Supporting Temporal and Spatial Isolation in a Hypervisor for ARM Multicore Platforms

***Abstract*—This paper addresses the problem of providing spatial and temporal isolation between execution domains in a hypervisor running on an ARM multicore platform. Isolation is achieved by carefully managing the two primary shared hardware resources of today's multicore platforms: the *last-level cache* (LLC) and the *DRAM memory controller*. The XVISOR open-source hypervisor and the ARM Cortex A7 platform have been used as reference systems for the purpose of this work.**

Spatial partitioning on the LLC has been implemented by means of *cache coloring*, which has been tightly integrated with the ARM *virtualization extensions* (ARM-VE) to deal with the memory virtualization capabilities offered by a two-stage *memory management unit* (MMU). Temporal isolation on the DRAM controller has been implemented by realizing a *memory bandwidth reservation* mechanism, which has been combined with the scheduling logic of the hypervisor.

An extensive experimental evaluation has been performed on the popular Raspberry Pi 2 board, showing the effectiveness of the implemented solutions on a case-study composed of multiple Linux domains running state-of-the-art benchmarks.

I. Introduction

The increasing computational power offered by modern computer architectures is pushing software developers to integrate a higher number of functions in the same embedded platform. For instance, in the automotive domain, the number of automated functions increased exponentially in the last years and this trend is expected to continue in the near future, since implementing each function in a dedicated electronic control unit (ECU) is no longer possible for problems related to space, weight, power, and cost (SWaP-C). Although merging applications on the same platform allows mitigating the SWaP-C problem, it creates new difficulties deriving from the reciprocal interference generated by the contention of the shared computational resources (i.e., processors, memory, bus, and I/O devices).

Without proper resource management strategies, such an interference introduces variable delays that may degrade the application performance in a way that is difficult to predict. This issue is particularly relevant when the platform is shared among applications with different level of criticality. For example, in automotive systems the same platform could run critical control tasks, in charge of managing sensible operations (e.g., related to vehicle dynamics, or even the combustion process of the engine), together with other less critical activities related to infotainment (e.g., navigation, wireless connectivity, and control of multimedia streams). In these cases, to guarantee a desired level of performance, critical applications should not be affected by the execution of non-critical ones, or at least the introduced delay should be contained within precise bounds. This feature is also particularly relevant to shield critical applications by misbehaviors or denial-of-service attacks occurring in non-critical applications.

Hardware virtualization achieved by hypervisors established as a de-facto solution to partition the computational resources of a computing platform among different application domains. However, while most hypervisors have been conceived to virtualize primary hardware resources, such as CPUs, memories, and I/O devices, they still lack of a proper management of other architectural resources that are implicitly shared by application domains running upon commercial off-the-shelf (COTS) multicore platforms. Most relevantly, even if domains are assigned separate and dedicated cores, access to memory can generate inter-domain interference. Specifically, one of the main source of interference is due to the contention of cache lines in the *last-level cache* (LLC) (shared by all the cores) [1]: data placed in the LLC by a domain can unpredictably be evicted to accommodate the data of another domain, and viceversa, with a resulting mutual increase of memory access times. Analogously, contention can arise when accessing the main DRAM memory, whose controllers are also known for re-ordering contending access requests to maximize throughput [2], thus further harming the system predictability. Providing hypervisors with isolation capabilities for such shared resources is therefore a prominent challenge in the design of virtualized mixed-criticality systems.

Contribution. This paper presents a hypervisor support for achieving isolation when accessing the LLC and the DRAM in ARM multicore platforms. Spatial partitioning of the LLC is achieved by means of cache coloring, while temporal isolation in accessing the DRAM is achieved through a memory bandwidth reservation mechanism. The support has been implemented within the XVISOR open-source hypervisor [3]. Experimental results to assess the effectiveness of the proposed support are reported for the popular Raspberry Pi 2 platform (equipped with an ARM Cortex A7 processor).

The presented results received considerable attention by the XVISOR community, which is integrating the developed techniques into the official release of the hypervisor.[1] The realized implementation is publicly available as open-source [4]. To the best of our knowledge, this is the first open-source hypervisor solution that supports both the proposed isolation capabilities.

Paper structure. The remainder of this paper is organized as follows. Section II reviews the essential background. Section III and Section IV present the proposed approach for supporting the isolation of the LLC and of the DRAM bandwidth, respectively, discussing their implementation in XVISOR. Section V reports on an experimental study that has been conducted to assess the effectiveness of the realized mechanisms. Section VI discusses the related work. Section VII concludes the paper and illustrates some possible future work.

[1]To date, the developed support for cache coloring is already publicly available in the XVISOR repository at https://github.com/avpatel/xvisor-next.

II. Essential background

To make the paper self-consistent and more accessible, it is fundamental to briefly recall some essential background about the hardware-based technology provided by ARM to implement virtualization (Sec. II-A) and the hypervisor adopted in this work (Sec. II-B).

A. ARM Virtualization Extensions

The ARM *virtualization extensions* (ARM-VE) allows executing multiple operating systems (OSes) upon the same platform, while providing to each of them the illusion of sole ownership of the system. This is accomplished by the introduction of new architectural features (i.e., with an hardware-based support) with respect to standard ARM architectures, which are **(i)** a new privilege mode for the processors, denoted as *hypervisor mode*; **(ii)** an additional layer for memory virtualization, which is managed by a two-stage *memory management unit* (MMU); **(iii)** an enhanced interrupt router; and **(iv)** the *hypervisor call* (HVC) instruction to implement hypercalls. Most relevant to this paper is feature (ii). While still disposing of traditional MMU capabilities, which enable memory virtualization for OSes, ARM-VE platforms include a second translation level for memory addresses. Specifically, the first stage translates virtual addresses to *intermediate physical addresses* (IPA), which will be then translated to the actual *physical addresses* (PA) by the second MMU stage. This feature is particularly relevant for virtualization purposes: the first stage can be configured by an OS without being aware of virtualization, while the second stage can be managed by a hypervisor to allow the coexistence of multiple OSes within the same memory space.

B. The XVISOR Hypervisor

XVISOR is an open-source type-1 hypervisor (i.e., native), which aims at providing a monolithic, light-weight, portable and flexible virtualization solution. It provides high performance and low memory footprint virtualization capabilities for various ARM architecture (with and without virtualization extensions) and for other CPU architectures including x86. The hypervisor allows executing multiple domains (also referred to as virtual machines or guests), where each of them can run a different instance of an OS (e.g., Linux, which is the primary OS supported by XVISOR). Each domain disposes of a set of *virtual* CPUs (VCPU), which are assigned to physical CPUs by the hypervisor scheduler. XVISOR comes with a generic VCPU scheduler that is pluggable with respect to a set scheduling strategies: to date, XVISOR supports fixed-priority scheduling with round-robin tie breaking and rate-monotonic. Virtualization of peripheral devices is achieved via emulation. Pass-trough access is also available for some devices.

III. Supporting LLC Isolation

Several approaches have been employed to achieve predictability in the presence of inter-core interference generated by shared levels of cache; the interested reader can refer to the excellent survey of Gracioli et al. [1] for a detailed literature review. At a high level, three major techniques have been proposed: **(i)** *index-based* cache partitioning, **(ii)** *way-based* cache partitioning, and **(iii)** cache *locking* (or *lockdown*). The first two methods provide a segmentation of the available cache memory by reserving specific cache sets or ways, respectively, to given cores. The latter aims at forbidding the eviction of cache lines by marking them as locked.

Cache coloring. In this work, isolation of the LLC has been achieved with *cache coloring* [5], a well-established software-based technique for index-based cache partitioning. This technique exploits the behavior of set-associative caches, which use part of the PA, denoted as *set index*, to identify the cache line to be used. The key rationale is that, if a domain accesses only PAs whose set indexes match a given pattern, then it will also access a restricted set of cache lines. In fact, cache coloring aims at reserving a subset of the bits composing the set index to identify a cache partition (i.e., a color). Such bits constitute the *color index*. Two colors are said to be contiguous if their color indexes differ by one. Also, an address a is said to match a color c if the bits of a reserved for implementing the coloring are equal to the color index of c. The *color size* (in bits) is defined as 2^k, where k is the position of the first bit (counting from the less significant one) that is used for the color index.

Cache coloring has been selected for three main reasons, which are advocated as particularly relevant in the context of hypervisors. First, because it is *practical*, i.e., cache coloring can be efficiently implemented, does not require specific or custom hardware support, and is *transparent* to the application programmer. Second, because hypervisors typically already manage the allocation of memory areas for the domains to virtualize their address spaces, and hence come with a software design that is prone to integrate coloring techniques at that stage. Third, because it allows exposing a relatively simple configuration interface to the system designer, which is also transparent with respect to the software executing within a domain. In fact, isolation can be controlled by assigning set of colors to domains, thus also seamlessly regulating the amount of cache memory that is reserved to them.

Proposed approach. The approach proposed in this paper has been conceived for ARM-VE platforms and consists in:

(i) *Assigning static colors to domains*. In the off-line configuration phase of the hypervisor, each domain is assigned a set of colors. Such colors are part of the *interface* exported by each domain (useful to compose the system without relying on details of the software running within domains). The overall set of available colors is platform-dependent: an example related to the reference platform used in this work is reported in Section III-B.

(ii) *Redefining the memory allocation strategy of the hypervisor*. When the hypervisor configures the second stage of the MMU (in charge of translating IPAs to PAs), the mapping of second-stage memory pages is performed by placing them into memory areas whose addresses match the colors associated to the domain. An example is illustrated in Figure 1.

In this way, strict partitioning of the LLC among domains running upon different cores can be achieved by just assigning them non-overlapping sets of colors. The resulting effect is that each domain will see a dedicated LLC partition, with the illusion of disposing of a smaller amount of cache memory in which it cannot suffer evictions caused by the software executing in the other domains. Although the reduction of the available cache memory may penalize the system performance (especially when average-case metrics are concerned), it makes each domain more predictable and more robust against misbehaviors or denial-of-service attacks originated in the other domains. Note that this approach also allows controlling isolation when multiple domains run on the same core. For instance, coloring can allow the segmentation of L1 caches, thus permitting the avoidance of evictions during preemptions among domains, or making the interference bounded as a function of the number

of preemptions (and hence, under control of the hypervisor), which can be achieved by assigning the same colors to all the domains running upon the same core. Nevertheless, such designs are argued as less attractive for safety-critical domains, for which the provision of dedicated cores is de-facto a more robust solution.

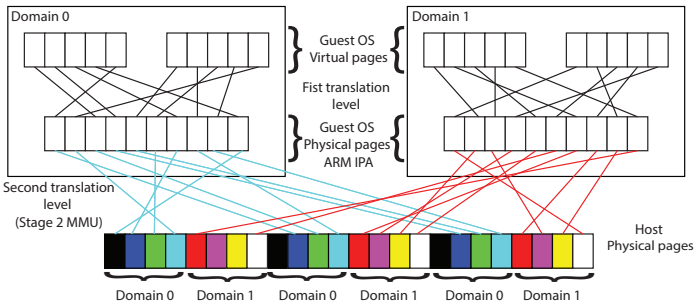

Figure 1. Example of memory mapping with cache coloring for two domains, each assigned four different colors. Their IPA space is allocated to physical memory in a discontinuous manner by the hypervisor to match the corresponding color indexes. For simplicity, the segmentation of the physical memory into colors has a granularity of one page.

A. Implementation

Before proceeding in detailing our implementation, it is necessary to briefly review how XVISOR handles memory virtualization. First of all, it is worth mentioning that XVISOR manages the system configuration by means of *device tree script* (DTS) files for each domain, which is a format borrowed from the Linux community. A DTS file must specify the memory layout that is seen by each domain, which is, in turn, organized into memory *regions*. According to the capabilities of ARM-VE platforms, such regions will be accessed by using IPAs, which can be configured with a specific field in the DTS file. At the system startup, XVISOR scans all the DTS files and reserves an area in physical memory for each region that is configured to represent a portion of RAM memory. A first-fit policy is used at this stage. When a domain attempts to access an IPA for which the Stage 2 page table of the MMU is not yet configured, an exception (also denoted as abort) is raised. Such an exception is handled by XVISOR, which takes care of allocating the memory page that includes the addressed IPA (different page sizes are available, with a default of 2MB). Allocation will take place in the area of physical memory that was reserved for the corresponding region during the startup.

Clearly, this strategy is not compatible with cache coloring, as the allocation of memory pages into physical memory must follow a specific pattern in order to match physical addresses that correspond to a given set of colors.

The first step that has been performed to support coloring consists in introducing a segmentation scheme where each region can be split into *maps*. Furthermore, regions have also been assigned a bitmap tag to represent the colors to which they are associated, which can be configured with a new field provided into the DTS file. Then, at the system startup, each region is split into multiple maps by following their corresponding colors, where each map has size equal to the color size. To reduce fragmentation, if a region is assigned contiguous colors, then they are merged into the same map. That is, a region R_i of size s_i, assigned to x contiguous colors, is split into $m_i = \lceil s_i/(x \cdot C) \rceil$ maps, where C is the color size. Once the segmentation into maps is performed, an area of physical memory is *reserved* for each map. The selection of these areas follows a first-fit strategy and is performed in such a way that their addresses match the corresponding colors. This phase is largely simplified by the fact that each map is aligned to the color size. Finally, when a Stage 2 MMU exception occurs for a given IPA, its corresponding map M and memory page P are identified: then, P is assigned to physical memory within the area reserved for M at the startup. The relative position of pages within maps is preserved in physical memory. This process is illustrated in Figure 2. Since cache coloring introduces a significant segmentation of the memory allocation, the memory page handled at this stage has been configured with a different size with respect to the one previously adopted by XVISOR (with a default of 4KB in place of 2MB). Note that such an allocation takes place only the first time a memory page is accessed, and its impact in terms of run-time overhead can be mitigated with a warm-up procedure at the system startup.

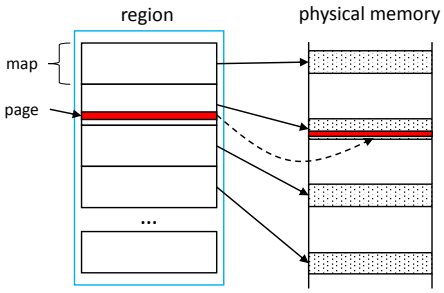

Figure 2. Illustration of the memory allocation of XVISOR for cache coloring. A memory region is split into maps, for which areas in physical memory (marked with dots) are reserved at addresses that are compatible with the assigned colors (note that the placement is fragmented). The dashed arrow represents the mapping of a page to physical memory.

B. Coloring on Raspberry Pi 2

In this work, the popular Raspberry Pi 2 has been used as a reference platform for testing and evaluating our implementation. The Raspberry Pi 2 disposes of a quad-core ARM Cortex-A7 processor with 512KB of L2 cache and 32KB L1 caches. Figure 3 illustrates the address layout of the reference platform with the set indexes for both the level of caches. Memory is accessed with a paging scheme with minimum page size of 4KB, which has been adopted in our implementation to cope with the fragmentation introduced by coloring. As it can be observed from the figure, the physical page identifier within the IPA overlaps with the L1 set index for just one bit, which implies that the L1 can be partitioned into only two colors. Conversely, the overlap with the L2 set index consists into 4 bits: however, if it is intended to only partition the L2 cache, only the bits that do not overlap with the L1 set index can be used, which are three (bits [15 13] in the figure). As a consequence, the available colors for LLC partitioning are eight.

IV. Supporting Memory Bandwidth Reservation

Although the hypervisor realizes isolation of the LLC by means of coloring, mutual interference among domains running upon different cores is still possible due to concurrent access to the DRAM main memory, e.g., in correspondence to L2 cache misses. A simple and practical technique to overcome this issue consists in realizing a memory bandwidth reservation mechanism that limits the number of memory accesses in a given time window, which was probably first proposed as a software-based solution by Yun et al. [6] in the context of non-virtualized multiprocessor systems.

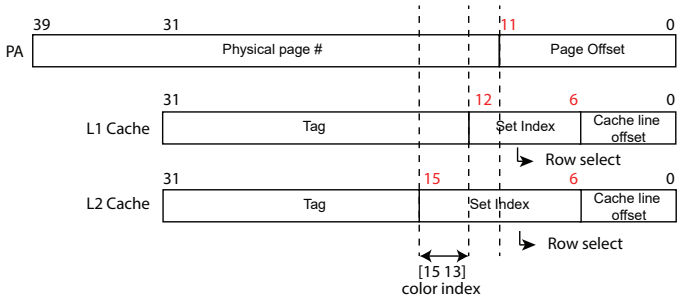

Figure 3. Address layout of ARM Cortex-A7 with 512KB of L2 cache and memory pages with size 4KB. Three bits ([15 13]) are available to implement coloring of the L2.

In this work, memory bandwidth reservation is implemented at the level of the hypervisor, where the latter provides to each domain a *budget* of memory accesses for each VCPU replenished in a *periodic* fashion. Both the budgets and the periods are static configuration parameters of the domains, for which specific fields have been provided into the DTS files. This approach allows reducing the interference incurred by domains due to memory contention, which becomes implicitly limited by the memory budgeting, as well as *independent* of the actual behavior of the software running within the other domains. The following sections present how the proposed approach manages the memory budgets (Sec. IV-A) and how the reservation mechanism has been integrated with the VCPU scheduler of the hypervisor (Sec. IV-B).

A. Managing memory budgets on ARM platforms

ARM processors include logic, denoted as *performance measurement unit* (PMU), which can collect statistics on the operations performed by the cores and the memory system during runtime. For instance, on the ARM Cortex-A7 processor, each core disposes of a PMU with four 32-bit counters, where each of them can track one configurable event among those that are present in a list available in the processor documentation. One of such events, denoted as *data memory access*, allows keeping track of the number of accesses to the DRAM memory, and has been used as the fundamental mechanism to manage memory budgets. This allows implementing a budget accounting mechanism as a pure hardware-based solution, hence without increasing the run-time overhead introduced by the hypervisor. Another relevant feature of PMU counters is their ability to generate an *interrupt* in correspondence to the overflow of the counter register. This feature has been used to implement the event of budget exhaustion, which again can be realized without wasting processor cycles for checking the current budget. Note that PMUs have a scope that is limited to the corresponding core, i.e., each of them must be configured by their core and interrupts will only be signaled to the latter. This fact determines the need for a distributed logic to manage memory budgets.

Overall, the following major steps have been required to use the PMUs to implement memory bandwidth reservation: **(i)** for each core, a counter of the corresponding PMU must be configured to keep track of the data memory access event and to generate an interrupt when a counter overflow occurs; **(ii)** to setup a budget of B memory accesses, the counter register of the selected counter must be configured to $2^{32} - 1 - B$. PMUs can be configured with *move to coprocessor from register* (MCR) instructions. The PMU setup required by our implementation consists in just five MCR instructions. Please refer to the online appendix [4] of this paper for further details.

B. Integration with the VCPU scheduler of XVISOR

The realized implementation allows reserving an individual memory bandwidth for each VCPU of each domain. To enable this feature, it was necessary to integrate the reservation mechanism with the VCPU scheduler of XVISOR. At a high level, the hypervisor has been modified to react to the events of memory budget exhaustion, which cause the suspension of the interested VCPUs, and to interleave different PMU configurations depending on the executing VCPUs. Also, periodic budget recharges and initialization procedures were required to be managed in conjunction to scheduling events. The resulting scheduling logic for a VCPU is illustrated as a state machine in Figure 4, whose most relevant state transitions are numerated from 1 to 9. A new VCPU state, named `RECHARGE`, has been introduced to keep track of a VCPU that is waiting for a memory budget recharge.

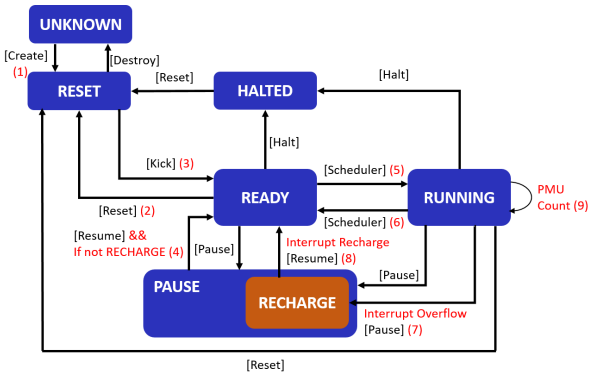

Figure 4. State machine of the XVISOR VCPU scheduler integrated with the memory bandwidth reservation mechanism.

When a VCPU is created (trans. 1), the domain control block is initialized with the memory reservation parameters stored into the corresponding DTS file. Then, when a VCPU starts being eligible for execution (trans. 3), a timer, denoted as *recharging timer*, is configured for raising a periodic interrupt to signal the event of budget replenishment. The recharging timer has been implemented by using the XVISOR timer infrastructure. In the corresponding handler, the timer is restarted to implement the periodic behavior, and is stopped when a VCPU is deactivated (trans. 2). As a function of the adopted scheduling policy (e.g., fixed-priority or round-robin), a VCPU alternates between the `READY` and the `RUNNING` states due to VCPU preemptions. Each time a VCPU is preempted (trans. 6), its current memory budget is first saved in the domain control block by reading the PMU counter register (PMXEVCNTR), then the register is reconfigured with the memory budget that is available for the preempting VCPU (trans. 5). When the memory budget exhausts, the PMU raises an overflow interrupt whose corresponding handler notifies the scheduler to suspend the VCPU, which transits from the `RUNNING` to the `RECHARGE` state (trans. 7). Consequently, the PMU is stopped. The VCPU will not be eligible for execution until the budget will be replenished. The budget is replenished when the recharging timer fires, whose corresponding interrupt notifies the scheduler to move the VCPU from the `RECHARGE` to the `READY` state (trans. 8). Finally, when a VCPU is in the `RUNNING` state, its memory accesses are counted by means of a PMU (trans. 9), hence without invoking any software mechanism.

V. Experimental Results

This section reports on an experimental evaluation that has been conducted to assess the effectiveness of the implemented

solutions. The popular Raspberry Pi 2 board, equipped with a quad-core ARM Cortex-A7 processor, has been used as a reference platform. The evaluation has been focused on the comparison between a standard release of XVISOR (v. 0.2.9) and the modified version of XVISOR (still based on v. 0.2.9) that integrates the proposed isolation capabilities, both serving two domains executing Linux 4.9.18. Each domain has been assigned a dedicated core. The first domain (Domain 0) has been used as a target for the measurements (interfered domain), while the second domain (Domain 1) has been used to generate interference to the first domain (interfering domain).

Domain 0 has been configured to execute a variant of the Isol-Bench benchmark suite [7], which consists in accessing a portion of memory of size N KB for M times. The memory is accessed in a sequential way using a cycle, and only the memory access time is measured. Conversely, to stress the implemented mechanisms and mimic the case of a misbehavior (or a denial-of-service attack), Domain 1 has been configured to continuously access a large portion of memory (10 MB). For both the two versions of XVISOR, the tests were performed **(i)** with only Domain 0 in execution (i.e., without interference), which is denoted as the *solo* case; and **(ii)** with the two domains in execution, which is denoted as the *corun* case.

A. LLC isolation

Four contiguous colors have been assigned to each domain, thus achieving an equal repartition of the L2 cache memory with 256 KB for each domain (see Section III-B). The size N of the memory accessed by Domain 0 has been varied from 8 KB to 512 KB with steps of 8 KB. For each value of N, the benchmark has been executed $M = 1000$ times collecting the running times. A timer has been used to carry out the measurements, and the overhead introduced by the hypervisor (e.g., periodic interrupts) has been filtered out. Cache warm-up has been performed by initializing the used memory areas before starting the actual tests. The maximum running times for $N \in [8, 256]$ KB are reported in Figure 5, while the ones for $N \in [256, 512]$ KB are reported in Figure 6. Average running times and other plots are available in an on-line appendix of this paper [4].

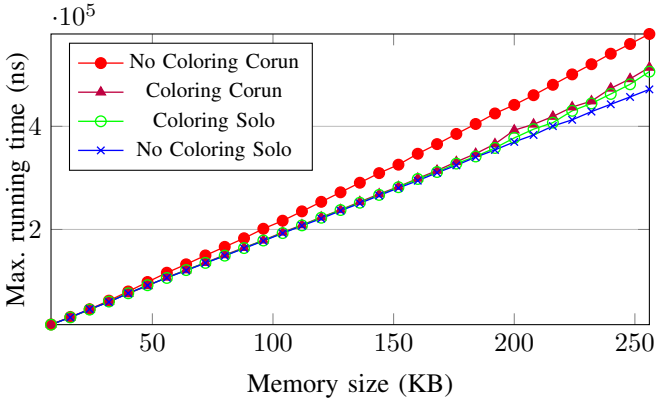

Figure 5. Maximum execution times of the benchmark when accessing a memory area with size 8 to 256 KB.

Note that Figure 5 illustrates the case in which Domain 0 accesses a portion of memory that can be contained *within* its reserved cache partition. As it can be observed from the graph, when cache coloring is not enabled, the difference of maximum running times between the cases with and without interference ('solo' vs. 'corun') increases as N increases, reaching a gap of 23%. Such a difference is the effect of the additional delays incurred by Domain 0 due to cache misses originated by the evictions caused by Domain 1. Conversely, when cache coloring is enabled, it is possible to observe that the maximum execution time is almost the same in both the cases, thus confirming the effectiveness of the implemented isolation mechanism. The small difference (lower than 3%) for $N > 180$ due to the intra-core interference caused by conflicts of cache set indexes.

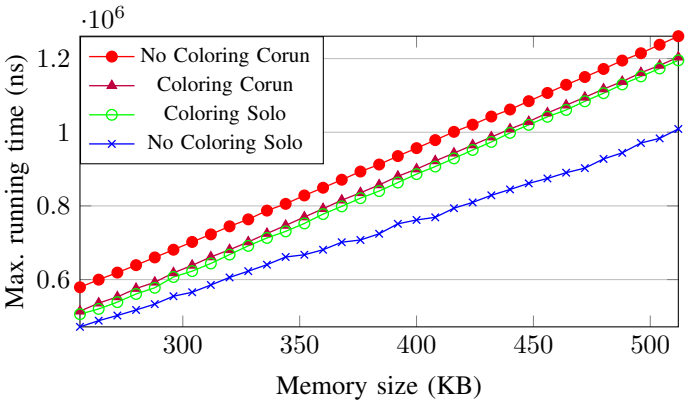

Figure 6. Maximum execution times of the benchmark when accessing a memory area with size 256 to 512 KB.

Figure 6 illustrates the case in which Domain 0 accesses a portion of memory that is larger than its reserved cache partition ($N > 256$ KB), but it can still be contained into the LLC ($N \leq 512$ KB). When cache coloring is not enabled, the difference of maximum running times between the cases with and without interference ('solo' vs. 'corun') is consistent, showing a gap up to 28%. Again, when cache coloring is enabled, the difference is lower that 3%. When no interference is present ('solo' case), it is worth observing that the difference between the cases with and without coloring is substantial: this is due to the fact that, independently of the behavior of the second domain, the coloring mechanism reduces the available space in the LLC. However, while representing a drawback for average-case scenarios, this phenomenon does not affect worst-case scenarios in the light of mixed-criticality systems, where another (possibly less critical) domain can continuously generate interference with an intense memory traffic. In fact, without cache coloring, in the worst-case, every cache line can be evicted by a conflicting domain.

B. Memory bandwidth reservation

To test the implemented reservation mechanism, the size N of the memory area accessed by Domain 0 has been varied in the range $[512, 10240]$ KB, with step 512 KB, thus forcing the domain to access the main DRAM memory (note that the size N is larger than the LLC size). The same experimental setup discussed in the previous section has been used, with cache coloring enabled when using our version of XVISOR. Memory bandwidth reservation has been enforced on Domain 1, whose bandwidth has been varied between 4, 20, 40 and 80 MB/s.

Figure 7 reports the maximum running times as a function of N for all the tested scenarios. As it can be observed from the graph, when adopting the standard version of XVISOR ('no res' case), the difference between the cases with and without interference ('solo' vs. 'corun') increases as N increases, with a very large gap up to 48%. Thanks to the implemented reservation mechanism, it is possible to limit the interference generated by Domain 1 (and make it predictable): as it emerges

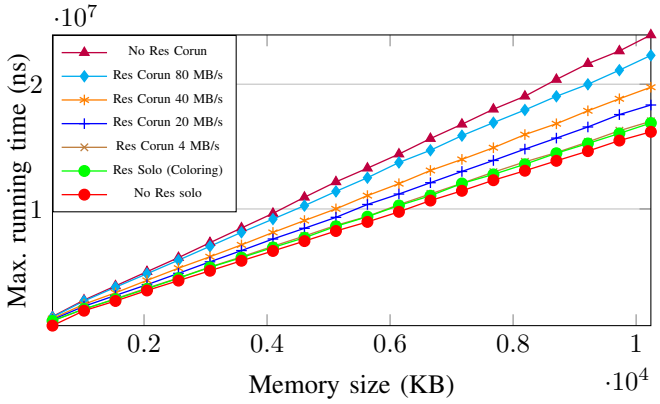

Figure 7. Maximum execution times of the benchmark when accessing a memory area with size 512 to 10240 KB. Different memory bandwidths are assigned to the interfering domain.

from the figure, the measured running times reduce as the bandwidth assigned to Domain 1 decreases, thus confirming the effectiveness of the reservation mechanism.

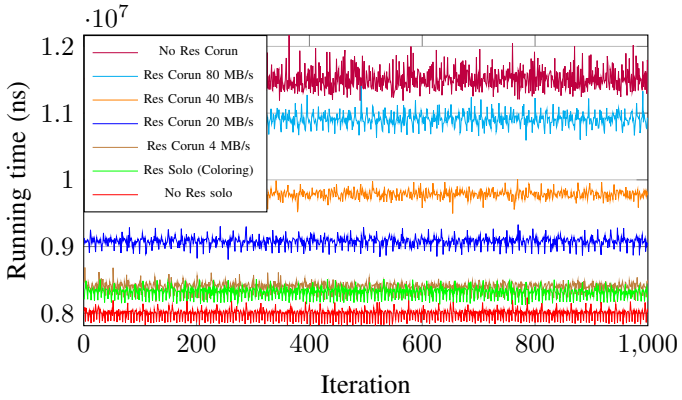

Figure 8. Individual execution times of 1000 iterations for accessing 5120 KB of memory. Different memory bandwidths are assigned to the interfering domain.

To illustrate the variability of the collected measurements, Figure 8 reports the running times of each execution of the benchmark (one iteration) for $N = 5120$ KB. The larger variability has been observed when no bandwidth reservation is applied, and the benefit of the bandwidth reservation mechanism can also be clearly observed by this graph.

VI. Related Work

During the last decade, the problem of architectural resource contention in multiprocessor systems received a lot of attention by multiple research communities. Considering the vast amount of presented results, a detailed literature cannot be reported here due to space limitations. Therefore, this section focuses only on the works that are closer to this paper.

Kim and Rajkumar [8] proposed two techniques for partitioning the LLC in virtualized systems. Their techniques are particularly focused on the knowledge of the tasks running within each domain, which is not the case of our work. The authors also implemented the proposed techniques in KVM. Xu et al. [9] proposed a technique for LLC partitioning by leveraging the Intel's Cache Allocation Technology (CAT), which then does not apply to ARM platforms. Their proposal has been implemented in the Xen hypervisor. The Quest-V separation kernel [10] also supports cache partitioning.

While cache partitioning has been widely investigated, less efforts have been spent on memory bandwidth reservation, especially in virtualized systems. To the best of our knowledge, the first software-based techniques for achieving isolation in accessing the DRAM have been proposed by Yun et al. [2], [6]. Such efforts, however, did not target hypervisors and were implemented in Linux. Recently, Ye et al. [11] proposed another memory reservation mechanism based on novel performance counters that are available on Intel platforms, which has been implemented in the Quest OS. All such works apply the reservation at a core level, while the approach proposed in this paper enforces individual reservations for each domain.

VII. Conclusion and Future Work

This paper addressed the design and the implementation of isolation mechanisms for the last-level cache (LLC) and the DRAM memory controller of an ARM multicore platform, focusing on their integration within XVISOR, an open-source hypervisor. Both the mechanisms have been tightly integrated with the virtualization mechanisms offered by XVISOR, namely memory allocation by means of two-stage memory management units and virtual CPU scheduling. Experimental results on the popular Raspberry Pi 2 platform confirmed the effectiveness of the implemented solutions, illustrating how the running time of a state-of-the-art benchmark is significantly reduced in the presence of isolation mechanisms.

The realized implementation is publicly available as open-source [4] and is going to be integrated in the official release of XVISOR. This work lays the foundation for several challenging future works, including DRAM bank-level partitioning [12], more dynamic cache partitioning techniques, improved memory reservation mechanisms with bandwidth reclaiming, reservation mechanisms for the DMA, and the support for the ARM TrustZone technology.

References

[1] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, "A survey on cache management mechanisms for real-time embedded systems," *ACM Comput. Surv.*, vol. 48, no. 2, Nov. 2015.

[2] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013, pp. 55–64.

[3] A. Patel, M. Daftedar, M. Shalan, and M. W. El-Kharashi, "Embedded hypervisor Xvisor: A comparative analysis," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, March 2015, pp. 682–691.

[4] [Online]. Available: https://github.com/pa007/xvisor-next/tree/cache-coloring_and_memory-reservation

[5] J. Liedtke, H. Hartig, and M. Hohmuth, "Os-controlled cache predictability for real-time systems," in *3rd IEEE Real-Time Technology and Applications Symposium*, Jun 1997, pp. 213–224.

[6] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memory access control in multiprocessor for real-time systems with mixed criticality," in *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*. IEEE, 2012, pp. 299–308.

[7] P. K. Valsan, H. Yun, and F. Farshchi, "Taming non-blocking caches to improve isolation in multicore real-time systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2016.

[8] H. Kim and R. Rajkumar, "Real-time cache management for multi-core virtualization," in *Embedded Software (EMSOFT), 2016 International Conference on*. IEEE, 2016, pp. 1–10.

[9] M. Xu, L. Thi, X. Phan, H.-Y. Choi, and I. Lee, "vCAT: Dynamic cache management using CAT virtualization," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2017 IEEE*. IEEE, 2017, pp. 211–222.

[10] R. West, Y. Li, E. Missimer, and M. Danish, "A virtualized separation kernel for mixed-criticality systems," *ACM Trans. Comput. Syst.*, vol. 34, no. 3, 2016.

[11] Y. Ye, R. West, J. Zhang, and Z. Cheng, "Maracas: A real-time multicore VCPU scheduling framework," in *IEEE Real-Time Systems Symposium (RTSS*, 2016, pp. 179–190.

[12] H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni, "PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms," in *19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2014.

APPENDIX A
EXAMPLE OF PMU CONFIGURATION

In order to provide a taste of how PMUs can be managed to implement the proposed approach, Listing 1 reports sample code extracted from our implementation (the GCC compiler has been adopted).

Function `init_PMU_counter` initializes the PMU counter with identifier `r` for counting the event with identified `evt`. The function first selects the counter with a write operation on Performance Monitors Events Counter Selection Register (PMSELR), and then sets the event by acting on the Performance Monitors Event Type Select Register (PMXEVTYPER). Function `setup_PMU_counter` configures the value `cnt` for the counter with identifier `r`. This is accomplished by first enabling the PMU setting the first and the third bit of the Performance Monitors Control Register (PCMR); the first to enable the counters and the third to use cycle counter divider 64. Then, after selecting the counter by means of the PMSELR, the function configures the actual counter value in the Performance Monitors Selected Event Count Register (PMXEVCNTR). Finally, the overflow interrupt is enabled and the counter is started by acting on the Performance Monitors Count Enable Set Register (PMCNTENSET). All these operations are implemented with a `MCR` instruction, which move to Coprocessor from ARM Register or Registers.

```
1  void init_PMU_counter(u32 r, u32 evt)
2  {
3  // Select event counter in PMSELR
4  asm volatile ("MCR p15, 0, %0, c9, c12, 5\t\n" ::
5  "r"(r));
6
7  // Set the event number in PMXEVTYPER
8  asm volatile ("MCR p15, 0, %0, c9, c13, 1\t\n" ::
9  "r"(evt));
10 }
11
12 void setup_PMU_counter(u32 r, u32 cnt)
13 {
14 // Enable PMU
15 asm volatile ("MCR p15, 0, %0, c9, c12, 0\t\n" ::
16 "r"(0x1 | 0x8));
17
18 // Select event counter in PMSELR
19 asm volatile ("MCR p15, 0, %0, c9, c12, 5\t\n" ::
20 "r"(r));
21
22 // Set PMXEVCNTR with value cnt
23 asm volatile ("MCR p15, 0, %0, c9, c13, 2\t\n" ::
24 "r"(cnt));
25
26 // Enable overflow interrupt
27 asm volatile ("MCR p15, 0, %0, c9, c14, 1\t\n" ::
28 "r"(0x00000001));
29
30 // Enable counting (PMCNTENSET):
31 asm volatile ("MCR p15, 0, %0, c9, c12, 1\t\n" ::
32 "r"(0x8000000f));
33 }
```

Listing 1. Sample code to manage ARM PMU counters.

APPENDIX B
AVERAGE BENCHMARK EXECUTION TIMES

Figures 9, 10, and 11 report the average execution times for the experiments discussed in Section V.

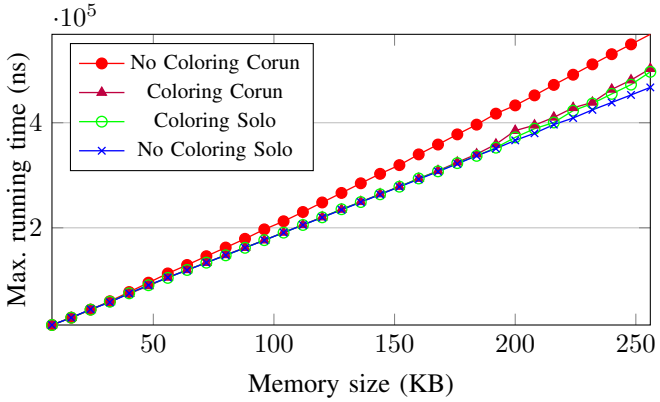

Figure 9. Average execution times of the benchmark when accessing a memory area with size 8 to 256 KB.

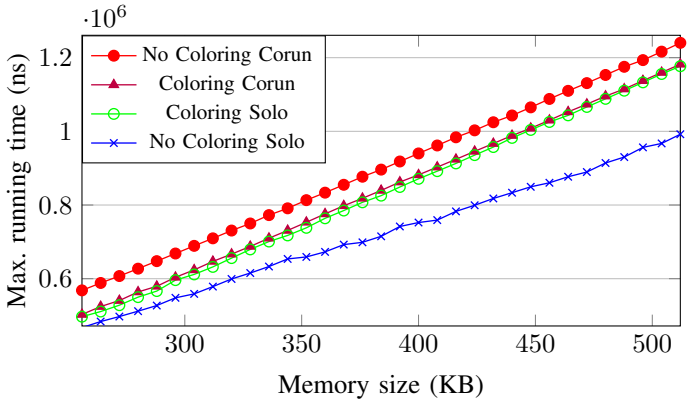

Figure 10. Average execution times of the benchmark when accessing a memory area with size 256 to 512 KB.

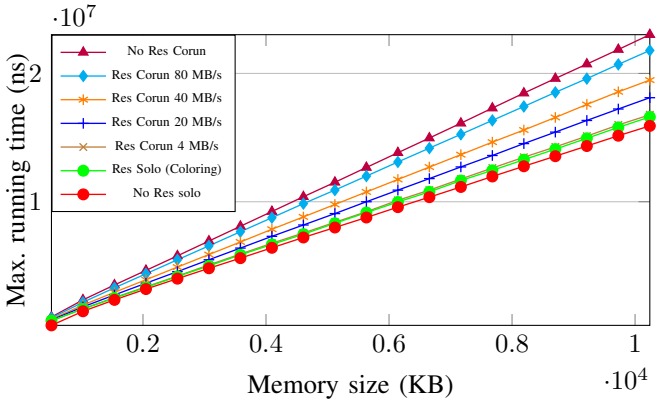

Figure 11. Average execution times of the benchmark when accessing a memory area with size 512 to 10240 KB. Different memory bandwidths are assigned to the interfering domain.