Inter-VM communication mechanism on Jailhouse: IVSHMEM.

Naddei Laura* L.NADDEI@STUDENTI.UNINA.IT
Università degli studi di Napoli Federico II

Spina Michele* MIC.SPINA@STUDENTI.UNINA.COM
Università degli studi di Napoli Federico II

Tutor: Daniele Ottaviano DANIELE.OTTAVIANO@UNINA.IT
Università degli studi di Napoli Federico II

Prof.: Marcello Cinque MACINQUE@UNINA.IT
Università degli studi di Napoli Federico II

1. Introduction

From 10 years apart there's a strong increasing trend in the use of virtualization technology. That's an obvious result due to the major advantages of this choice: flexibility, costs, isolation, and efficiency.

This technology is being adopted also in embedded systems and in mixed-criticality contexts thanks to one of the major benefits: the isolation.

This combination brings with it a lot of still open problems, in particular in real-time context.

The approaches to real time virtualization are multiple and different:

- Microkernel and separation kernel
- Adaptation of general-purpose hypervisor
- Based on advanced HW architectural features

In our case of study, we will focus on *Jailhouse*, which is an hypervisor of the first category.

So, in this paper, we will explore the problem of RT virtualization, the solution proposed by Jailhouse hypervisor, the communication mechanism provided by *IVSHMEM* in the Jailhouse's adaptation (*IVSHMEMv2*), and a complete example on the *Zynq UltraScale+ ZCU104* architecture using this protocol.

* These authors contributed equally

2. Case of study

The goal of our experiment is analysing the Inter-Virtual Shared Memory (*IVSHMEMv2*), an inter communication mechanism between virtual machines adopted by Jailhouse hypervisor and realize a use case in which two partitions exchange information using this protocol on an ARM architecture (*Zynq UltraScale+ MPSoC ZCU104*).

3. Real time virtualization

As already said, virtualization technology is gaining space in the embedded domains and in real time context thanks to the interest in mixed-criticality system (*MCS*). MCSs are systems that present different levels of criticality on the same hardware platform. This definition is enough to understand the power of virtualization in this context: isolate the environment of different criticality levels is the solution to the main problem of MCS, the interferences between different parts of the system. That also facilitates the development of software by decoupling from the real hardware reducing development cost. At last, but not least, encapsulation of high critical subsystem permits to certify this part independently of the other subsystems.

For those reasons, this approach is being largely used in IoT and automotive industry, where there are different situations characterized by different levels of criticality.

Of course, a new technology always brings with it new problems! First of all, when we talk about

real-time we must mention latency issue mainly caused by non-preemptive sections in the *VMM*. Another problem affects the scheduling: how to schedule vCPU assigned to virtual machines?

Obviously, also the memory is affected by problems derived from sharing the same hardware platform by different VMs. I/O as well as memory.

So, these four principal problems are differently dealt by three main hypervisor categories:

- *Microkernel and separation kernel* face the problems with a static approach. Jailhouse uses this type of mechanism.
- *Adaptation of general-purpose hypervisor*, like *Xen* or *KVM*, to RT context.
- *HV based on specific hardware features* such as *ARM Trust Zone*, that provides two state of execution, *secure* and *non-secure*.

4. Jailhouse

Jailhouse is a static partitioning hypervisor based on *Linux* that allows partitioning of a single physical platform into multiple isolated environments called "*cells*". Each cell with its own set of resources (CPU, memory, I/O devices), runs its own operating system (or bare metal) and applications and is isolated from the other cells and from the host system. So, Jailhouse is well known for its strong reservation, there's no sharing between VMs and that completely eliminates the problem of vCPU scheduling (thanks to static partitioning of physical resources). It's small, we're talking about less of 10k lines of code, and that is an enormous plus for certification. Of course, a strong isolation brings with it real time guarantees but also inefficiency in resource utilization!

In a MCS context, static partitioning is an interesting and strong solution to isolate high criticality level parts from the less ones.

A particularity but also a limitation of Jailhouse is that it can't be implemented without Linux OS.

In fact, once Linux has been loaded on the system, Jailhouse hypervisor is initiated and loaded by command line and takes the control of the system making Linux the "*root cell*", a cell that owns all the resources and is configured via configuration file.

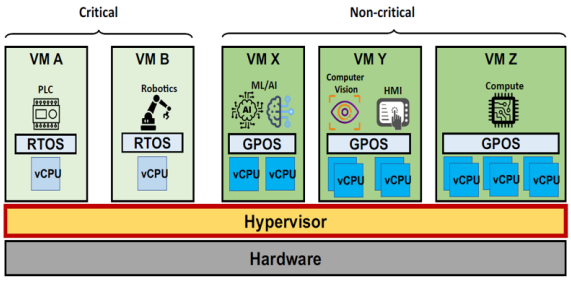

Figure 1: MCS typical architecture with statical partitioning.

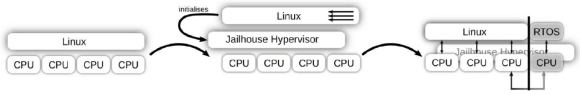

Figure 2: Jailhouse bootstrap process.

At this point, the non-root cells can be created by subtracting resources from root cell in base of configuration files.

Jailhouse is loaded as a firmware image that resides in a dedicated memory region. In particular, *jailhouse.ko* loads the firmware and creates */dev/jailhouse* that is used by *Jailhouse user-space tool* for managing cells' life cycle (so without any HV logics).

As already said, cells represent VMs and software running on these are called *inmates*.

There're also two execution modes:

- *Open model*: Linux, as a root cell, totally controls the other cells that are not involved in decisions. That implies that the root cell is trusted!
- *Safety model*: Linux, as a root cell, controls other cells but there's the possibility to configure cells to get involved in decisions.

In our case (as most cases) we use the Open model.

Jailhouse is an hypervisor that rarely comes to play, only when it's strictly necessary. In fact, the cases of Jailhouse intervention are:

- Memory access violation

- Interrupt re-injection.
- Intercept hardware resources that are not virtualized.
- Emulation of privileged instructions.

4.1. Configuration files

To use Jailhouse on a system is mandatory to pre-allocate contiguous RAM blocks to hypervisor and cells. This operation can be done during the boot-up. In particular, for ARM family, the memory seen by the Kernel is reduced via the command mem or modifying the device tree.

Once memory for hypervisor and cells has been allocated, we must statically configure each cell by assigning values to various C structures:

- *mem_regions*: specifies memory areas used by the cell.
- *irqchips*: specifies which IRQ can be assigned to the cell.
- *pci_devices*: specifies PCI devices assigned

Of course, also the number of CPU must be assigned through a *__u64* data type.

5. IVSHMEM

Jailhouse focuses on giving isolation to the applications, but this can be seen as a limitation as it may restrict communication between applications running in different partitions. The *IVSHMEM* (Inter-VM Shared Memory) is a mechanism for communication between multiple virtual machines running on the same physical host. This mechanism was firstly implemented for *QEMU* and was later released in its second improved version providing the adaptation in modern virtualization environments such as the Jailhouse one.

IVSHMEM's goal is to realize a memory area shared by the cells that is opaque to the hypervisor to give guests the complete control on it.

The main features provided by the IVSHMEM mechanism are:

- Multi-purpose shared memory region
- Event signaling via interrupts.
- Support for LCM via state value exchange and interrupt.
- Free choice of protocol to be used on top.
- Register can be implemented either memory-mapped or via I/O, depending on platform support and lower VM-exit costs.

5.1. Programming Model

To set up a communication channel between two or more cells we have to add memory regions to each cell, in particular:

- Read-only region
- Common read/writable region for all peers
- Output regions

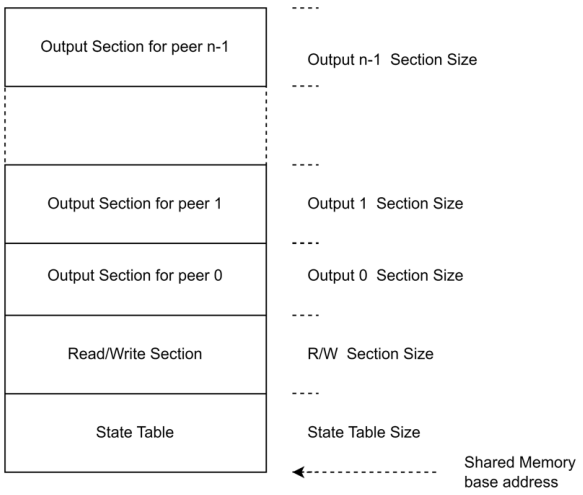

Figure 3: IVSHMEM regions.

The read-only region is a piece of memory starting at the shared memory base address and typically one page large. It holds the State Table that contains 32-bit state value for each of the peer and is updated on each state change of a peer. So, whenever a user writes a value to the Local State register, an interrupt is triggered, and the value is copied into the corresponding entry of the State Table.

The common readable/writable region for all peers is a contiguous block of memory that is shared

between the VMs and used for communication between cells. It is an optional region; this means that it can be zero-sized.

The last regions are the output regions, one per peer, these must have the same size. Each of these sections is readable/writable for the corresponding peer and read-only for the others.

After creating the memory regions, also a PCI device needs to be added to each connected cell to implement signaling between cells. For each PCI device two important registers areas are the *register region* and the *header register*.

The *register region* is a small area of memory that may be implemented as MMIO, or I/O mapped. Its purpose is containing registers dedicated to control and monitor the shared memory region. Whenever an application needs to interact with the device driver to perform a task, it will read or write from/to the registers. In particular, the registers provided are:

- *ID*
- *Maximum Peers*
- *Interrupt Control*: enables reception of interrupt by other peers.
- *Doorbell*: triggers an interrupt vector in target device.
- *State*

In particular, the *Doorbell register* is split in two parts, first 16 bits are called "Vector number" and the other 16 are called "Target ID".

So, there's a correspondence between pairs of bits, like 0-15, 1-16 etc. When a bit in the vector number is 1, it enables the interrupt for the corresponding device. When the PCI device changes its *Local State Register*, the alteration is propagated to the *State Table* and devices specified in the *Doorbell Register* receive an interruption.

The *header register* is used to store information about the device, such as its vendor ID, device ID, class code, interrupt line, base address registers, etc. These parameters are used by the hypervisor to configure the IVSHMEM device and to allow the cells to access the *shared memory region* (via BAR2) or the *register region* (via BAR0).

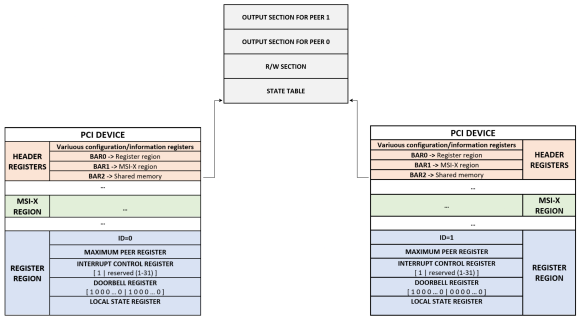

Figure 4: IVSHMEM idea.

As we can see, an important aspect of the IVSHMEM protocol is that it is based on pointers and offsets. This approach has several advantages. Firstly, it allows for more flexible and dynamic memory allocation, as the VMs can allocate and deallocate memory as needed, without having to worry about absolute memory addresses. Secondly, it makes it easier to handle different memory layouts in different VMs, as the pointers and offsets can be adjusted to match the local memory layout. Finally, it provides a more secure mechanism for communication between VMs, as the VMs cannot access each other's memory without explicit permission because of different address spaces for each cell.

5.2. Shared Memory configuration

To define a shared memory region, we must configure it in the cell configuration file in according to the protocol IVSHMEMv2. So, in the *.mem_region* section of the configuration file must be defined:

- Physical start address
- Virtual start address
- Size
- Flags

for each section of the shared memory (state table, R/W section, output sections). A non-root cell that shares memory with the root cell needs the memory flag *JAILHOUSE_MEM_ROOTSHARED* on the region. Jailhouse provides a standard macro called *JAILHOUSE_SHMEM_NET_REGIONS (...)* that reserves 1MB of memory at a specified base address and assigns access according to the specified ID.

```
.mem_regions = {
        /* IVSHMEM shared memory regions (demo) */
        // state region
        {
                .phys_start = 0x7fb00000,
                .virt_start = 0x7fb00000,
                .size = 0x1000,
                .flags = JAILHOUSE_MEM_READ,
        },
        // R/W region
        {
                .phys_start = 0x7fb00000 + 0x1000,
                .virt_start = 0x7fb00000 + 0x1000,
                .size = 0x4000,
                .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE,
        },
        //OUTPUT ROOT
        {
                .phys_start = 0x7fb00000 + 0x5000,
                .virt_start = 0x7fb00000 + 0x5000,
                .size = 0x4000,
                .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE,
         },
        //OUTPUT INMATE
        {
                .phys_start = 0x7fb00000 + 0x9000,
                .virt_start = 0x7fb00000 + 0x9000,
                .size = 0x4000,
                .flags = JAILHOUSE_MEM_READ,

         },
```

Figure 5: Shared memory configuration.

After memory region has been created, we must define the PCI device for the signaling mechanism. For this purpose, field must be set as follows:

- *.type* must be *JAILHOUSE_PCI_TYPE_IVSHMEM*
- *.bdf* (*bus-device-function*) must be the same for all connected peers
- *.domain* can be different between peers
- *.bar_mask* must be *JAILHOUSE_IVSHMEM_BAR_MASK_MSIX* or *JAILHOUSE_IVSHMEM_BAR_MASK_INTX*, depending on the interruption technology.
- *.shmem_regions_start* indicates the first shared memory region used by device
- *.shmem_dev_id* is the ID of the peer
- *.shmem_peers* indicates the maximum number of connected peers
- *.shmem_protocol* defines the shared memory protocol used over the link (self-made or standard such as *VETH*)

```
.pci_devices = {
        /* 0001:00:01.0 */ {
                .type = JAILHOUSE_PCI_TYPE_IVSHMEM,
                .domain = 1,
                .bdf = 1 << 3,
                .bar_mask = JAILHOUSE_IVSHMEM_BAR_MASK_INTX,
                .shmem_regions_start = 0,
                .shmem_dev_id = 0,
                .shmem_peers = 2,
                .shmem_protocol = JAILHOUSE_SHMEM_PROTO_UNDEFINED,
        },
},
```

Figure 6: PCI device configuration.

6. Zynq Ultrascale+ ZCU104 configuration

Our goal is to analyse the IVSHMEM protocol also in a practical way and to do that we must execute Jailhouse on a supported architecture such as a *x86*, *ARM* and *ARM64*. In our case, the *Zynq Ultrascale+ ZCU104* has been an idoneous choice (ARM64).

It's an AMP system but, despite the presence of RPU, GPU and FPGA, we'll use only the APUs.

An important limitation of Jailhouse is that is mandatory to start from a Linux distribution. To do that, we have to build a Linux image which has Jailhouse and is appropriate to the *Zynq ZCU104*. We've used *Petalinux (2019.1)* on an *Ubuntu 18.04* VM and, once kernel image has been successfully created and Jailhouse successfully cross-compiled, an SD card has been made bootable for the Zynq board. [for technical detail look at the attachment]

In order to make Jailhouse successfully cross-compiled for the Zynq ZCU104, two important steps must me made:

1. generate the *.dts* from the *.dtb* provided by Xilix and include it in *configs/arm64/dts*
2. create *config.h* in */include/Jailhouse* and define in it two flags.

7. Use case

IVSHMEM is a useful mechanism that can be used for different purposes. For example, we used it to implement a *keepalive* protocol.

The first step is configuring the Linux root cell and a bare metal non-root cell. In particular, to

use the IVSHMEM is mandatory to set the shared memory regions and the PCIs as already said in the 5.2 section.

Despite the several problems (lately described) faced during the configuration and the implementation of the use case, it is based on a simple mechanism:

1. the non-root cell and the Linux controller initialize the pointers to the readable/writable region.
2. Linux waits for the *keepalive* (in polling);
3. the non-root cell periodically writes in the R/W region and flushes the cache.
4. Linux reads the R/W region and lowers the keepalive signal.
5. Linux returns to polling.

This workflow, of course, represent the standard behaviour assuming a correct working of the non-root cell.

Instead, if the non-root cell fails and doesn't send the *keepalive* signal, the root cell will restart the non-root cell.

7.1. Problems and open issues

First of all, the initial idea was to implement the task of the non-root cell with FreeRTOS and not bare metal. Despite of several attempts to integrate FreeRTOS libraries with Jailhouse using Vitis, we had to give up the idea (this issue could be fixed by dedicating more time to it).

Awareness of that, we've tried the *ivshmem-demo* provided by Jailhouse that initiates the pci device by reading the cell configuration and trigger some interrupts. Unfortunately, the interrupts are not triggered, as shown by analysing the */proc/** data.

This is a well-known issue for ARM64, as can be read in the official mailing list of Jailhouse.

Another point is the driver of vPCI for Linux. This topic has not been explored in a practical way because even the bare metal interrupts don't work. By the way, vPCI devices are not detected by Linux vanilla and there are two solutions:

1. Use Kiszka's Linux kernel
2. Use a special "UIO driver" designed for Linux and provided by a side git project, called the "*ivshmem-guest-code*".

Once we succeeded in getting the two cells communicate via the pointer-based mechanism, an additional problem related to the address assignment to the IVSHMEM was detected. In particular, some locations were overwritten with uncontrolled values.

Probably that behaviour is due to the overlap of the root cell RAM section with the shared one. This makes it possible for Linux to use that space for other things.

To solve this problem, we avoid the overlap by configuration files.

Another possible reason is related to paging issues.

At this point, a peculiar behaviour occurred: only by restarting the non-root cell the changes made by it became effective even for the root cell.

In addition, if the non-root cell starts before the root cell and the last one writes in the shared memory, the changes don't affect the non-root cell. On the other way, if the non-root cell starts after the root cell, the changes are detected by the non-root cell.

These behaviours suggest a missed cache coherence. In fact, by manually flushing the cache using a Jailhouse Assembly function and by adding it in our code, we fixed the problem.

So, the non-root cell doesn't perform the write-back at every write in the shared memory and, also, Jailhouse doesn't have flags to make pages non-cacheable.

This represents a major limitation of IVSHMEM in terms of performance (a flush with each write) and implementation of protocols between the two cells (such as in our case).

8. Test and results

In the context of a *keepalive* protocol between two cells, we tested the recovery time, which is the time between the failure of the bare-metal cell and the restoration of the keepalive protocol.

In particular, we've made four kind of test:

- **Shutdown**: the bare metal cell is restored using the command *shutdown* in a non-stressed environment. The shutdown command is:

```
jailhouse cell shutdown 1
```

- **Shutdown with stress**: the bare metal cell is restored using the command *shutdown* in a **cpu-cache** stressed environment.
- **Destroy**: the bare metal cell is restored using the command *destroy* and then re-created in a non-stressed environment. The destroy command is:

```
jailhouse cell destroy 1
```

- **Destroy with stress**: the bare metal cell is restored using the command *destroy* and then re-created in a **cpu-cache** stressed environment.

For the stress of the cpu-cache we used **stress-ng** with the command

```
stress-ng --class cpu-cache --all 1
```

that stresses each CPU's cache (three for the root cell) with all the stressor that stress-ng provides.

Due to the version of petalinux (2019.1), it was not possible to load stress-ng on the kernel image generated with the tool, so we had to build it statically and then load it on the board (as shown in the appendix).

The data collected for each test consisted of 5373 samples. The distributions of these samples and the parameters of main interest are shown below (results are in nanoseconds).

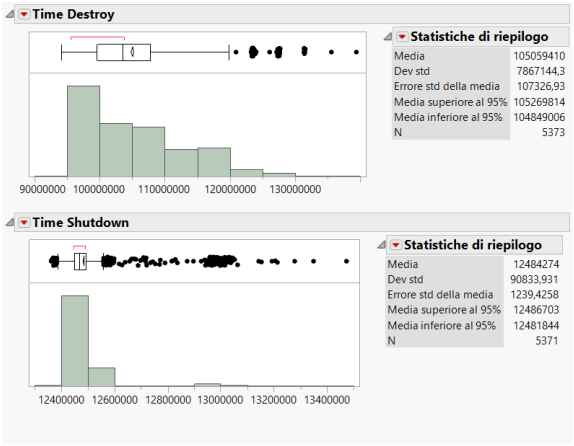

Figure 7: Data distributions without stress.

Specifically, the means are:

- **Destroy**: 105.059410 ms
- **Shutdown**: 12.484274 ms

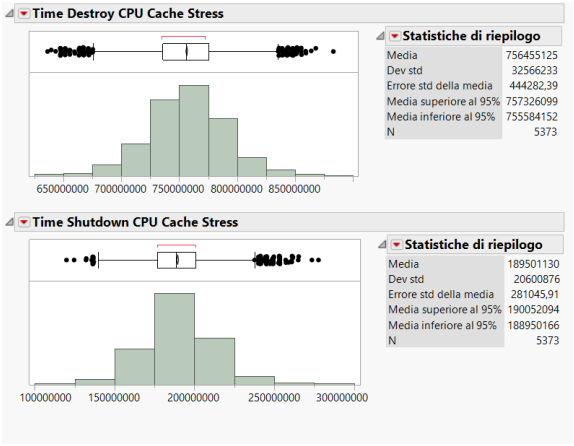

Figure 8: Data distributions.

- **Destroy with stress**: 756.455125 ms
- **Shutdown with stress**: 189.501130 ms

Following the box plot comparison between shutdown with and without stress, as well as for the destroy:

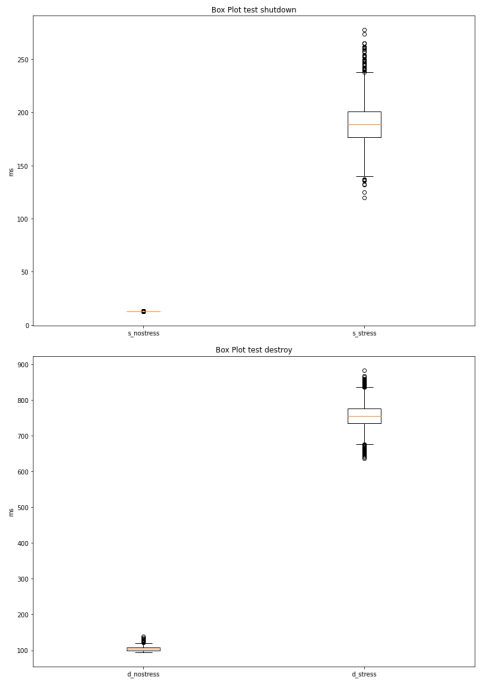

Figure 9: Box plot comparison.

The test results revealed three facts:

- use the *destroy* command instead of *shutdown* (as might be expected) implies significantly longer recovery times
- the system suffers from cpu-cache stress
- the deterioration in recovery time due to cpu-cache stress is double in the shutdown case compared with destroy. Specifically, with the destroy command the worsening is in the order of 7x while for shutdown we have a worsening in the order of 15x.

The first conclusion is rather obvious, the *destroy* command reallocates the resources reserved for the non-root to the root cell, after which the *create* command performs the reverse operation. The *shutdown* command simply shuts down the cell, and this results in a significant speedup in cell recovery (we are talking about ten times faster recovery).

The second conclusion is probably due to the fact that although Jailhouse is a static partitioning hypervisor, the memory controllers remain unique and shared between cells.

9. Conclusions

First of all, there're many open issues on this topic, in particular because of ARM64 architecture.

The IVSHMEMv2 protocol is theoretically well founded, and beyond that, shared memory between two static cells undoubtedly opens up several possibilities.

The keepalive protocol is just one of the possibilities but has proven to be a tricky implementation because of the cache's failure to writeback. This limits the implementation of reactive protocols between two cells.

Tests also showed significant slowdown by the hypervisor (on the order of x7 and x15) in cell recovery under cpu-cache stress conditions, this is a major limitation in a Mixed Criticality context. This can certainly be mitigated by the use of cache coloring coupled with a bandwidth control on the data bus.

However, a crucial aspect is the low compatibility of IVSHMEM with ARM64 architectures, a problem also recognized by the Jailhouse community and which has prevented us from using the protocol as it was designed, i.e., with interrupt mechanisms that we were unable to use.

References

M. Cinque. Partitioning hypervisor.

D. Ottaviano. Openamp guide: Rpu usage on zynq ultrascale+. https://github.com/DanieleOttaviano/OpenAMP_tests, 2022.

J. Kiszka R. Ramsauer. Jailhouse documentation. https://github.com/siemens/jailhouse/tree/master/Documentation.

D. Ramos. Exploring ivshmem in the jailhouse hypervisor. https://cister-labs.pt/docs/exploring_ivshmem_in_the_jailhouse_hypervisor/1596/view.pdf, 2019.

L. De Simone. Real time virtualization partitioning hypervisor.

Appendix A. Setup and Configurations

First of all, we installed Petalinux 2019.1 following the guide provided in
https://github.com/DanieleOttaviano/OpenAMP_tests
and then following
https://github.com/siemens/jailhouse/blob/master/Documentation/setup-on-zynqmp-zcu102.md
we made the setup on the Zynq Ultrascale+ ZCU104.

Of course, some changes have been done to adapt the setup to the Zynq Ultrascale+ ZCU104 instead of Zynq Ultrascale+ ZCU102. In particular we had to download the correct .bsp from
https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-design-tools.html.
To generate the .dts from the .dtb provided by Xilinx (lately included in configs/arm64/dts) we used the device tree compiler (dtc).

In the "config/arm64" directory the two cells configuration files are included.

In order to statically build *stress-ng*, after cloning the stress-ng repository the following command has to be executed:

```
CC=aarch64-linux-gnu-gcc-4.9 STATIC=1 make
```

In the non-root cell it is necessary to flush the cache at every write operation. To do this it is necessary to:

- copy "dcaches.h" in inmate/lib/arm_common/include/asm directory.
- copy "caches.s" in inmates/lib/arm64 directory
- add the code line "lib-y += caches.o" to the makefile in inmates/lib/arm64 directory
- Now including "dchaches.h", the arm_dcaches_flush() function can be invoked at each writing.

Appendix B. Code

The code developed and used is available at the following link:
https://github.com/MicheleSpina99/jailhouse-ivshmem-zynqZCU104