (no col), or coloring enabled with hypervisor coloring too (hyp col). Figures 3 and 4 compare the results obtained in the different configurations.

First of all, the impact of interference, caused by stress workloads, on the calculation time is well known, and, as expected, it affects the calculation time when no cache coloring is enabled. This is due to the increased L2 cache misses caused by the interference. Cache coloring has been shown to be an effective way to mitigate this interference, as it can reduce L2 cache misses even when running stress workloads on Linux to the same level as in non-stress conditions (and coloring enabled). However, it is important to note that cache coloring does not completely eliminate interference: while it can help reduce L2 cache misses, interference can also occur at other levels of the hierarchy, such as interconnects and memory controller, which can also impact the performance of the system. Finally, hypervisor image coloring seems to provide no considering benefits compared to cache coloring without hypervisor image coloring. In fact, hypervisor coloring reduce the number of colors available to the guest VMs, which can result in increased L2 cache misses, leading to degraded performance.

8.2 Fault Injection results

8.2.1 Configuration Faults results

The results obtained in the configuration fault injection tests are the following:

- Assign a partition more cpus then available (performed on both partitions):
  Output:
  No error message reported
  Result:
  Incorrect partition doesn't start; Correct partition starts and runs.

- Assign to FreeRTOS guest's image the memory area already assigned to Linux guest's image:
  Output:
  BAO_ERROR(no handler for abort ec = 0x20)
  Result:
  FreeRTOS doesn't start; Linux starts and runs.

- Assign a partition a memory area already assigned to the hypervisor image (performed on both partitions):
  Output:
  BAO_ERROR(no handler for abort ec = 0x20)
  Result:
  Incorrect partition doesn't start; Correct partition starts and runs.

The results show that fault isolation has been guaranteed in all tests: when a partition is incorrectly configured, the other partition is not affected and runs properly.

8.2.2 Fault Injection in cache coloring mechanism results

In total 14 tests have been conducted. The results obtained in the cache coloring mechanism fault injection tests are the following; the tests having the same output and result have been grouped:

- pp_alloc_clr(pool, 0, 0, &pages);
  pp_alloc_clr(pool, numpages, 0, &pages);
  Output:
  No error message reported
  Result:
  Both VMs do not start.

- pp_alloc_clr(pool, 0, colors, &pages);
  Output:
  BAO ERROR: failed to alloc page table
  Result:
  Both VMs do not start.

- pp_alloc_clr(NULL, numpages, colors, &pages);
  pp_alloc_clr(pool, numpages, colors, NULL);
  pool->bitmap=0;
  Output:
  BAO ERROR: cpu internal hypervisor abort - PANIC
  Result:
  Both VMs do not start.

- pool->base=0;
  Output:
  Not interpretable output
  Result:
  Both VMs do not start.

- pool->size=0;
  pool->free=0;
  Output:
  BAO ERROR: failed to allocate shared memory
  Result:
  Both VMs do not start.

- pool->last=0;
  pool->lock=0;
  ppages.base=-1;
  ppages.numpages=-1;
  ppages.colors=0;
  Output:
  No error message reported
  Result:
  Both VMs start and run.

Note that in the last tests in the above list have not caused any error; that is because the parameters values do not represent invalid inputs or they are overwritten or reset inside the function code.

8.2.3 Fault isolation between VMs results

The results obtained in the fault isolation between VMs tests are the following:

- Causing a crash in Linux guest by running echo c > /proc/sysrq-trigger
  Output:
  No error message reported
  Result:
  Linux crashes, FreeRTOS runs properly.

- Try to access from FreeRTOS invalid memory location
  Output:
  No error message reported
  Result:
  FreeRTOS crashes, Linux runs properly.

- Saturate FreeRTOS heap area
  Output:
  No error message reported
  Result:
  FreeRTOS crashes, Linux runs properly.

The results show that fault isolation has been guaranteed in all tests: when a partition crashes, the other partition is not affected and runs properly. All tests have been repeated during inter-VMs communication via the shared memory and have reported the same results, so the inter-VMs communication mechanism has not caused any propagation of the fault.

9 Conclusion

In this work we have tested and evaluated Bao's temporal, spatial and fault isolation. The results show that Bao has strong overall isolation properties, provided by its static partitioning architecture and specific mechanisms such as cache coloring. However, despite the strong focus on isolation, VMs still share micro-architectural state, such as interconnects and memory controller, thus not guaranteeing full temporal isolation. Nevertheless, Bao is a promising recent project and is constantly being updated, so it has a lot of room for improvement.

References

[1] Bao's github official repository. https://github.com/bao-project/bao-hypervisor.

[2] Arm. Learn the architecture: Aarch64 virtualization. https://developer.arm.com/documentation/102142/0100, 2022.

[3] M. Cinque, D. Cotroneo, L. De Simone, and S. Rosiello. Virtualizing mixed-criticality systems: a survey on industrial trends and issues, 2021.

[4] J. Corbet. Bao: a lightweight static partitioning hypervisor. https://lwn.net/Articles/820830, 2020.

[5] FreeRTOS. Documentation: configuration. https://www.freertos.org/a00110.html.

[6] N.P. Kropp, P.J. Koopman, and D.P. Siewiorek. Automated robustness testing of off-the-shelf software components. 1998.

[7] J. Martins and S. Pinto. Bao: a modern lightweight embedded hypervisor, 2020.

[8] J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto. Bao: a lightweight static partitioning hypervisor for modern multi-core embedded systems, 2019.

[9] Samuel Pereira, Joao Sousa, Sandro Pinto, Jose Martins, and David Cerdeira. Bao-enclave: Virtualization-based enclaves for arm, 2022.

[10] Ubuntu Wiki. stress-ng manual. https://manpages.ubuntu.com/manpages/bionic/man1/stress-ng.1.html.