Distributed Real-time Architecture for Mixed Criticality Systems

Implementation of real-time scheduling heuristics and coordination for the KVM hypervisor

D 2.2.3

Project Acronym	DREAMS	**Grant Agreement Number**		FP7-ICT-2013.3.4-610640	
Document Version	1.0	**Date**	2016-05-04	**Deliverable No.**	[D 2.2.3]
Contact Person	Alexander Spyridakis	**Organisation**		Virtual Open Systems	
Phone	+33 (0)6 52 52 22 58	**E-Mail**		a.spyridakis@ virtualopensystems.com	

Contributors

Name	Partner
Nicolas Dagieu	VOSYS
Jeremy Fanguede	VOSYS
Alexander Spyridakis	VOSYS

Table of Contents

1 Introduction

This document is the deliverable D2.2.3 of the DREAMS project. It is the last deliverable of task *T2.2 – Resource Management and Adaptation Services for Mixed Criticality* of work package *WP2 – Multicore Virtualization Technology*. This deliverable, *D2.2.3 – Implementation of real-time scheduling heuristics and coordination for the KVM hypervisor,* presents the next set of scheduling enhancements implemented for KVM and their improved integration with the DREAMS hardware and software technological results.

D2.2.3 is the continuation of the first implementation in *D2.2.1 - Optimized hierarchical real-time scheduling heuristics.* In this document the overall status of the scheduling extensions is covered, such as the concept of guest – host scheduling coordination in KVM and how they are complemented and improved with additional infrastructure from the previous deliverable.

1.1 Position of the Deliverable in the Project

The objective of WP2 is to develop the chip-level platform of the DREAMS architecture, which will encompass a novel multi-layered HW/SW infrastructure with inherent temporal and spatial partitioning, real-time support, built-in security mechanisms and energy-awareness. Another critical technology, concerns resource sharing of key subsystems in the multicore SoC architecture, leveraging advances from component-based design, distributed communication and computation oriented monitoring facilities and hierarchical real-time scheduling to separate conflicting system, personal and business-owned application requirements. Modularity in the composition of hierarchical scheduling algorithms with high- and low-level schedulers will allow seamless support of different local schedulers by simply running a different guest OS.

D2.2.3 is part of T2.2, which considers the necessary extensions for priority based preemptive scheduling and context switching heuristics to bind task priority assignments (which are relative) to real-time constraints (which are absolute). Hierarchical scheduling heuristics based on cluster-level and core-level enable distributed decision making by independent, low-level local schedulers implementing space- and time-sharing of resources with different criticality structures to enable hard, soft and best-effort scheduling strategies, configured with a high degree of system modularity.

In regards to KVM, the T2.2 and D2.2.3 is focused on providing a set of mechanisms and additional infrastructure to enhance the performance of guest scheduling (e.g. CPU, disk I/O, memory bandwidth) for soft real-time scenarios in systems which their resources are over-committed. For hard real-time support, as it will be documented in *D2.3.2 - Firmware monitor layer implementation for the concurrent execution of an RTOS and Linux/KVM (M34)*, hard real-time capabilities are off loaded to an isolated RTOS which is executed securely on the same resources as Linux/KVM, by utilizing the TrustZone security extensions. This combination of hard/soft real time workloads, in different subsystems, requires further coordination, to ensure time criticality and overall system responsiveness.

The confidentiality level of this deliverable is public (PU) and it will be published on the DREAMS website, once approved by the European Commission.

1.2 Contents of this Deliverable

In chapter 2, we detail the problem of memory bandwidth in mixed-criticality scenarios and the extension of policies for guest systems, while on chapter 3, improvements of the previous coordinated scheduling approach (D2.2.1) are reported. Finally, in chapter 4 we conclude the status of the current work and the next directions targeting the Healthcare demonstrator.

2 Memory bandwidth policies extensions

In *D2.2.1 - Optimized hierarchical real-time scheduling heuristics*, the concept of coordinated scheduling between Linux host/guests was explored, providing a proof of concept implementation with some initial metrics on the performance improvement that such a design can offer. For this deliverable we explore a similar concept, but instead of disk I/O or task scheduling, virtual machines can communicate with the host to fine-grain their use of memory bandwidth. First we take a look at Memguard, a memory bandwidth aware scheduler, this solution is modified and ported for the use in the DREAMS ARMv8 platform, the Juno development board. The core functionality of Memguard is assessed with a number of benchmarks run on Juno and subsequently guest systems are exposed to the Memguard mechanism through a communication interface with the host.

2.1 Introduction

Nowadays, computers and embedded systems are based on multi-component architectures, which require at least a microprocessor, some RAM and other optional peripherals and storage devices. Over the last decades the performance of CPUs has been increasing steadily but memory, on the other hand, hasn't followed this trend, as such, computer systems are facing the "Memory Wall" problem. Even if new solutions like High Bandwidth Memory (HBM) or "stacked memory" are attempts to solve this problem [1], most actual platforms are based on standard DRAM. In this context it is difficult to provide a guaranteed bandwidth to an application, especially "real-time" (i.e. soft or hard real-time) applications executed together with other tasks. In this context, bandwidth is a major part of the system, especially on multi-core systems (which share memory).

The performance bottleneck of memory has been extensively studied and several solutions have been implemented. Most of them are hardware solutions [2] [3], at the memory controller level. Few solutions have been proposed at the software level [4], mostly for server distributed large scale systems [5]. Instead in this chapter we will take a look on how memory bandwidth management can be exposed to Virtual Machines, and how a coordination can be applied between the host and guest systems.

Since the bottleneck of memory performance can be pronounced even more in mixed-criticality environments. In consequence of this problem, tasks are starved during execution due to the lack of available memory bandwidth at the right time, which can significantly reduce the performance of the system. Experimental results show this bottleneck and highlight the importance of a scheduling method to solve it. A solution based on Memguard is explored and implemented to solve this issue on an ARMv8 SoC, which is representative of an actual high-end embedded computer system. The extension to Memguard also relies on *Completely Fair Scheduler* (CFS) the standard task scheduler of Linux, it involves a new scheduling mechanism to take care of the bandwidth and to manage tasks depending on their memory bandwidth usage.

2.2 Memguard

Memguard [7] is a memory bandwidth aware scheduler, it distinguishes memory bandwidth in two parts, guaranteed and best-effort. It provides guaranteed bandwidth for temporal isolation and best-effort bandwidth to use as much as possible the spare bandwidth (after all cores are satisfied). Memguard is designed to be used on actual systems using DRAM as main memory.

The common DRAM architecture consists of banks with different rows/columns. Maximum memory bandwidth can be achieved in the case where data are located in different banks, in other cases the memory bandwidth can be limited and to address this bottleneck a solution named Memguard is used, which takes care of scheduling the memory bandwidth to provide the desired Quality of Service.

Memguard is implemented as a Linux kernel module, which is based on the use of the Performance Monitor Unit (PMU). It captures the memory usage of each core by reading the Performance Counter Monitor (reading memory requests if used with PCM version < 2.4 and memory reads and writes with PCM version > 2.4).

The module architecture is based on two parts, the first being the Reclaim Manager which stores and provides bandwidth allocation to all per-core B/W regulators, while the other part is the per-core B/W regulator that monitors (thanks to the PCM) and regulates the memory bandwidth usage of each core. Memguard is linked to physical cores, the regulation process works only at the core level. Due to this architecture, regulating a process running on several cores at once is not easily feasible.

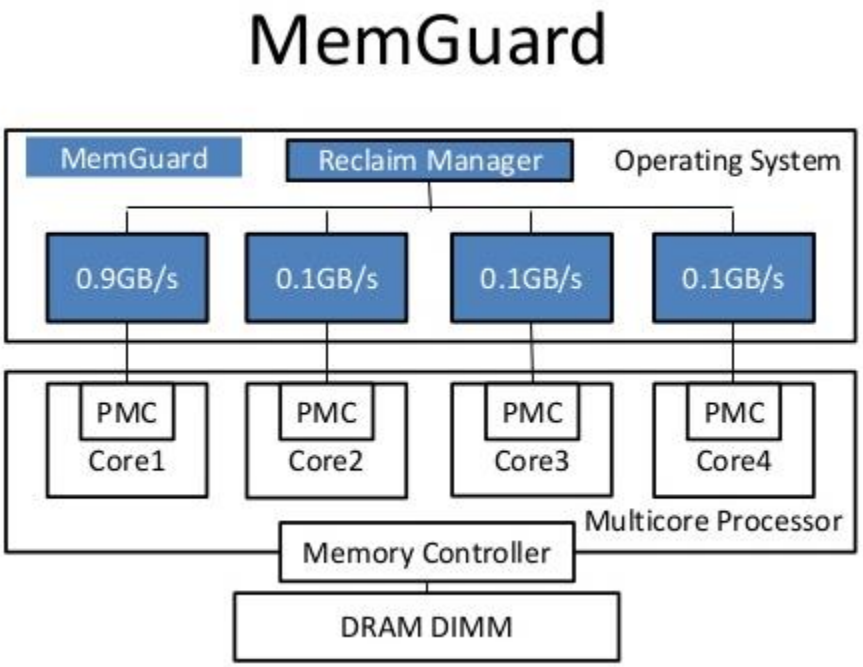

Figure 1: Memguard architecture overview

We can describe the architecture as follows:

- The global budget manager also known as ***Reclaim manager***: It handles the memory budget on each core of the CPU. Every scheduler tick (1 ms) if the predicted budget of each core is under the assigned (fixed) budget of overall system, a *memory budget tank* is set to give more bandwidth during the future time slice if a task need to access to more B/W than required (and some B/W is available in the reclaim manager).
- The per core ***B/W regulator***: It handles the memory management for each core, updating the actual used budget with the PCM value, configuring the PCM to generate an overflow when all memory budget is used and reclaiming more bandwidth from the reclaim manager if needed.

Beside the overall architecture, Memguard has different features. Its major functionality is bandwidth management limiting, allowing a user to set a limit (in MB/s, “weight” or in percent). Another feature is the per-task mode, it uses task priority as a core's memory weight. The last major feature is the “reclaim bandwidth” functionality, distributed any leftover bandwidth that was not consumed. This last feature enables to use as much as possible memory bandwidth. When not in use, the available bandwidth is equal to the max-bandwidth setting set at start (or updated later).

Memguard can be used in different ways. The simplest use of Memguard is to balance workloads, reducing the memory bandwidth of a task to preserve memory-bandwidth for others. Memguard usage is linked to the physical cores of the CPU, consequently the application level use is

complicated and must be done manually. Memguard requires setting the bandwidth manually, as such users must be careful on which core, applications are running on, and adjust meaningfully each application's B/W needs.

2.3 Trends and virtualization

In the past most actual embedded systems were designed to handle standalone actions within simple applications. Nowadays, more and more complex tasks are used through embedded systems. Multimedia applications and database analysis are now common. Embedded systems are actually designed with several micro-controllers communicating with each other (and/or with a master), increasing cost and decreasing the MTBF (mean time before failure).

New kind of needs appear, requiring powerful embedded systems with a large number of connection interfaces. In the near future, most actual multi-chip embedded systems will be replaced by a central unit performing all computation and networking tasks. This embedded systems architecture direction raises the problem of mixed-criticality which is at the heart of the system. If a single platform is used to run different criticality software, some requirements are needed.

Mixed-criticality means running some hard-real-time application with soft-real-time or standard application at the same time on the same processing unit. Also this kind of system, needs to provide security separation between tasks to ensure data/program isolation. Cooperation between hard and soft real time processes pulls all the software interface to be more cooperative and resource/need aware. Each program needs to exchange information in order to provide Quality of Service.

Virtualization is the last component of future unified embedded systems architecture. Virtual Machines give the possibility to ensure the security and resource isolation between tasks. Each task, for example a video processing task (capture video from a sensor and process the image to find particular patterns) could be executed at the same time as video playback and/or a more critical task. Each task can then be executed in a separated VM with all the software needs and the correct amount of processing/memory bandwidth reserved.

In this context, the memory bandwidth management becomes the bottleneck of the system not only because all cores use the same memory but also because all different VM are running “simultaneously”. Each VM handles a certain software environment, with a specific priority and memory bandwidth need. The priority of the guest can already be solved by a priority scheduling mechanism which we explored in D2.2.1, while the memory bandwidth must be managed to reduce memory bottlenecks.

2.4 Memguard results on Juno

For this work we utilize a specific benchmark suite, composed by a virtualized environment and different benchmark software. The environment is Linux 4.3.0 kernel with an open-embedded file-system, QEMU/KVM is the selected virtualization solution. The actual benchmark platform is Juno r0 an ARMv8 development board with 2*Cortex-A57 and 4*Cortex-A53 cores. For testing only A57 cores are used to run the needed number of guests, as the memory bandwidth difference between A57 and A53 cores is too large to include both types of cores(from 2500MB/s to 1500MB/s). The *taskset* utility is used to set guests on specific cores. Guests are running with a 4.3.0 Linux kernel and a minimal filesystem, including the benchmark software suite.

The first benchmark used is a program used by the original author of memguard, this program is used to get a point of comparison between our platform and the author's one. It consists in a simple buffer copy-process program to use a large amount of memory bandwidth, it simply provides a number of “processed” frames per second. The second program is the well-known Mplayer video suite. Mplayer was chosen to represents some multi-media use-case in a mixed-criticality environment. Mplayer is used with the benchmark option to see if a high-bitrate video decoding (two videos are used, 5Mb/s and 1Mb/s) process is runnable in the benchmark environment. The

last benchmark is an FFT program, simulating a capture and process task in soft real-time constraints. The FFT benchmark is called periodically and sets a buffer to process FFT computation on it, the output result is a the number of processed buffers per second.

The first test (Figure 2) is the memory bandwidth limitation mechanism. For this purpose, four different tasks are launched at the same time. A different memory bandwidth weight will be associated to each core/task. Each task is running on a specific core (one core = one task).

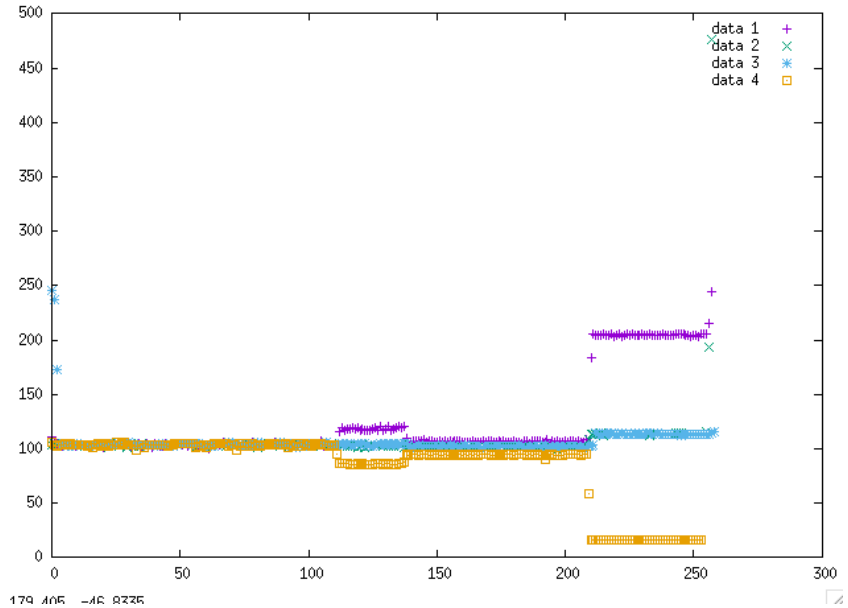

Figure 2: Memory bandwidth limitation in different tasks (x axis in seconds, y axis in MB/s)

During the first 100 seconds, Memguard was not activated. Memguard is enabled after the 120th second. After 220 seconds, the Memguard module is working with different weights to highlight the memory bandwidth limitation on each task. Task 1 has the maximum weight while task 4 has the lowest (tasks 2 and 3 have the same weight). The results are expected, it shows that the Memguard module is regulating the memory bandwidth of each task.

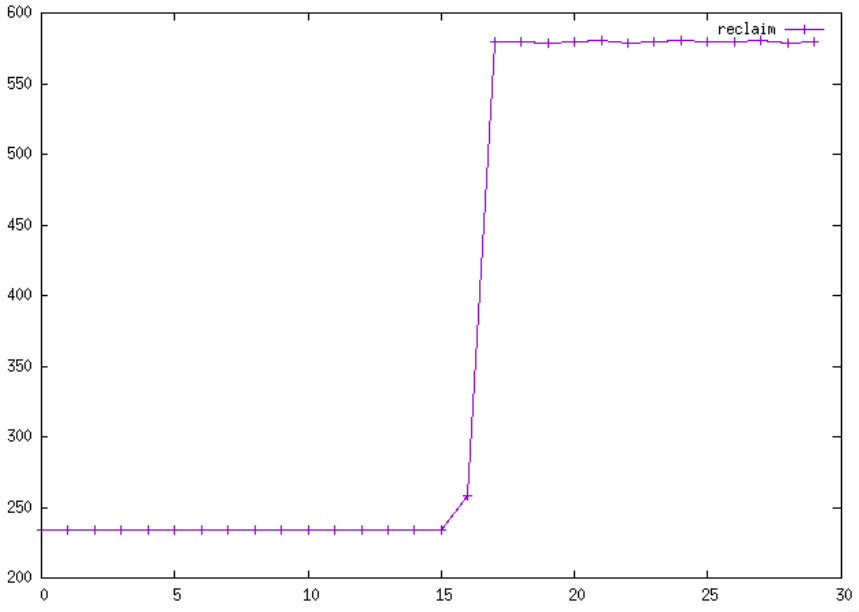

Figure 3: Memory reclaim results (x axis in seconds, y axis in MB/s)

The second experiment (Figure 3) is about the reclaim feature, a simple task will be used to test if Memguard can release more bandwidth than the chosen limitation. The task is running between 0 to 15 seconds with an under-estimated memory bandwidth limit. The limit was set to 240MB/s, when the reclaim feature is enabled the memory bandwidth reaches 590MB/s. This experiment shows that the reclaim feature can provide more than twice the original memory bandwidth limitation if more bandwidth is available.

The CPU overhead of Memguard on the global performance was measured to understand how to use Memguard in order to reduce as much as possible the overhead. The test uses Memguard with the reclaim feature activated. The overhead test (Figure 4) shows that Memguard is performing well under a large memory-bandwidth allocation per core. This drawback comes from the way Memguard is working, if a core is under-estimated when memory-bandwidth is set, Memguard produces a large overhead due to the reclaim feature.

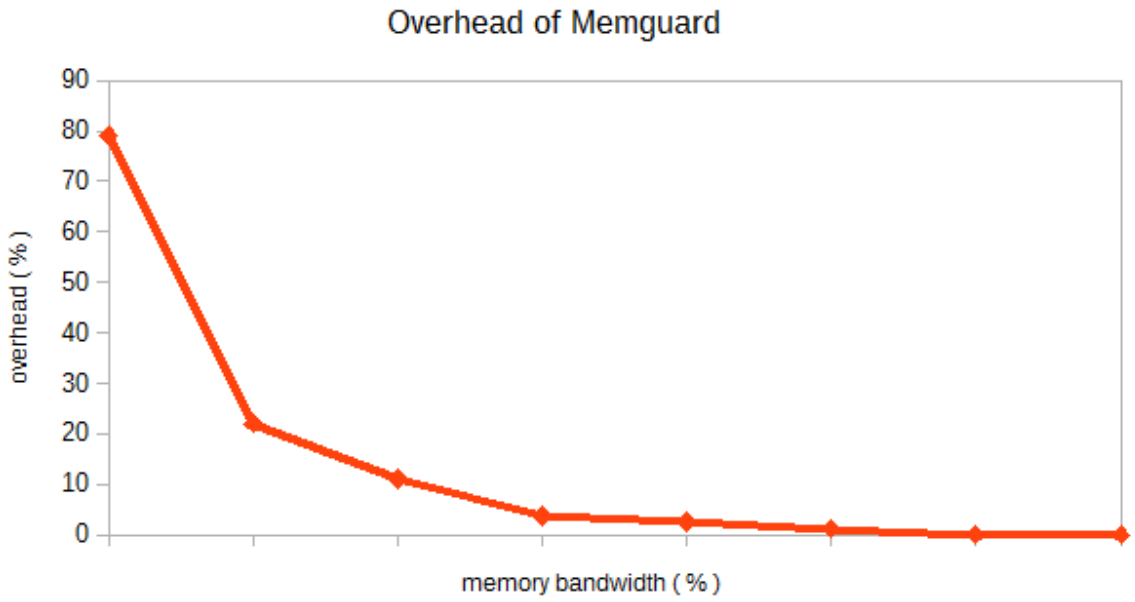

Figure 4: Memguard CPU overhead in relation to memory bandwidth limiting

Finally, in order to understand the way Memguard can be used in real-life scenarios, a test with video playback has been done. This test highlights the memory-bandwidth reservation capability of Memguard. Without any memory bandwidth limitation, 60s (approximately) are needed to decode the video, whereas when Memguard is enabled, decoding lasts 58s. The interest of Memguard resides in the memory-bandwidth temporal reservation. A core can be limited to let others cores use as much as possible the available memory bandwidth.

Plain Linux with 2 cores executing the same video task (mplayer)	Core 0: 60.318s Core 2: 60.320s
Memguard with under estimated bandwidth : 20 MB/s on all cores	Core 0: 313.313s Core 2: 311.306s
Memguard with correct estimated b/w for core 0 (250 20 20 20)	Core 0: 58.836s Core 2: 276.001s
Memguard with correct estimated b/w for core 0 and best-effort policy	Core 0: 59.881s Core 2: 95.619s

Table 1: Different rendering times with different memory b/w settings

2.5 Memory bandwidth policies in embedded virtualization

Since with QEMU/KVM a virtual machine is seen as an additional task to schedule in the host, then the memory-bandwidth bottleneck becomes a limiting factor. Every guest is using the same memory bandwidth and no hierarchy is implemented (like in a CPU scheduler) between guests. This memory bandwidth bottleneck can eventually affect the performance of guests in scenarios where memory is aggressively utilized.

When Memguard is used to regulate guests, users must launch each guest on one specific core (or several but, a core must be reserved to each guest), reducing the interest of using Linux with KVM, with the load balancing between cores. The use of a virtualized environment introduces also another use-case, VMs are highly dynamic processes for the host, as they have dynamic workloads and there is a need to change their memory bandwidth limit whenever needed. This results in the need for Memguard to be more flexible and be able to regulate on a process granularity instead of cores.

2.6 Architecture and implementation

The aforementioned problem in virtualized environments can be solved using a memory bandwidth scheduler. The solution is based on a new architecture involving all layers of the computing chain (from guest to kernel of the host). It can deliver messages and regulate the memory bandwidth dynamically. The architecture of the solution is split in three main parts: the guest level API, the host message exchange mechanism and parts of Memguard linked to CFS. The selected architecture helps to keep a simple yet flexible mechanism. The first part is composed of a simple debugfs interface, allowing user to write/read from a simple file to set the needed memory-bandwidth value. It allows setting the memory bandwidth from another program (like a local resources manager).

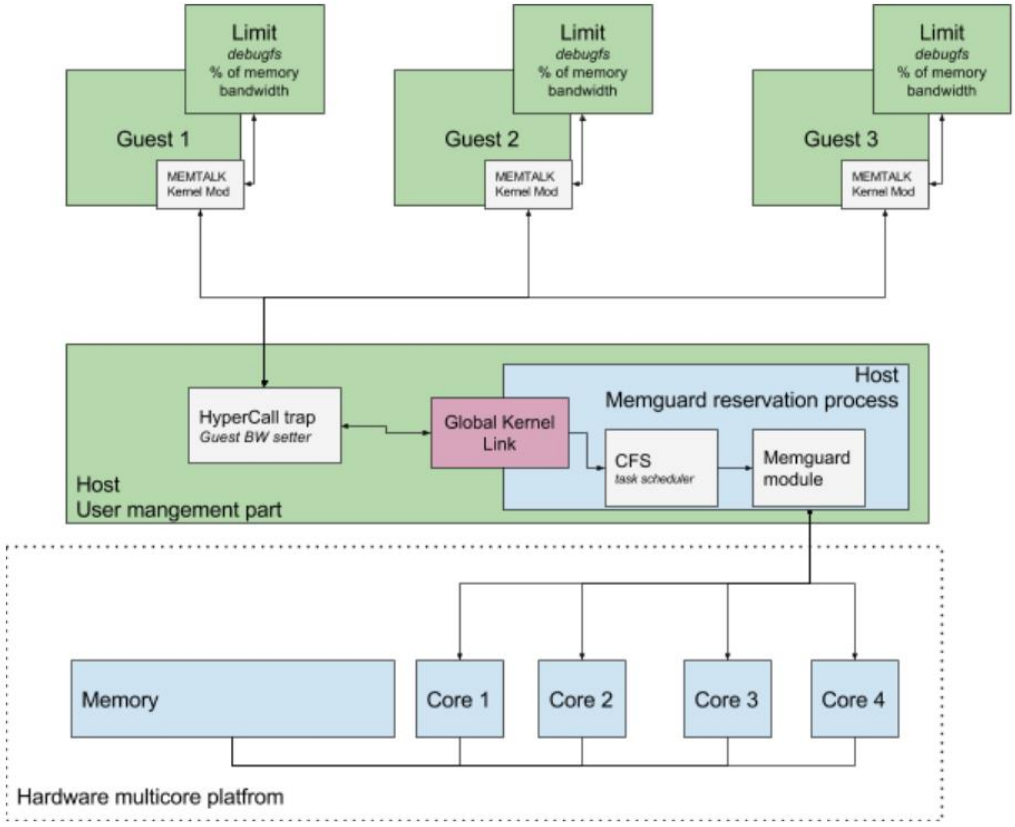

Figure 5: Architecture overview of Memguard with virtualization extensions

Every call is made with:

- **Request ID**: Host is aware that this call is a guest request
- **Request type**: Host is notified if guest wants to update the bandwidth or be removed from the guest reservation process
- **Value**: A 64-bit variable to exchange information (e.g. bandwidth need: 70%)

The second part is the hypercall module, which is processed by KVM in the host, every hypercall is trapped, filtered and processed. The hypercall process will be described in detail in the next chapter.

After the guest issues a hypercall KVM traps the guest and the memory-bandwidth request is stored in the host kernel. The kernel structure for the information needed is composed of:

- **memguard_sched_guests**: The number of guest executed with memguard reservation enabled
- **memguard_sched_PID**: List of guests PIDs
- **memguard_sched_BW**: Bandwidth request of guest
- **memguard_update_bandwidth**: A pointer to the Memguard callback function

The third part is the mechanism which regulates the bandwidth applying the requested bandwidth that was stored. This part is composed of two components, CFS, the Linux scheduler and Memguard, the kernel module, regulating memory-bandwidth at core level. CFS was selected because it is the default Linux scheduler and is fair between tasks.

The following pseudo code snippet is a method that calls Memguard when the guest vCPU process is being scheduled:

```
function memguard_guest_update(cpu_number)
        if next_task = a_guest_in_the_list
                callback_to_memguard()
```

When CFS has scheduled the next task, a callback to Memguard is executed which then enforces the memory bandwidth regulation. It is also worth mentioning that Memguard had to be also modified in order for it to handle the callback from CFS. This function in Memguard updates the memory bandwidth of the core corresponding to the linked guest.

```
update_budget_sched(int cpu_n, long bw_n)
        convert_bandwidth_to_cache_event()
        set_the_core_budget()
        initialize the memguard statistics()
```

The actual implementation has several benefits. The first one is the limited overhead due to a change in the memory bandwidth requested by the guest, as a hypercall is performed only when needed, reducing the total time processing the bandwidth modification. The second benefit relates to the use of the CFS scheduler. This significantly reduces the complexity of integrating the solution, and the overhead is kept to a minimum. The last benefit comes from the Memguard callback, which provides memory bandwidth reservation and limitation functionalities.

As discussed previously, the target is to define a virtualized mixed-criticality based solution to regulate memory bandwidth. The guest user is able to set the needed bandwidth or let the system take care of this transparently. This results in the possibility to dynamically adjust memory bandwidth, which allows the regulation of tasks between them, reducing the memory bandwidth of a task enabling other tasks more resources.

2.7 Experimental results

Following are some tests and benchmarks which show how the Memguard extensions can be used to get better performance. The first test (Figure 6) shows the problem of the memory bottleneck. When two guests are running on the same core, both tasks are limited. As in the first set of tests, in the virtualized environment the bottleneck remains the same.

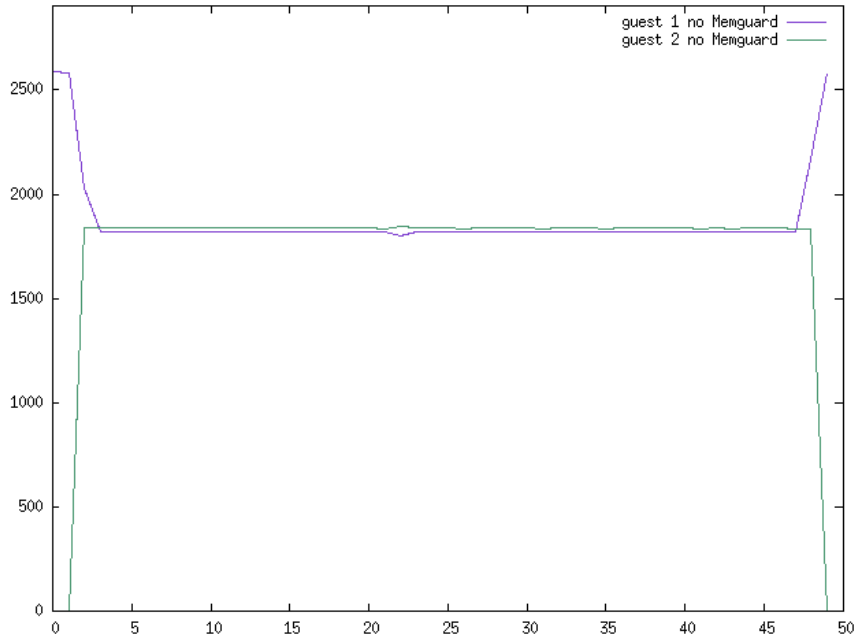

Figure 6: Memory bandwidth sharing between two guests (x axis in seconds, y axis in MB/s)

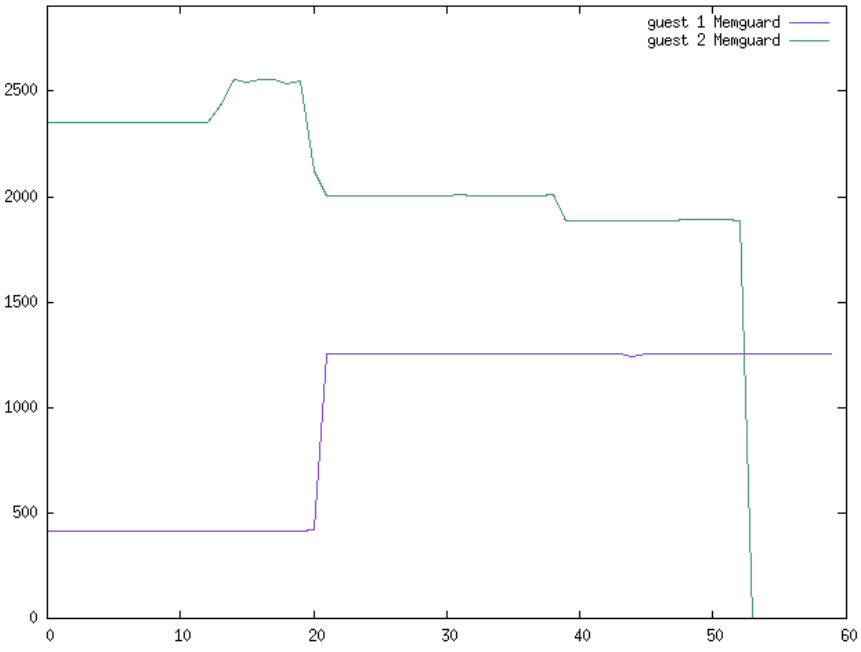

Figure 7: Memory regulation between guests (x axis in seconds, y axis in MB/s)

The second test (Figure 7) highlights the gains of the Memguard extensions. Initially, both tasks are bandwidth limited, the first guest at 70% of the guaranteed bandwidth while the second guest at 20%. When the Memguard module is disabled (between 13th and 20th second) the first guest can reach the maximum bandwidth. After 20 seconds the first guest requests more bandwidth, which

results for less bandwidth for the second guest. The interesting point is that both guests are executed with a different bandwidth percentage, which allows for a hierarchical differentiation between them.

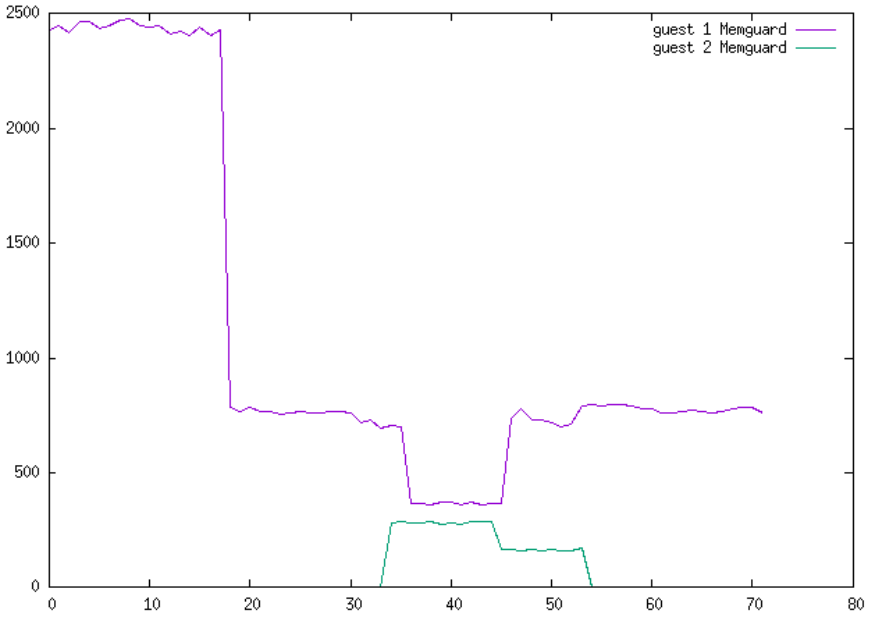

Figure 8: Memory separation

Figure 8 demonstrates the memory separation between guests. The first guest initially is running without a limit, after 17 seconds, a limit is enforced, a second guest is launched after 33 seconds with a limited bandwidth. The memory-bandwidth fall is due to the CPU time shared between both guests (running on the same core). The extended version allows Memguard to regulate memory bandwidth from a core level to the guest level.

Plain Linux 2 cores executing the same video task (mplayer)	Guest 1: 62.112s Guest 2: 67.968s
Memguard with under estimated bandwidth: 20 MB/s on all cores	Guest 1: 386.893s Guest 2: 384.655s
Memguard with correct estimated b/w for core 0 (250 20 20 20)	Guest 1: 57.947s Guest 2: 312.014s
Memguard with correct estimated b/w for core 0 and best-effort policy	Guest 1: 60.911s Guest 2: 97.665s

Table 2: Different rendering times in guests with different memory b/w settings

Finally, the Mplayer benchmark was done with a decoding process per Guest. The results are following ones produced without the guest environment. The current implementation is giving at least the same results as standard Memguard.

3 Coordinated scheduling enhancements

To complement the work being initiated in *D2.2.1 - Optimized hierarchical real-time scheduling heuristics*, the initial proof of concept for disk I/O and task scheduling has been extended to provide a more robust implementation. This includes the usage of a standard Linux interface for paravirtualization which improves the overall infrastructure for the coordinated scheduling mechanism. Additionally the Linux *sysfs* interface is being utilized both in the host and the guest system in order for users to be able to interact and control the scheduling policies involved.

3.1 Paravirt ops interface for KVM on ARM

Linux already provides a way to perform some paravirtual actions through an infrastructure named *paravirt-ops* (pv-ops for short) [8]. This API is used to run para-virtualized virtual machines on multiple hypervisors with the same kernel binary. That is to say the same kernel binary can run on bare hardware, or on hypervisors such as VMWARE VNI or Xen, it can be para-virtualized or full virtualized [9]. This infrastructure exists for multiple architectures and hypervisors, but not for KVM on ARM, one of the virtualization solutions for DREAMS. Therefore, a basic *paravirt operator* was developed in order to implement the coordinated scheduling extensions with a ready to use infrastructure, providing better flexibility and maintainability for new features. The approach is based on a previous work that enables paravirt-ops for Xen on ARM/ARM64 [10], thus in this chapter we focus on the implementation for KVM on ARM.

Paravirtual functions require a *hypercall* implementation, in order to be able to send information to the host system. Therefore, hypercall functions specific to KVM have been implemented into the KVM code base of the Linux kernel. These functions use the *HVC* (hypervisor call) instruction of the ARM architecture, with the immediate argument of the HVC instruction being a constant integer used to recognize a paravirt call (from a Power State Coordination Interface call, for instance, which can also use an HVC instruction [11]). The parameters of the hypercall are passed through the scratch registers, r0 contains the identification number of the hypercall and registers r1 to r3 represent the potential arguments for this hypercall. Figure 9, details the implementation of the *kvm_hypercall1* which is the hypercall implementation with one parameter.

```
static inline int kvm_hypercall1(u32 num, u32 arg1)
{
    register u32 n asm("r0");
    register u32 r asm("r0");
    register u32 a1 asm("r1");

    n = num;
    a1 = arg1;
    __asm__ __volatile__(
        __HVC(KVM_IMM)
        : "=r" (r) : "r" (n), "r" (a1) : "memory"
        );

    return r;
}
```

Figure 9: Code example for the hypercall '1' implementation on KVM

Those hypercalls are called from the paravirt-ops implementation of each paravirtualized subsystem. In our case it consists of pointers to functions, stored in a structure that represents the paravirt subsystem. Those functions are called if the paravirt-ops infrastructure is enabled for the hypervisor on which the virtual machine is running. For our needs a paravirt-ops interface named *pv_cosched_ops* was added. Along with a new hypercall named *KVM_HC_COSCHED*. The *pv_cosched_ops* paravirt interface in the guest contains four functions:

- **Register VM:** During boot the virtual machine will issue a register call to the hypervisor, if the hypervisor doesn't support the *pv_cosched_ops* interface, then all coordinated functionality is disabled. The registration procedure, if successful, also enables a *sysfs* entry in the Linux host, where the user can selectively enable/disable or even fine tune the priority of a guest.

- **Deregister VM:** At any point in time, the host might decide that coordination is no longer desirable during runtime. If a hypercall attempt for the guest is denied the next action of the guest is to request its deregistration from the host. Additionally the guest can also request deregistration if the guest user decides to do so, through a Linux *sysfs* entry.
- **New task**, *new_task()*: Called each time a new process is created. We use this function to implement a heuristic mechanism to detect which are the tasks that need to be prioritized. This function is called from *wake_up_new_task()* in the Linux kernel code (kernel/sched/core.c) [12].
- **Activate task**, *activate_task()*: Called each time a task becomes runnable. That is to say, each time a task which was waiting voluntary or due to an I/O wait becomes runnable again. We also use this function for the detection mechanism of the task to prioritize. This function is called from the function *activate_task()* in the Linux kernel (kernel/sched/core.c).
- **Schedule**, *schedule()*: Called each time a new task is scheduled. It is in this paravirt function that the hypercall *KVM_HC_COSCHED* is performed; to request a higher or lower priority. This function is called from *__schedule()* in the Linux kernel code (ker nel/sched/core.c).

On the host side, HVC instructions executed by the guest are trapped by KVM (in function *handle_hvc()* in arch/arm/kvm/handle_exit.c), and thus, can be handled correctly, the immediate argument of the HVC instruction is also checked to be sure that it is a hypercall and not something else (e.g. a PSCI call, or an invalid call). Then, KVM can perform the corresponding action to this hypercall according to the value retrieved from r0. For the hypercall we added, *KVM_HC_COSCHED*, it takes only one argument, which is an integer, set to 1 if the guest needs to be prioritized and 0 if it doesn't need this anymore.

3.2 Paravirt-ops host side implementation

The modifications done in the host side are located in the KVM and the scheduler code base of Linux. We had to implement the "backend" of the KVM_HC_COSCHED hypercall, which retrieves the argument of the hypercall and performs the corresponding actions. Thus, according to the argument, which could be 0 or 1 the hypercall handler will finally invoke the functions *coshed_boost_task()* or *coshed_deboost_task()* on task current. The task current is always a vCPU thread in that case.

The added function *cosched_boost_task()* lives in the scheduler code base of Linux (kernel/sched/core.c), it takes a struct task_struct as an argument, which is the task to prioritize (although in our case this function is always called with current as an argument). It boosts the priority of all threads associated to this task, i.e. the potential other vCPU threads and the I/O threads. We choose to prioritize those processes with a *SHED_RR* policy of priority 1. For this purpose it invokes the function *sched_setscheduler_nocheck()* to change the scheduling policy of these tasks.

The function *cosched_deboost_task()* does the reverse operation, that is to say it lower the priority of all the threads related the virtual machine to the default one, so that the policy of the processes is reset to *SCHED_NORMAL*.

3.3 Paravirt-ops guest side implementation

On the guest side the modifications consist in calling the hypercall to request a higher or lower priority at the right time. Therefore, the *schedule()* paravirt function of *pv_coshed_ops* is called from the core *__schedule()* function (in kernel/sched/core.c) [x] equipped with the next task to schedule as a parameter. A test on this future process to run is performed to determine if this process needs to be prioritized or not, according to this information the hypercall is executed with the correct argument (raise or lower priority).

```
function pv_cosched_ops.schedule(next_stask):
 static start_time
 static boosted = false
    if need_to_be_boosted(next_task):
       start_time = current_time()
       if not boosted:
          kvm_hypercall1(KVM_HC_COSCHED, 1)
          boosted = true
    else if boosted:
       now = current_time()
       if (start_time + MIN_BOOST_TIME) <= now:
          kvm_hypercall1(KVM_HC_COSCHED, 0)
          boosted = false
```

Figure 10: Pseudo-code of the paravirtual "schedule" function

Importantly enough, the time needed to perform a hypercall is not negligible, especially because of the HVC instruction, trapped by KVM. We estimate this guest to host plus host to guest context switch at around 1500 clock cycles for a Cortex-A53 core on ARM's Juno development platform. Thus, if the number of hypercalls is too frequent the performance will be worse than without the co-scheduling mechanism due to this overhead. So in order to solve this problem, the guest will request higher priority for a process, for at least a minimal period of time, i.e. the guest guarantees that it will not require a prioritization period inferior of the minimal raising time. The pseudo-code of this paravirtual *schedule()* function is detailed in Figure 10.

Function *need_to_be_boosted()* determines whether a task deserves to be prioritized or not. All tasks managed by a real time policy (i.e. *SCHED_FIFO, SCHED_RR and SCHED_DEADLINE*) are qualified for being prioritized, it corresponds to all the tasks that have the prio field of the struct task_struct strictly inferior to 100. For tasks managed by the fair policies (i.e. *SCHED_NORMAL* and *SCHED_BATCH*), a linked list of all tasks to prioritize is maintained, this is where the two other paravirt functions are useful: New task and Activate task.

Each time a new task is created the paravirt function *new_task()* adds this task to the prioritized list of tasks and each task has a counter associated and initialized to a positive value. This paravirt function is called form *wake_up_new_task()* in Linux (kernel/sched/- core.c). Each time a task of this list is scheduled, its counter is decremented (in the *schedule()* paravirt function), and when it reaches 0 the task is removed from the list. The counter is incremented each time a task is woke-up from a voluntary sleep, that is to say, a sleep caused by the task itself, e.g. a wait for a I/O job or a timer, this is done in paravirt *activate_task()* which is called from *activate_task()* in Linux (kernel/sched/core.c).

3.4 Host sysfs user interface

Up to now the coordination mechanism was transparently initiated between the host and guest kernel, without any user interaction. This can end up in scenarios were multiple guests are competing with each other for resources (essentially having the same priority between them), or cases where a non-trusted guest can exploit coordination starving other tasks in the system. In order for the user to have more control over the host system, a set of Linux *sysfs* entries are created when a guest is using the coordinated scheduling extensions.

By default, the Linux host/guest provide a sysfs interface to the user in which coordination can be enabled or disabled for each type of scheduler. For example in the case of disk I/O scheduling the following entry is created when the each system boots:

```
/sys/block/sda/queue/vbfq/enable
```

The user can then issue a simple command to completely enable or disable the coordination enhancements of the scheduler:

```
# echo 1 > /sys/block/sda/queue/vbfq/enable

# echo 0 > /sys/block/sda/queue/vbfq/enable
```

When a guest is booted, the first step of the respective scheduler is to issue a registration hypercall to the host system. If registration is successful, then the guest is in a position to continue with normal coordination hypercalls. If registration is denied, or if any coordination attempts fail to be completed, then the guest scheduler is no longer issuing any hypercalls, avoiding unnecessary context switches for handling the hypercall between the host/guest.

From the host side, once a guest is successfully registered another set of *sysfs* entries are populated, where the user can have a more fine-grained control for each registered guest. Each registered VM has its own entry in *sysfs* with a listing of all VM processes that are involved, along with an entry to selectively enable or disable coordination for a particular VM. Additionally a priority entry is provided where the user of the host system can select if a guest should be further prioritized among other different guests that use coordination (by default they share the same priority). In the following *sysfs* example 10301 is the PID of the registered guest:

```
/sys/block/sda/queue/vbfq/10301/enable

/sys/block/sda/queue/vbfq/10301/priority
```

3.5 Hypercall integration with the secure monitor firmware

D2.3.2 - Firmware monitor layer implementation for the concurrent execution of an RTOS and Linux/KVM (M34), in order to be able to execute mixed-criticality workloads and properly guarantee hard and soft real-time latency, a secure monitor firmware layer has been implemented specifically for the needs of the Healthcare demonstrator and the DREAMS project. This firmware essentially allows the concurrent execution of two different operating systems, ensuring their temporal and spatial isolation by means of hardware and software support.

The secure monitor firmware implementation is based on the TrustZone security extensions, which is supported by most modern ARMv7 and ARMv8 processors. TrustZone implements in hardware the concept of different execution modes, called the Secure and Non-secure world. Additionally, properly supported resources can be partitioned to Secure and Non-secure, as for example, memory, peripherals, interrupts and even timers. Secure world protection is ensured by monitoring physical access to memory or peripherals, therefore, a trusted OS, running in Secure world, is totally isolated from applications executing in the Normal world.

Combined with the paravirt-ops interface of this deliverable, the secure monitor firmware can enable Virtual Machines to access secure services (such as encryption, DRM, etc), either as part of the secure monitor, or even from another operating system executed in the Secure world. Usually in this kind of system the host operating system can access secure services by issuing the Secure Monitor Call instruction (SMC). From the guest side calling an SMC instruction will immediately trap to the KVM hypervisor and abort guest execution. In our case by utilizing the same principle as hypercalls we can implement a paravirt operator for secure services. When called, KVM will first trap the guest, and subsequently make a real SMC call which will end up in the Secure Firmware service routine or the Trusted OS implementing the requested service.

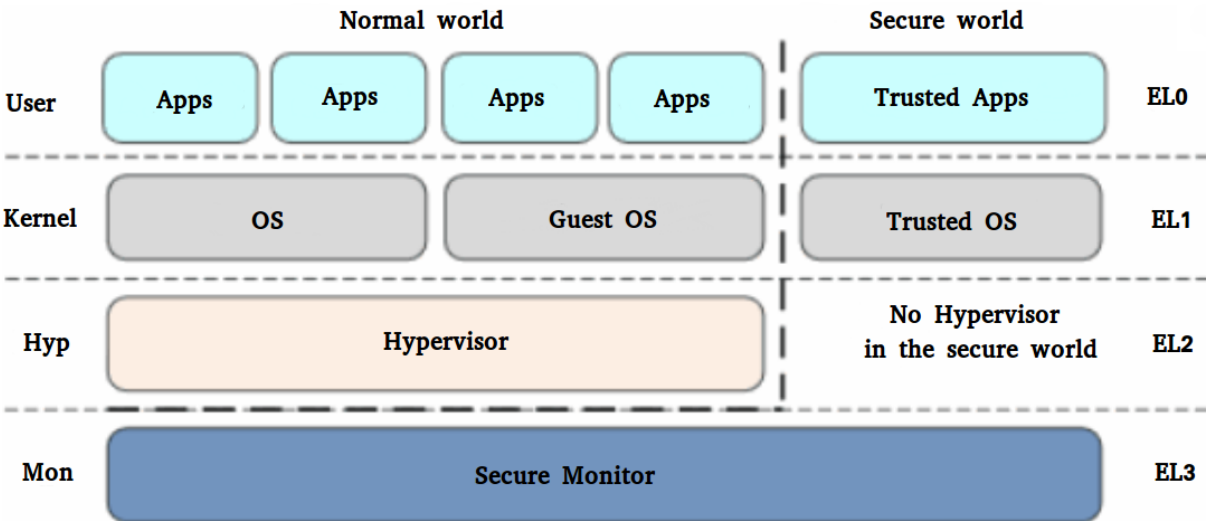

Figure 11: Execution mode overview on ARMv8 and how they relate between them

At this point no real service has been implemented yet, although simple exchange of information is possible. In order to measure the communication latency between a guest and a trusted OS we are utilizing the PMU cycle counter of ARM cores. Figure 12 shows the interactions and the traversal of SMC calls between the Linux guest, KVM and FreeRTOS (as the trusted OS).

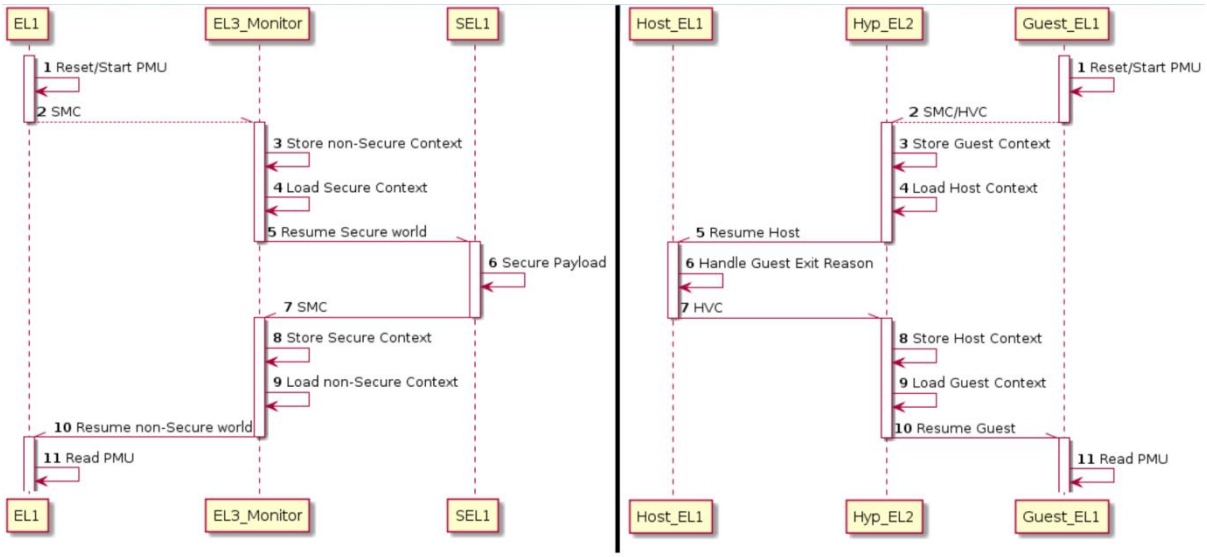

Figure 12: Right side, World switch (Linux – Monitor – FreeRTOS) – Left side, VM context switch (guest – Linux/KVM)

We measure two different paths, first the world switch latency response. This includes going from Linux to the Secure Monitor, then to the Trusted OS (FreeRTOS) and back. For this scenario the latency is in the range of 2300 clock cycles, with a frequency of 700 MHz on a Cortex-A53, this translates roughly to 3,2µs of latency.

The second measurement is the VM context switch latency needed between the guest to Linux/KVM and back. For the same processor and clock frequency as before the average clock cycles needed to do the full path context switch is in the range of 1500 clock cycles or roughly 2,15µs of latency. With this in mind we can estimate that the total time needed for a guest to interface with a trusted OS is in the range of 5,4µs.

As a next step in this direction, the VM context switch latency can be reduced by issuing an SMC call in Hyp_EL2 instead Host_EL1, this will allow for avoiding 2 additional world switches dropping significantly the latency between a guest and a secure service.

4 Conclusion and future work

In this deliverable we report the latest scheduling enhancements that have been implemented for task and I/O scheduling, work that was first initiated during deliverable D2.2.1. Additionally, memory bandwidth policies have been considered and utilized on the DREAMS platform (Juno development board), where the concept of coordination has been adapted for and guests can ask for more or less memory bandwidth depending on their scheduled tasks. Further effort has also been dedicated towards a range of features which give the user better system control of the scheduling policies.

Another aspect for the activity in this deliverable, is the convergence of the hypercall infrastructure for guests with the secure firmware monitor (main part of D2.3.2), where a guest can eventually request access to secure services, which are exposed to either the Linux host or guests through the most privileged execution mode in TrustZone.

The continuation of this work includes the tight integration with all other technological results on the DREAMS Healthcare demonstrator and extensive testing with the userspace software involved in the healthcare use case. Additionally, more detailed metrics and performance comparisons are expected to be part of the final assessment report in *D8.3.2 - Assessment report for mixed-criticality healthcare and entertainment use cases*, where the coordinated scheduling aspect is going to be highlighted in cases where resource over-commitment can hinder the performance of the system.

5 Bibliography

[1] Seth H Pugsley, Jeffrey jestes, Huihui Zhang on NDC: Analyzing the Impact of 3D-Stacked Memory+Logic Devices on MapReduce Workloads http://www.cs.utah.edu/~rajeev/pubs/ispass14.pdf

[2] Optimizing Memory Bandwidth in Systems-on-Chip Krishnan Srinivasan http://sonicsinc.com/wp-content/uploads/2012/09/Presentation_Multicore_final.pdf

[3] Self-Optimizing Memory Controllers: A Reinforcement Learning Approach Engin Ipek http://www.csl.cornell.edu/~martinez/doc/isca08.pdf

[4] Dynamic Round-Robin Task Scheduling to Reduce Cache Misses for Embedded System, Ken W. Batcher, Robert A. Walker http://www.date-conference.com/proceedings/PAPERS/2008/DATE08/PDFFILES/IP1_04.PDF

[5] ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers, Yoongu Kim, Dongsu Han, Onur Mutlu, Mor Harchol-Balter https://users.ece.cmu.edu/~omutlu/pub/atlas_hpca10.pdf

[6] Parallel Application Memory Scheduling, Eiman Ebrahimi, Rustam Miftakhutdinov, Chris Fallin, Chang Joo Lee, José A. Joao, Onur Mutlu, Yale N. Patt *http://hps.ece.utexas.edu/people/joao/pub/ebrahimi_micro11.pdf*

[7] MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms, Heechul Yun https://pdfs.semanticscholar.org/cb5e/817da1eac5b5b6fe840b6e0c30c89ea751c1.pdf

[8] Linux kernel paravirt ops documentation, http://lxr.free-electrons.com/source/Documentation/virtual/paravirt ops.txt

[9] Xen Paravirt ops, http://wiki.xen.org/wiki/XenParavirtOps

[10] Xen ARM/ARM64 CONFIG PARAVIRT patch series, S. Stabellini, http://lists.xen.org/archives/html/xen-devel/2014-01/msg00851.html

[11] PSCI Linux documentation, http://lxr.free-electrons.com/source/Documentation/devicetree/bindings/arm/psci.txt

[12] Linux process scheduler core code file, http://lxr.free-electrons.com/source/kernel/sched/core.c