Memguard: A Memory Bandwidth Management in Mixed Criticality Virtualized Systems
Memguard KVM Scheduling

Nicolas Dagieu, Alexander Spyridakis, Daniel Raho

Virtual Open Systems
Grenoble - France
Email: {n.dagieu,a.spyridakis,s.raho}@virtualopensystems.com

***Abstract*—Memory bandwidth in standard computing architectures using DRAM (Dynamic Random Access Memory) is one of the most critical parts of the system, mainly responsible for performance degradation in memory bandwidth demanding computations. Memguard is a kernel module designed to solve this issue, created with the goal to schedule memory bandwidth at the CPU (Central Processing Unit) core level and enabling bandwidth regulation functionalities. In this paper we propose a new implementation of Memguard that can also be utilized in mixed-criticality virtualized computing environments. This involves the regulation of memory bandwidth at the guest level and forwarding memory bandwidth needs to the host, where the requests are enforced. Introduced changes include modifications to the CFS (Completely Faire Scheduler) Linux scheduler to work with the modified Memguard kernel module. The original kernel module and the proposed implementation have been tested on an ARMv8 platform to demonstrate the performance and viability of such extensions on future embedded systems. A specific benchmark suite was used to stay as close as possible to common scenarios, measuring the memory bandwidth and the performance gain when scheduling at this level is introduced.**

Keywords–Memguard; memory bandwidth scheduling; CFS; virtualization; KVM/ARM.

I. INTRODUCTION

Nowadays, computers and embedded systems are based on a multi-component architecture, which requires at least a microprocessor, some RAM (Random Access Memory) and other optional peripherals and storage devices. Over the last decades the performance of CPUs (Central Processing Unit) has been increasing steadily but memory, on the other hand, has not followed this trend, as such, computer systems are facing the Memory Wall problem [1][2]. Even if new solutions like HBM (High Bandwidth Memory) or stacked memory are attempts to solve this problem [3], most actual platforms are based on standard DRAM (Dynamic Random Access Memory). 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, as such, memory-bandwidth remains the most critical part of the system, especially on multicore systems (where memory is shared).

The performance bottleneck of memory bandwidth has been extensively studied, and several solutions [4] have been implemented. Most of them are hardware solutions [5][6], at the memory controller level. Few solutions have been proposed at the software level [7][8][9], mostly for server distributed large scale systems [10]. In this paper, memory bandwidth management has been considered as a solution to regulate virtualized environments.

A. *Contribution of this paper*

The existence of memory bottlenecks in actual computing systems is highlighted which results in degraded performance. In a mixed criticality and virtualized system, it also decreases the interactivity (interrupts processing can be slowed down). There is a need to implement a memory bandwidth reservation service to solve this issue.

A solution, called memguard was chosen as the memory bandwidth reservation system. The need to experiment with memory bandwidth regulation features on an embedded system, resulted in porting Memguard from x86 to the ARMv8 architecture and benchmarks demonstrate that even on ARMv8 Memguard can be beneficial as a memory bandwidth reservation system.

In the context of virtualization and embedded mixed-criticality systems, a communication interface between guests and the host was designed which forwards memory bandwidth requests to the Memguard kernel module. This new design also makes use of CFS [11] to sync the memory bandwidth reservation of tasks with the default scheduler of Linux.

An ARMv8 platform [12] was used to run experimental tests, which represents actual high-end embedded computer systems. This platform was selected to demonstrate that actual systems can optimally run mixed-criticality workloads by utilizing a memory bandwidth mechanism with virtualization in mind. Qemu/KVM (Kernel-base Virtual Machine) [13] was used as the virtualization solution to run experiments, as it is the most popular embedded virtualization solution.

B. *Organization of the paper*

The rest of this paper is organized as follows. Section II describes the state of the art of Memguard. Then, Section III lists the problems with virtualized mixed-criticality systems. Methods and benchmarks are explained and detailed in Section IV while initial results are reported in Section V. Possible implementations and solutions are detailed in Section VI and experimental results in Section VII. Finally, Section VIII summarizes the findings and directions for future work.

II. Memguard kernel module

Memguard [14][15] 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 available bandwidth [16] (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 [17]. Maximum memory bandwidth can be achieved in the case where data are located in different banks [18], in other cases the memory bandwidth can be limited and in such cases, Memguard can improve performance by scheduling the memory bandwidth to provide the desired Quality of Service.

A. Memguard architecture

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 request if used with PCM lower than 2.4 and memory reads and writes if PCM upper than 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.

We can summarize the Memguard architecture components as follows:

The global budget manager aka Reclaim manager, which 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 the overall system, a memory budget tank is set to give more bandwidth during the future time slice for tasks that need to access more B/W than required (and some B/W is available in the reclaim manager).

The per core bandwidth regulator, handles the memory management for each core, updating the actual used budget with the PCM (Performance Counter Unit) value, and configuring the PCM to generate an overflow when all memory budget is used. Additionally it requests more bandwidth from the reclaim manager if needed.

B. Memguard functionalities and use-cases

Beside this architecture, Memguard has different features. Its major functionality is the bandwidth limiting management, allowing users to set a limit (in MB/s, weight or in percent). Another feature is the per-task mode, where it uses task priority as the core's memory weight. The last major feature is the reclaim bandwidth functionality, distributing any leftover bandwidth that was not consumed, enabling the most effective use of memory bandwidth. When not in use, the available bandwidth is equal to the max-bandwidth setting set at start (or updated later).

The simplest use case for 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 application level use is complicated and must be done manually. Memguard usage requires to set the bandwidth manually, thus users must be aware of application B/W needs and on which core they are being executed.

III. Memory management in embedded virtualized environments

A. Context of use

In the past most actual embedded systems were designed to handle standalone actions within simple scenarios. Nowadays, more and more autonomous and network related tasks are utilized for embedded systems, as well as multimedia applications and database analysis workloads. At the same time embedded systems are designed with several small micro-controllers to communicate with each other (and/or with a master), resulting in increasing costs and decreasing the MTBF (mean time before failure).

As more demanding usage patterns emerge, most actual multi-chip embedded systems are being replaced by a central unit, performing most of the computation and networking related workloads. This paradigm shift 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, additional resource and safety constraints are created.

Mixed-criticality is essentially the concurrent execution of hard real-time application together with soft real-time or standard applications [13] on the same processing unit. As such, this kind of system needs to provide spatial and temporal isolation of system resources, and in addition proper scheduling between hard and soft real time processes, as well as Quality of Service.

Virtualization is the last component of a future unified embedded system 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 proceeding the image to find particular patterns) could run at the same time as a video playback workload and/or additional critical tasks. Each task can then be executed in a separate VM with all the software needed and the correct amount of processing/memory bandwidth reserved.

B. Requirements

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 VMs are running simultaneously. Each VM handles its own software environment, with a specific priority and memory bandwidth. The priority of the guest is already solved with a priority scheduling mechanism[14], but for memory bandwidth this is not the case and it must be managed to reduce memory performance degredation.

As Memguard was designed to be used at the core level, its use at the Guest level in a virtualized environment involves to utilize Memguard in a different manner and to modify/extend parts of it. For now Memguard is restricted to be used with cores, and can't be linked to a program executed on different cores (scheduler balancing activated). As a result it's impossible to set a memory bandwidth policy on a guest to restrict its bandwidth and allow other guests make use of the remaining available bandwidth.

The primary target of this paper is to suggest a solution which enables a guest to set its memory bandwidth requirements. This would allow to set manually or automatically memory bandwidth in order to use it as efficiently as possible. The second target is to produce a tool which schedules the bandwidth between guests in order to limit some of them while letting others to maximize their usage. The possibility to schedule in that way, would allow to preserve some tasks (guests), making sure that they always have the correct amount of bandwidth. This would create a temporal memory separation and provide even more security between guests.

IV. Methods and Benchmarks

This paper uses a specific benchmark suite, composed by a virtualized environment and various software benchmarks. The experimental environment is based on the Linux 4.3.0 kernel with an open-embedded file-system, while Qemu/KVM is the selected virtualization solution. The actual benchmark platform is a Juno r0 development board with 2 Cortex-A57 and 4 Cortex-A53 cores. 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, which they are based on 4.3.0 Linux and a minimal file-system, 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 of a simple buffer copy-process application which utilizes a large amount of memory bandwidth, while providing a number of processed frames per second. The second program is the well-known Mplayer video suite. Mplayer was chosen to represent multimedia use-case in a mixed-criticality environment and 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 workload in soft real-time constraints. The FFT benchmark is called periodically and allocates a memory buffer for FFT computations, the output is a number of buffers processed per second.

V. Experimental Results

A. Memory bandwidth limitation

The first test (Fig. 1) highlights the memory bandwidth limitation mechanism. For this purpose, four different tasks will be 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).

During the first 120 seconds, Memguard is not loaded. After 220 seconds, Memguard 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

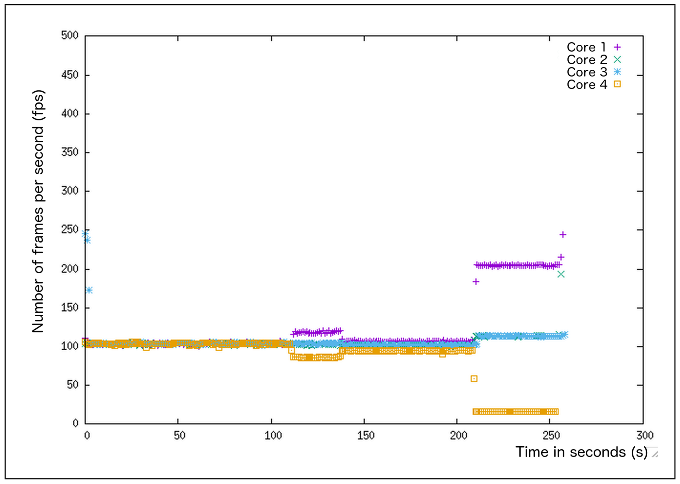

Figure 1. Memory bandwidth reservation on different cores

(tasks 2 and 3 have the same weight). The results are equivalent to the author ones, and show that Memguard is regulating the memory bandwidth of each task.

B. Memory bandwidth reclaim feature

The second experiment (Fig. 2) highlights the reclaim feature, a simple task is used to test if Memguard can release more bandwidth than the applied limit.

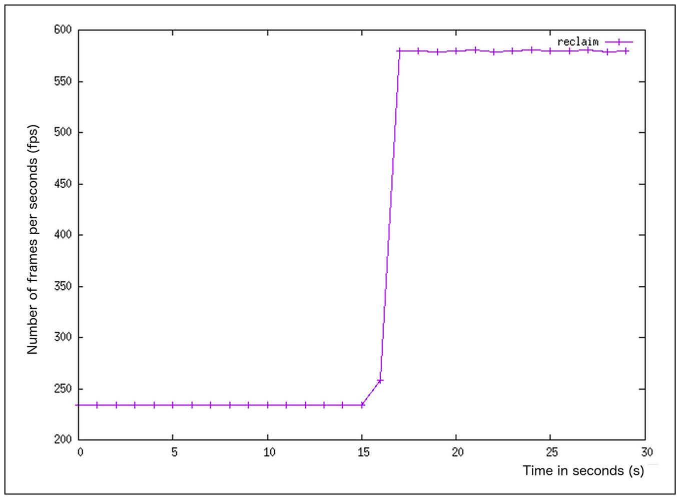

Figure 2. Memory bandwidth reclaim feature

The task is running between 0 to 15 seconds with an under-estimated memory bandwidth limit. The limit was set to 240MB/s, and 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.

C. Memguard's overhead

The CPU overhead of Memguard was measured to understand how to efficiently use Memguard in order to reduce this overhead as much as possible. The experiment uses Memguard with the reclaim feature activated.

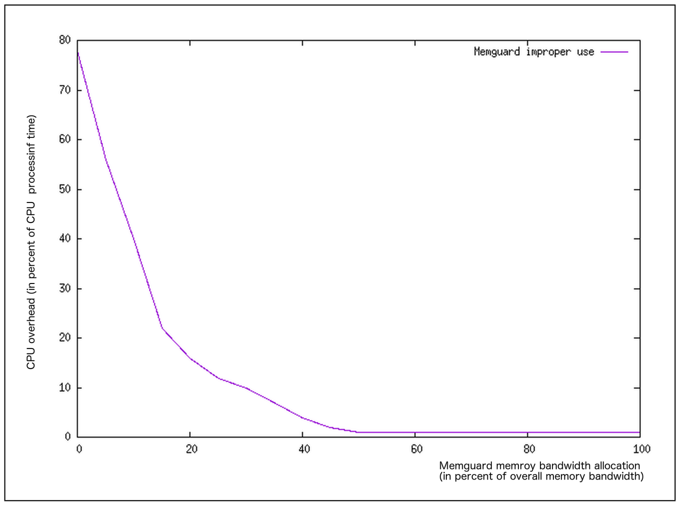

Figure 3. Relation of memory bandwidth allocation and CPU utilization

Depending on the memory reservation policy set by the user, Memguard can introduce a significant overhead to the system, in terms of CPU utilization. Fig. 3 shows the relation between CPU utilization and memory bandwidth allocation of an application. In cases where a large memory bandwidth is allocated, CPU utilization remains low, but when memory bandwidth for an application is underestimated, Memguard produces a large overhead due to the reclaim feature. This feature throttles the core and reallocates a fixed memory bandwidth amount, if the available bandwidth is too high, the produced overhead can reach up to 78 percent of CPU usage.

D. Example of use

In order to understand the way Memguard can be used in real-life computation, an experiment with video playback has been done (Table I).

TABLE I. VIDEO DECODING BENCHMARK

Environment of execution	*Processing time*
Plain linux 2 cores running same video task (mplayer)	Core 1 : 60.318s (decoding time) Core 2 : 60.320s (decoding time)
Memguard with under estimated bandwidth : 20 MB/s on all cores	Core 1 : 313.313s (decoding time) Core 2 : 311.306s (decoding time)
Memguard with correct estimated b/w core 1 (250 20 20 20)	Core 1 : 58.836s (decoding time) Core 2 : 276.001s (decoding time)
Memguard with correct estimated b/w core 1 and best-effort policy activated	Core 1 : 59.881s (decoding time) Core 2 : 95.619s (decoding time)

This experiment highlights the memory-bandwidth reservation capabilities of Memguard. When standalone Linux is executed, 60s (approx.) 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 other cores to use as much as possible the remaining memory-bandwidth (last case).

E. Memory bottleneck

Memory bottleneck conditions are highlighted in the first experiment (Fig. 1). Executed applications are all performing with similar results at the start of the test, where the bandwidth is divided equally to the guests. When Memguard is enabled, task number 1 reaches more than twice of the original memory bandwidth usage. The memory bottleneck is obvious, and if the user wants to prioritize a task due to its criticality, it's impossible to do so without Memguard. As such, memory bandwidth is the limiting parameter of the whole system, introducing increased latency and overall reduced performance.

F. Embedded virtualization problems

Since with QEMU/KVM a virtual machine is just another task to be scheduled by the host, memory bandwidth can have a significant role in performance. Every guests 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, the user must launch each guest on one specific core (or several but, at least one core must be reserved to each guest), reducing the interest of using Linux with KVM, with the load balancing between cores. From the host's viewpoint, VMs are highly dynamic processes with varying workloads that may need different amounts of memory bandwidth. This results in the need for Memguard to be more flexible and be able to regulate on a process granularity instead of cores.

VI. Solution

The aforementioned problem in virtualized environments can be solved using a memory bandwidth scheduler.

A. Architecture and implementation

The solution is based on a new architecture involving all layers of the virtualized computing chain (from the guest to the host kernel), which can deliver messages and regulate the memory bandwidth dynamically.

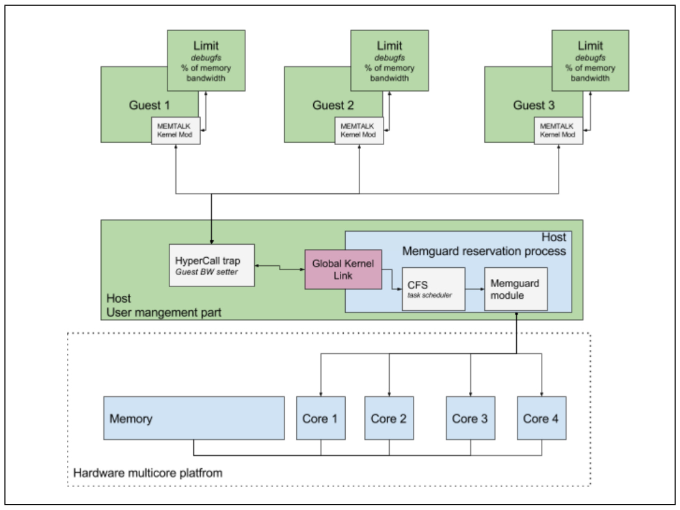

Figure 4. Proposed extensions to Memguard's architecture

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, enabling the user

```
memguard_guest_update(cpu_number){
   if next_task = a_guest_in_the_list
      callback_to_memguard()
}
```

Figure 5. Pseudo code to call memguard from CFS

```
update-budget-sched(int cpu-n, long bw-n){
   convert-bandwidth-to-cache-event()
   set-the-core-budget()
   initialize-the-memguard-statistics()
}
```

Figure 6. Pseudo code to update the per-core budget, called from CFS

to write/read from a simple file to set the needed memory bandwidth value, which also allows to set the bandwidth from other applications (e.g., a local resource manager).

Every call is made with the following:
ID of request: Host is aware that this call is a guest request
Request type: Host is notified if a guest wants to update bandwidth or be removed from the guest reservation process
Value: A general purpose 64bit variable to send information (e.g., bandwidth need: 70 percent of total BW)

The second part is the HyperCall module, which is processed by KVM in the host; every HyperCall is trapped, filtered and processed by the hypervisor. The HyperCall process has been described in detail previously[4], it traps the guest memory bandwidth request and stores it in the GlobalKernelLink.

The GlobalKernelLink is the bridge between the frontend (guest's API) and the backend, which is a hidden mechanism regulating the guest's memory bandwidth. A structure composed of several variables, exported across the host kernel called the GlobalkernelLink is responsible for handling all needed information for the solution, composed of
Memguard_sched_guests: number of guests running with memguard reservation enabled
Memguard_sched_PID: Tab to store guests PID
Memguard_sched_BW: Tab to store Bandwidth need of guests
Memguard_update_bandwidth: pointer to memguard callback function

The third part is the mechanism which regulates the bandwidth, applying the requested memory-bandwidth that was previously 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 main Linux scheduler and is fair between tasks. We implemented a method to call Memguard when guests are running.

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.

CFS is an asynchronous scheduler, no fixed length scheduling clock is used during the scheduling (except the minimum execution time 4ms). Contrariwise Memguard has a fixed length scheduling clock (1ms), this scheduling mechanism difference raises a problem when merging both parts of the proposed solution. In order to address this issue, Memguard was modified to start a new scheduling period when CFS is calling-back Memguard. The resulting solution synchronizes both parts, CFS is unchanged and Memguard's scheduling tick is synchonized with CFS. Changes made in CFS introduce a small slowdown due to the processing time needed to check tasks' membership.

B. Benefits

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 spent when adjusting the value. 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.

C. Mixed criticality enhancement

As discussed previously, the target of the actual paper is to define a virtualized mixed-criticality solution to regulate memory-bandwidth. The solution provides a global answer in order to schedule dynamically memory-bandwidth, as the guest user can either set the needed bandwidth manually or let an automatic system take care of it. This results in the possibility to dynamically adjust the memory bandwidth and to regulate tasks between them, reducing the bandwidth of a task to let others use the remaining.

VII. Experimental results with new Memguard architecture

In this section, experiments and benchmarks are presented in order to highlight how Memguard extensions can be used in a mixed-criticality virtualized system.

The first benchmark (Fig. 7) shows the problem of the memory bottleneck. When two guests are running on the same core, the memory bottleneck limits the memory-bandwidth of both tasks. As in the first experiment (Fig. 1), in a virtualized environment, the bottleneck is the same.

This second experiment (Fig. 8) shows the interest of Memguard solution, at first both tasks are memory-bandwidth scheduled, the first curve (top one) at 70 of the guaranteed bandwidth and second curve (bottom one) at 20 of the memory bandwidth. When Memguard is disabled (around 13 seconds to 20 seconds) the first guest can reach the maximum bandwidth; after 20 seconds the second guest increases its memory-bandwidth reservation, resulting in less bandwidth available for the first guest. The interest is that both guests are running

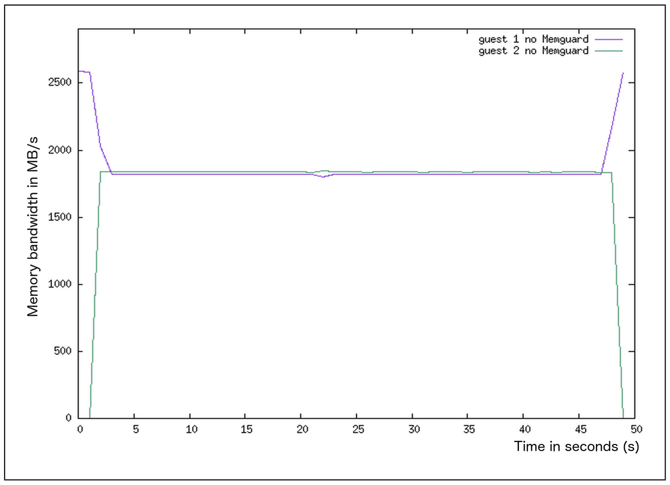

Figure 7. Memory bandwidth degradation between two guests

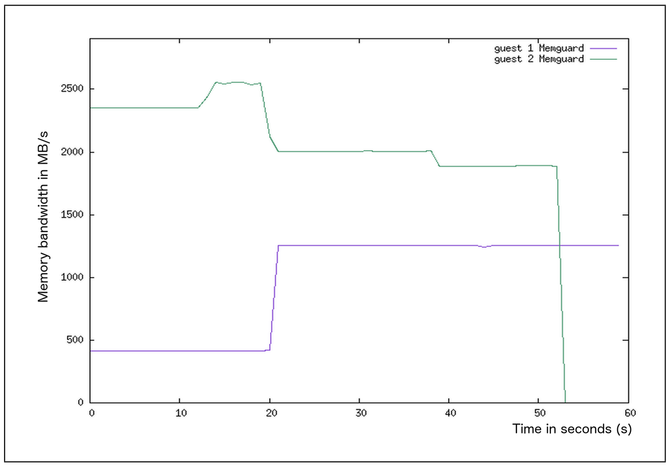

Figure 8. Guest memory bandwidth separation with Memguard

at different memory-bandwidth limits enabling a memory-bandwidth hierarchy between them.

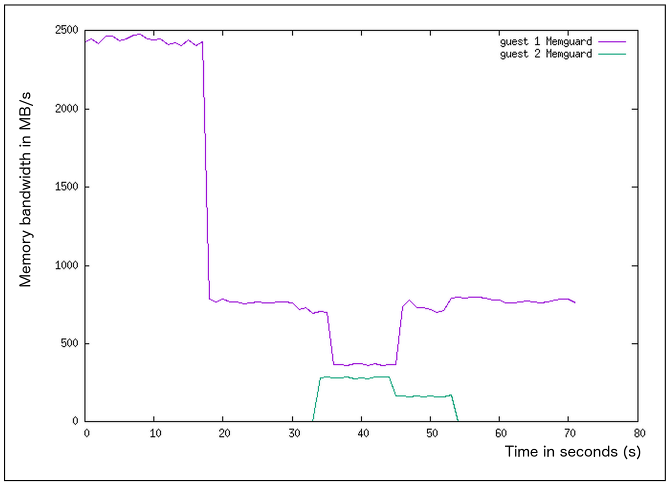

Figure 9. Guest memory bandwidth separation with Memguard (1 core of execution)

The third benchmark (Fig. 9) demonstrates the memory separation between guests. The first guest (top curve) is running unregulated at start, after 17 seconds, a limit is set, and a second guest (bottom curve) is launched after 33 seconds with a limited bandwidth. The reduction of bandwidth is due to the CPU time shared between both guests (running on the same core), when both guests are running, each has a specific memory bandwidth allocation which highlight the memory separation of guests running on the same core.

TABLE II. Video decoding benchmark

Environment of execution	*Processing time*
Plain linux 2 cores running same video task (mplayer)	Guest 1 : 62.112s (decoding time) Guest 2 : 67.968s (decoding time)
Memguard with under estimated bandwidth : 20 MB/s on all cores	Guest 1 : 386.893s (decoding time) Guest 2 : 384.655s (decoding time)
Memguard with correct estimated b/w core 1 (250 20 20 20)	Guest 1 : 57.947s (decoding time) Guest 2 : 312.014s (decoding time)
Memguard with correct estimated b/w core 1 and best-effort policy activated	Guest 1 : 60.911s (decoding time) Guest 2 : 97.665s (decoding time)

The Mplayer benchmark (Table II) was accomplished with an Mplayer decoding process running on two Guests. The results are following the ones done with no Virtualized environment and it demonstrates that the solution is not reducing the performance of the overall system.

TABLE III. FFT ”real time” benchmark

Process used	*Processing speed*
FFT	78 033 sec/frame
FFT Database	148207 sec/frame 1450 MB/s
FFT (high priority) Database (BW reduced)	81014 sec/frame 800 MB/s

The last Benchmark (Table III) involves two guests, one is a camera capture-process VM and the second one is a memory intensive program (equivalent to a database explore task). Overall we can see that with Memguard plus the virtualization extensions, the performance of mixed-criticality use cases can improve significantly due to the additional QoS features implemented. When Memguard is not used, the FFT task has a large slowdown due to a lack of available memory-bandwidth, where if Memguard keeps the database task to a certain level of memory-bandwidth usage (800 MB/s), the FFT task can almost reach its full performance.

VIII. Conclusion and Future Works

In this paper, we highlighted the memory bottleneck on multi-core CPUs and the need to use a memory bandwidth reservation mechanism. In answer Memguard has been tested and extended for its use on ARM platforms. Due to the pervasive nature of virtualization even on embedded systems, Memguard has been adapted to fit this need.

A new architecture forwarding guests' memory requirements to Memguard has been implemented, working with CFS, Memguard has been modified to be synced with the scheduler. In result we obtained a memory reservation service which can throttle memory-bond tasks in favor of high criticality tasks. The actual implementation has several benefits and allows to increase the performance of tasks in mixed-criticality use cases. The overhead is kept to a minimum and the communication mechanism is easy to use from user space or other applications.

The proposed extensions to Memguard are still a proof of concept, and some improvements can be achieved when several guests are running on the same core to improve the tasks' memory separation.

ACKNOWLEDGMENT

This project has received from the European Unions FP7 research and innovation programme, Dreams, under grant agreement N 610640. This work reflects only authors view and the EC is not responsible for any use that may be made of the information it contains

REFERENCES

[1] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, Memory access scheduling. ACM, 2000, vol. 28, no. 2.

[2] D. Field, D. Johnson, D. Mize, and R. Stober, "Scheduling to overcome the multi-core memory bandwidth bottleneck," Hewlett Packard and Platform Computing White Paper, 2007.

[3] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "Ndc: Analyzing the impact of 3d-stacked memory+ logic devices on mapreduce workloads," in Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on. IEEE, 2014, pp. 190–200.

[4] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers," in High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on. IEEE, 2010, pp. 1–12.

[5] K. Srinivasan, "Optimizing Memory Bandwidth in Systems-on-Chip," ESC conference, 2011, http://sonicsinc.com/wp-content/uploads/2012/09/Presentation_Multicore_final.pdf.

[6] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," in Computer Architecture, 2008. ISCA'08. 35th International Symposium on. IEEE, 2008, pp. 39–50.

[7] K. W. Batcher and R. A. Walker, "Dynamic round-robin task scheduling to reduce cache misses for embedded systems," in Proceedings of the conference on Design, automation and test in Europe. ACM, 2008, pp. 260–263.

[8] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt, "Parallel application memory scheduling," in Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture. ACM, 2011, pp. 362–373.

[9] W. Jing, "Performance isolation for mixed criticality real-time system on multicore with xen hypervisor," 2013.

[10] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on. IEEE, 2010, pp. 65–76.

[11] J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, and A. Fedorova, "The linux scheduler: a decade of wasted cores," in Proceedings of the Eleventh European Conference on Computer Systems. ACM, 2016, p. 1.

[12] ARM, "Technology Preview: The ARMv8 Architecture," ARM white paper, https://www.arm.com/files/downloads/ARMv8_white_paper_v5.pdf.

[13] Qumranet, "KVM: Kernel-based Virtualization Driver," White paper, http://www.linuxinsight.com/files/kvm_whitepaper.pdf.

[14] H. Yun, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th. IEEE, 2013, pp. 55–64.

[15] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memory bandwidth management for efficient performance isolation in multi-core platforms," 2013.

[16] H. Yun, "Improving real-time performance on multicore platforms using memguard," 2013.

[17] V. Cuppu, B. Jacob, B. Davis, and T. Mudge, "A performance comparison of contemporary dram architectures," in ACM SIGARCH Computer Architecture News, vol. 27, no. 2. IEEE Computer Society, 1999, pp. 222–233.

[18] IBM, "Understanding DRAM Operation," Application note, https://www.ece.cmu.edu/ ece548/localcpy/dramop.pdf.