| | |
|---|---|
| Project title: | High-Performance Real-Time Architectures for Low-Power Embedded Systems |
| Acronym: | HERCULES |
| Project ID: | 688860 |
| Call identifier: | H2020 – ICT 04-2015 – Customized and low power computing |
| Project coordinator: | Prof. Marko Bertogna, University of Modena and Reggio Emilia |


# D4.5: Multi-OS Integration and Virtualization

| | |
|---|---|
| Document title: | Multi-OS Integration and Virtualization |
| Version: | 1.3 |
| Deliverable No.: | D4.5 |
| Lead task beneficiary: | EVI |
| Partners involved: | EVI, UNIMORE, CTU |
| Author: | C. Scordino, L. Cuomo, M. Solieri, M. Sojka |
| Status: | Final |
| Date: | 2018-12-30 |
| Nature: | S |
| Dissemination level: | PU – Public |

## Purpose & Scope

Deliverable D4.5 is a package containing the software for the concurrent execution of multiple operating systems (i.e., the Real-Time Linux developed in Task 4.1 and the Lightweight Real-Time Operating System (RTOS) developed in Task 4.2) on the same multi-core System-on-Chip (SoC). In addition, the software implements proper mechanisms (namely, cache colouring, memguarding and co-scheduling) to limit the interference on shared hardware resources between the different Operating Systems (OSs).

This document aims at illustrating the code developed, providing information about how to configure and use the provided software. The developed source code is available in the package provided along with this document. A subset of the developed code has been also released as open-source software [JailEvi, JailTX1, JailTX2]. The remaining part will we be similarly published after the project conclusion.

## Revision History

| Version | Date | Author/Reviewer | Comments |
|---|---|---|---|
| 0.1 | 2018-06-21 | EVI | First version |
| 0.2 | 2018-06-26 | UNIMORE | LaTeX conversion |
| 0.3 | 2018-07-27 | UNIMORE | Revised structure, added description and guide for cache coloring support, fixed acronyms |
| 1.0 | 2018-07-29 | CTU | Added MemGuard description |
| 1.1 | 2018-10-30 | UNIMORE | Extended PREM guide, general revision |
| 1.2 | 2018-10-30 | UNIMORE | Included reviewers comments |
| 1.3 | 2018-12-30 | UNIMORE | Final revision |

HERCULES project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement: 688860.

# Contents

# Acronyms

| | |
|---|---|
| **API** | Application Program Interface . . . 6 |
| **ARXML** | AUTOSAR XML . . . 7 |
| **AUTOSAR** | AUTomotive Open System ARchitecture . . . 3 |
| **ECU** | Electronic Control Unit . . . 3 |
| **CPU** | Central Processing Unit . . . 4 |
| **DRAM** | Dynamic Random Access Memory . . . 14 |
| **GA** | Grant Agreement . . . 2 |
| **GPOS** | General-Purpose Operating System . . . 3 |
| **GNU GPL** | GNU General Public License . . . 4 |
| **I/O** | Input/Output . . . 5 |
| **LLC** | Last-Level Cache . . . 7 |
| **MPSoC** | MultiProcessor System-on-Chip . . . 10 |
| **PREM** | PReditable Execution Model . . . 14 |
| **OS** | Operating System . . . i |
| **OSEK/VDX** | Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen / Vehicle Distributed eXecutive . . . 3 |
| **RAM** | Random-Access Memory . . . 4 |
| **RTE** | RunTime Environment . . . 7 |
| **RTOS** | Real-Time Operating System . . . i |
| **SDRAM** | Synchronous Dynamic Random-Access memory . . . 5 |
| **SoC** | System-on-Chip . . . i |
| **US+** | Xilinx Zinq UltraScale+ . . . 7 |

# 1 Introduction

## 1.1 Objectives

As illustrated in the Grant Agreement (GA),

> [Task 4.5 of HERCULES] aims at investigating, designing and developing efficient mechanisms for the concurrent execution of multiple OS on the same SoC. In particular, it focuses on supporting the concurrent execution of the Linux and ERIKA Enterprise OS on the reference platform selected in WP2. Besides the mere concurrent execution of different OSs on the same platform, the task also investigates and designs mechanisms for an efficient synchronization and communication of data between the OSs.

As part of this task, several activities have been carried out by the consortium:

- The investigation and selection of a suitable hypervisor for the envisioned scenario.
- The porting of the selected hypervisor onto the reference platforms.
- The design and implementation of the cache colouring mechanism (resulting from Task 5.1).
- The design and implementation of the co-scheduling mechanism (resulting from Task 5.3).
- The design and implementation of the memguarding mechanism (resulting from Task 5.3)

This document will describe all these parallel activities, providing insights about the implementation and information about using the software.

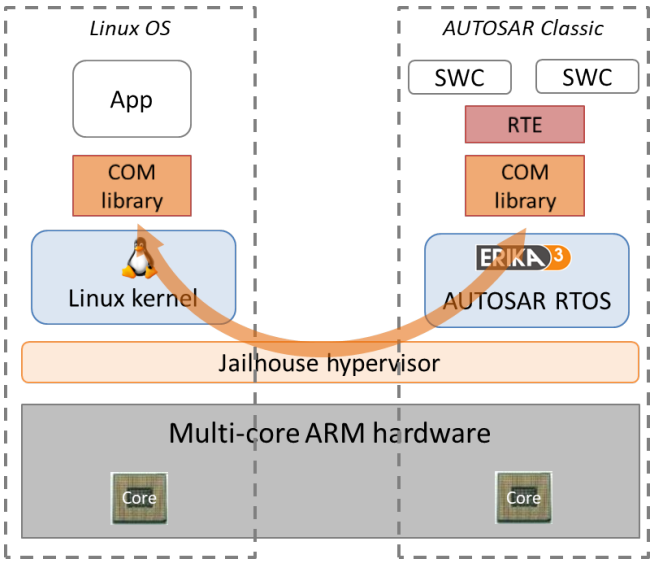


Figure 1: Overall software infrastructure.

# 2 Hypervisor

## 2.1 Introduction

Software running safety-critical tasks must be developed according to certified development processes, and to strict safety standards. This process aims at preventing possible injuries due to misbehaving or faulty software. In the case of automotive Electronic Control Units (ECUs), in particular, the Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen / Vehicle Distributed eXecutive (OSEK/VDX) and the newer AUTomotive Open System ARchitecture (AUTOSAR) standards already established a set of rules and constraints for operating systems design, including memory and timing protection, and stack monitoring. These standards impose the usage of a RTOS implementing a set of well-known scheduling algorithms and techniques. Usually, the size of the RTOS is kept at a bare minimum to reduce the code complexity (hence the number of possible bugs) and the costs of the verification process. In addition to this, there is an ever growing interest from the industry for using the advanced features and potential of modern computing platforms, including networking and graphics subsystems, typically supported by a General-Purpose Operating System (GPOS) like Linux. Finally, there is an increased need to integrate together on the same SoC legacy automotive applications based on such OSEK/VDX or AUTOSAR standards, without radically changing the application source code. A hypervisor solution like the one devised in HERCULES is able to fulfill these three requirements, allowing the coexistence of different operating system, guaranteed real-time behavior and integration of legacy applications. In particular, the HERCULES project has chosen to implement a Multi-OS paradigm in which the Linux GPOS and the ERIKA Enterprise RTOS are executed concurrently under the supervision of a hypervisor for proper isolation of the safety-critical activities. The overall infrastructure is the one illustrated in Figure 1.

## 2.2 Hypervisor Selection

Several interesting open-source technologies were available to be used as the basic brick for the hypervisor to be developed in HERCULES. Among the most interesting and actively maintained projects, we can cite:

- Jailhouse [KSa18]

- KVM [KVM; Wik; Red]
- Xen [Xen]
- Xvisor [Xvia; Xvib]

When selecting the most suitable hypervisor for HERCULES project's scope, we focused on the following requirements:

- Support for the reference hardware; i.e. ARM.
- Real-time capabilities and, especially, the possibility of handling low-latency control).
- Availability of the source code under some kind of open-source license (e.g. BSD license, GNU General Public License (GNU GPL), MIT licence).
- Good project activity by the development and maintaining community.
- Small codebase, which paves the way to a possible certification.

These requirements have allowed to narrow the investigation, and eventually to choose Jailhouse.

## 2.3 Jailhouse

Jailhouse [KSa18; Lin] is a project aiming at creating a small and lightweight hypervisor targeting industrial-grade applications. Despite its youth (born in 2013), the project is sponsored by a big company (i.e. Siemens), it received support into the official Linux kernel, and it is getting visibility and traction in the open-source community. The hypervisor supports both the x86-64 architecture, employed by both Intel and AMD, and the 32-bit and 64-bit variants of ARM architectures, provided that the instruction set has support for hardware-assisted virtualization.

The source code is released under GNU GPL version 2 and hosted on GitHub. Partner EVI worked with the open-source community to change the licensing of some parts of the guest libraries to solve potential issues in commercial usage [Evid].

Jailhouse is a type-1 *partitioning* hypervisor, more concerned with isolation than virtualization. Like Xen, Jailhouse requires Linux because its management interface has been based on the Linux infrastructure. This design choice has allowed to keep the size of Jailhouse's source code as small as possible. Like KVM, it is loaded from a normal Linux system. Once the hypervisor is started, however, it takes full control of the hardware and splits the hardware resources into isolated compartments (called *cells*) that are fully dedicated to guest software programs (called *inmates*). One of these cells runs the Linux OS, and is known as the *root* cell. The role of this cell role is somewhat similar to that of dom0 in Xen. However, the root cell doesn't assert full control over hardware resources as dom0 does; instead, when a new cell is created, the root cell cedes to that new cell control over some of its Central Processing Unit (CPU), device, and memory resources. This process is called *shrinking*. Figure 2 shows how this partitioning works [Lin].

Jailhouse is able to run bare-metal applications or (adapted) operating systems, but one operating system must necessarily be Linux. It cannot run OSs (like Windows or FreeBSD) unmodified. Even the Linux OS, for running on additional non-root cells, needs to be properly configured. Due to its static design:

- Guests cannot share a core because there is no scheduling.
- It does not support over-commitment of resources (like CPUs, Random-Access Memory (RAM) or devices).

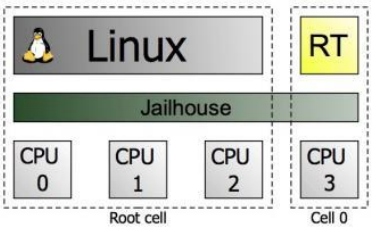


Figure 2: Partitioning in the Jailhouse hypervisor.

- It does not emulate hardware resources not available and the available hardware resources are statically assigned to a single guest OS (so that there is no interference between the OSs when the interrupt occurs).

Thanks to the full partitioned approach, the only source for interference between OSs running on different cores is caused by access to shared hardware resources (i.e. caches, Synchronous Dynamic Random-Access memory (SDRAM), Input/Output (I/O) buses). Jailhouse does not currently provide any mean for lowering such interference. For this reason, the consortium of HERCULES has designed and developed several mechanisms to limit or control this interference, as illustrated in the next sections.

For what concerns fault-tolerance, Jailhouse allows restarting non-root cells that entered a deadlock state. Even the case of deadlock int the root cell (e.g. a Linux kernel crash) does not affect other cells' execution (however, it makes impossible to use the Jailhouse's management interface, because based on the root infrastructure). Partner EVI worked with the Jailhouse development community for further increase the fault tolerance by addressing the case of guests that do not respond anymore [Evie].

Jailhouse also contains features targeting safety-critical applications, such as locking of the configuration interface for some cells. This allows, for example, preventing Linux from unintentionally shutting down a safety-critical cell.

From a technical viewpoint, Jailhouse is loaded as a firmware image (the same way Wi-Fi adapters load their firmware blobs) and resides in a dedicated memory region that must be reserved at Linux boot time. Jailhouse's kernel module (`jailhouse.ko`, also called "driver") loads the firmware and creates `/dev/jailhouse` device, which the Jailhouse user-space tool uses, but it does not contain any hypervisor logic.

# 3 Contributions

## 3.1 Porting on the Reference Platforms

As resulting activity from the HERCULES project, partner EVI has ported the Jailhouse hypervisor onto the Nvidia Jetson TX1 and the TX2 platforms. Such support has been already integrated onto the official Jailhouse hypervisor [Evib; Evic]. In addition, partner EVI has maintained the official version of Jailhouse working on the Nvidia Linux distribution [Evia]. Besides supporting the vendor Linux kernel provided by Nvidia, such version of Jailhouse also integrates the IVSHMEM mechanism for communication between guests on the ARM platform. Such support is not yet available on the official Jailhouse version. For sake of completeness, the source code of this tree of Jailhouse is included in the `linux-jailhouse-jetson` directory in the package provided together with this document.

Concerning the other HERCULES reference platform (i.e. Xilinx Ultrascale+), no porting activity has been needed, as the platform was already officially supported by Jailhouse.

As previously mentioned, partner EVI has also collaborated with the Jailhouse community for enhancing the hypervisor, e.g. by increasing fault tolerance [Evie], or modifying the licensing of the inmate library [Evid]. This is a list of the most relevant contributions made by partner Evidence onto the official Jailhouse tree:

| *Commit hash* | *Message log* |
|---|---|
| `1b7a63610ea8078d940410eb4c1f6c0764f2c3bb` | Jetson TX2: fix root cell config for GPU acceleration |
| `108d84d82be82bfbc23af3545515440f934599e9` | Jetson TX2: add demo cell config |
| `48c4909226cb1177083e48a665c7896160ece0cf` | Jetson TX2: add inmate support |
| `765365f4c4868e0e9ad30d7ca6e2e094b0c405ce` | Jetson TX2: root cell config |
| `3f1eba5aa77b42650eae2613f0624546acb6797d` | Stop waiting for message reply after a certain amount of time |
| `607251b44397666a3cbbf859d784dccf20aba016` | Provide the inmate library under BSD-2 license too. |
| `87c82cfb06e6a207048075de4401ce1584ea6747` | Update config files for TX1 |
| `eb8ad9089979b980e5c149319eac42e704af5948` | Split jetson-demo config between TK1 and TX1 |
| `0b165bb0c87780e9c05bd8f8754607c5c4fbe51c` | configs: add config files for Jetson TX1 |
| `1be5fdef659bd83c221228b01cffa9b427642dda` | inmates: add support for Tegra TX1 |
| `fbeafd5b3251cec198544865db6d68667548413b` | Additional headers for old kernel versions |

As a result of HERCULES, partner EVI also integrated and released a VirtualBox [Ora] virtual machine containing the ERIKA Enterprise RTOS and the RT-Druid development environment already installed for the TX1 and TX2 platforms. Such virtual machine is publicly available for download [Evif].

## 3.2 Inter-Guest Communication Mechanism

Jailhouse provides a quite complicated mechanism for communication between different guests. Such mechanism closely follows the ”ivshmem” device model from the Qemu project, which is based on an abstraction of PCI devices. The guests therefore need a preliminary phase for discovering the PCI addresses where the communication mechanism has been mapped. This mechanism is not practical, and thus the community is looking to alternatives like virtio [Kis]. Moreover, this mechanism is currently available only for x86-64 platforms, as the ARM support has not yet been upstreamed.

Within the EUROSTARS RETINA project [RET] partner EVI has worked for designing an AUTOSAR-compliant communication library working on top of Jailhouse. Rather than re-inventing the wheel, partner EVI has chosen to re-use such library, proving its effectiveness on the HERCULES use-cases. The library (also shown in Figure 1) basically creates some periodic threads that take care of sending/receiving messages through either blocking or non-blocking calls. The Application Program Interface (API) is very simple and it mainly consists of the following functions:

```
Com_StatusType Com_GetStatus(void);

uint8 Com_SendSignal(Com_SignalIdType SignalId, const void *SignalDataPtr);
uint8 Com_ReceiveSignal(Com_SignalIdType SignalId, void* SignalDataPtr);

uint8 Com_SendDynSignal(Com_SignalIdType SignalId, const void *SignalDataPtr,
       uint16 Length);
uint8 Com_ReceiveDynSignal(Com_SignalIdType SignalId, void* SignalDataPtr,
       uint16* Length);

uint8 Com_SendSignalBlock(Com_SignalIdType SignalId, const void* SignalDataPtr);
uint8 Com_ReceiveSignalBlock(Com_SignalIdType SignalId, void* SignalDataPtr);

uint8 Com_ReceiveDynSignalBlock(Com_SignalIdType SignalId, void* SignalDataPtr,
       uint16* Length);
uint8 Com_SendDynSignalBlock(Com_SignalIdType SignalId, const void* SignalDataPtr,
       uint16 Length);
```

The configuration of the channels and messages is automatically handled by the RunTime Environment (RTE) generator which, starting from the AUTOSAR XML (ARXML) files, generates the `.c` and `.h` files to be linked against the library and the ERIKA RTOS.

Due to confidentiality concerns, the source code of this library is not provided as part of the deliverable, but it is maintained on EVI's private servers and available to the European Commission and to the reviewers upon request.

## 3.3 Cache Coloring

Cache coloring is a software-level technique that, exploiting the placement function of last level caches—which assigns a unique set to any given memory address—leverages on physical memory partitioning to define non-overlapping cache partitions called *colors*. If the system memory is allocated to inmates so that different inmates are allocated different colors, mutual interference in shared Last-Level Cache (LLC) is avoided by construction. The solution implements cache partitioning without the need of any hardware assistance.

UNIMORE implemented cache coloring support in the Jailhouse hypervisor, which allows cells to be assigned memory regions with a selection of cache colors. This contribution is integrated under `jailhouse` in the `D4.5 Software` folder, together with the other Jailhouse extensions.

A more detailed introduction to cache coloring, a description of UNIMORE implementation, and an extensive evaluation on NVIDIA Tegra X1 performed by UNIMORE is reported in [D5.1], which is devoted to software techniques and scheduling algorithms for HERCULES platform CPUs. Experimental results presented there successfully confirm the efficacy of cache coloring in eliminating contention on last level cache, thus improving the predictability of the system and, consequently, the tightness of the analysis of this latter. Collaborating with a third-party contractor, UNIMORE also successfully tested the cache coloring implementation on Xilinx Zinq UltraScale+ (US+), and obtained brilliant benchmark results of attainable latency and jitter.

## 3.4 MemGuard

MemGuard is a mechanism for managing memory bandwidth available to CPU cores in a multi-core processor. It was first introduced by Yun at al. [Yun+15] as a Linux kernel module. The memory is bandwidth consumption is monitored by a Performance Monitoring Unit (PMU) available on most modern processor. When memory bandwidth consumption by a single CPU core is higher than a predefined threshold, the core it throttled.

CTU implemented a similar mechanism in the Jailhouse hypervisor. It uses ARM's PMU and a hypervisor timer to implement memory bandwidth monitoring and throttling. The mechanism is controlled by a hypercall interface described in Section 4.3.

## 3.5 Memory Co-Scheduling

The PRedictable Execution Model provides timing guarantees by imposing tasks to treat the shared memory as a resource to be used exclusively. This ensures that any additional latency with respect to optimal performance that can be experienced by a task is introduced only by waiting for memory exclusive access, and not by the contention for memory bandwidth.

UNIMORE and CTU implemented memory co-scheduling support in the Jailhouse hypervisor version 0.9.1. This contribution is integrated under `jailhouse` in the `D4.5 Software` folder, together with the other Jailhouse extensions. A detailed description and evaluation is reported in deliverable [D5.3], dedicated to software techniques and scheduling algorithms for the whole HERCULES platform, i.e. considering both CPU and accelerators.

# 4 User's Guide

This section provides user guidance for installation, configuration and usage of software delivered in D4.5.

## 4.1 Hypervisor Installation

A guide for Jailhouse installation is available together with the software distribution, in the `README.md` file.

## 4.2 Cache Coloring Configuration

In this section we describe Jailhouse configurations that are specific to cache coloring.

### 4.2.1 Hypervisor configuration

Jailhouse supports various static compile-time configuration parameters, such as platform specific settings and debugging options. Those settings can optionally be defined in `include/jailhouse/config.h`. Every configuration option should be defined to value `1` or not be in the file at all—defining any other value can cause unexpected behaviour.

By default, the following settings are enabled:

`CONFIG_TRACE_ERROR` Print error sources with filename and line number to debug console

`CONFIG_CRASH_CELL_ON_PANIC` Set instruction pointer to 0 if cell CPU has caused an access violation. Linux inmates will dump a stack trace in this case.

### 4.2.2 Preliminaries

Cache hierarchy of a modern multi-core CPU typically has first levels dedicated to each core (hence using multiple cache units), while the last level is shared among all of them. Such configuration implies that memory operations on one core (e.g. running one Jailhouse inmate) are able to generate interference on another core (e.g. hosting another inmate). Cache coloring allows eliminating this mutual interference, and thus guaranteeing higher and more predictable performances for memory accesses.

The key concept underlying cache coloring is a fragmentation of the memory space into a set of sub-spaces called *colors* that are mapped to disjoint cache regions. Look at the ASCII art in Figure 3. Technically, the whole memory space is first divided into a number of subsequent regions. Then each region is in turn divided into a number of subsequent sub-colors. The generic $i$-th color is then obtained by all the $i$-th sub-colors in each region.

There are two pragmatic lessons to be learnt.

1. If one wants to avoid cache interference between two inmates, different colors needs to be used for their memory.

2. Color assignment must privilege contiguity in the partitioning. E.g., assigning colors $(0, 1)$ to inmate $I$ and $(2, 3)$ to inmate $J$ is better than assigning colors $(0, 2)$ to $I$ and $(1, 3)$ to $J$.

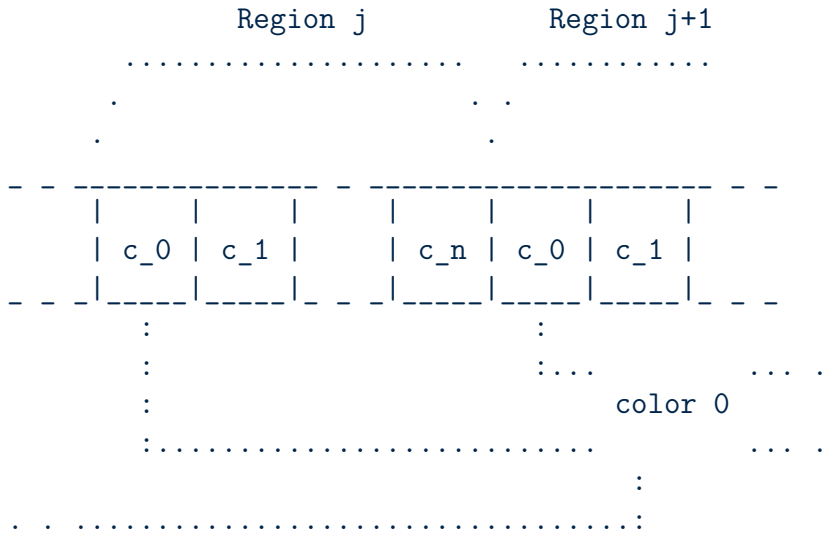


Figure 3: Cache colors

**Coloring configuration** Jailhouse configurations needed to create a cache-colored Jailhouse setup are split into two conceptually separate levels: one for developers/vendors and one for users/clients.

**Definition** this portion is placed in the root cell configuration, and defines the colouring scheme, depending on user requirements and according to the specific cache system.

**Assignment** this part is placed in non-root cell configuration, and define how the set of available set colors is split among inmates, depending on user requirements.

### 4.2.3 Color Definition Configuration

This configuration greatly depends on the specific CPU cache, and it is therefore recommended to be setup by an expert technician.

The root cell configuration for coloring support is defined by the *coloring description*, that has to be inserted within the structure `platform_info`, and that contains the three parameters, whose values are sizes expressed in bytes.

**`cachesize`** is the size of the whole last-level cache.

**`fragment_unit_size`** determines the size in bytes of a sub-color, and should be a multiple of the size of pages that is set up by the root cell Linux (`PAGE_SIZE`).

**`fragment_unit_offset`** defines the size in bytes of a region, and must be obtaining dividing ‘cachesize‘ by the number of ways of the cache.

As an example, we consider the configuration of the US+ MultiProcessor System-on-Chip (MPSoC), which has a A53 CPU equipped with a 1 MiB 16-ways associative L2 cache [Xil]. This gives us immediately first and third parameters: 1 MiB and 64 KiB = 1 MiB/16, respectively. Assuming that the page size is setup by Linux at the default value of 4 KiB, we may opt for setting the second parameter at the smallest possible value. See lines 13-17 of Listing 1.

Listing 1: Example of `platform_info` configuration

```
1  platform_info = {
2    .pci_mmconfig_base = 0xfc000000,
3    .pci_mmconfig_end_bus = 0,
4    .pci_is_virtual = 1,
5    .arm = {
6      .gic_version = 2,
7      .gicd_base = 0xf9010000,
8      .gicc_base = 0xf902f000,
9      .gich_base = 0xf9040000,
10     .gicv_base = 0xf906f000,
11     .maintenance_irq = 25,
12   },
13   .coloring_desc = {
14     .fragment_unit_size = 0x1000, // 4 KiB
15     .fragment_unit_offset = 0x10000, // 64 KiB
16     .cachesize = 0x100000, // 1 MiB
17   },
18 },
```

Observe that the first two configuration parameters determine the number of available colors to be available for inmate cells:

$$\text{colors} = \frac{\texttt{fragment\_unit\_offset}}{\texttt{fragment\_unit\_size}} \tag{1}$$

Back to the example, in the US+ board the previous configuration thus defines 16 = 64 KiB / 16 KiB colors. This is the maximum number of colors configurable on such platform, because we chose the smallest `fragment_unit_size`, which allows the greatest flexibility in the configuration. For the sake of completeness, the whole `platform_info` is reported in Listing 1.

### 4.2.4 Color Assignment Configuration

The inmate configuration is supposed to be make by the final user and for this reason it as been kept as simple as possible. The user does not necessarily need to know how the coloring policy is implemented and what the board configuration is, but he can just choose the color assignment, depending on its specific needs.

In particular, we allow the user to put a given memory region mapping, as usually specified in a inmate cell configuration file, in a new kind of structure called `memory_regions_colored`. This structure is equipped with a `colors` parameter, which a bit mask which specifies which are the colors that should be used within the memory mappings.

If, for instance, we have 16 colors as in the root-cell configuration example about US+, we may want to configure an inmate as in lines 16-29 of Listing 2, where we assign a 128 KiB region, and restrict the reservation to the lower half of colors. This makes 64 KiB of memory available to the inmate and visible from virtual address `0x20000` to `0x30000`. The unused colors in the very same memory area can be assigned to other inmates by using the same snippet and changing only the `colors` parameter. On the other hand, the same color bit cannot be asserted in two different cell, because this would cause memory chunks to be shared between them.

Adding a `memory_regions_colored` involves a process almost identical to that of adding a new Jailhouse memory region.

Listing 2: Example of non-root cell configuration

```
struct {
  ...
  /* Declare the array */
       struct jailhouse_memory_colored memory_regions_colored[1];
  ...
} __attribute__((packed)) config = {

  .cell = {
    ...
    /* Set the memory_regions_colored number */
    .num_memory_regions_colored = ARRAY_SIZE(config.memory_regions_colored),
    ...
  },
  ...
  /* Define the array data*/
 .memory_regions_colored = {
  {
    /* RAM */
    .memory = {
      .phys_start = 0x800620000,
      .virt_start =   0x20000,
      .size       =    0x20000,
      .flags      = JAILHOUSE_MEM_READ |
                    JAILHOUSE_MEM_WRITE |
                    JAILHOUSE_MEM_EXECUTE |
                    JAILHOUSE_MEM_LOADABLE,
    },
    .colors = 0x00ff,
  },
},
```

1. One first needs to declare an array of type `jailhouse_memory_colored` in the first part of the cell configuration file.

2. Then the memory region(s) should be defined, accordingly to the length of the declared array.

An example is in Listing 2.

## 4.3 MemGuard Configuration

MemGuard is enabled by default in the Hercules version of Jailhouse. It can be controlled via two Jailhouse hypercalls. The `gprem` hypercall is always enabled and it allows controlling both the memory co-scheduling and memguard via its parameters. The signature of this hypercall is shown in Listing 3.

MemGuard is controlled with parameters `memory_budget`, `period` and `flags`.

The second hypercall (`memguard`) (Listing **??**) is enabled only if `CONFIG_DEBUG_MEMGUARD` is defined as 1 in *include/jailhouse/config.h*. It has only three parameters (see below) and the difference from the `gpem`

Listing 3: `gprem_set` Hypercall Signature

```
int gprem_set(enum prem_phase phase,
              unsigned long memory_budget,
              unsigned long timeout,
              unsigned long period,
              unsigned long flags);
```

Listing 4: `memguard` Debug Hypercall Signature

```
long memguard(unsigned long budget_time, unsigned long budget_memory,
              unsigned long flags);
```

hypercall is (besides the co-scheduling behavior) in its return value, which can be used for profiling of the PREM phases.

This hypercall setups time and memory budgets for the calling CPU core. The subsequent call then returns statistics since the preceding call. Time budget overrun is reported with MGRET_OVER_TIM_MASK bit, memory budget overrun with MGRET_OVER_MEM_MASK. Returned statistics also include the total time and total number of cache misses since the preceding call, which can be used for application profiling.

The memguard functionality can be influenced by the following flags:

**MGF_PERIODIC** When set, the memguard timer is set to expire periodically every `budget_time`. Overrunning memory budget causes the CPU to block (enter low-power state) until the next expiration of the memguard timer. At every timer expiration, memory budget is replenished with `budget_memory` value.

Note that with this flag unset, memguard only reports budget overrun in statistics. No blocking happens.

**MGF_MASK_INT** When set, memguard disables interrupts that can be disabled and are not needed for proper memguard functionality. This is to ensure (almost) non-preemptive execution of PREM predictable intervals that is required to reduce number of unpredictable cache misses.

The parameters have the following meaning:

**period** Time budget in microseconds. When zero, time monitoring is switched off.

**memory_budget** The number of cache misses allowed. When zero, memory access monitoring is switched off. Must be non-zero when `MGF_PERIODIC` flag is set.

**flags** Flags – see MGF_* constants above.

**Return value** Statistics since the preceding memguard call and/or the error flag. These are encoded in different bits as follows:

- 0–31 – Total number cache misses
- 32–55 – Total time in microseconds
- 61 – Memory budget overrun
- 62 – Time budget overrun
- 63 – Error

See also MGRET_* constants.

## 4.4 Memory Co-Scheduling Configuration

PReditable Execution Model (PREM) support is enabled by default in the Hercules version of Jailhouse. It can be controlled by the `gprem_set` hypercall, whose signature was provided Listing 4.

The strictly relevant arguments are the first and third ones.

`phase` is value in an enumeration of possible phases. Currently supported values are:

**Memory** (numerical value: `1`) it should be used for highly memory bound execution segments, like cache prefetch or flush operations. This hypercall guarantees exclusive access to Dynamic Random Access Memory (DRAM), granted following the statically assigned priority. This call enter in a blocking spinlock at hypervisor level, blocking any other execution on the callee's core until the lock release.

**Computation** (numerical value: `2`) it is meant, on the contrary, for code segments where data is ideally never accessed from/to central memory, but from/to cache only. This can happen only thanks to a previous prefetch phase. If possessed, a callee's core lock is released upon this call.

**Compatible** (numerical value: `0`) it should be used for mixed or unknown code or foreign calls. If possessed, a callee's core lock is released upon this call.

`timeout` is the amount of microsecond to wait before to forcibly recover memex lock from a misbehaving guest which previously obtained it before.

# References

[D5.1] Hercules consortium. *D5.1: Power-Aware Scheduling Algorithm for Host*. Deliverable of the HERCULES project. June 2018.

[D5.3] Hercules consortium. *D5.3: Integrated Schedulability Analysis*. Deliverable of the HERCULES project. June 2018.

[Evia] Evidence Srl. *Jailhouse for default Linux kernel of Nvidia Jetson platforms*. https://github.com/evidence/linux-jailhouse-jetson.

[Evib] Evidence Srl. *Jailhouse support for Nvidia Jetson TX1*. https://github.com/siemens/jailhouse/blob/master/configs/arm64/jetson-tx1.c.

[Evic] Evidence Srl. *Jailhouse support for Nvidia Jetson TX2*. https://github.com/siemens/jailhouse/blob/master/configs/arm64/jetson-tx2.c.

[Evid] Evidence Srl. *Jailhouse, Provide the inmate library under BSD-2 license too*. https://github.com/siemens/jailhouse/commit/607251b44397666a3cbbf859d784dccf20aba016.

[Evie] Evidence Srl. *Jailhouse, Stop waiting for message reply after a certain amount of time*. https://github.com/siemens/jailhouse/commit/3f1eba5aa77b42650eae2613f0624546acb6797d.

[Evif] Evidence Srl. *VirtualBox virtual machine for TX1 and TX2*. http://www.erika-enterprise.com/index.php/download/virtual-machines.html.

[Kis] Jan Kiszka. *Jailhouse 0.9 released.* LWN.net, https://lwn.net/Articles/756440/. Accessed: 2018-06-26.

[KSa18] Jan Kiszka, Valentine Sinitsyn, and Henning Schild and contributors. *Jailhouse hypervisor*. Siemens AG on GitHub, https://github.com/siemens/jailhouse. Accessed: 2018-03-31. 2018.

[KVM] KVM contributors. *Linux Kernel Virtual Machine*. http://www.linux-kvm.org/page/Main_Page.

[Lin] Linux Journal. http://www.linuxjournal.com/content/jailhouse.

[Ora] Oracle. *VirtualBox*. https://www.virtualbox.org/.

[Red] Red Hat. *What is KVM?* https://www.redhat.com/en/topics/virtualization/what-is-KVM.

[RET] RETINA partners. *RETINA EUROSTARS project*. http://www.retinaproject.eu/.

[Wik] Wikipedia contributors. *Kernel-based Virtual Machine*. https://en.wikipedia.org/wiki/Kernel-based_Virtual_Machine.

[Xen] Xen contributors. *Xen project*. https://www.xenproject.org/.

[Xil] Xilinx. *Zynq UltraScale+ MPSoC Data Sheet: Overview*. https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf.

[Xvia] Xvisor contributors. *Xvisor project page*. http://xhypervisor.org.

[Xvib] Xvisor contributors. *Xvisor source code*. https://github.com/xvisor/xvisor.

[Yun+15] Heechul Yun et al. “Memory Bandwidth Management for Efficient Performance Isolation in Multi-core Platforms”. In: *IEEE Transactions on Computers* (2015).