The key role of memory in next-generation embedded systems for military applications

Ignacio Sañudo[1], Paolo Cortimiglia[2], Luca Miccio[1], Marco Solieri[1], Paolo Burgio[1], Christian Di Biagio[2], Franco Felici[2], Giovanni Nuzzo[2], and Marko Bertogna[1]

[1] Università di Modena e Reggio Emilia, Italy
[2] MBDA SpA, Italy
{ignacio.sanudoolmedo, luca.miccio, marco.solieri, paolo.burgio, marko.bertogna}@unimore.it
{paolo.cortimiglia, christian.di-biagio, franco.felici, giovanni.nuzzo}@mbda.it

Abstract. With the increasing use of multi-core platforms in safety-related domains, aircraft system integrators and authorities exhibit a concern about the impact of concurrent access to shared-resources in the Worst-Case Execution Time (WCET). This paper highlights the need for accurate memory-centric scheduling mechanisms for guaranteeing prioritized memory accesses to Real-Time safety-related components of the system. We implemented a software technique called cache coloring that demonstrates that isolation at timing and spatial level can be achieved by managing the lines that can be evicted in the cache. In order to show the effectiveness of this technique, the timing properties of a real application are considered as a use case, this application is made of parallel tasks that show different trade-offs between computation and memory loads.

Keywords: Real-Time systems · Multi-core · Determinism · Memory interference.

1 Introduction

The exponential increase in the embedded software's complexity and the integration of multiple functionalities in the same computing platform have completely changed the way in which vendors designed their solutions. Nowadays, many software manufacturers in the avionic domain are deploying solutions on top of multi-core processors whose cores are all disabled but one [1]. The main reason behind this decision is the lack of methods for guaranteeing core isolation, which in turn is mostly due, especially in the latest generation architectures, to the limited, or absent support of hardware mechanisms for the management of core-shared resources, e.g. the memory hierarchy or the I/O devices [2]. In the most safety-critical systems, isolation is achieved either by integrating software components on a dedicated single-core System-on-Chip (SoC) which is used exclusively for one specific purpose, or by allocating all tasks (i.e. threads or processes) on the same core. As a matter of fact, the development of safety-related software components running on top of a heterogeneous multi-core system is still difficult, since achieving core-isolation and the consequent predictability is a challenging problem.

Indeed, on the certification level, the design, validation and integration of software components with different criticality levels on top of multi-core platforms is still hard. In order to ensure safeness in such process, different tight domain-specific standards have been proposed to provide a reference guidance — e.g. ISO-26262 [3] in the automotive domain, and DO-178C [4] in the avionic one. For the sake of simplicity, but without losing much generality[3]from now on we will focus on the considerations and terminology used in MCP CAST-32 [5] and MCP-CRI (still under development). The purpose of both documents is: to "*identify topics with Multi-Core Processors (MCP) with two active cores that could impact the safety, performance and integrity of the software for a single airborne system executing on MCPs*". Both documents also identify in the *temporal and spatial partitioning* the key requirement for the deployment of software components with different Development Assurance Level (DAL) in the same sub-system. In this way, a software component with a low DAL can be prevented from propagating it to a higher-DAL component, thus reducing design and production costs, by simply allocating the two in different partitions. Similarly, temporal and spatial partitioning mandates that variation in a task's computation time and memory usage, respectively, does not interfere with those another one's.

At system level, software components can be *interfered* in different ways. For example, one application can run for longer a time, thus increasing the execution time of another application and potentially causing execution starvation. Also, an application can obtain and not release a shared resource, which may block another application that needs it, or cause even more severe pathologies like deadlocks or livelocks. Now, these interferences dramatically affects the estimation of the Worst-Case Execution time (WCET). Considering for instance the pervasiveness of image processing or neural networks components, it is easy to see that applications are becoming more memory-intensive. This accordingly shifted the concerns of authorities and industries towards the need of understanding and taming the dominating part of WCET estimation — the access to shared resources, such as the memory subsystems.

Specifically, without applying *memory partitioning* techniques, multi-core systems feature DRAM banks and last-level caches that are extremely often accessed, but shared, hence prone to mutual inter-core contention. Safety-critical tasks can therefore be interfered both at time and space level. For instance, a cache line is evicted/replaced, the effective task execution time is impacted by

[3] Safety-critical software guidelines in different domains like ISO-26262 or IEC-61508 [6] are quite similar in many technical aspects.

the need of experiencing a miss event and a data fetch from central memory, which may even double the memory access latency. This leads to quite pessimistic assumptions in the WCET estimation, exacerbating the difference with respect to the average execution time (ACET) thus causing a reduction of the total CPU utilization.

At *cache* level, isolation or partitioning can be provided by using hardware mechanisms like cache locking, available on the old 7th generation of the ARM architecture, or cache partitioning, equipping a high-end server segment of Intel Xeon x86 architecture. The ARMv8 architecture, which powers most of the medium-to-high-performance modern system has instead no hardware assisting technology[4]. Now, the most prominent software technique that allows to implement cache partitioning is called *page coloring* [8] and consists in exploiting the cache mapping function to isolate non-contiguous memory partitions that injectively maps to cache indices partitions. This technique is founded on virtual address translation, which allows the striped allocation to be hidden at application level. An implementation at OS level would suffer both: great expensiveness, due to the sparseness in the OS choices by the embedded software companies; scarce applicability caused by the presence of custom OS. We advocate instead an easy-to-deploy, legacy-friendly *hypervisor* solution which additional enable seamless consolidation of single-core systems — cache coloring is placed below bare-metal applications and OSs. More precisely, we chose a novel extension [9], to the Jailhouse hypervisor [10], an open-source, minimal, partitioning hypervisor designed for real-time and/or safety-critical use cases. The solution has been developed in the context of the EU Horizon 2020 HERCULES research project [11], whose goal is to develop a system stack for a the next-generation, high-performance multi-core real-time systems. We consider a representative system where a real application from military domain runs on top of next-generation heterogeneous embedded platform, namely a Xilinx Zynq UltraScale+ MPSoC.

The remainder of the paper is organized as follows: Section 2.2 presents an analysis of interference patterns caused by memory contention. The reference application is presented in Section 3.1, showing some result of the capabilities of cache coloring to provide isolation at memory level in Section 3.3, finally a concluding discussion is presented in Section 4.

2 Platform Memory Profiling

In this section, we present the experiments conducted to determine the impact on memory latency caused by multiple sources of aggressive memory accesses, so to characterize the amount of interference.

2.1 Reference architecture

We consider a widely adopted modern heterogeneous multiprocessor system on chip (MPSoC), the Xilinx Zynq UltraScale+, mounted on the ZCU102 board.

[4] The recently announced ARMv8.4-A will include the Memory System Resource Partitioning and Monitoring system, whose specification have been just released [7].

This platform is also used in the following to present the cache coloring mechanism. The hardware architecture considered in this work, the Xilinx Ultrascale+, has a 64-bit quad-core ARM Cortex-A53 CPU coupled with programmable logic in a single device, cache-coherent interconnect, and a shared memory system, among other peripheral interfaces. Figure 1 depicts potential contention points

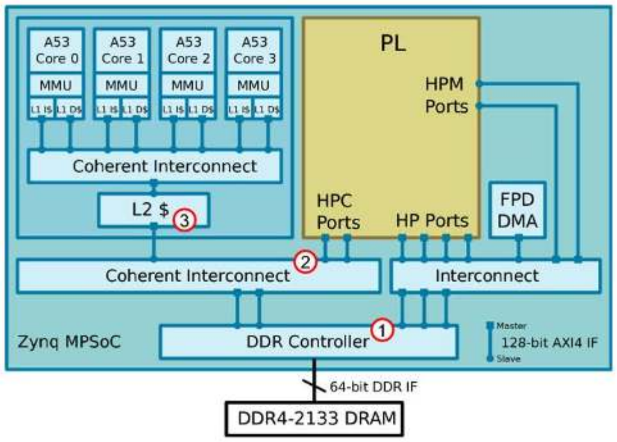

Fig. 1. Xilinx Zynq UltraScale and MPSoC.

that can be found in the platform architecture. As can be observed in Figure 1, the main source of contention between the processors and the programmable logic is found in the DDR memory controller when the DMA transfer is produced (1). Other sources of contention are the cache coherent interconnection (2) and the L2 cache (3).

2.2 Evaluation

Empirical activities have been performed within the HERCULES EU project [11] and are reported in a dedicated deliverable [12]. Images in this section are courtesy of respective authors and all rights belong to their respective owners. Along a first axis, experiments focused on three variants depending on where the measurement and interference run: CPU-CPU, CPU-FPGA and FPGA-CPU. On the CPU, it was employed the LMBench synthetic benchmark [13], which measures the average latency to perform a read memory operation; it was hosted on one of the CPU cores and configured to test both (i) sequential, and (ii) random stride reads. On the FPGA, a custom implementation has been realised and used.

CPU impact on CPU In this experiment is outlined the interference produced at intra core level (Figure 2). In the 'x' axis the Working Set Size (WSS) in bits is shown, while in the 'y' axis is displayed the relative execution time in nanoseconds (ns). As can be observed from the charts, the impact is negligible whilst the WSS fits in the private L1 cache (32 KiB). The interference starts when we increase the working set, which increases by at most 2X. The latency starts to be significant when the WSS reaches the size of the shared L2 cache (1 MiB), in this case, the latency is greatly impacted by the presence of interference, 7X and 16X for sequential and random reads respectively. The latency slowdown gets serious when the WSS does not fit the L2 cache. In this case, the latency converges to a delay proportional to the number of interfering cores. The impact for random interference patterns is less pronounced, however it reaches a 2X and 1.5X for sequential and random reads, respectively.

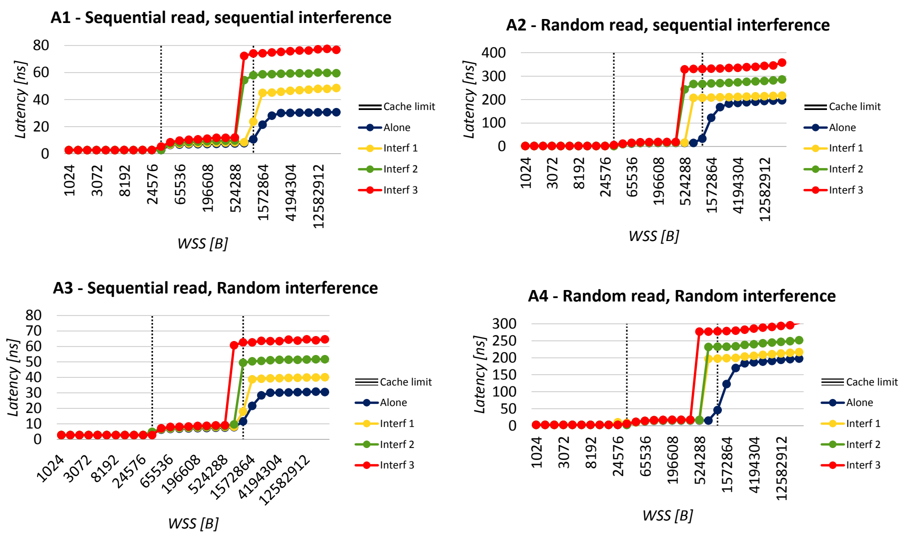

Fig. 2. CPU interference CPU.

FPGA impact on CPU In this experiment, authors characterized the interference experienced by the CPU when there is DMA activity with data required by the FPGA accelerator. The CPU cores perform sequential (D1) and random (D2) reads while FPGA accelerator leads to memory accesses with different bandwidth ratios. Figure 3 shows that, in both cases, memory interference degrades CPU performance. The 'X' axis plots the working set (WSS) in bits, that is, the amount of data accessed by the benchmark, while the 'Y' axis shows the corresponding execution time or latency to perform the operation of the target application in nanoseconds (ns). We see that, when the WSS becomes larger than the L2 cache, it is experienced a heavy performance degradation (almost 3×) if the FPD-DMA module transfers data at maximum bandwidth.

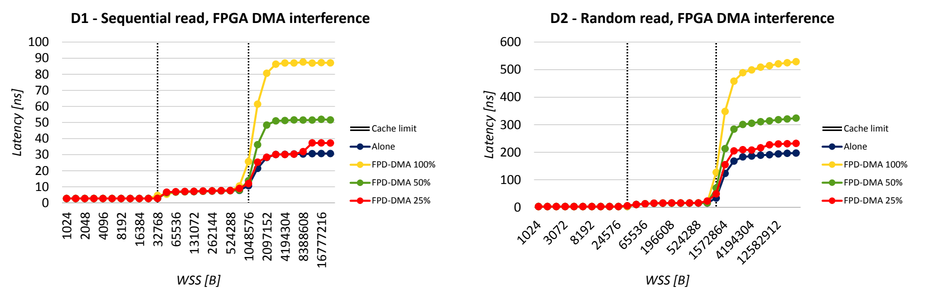

Fig. 3. FPGA interference to CPU.

CPU impact on FPGA The purpose of this experiment was to find out the interference caused by the CPU to the FPGA accelerator. The results of this experiment are presented in the figure 4. In (E1) the FPGA accesses memory via burst transfers (DMA), while the host cores perform sequential reads. In (E2) the CPU cores perform random interference. From these results, it is evident that the performance decreases proportionally to the number of interfering cores, as in the experiments shown above.

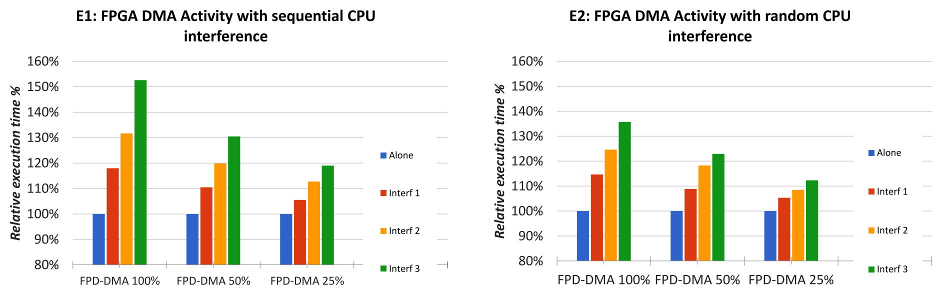

Fig. 4. CPU interference FPGA.

Discussion The experiments presented in this section show that memory-intensive tasks can significantly influence the others task timing behavior. This issue is highly problematic for safety-critical applications and it cannot be mitigated in the discussed platform because the Xilinx Ultrascale+ does not present the capabilities to manage the memory accesses or infer the task priority when memory access are performed. To solve this problem, we will now briefly introduce an effective approach for resource partitioning, that mitigates this effect while providing higher bounds of predictability, this software technique is called cache coloring.

3 Software solution for seamless cache partitioning

Page coloring In order to enforce a deterministic cache hit rate on the most frequently accessed memory pages we leveraged on a software cache partitioning technique called *cache coloring*. By using cache coloring, virtual memory pages are mapped to contiguous memory sets into the physical cache by "coloring" the physical pages. By doing so, tasks are allocated a subset of the cache that cannot be evicted by other concurrent tasks. Page coloring and cache lockdown [14,15] mechanisms make easier the achievement of the goals established in the rationale defined in `MCP_Determinism_8-11` (MCP-CAST 32) that are related to shared-resources.

Hypervisor solution Jailhouse [10] is an open-source, minimal, partitioning hypervisor designed for real-time and/or safety-critical use cases. It partitions hardware resources (e.g. CPUs, memory regions, PCI devices, interrupts) to cells, and assigns them to guests OS's or bare-metal applications called inmates. Jailhouse does not implement any form of resource sharing, scheduling or emulation. Jailhouse design's philosophy focuses on simplicity and openness it features a small code base ranging between 7 and 9.7 KLoC depending on the architecture (ARMv8 and x86, respectively), implying much lower certification costs. It is licensed under GPLv2.

Coloring support The software mechanism proposed for the considered problem was implemented on top of the Jailhouse hypervisor [9] and Xilinx Zynq UltraScale+. This approach has been successfully evaluated in a demanding real-time setup [16].

3.1 Use Case Architecture

Application software for military systems exposes high levels of functionality, integrity and dependability for company missile systems, support facilities and test applications. This software is typically multi-tasking and it is "hard real-time", i.e, the software tasks have to work within specific time constraints, and any minimal failure results in a complete loss of the missile and an unsuccessful engagement, hence it is essential that it is designed and verified to very exacting standards, and it exhibits very high levels of dependability. Dependability is a term that encompasses the needs associated with software systems that required

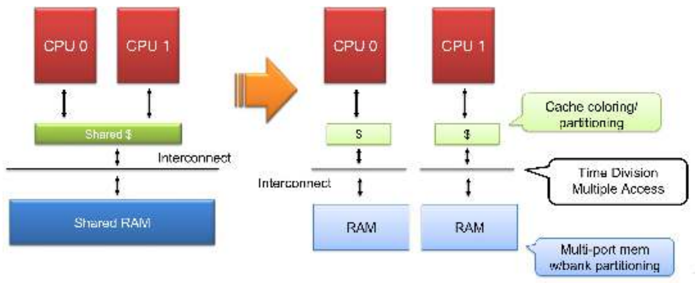
CPU 0
CPU 1
Shared $
Interconnect
Shared RAM
CPU 0
CPU 1
$
$
Interconnect
RAM
RAM
Cache coloring/ partitioning
Time Division Multiple Access
Multi-port mem w/bank partitioning

Fig. 5. Techniques to reduce memory interference.

levels of Safety, Security, Reliability, Integrity, Mission Criticality etc. In this context, most of the products provided by MBDA are mission critical, i.e., a malfunction could result in the equipment under its control (a missile, radar, launcher, etc.) failing to operate correctly and thus failing to meet its operational need. However many of these products now contain features that require the development to meet the other aspects of Software Dependability. Embedded systems have very often real-time constraints dealing with predictability and determinism more than raw speed. Take for example flight software as a tactical application to support missile engagement mission: it runs guide, navigation and control algorithms in a real-time environment supported by multitasking O.S. in order to guarantee timing constraints with the objective to reach acquired target computing flight asset and adjusting its own route. Sensors management, data validity and actuators control must be very strictly related to time schedule and reaction performing the consequent task according to the real environment.

Figure 6 shows the task decomposition of the application that we will consider in this work. The application is composed of different tasks and timers, the figure highlights the parallel execution of different threads ("Active nodes") and the data they exchange (junction nodes). We distinguish between threads executing algorithms, in circular boxes marked with "Algo", and the ones performing I/O activities (triangular boxes, marked as "Drivers") and interacting with external equipment, which are not shown in the diagram. Dashed lines represent movements of data, while solid, "stim" lines are events triggering node execution, that can optionally include also data movement. This application (developed by MBDA) was designed to execute on a single ARM core of the target board, clocked at 800MHz, using a memory pool of 500MiB.

3.2 Use Case Application

For the sake of simplicity we will consider only a subset of the taskset presented in the last section.

Test application The object under test is a single-core bare-metal application composed by two parts:

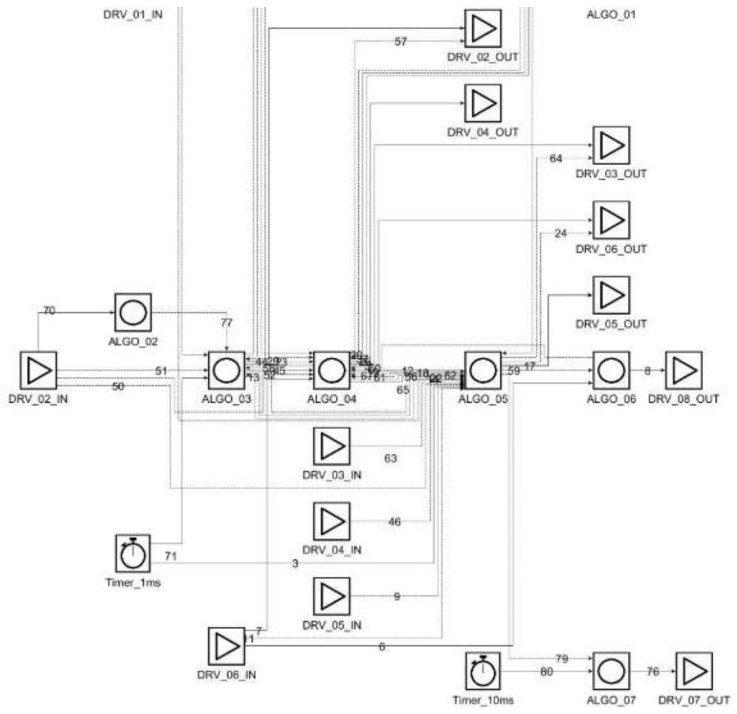

Fig. 6. Target application architecture.

- *Test* A high-criticality, small-sized (512 B), either instruction- or data-bound, routine, whose periodic execution (from 200 us to 3 ms) is triggered through an AXI timer interrupt, in turn raised by the PL.
- *Internal interference* A lower-criticality, medium-sized (32 KiB) periodic (500 us) routine, independently executing on the same core as Test.

Observe that Internal interference is able to evict Tests instructions, or data, from L1 cache, forcing them to be retrieved in L2.

Interference application A quite pessimistic model of memory activity is implemented in External Interference, a second single-core bare-metal application. It repeatedly and endlessly copies a 2 MiB memory segment into another one of equal size. To approximate saturation of the system memory bandwidth, and to maximise L2 cache pollution, we consider two running instances of External Interference.

Evaluation We measured the execution time of Test under three different setup configurations:

- *Baseline* Only the test application is deployed on the original Jailhouse version 0.8.

– *Contention* We just add the interference application to the Baseline configuration.
– *Contention&coloring* We enable the coloring support to the Contention configuration. In particular, we assign:
  - half of the L2 cache to the test application, i.e. 8 colors, or 512 KiB;
  - quarter of the L2 cache to each inmate of the interference application, i.e. 4 colors, or 256 KiB.

In summary, we obtain the 48 configuration combinations that are reported in Table 1, each of which was run 10 K times. Results are shown in Figure 7.

	Test	*Internal*	*External*
Core (id)	3	3	{none, 1-2}
Period (ms)	{0.2, 0.4, 0.6, 0.8, 1, 1.5, 2, 3}	0.5	0
Footprint size (B)	512	32 Ki	2 Mi
Footprint kind	{instr., data}[5]	{instr., data}[6]	data

Table 1. Test configurations summary. 8 and 9 are assumed equal.

3.3 Discussion

We shall first comment on the bare-metal implementation, then compare it first to the hypervisor-hosted variation without coloring, and lastly discuss the colored version.

Baseline Results for this configuration are plotted in the blue box plots of Figure 7. This results represent our golden standard for attainable performances on this platform, and we notice it is indeed quite stable. Variations to the execution period produce effects on the execution time that are negligible both for the data-bound test, where no significant correlation is observable, and for the instruction-bound one, where even changes are hard to be measured. Average and maximum values are respectively around 0.13 and 0.17 μs for the instruction-bound routine, and around 0.23 and 0.61 μs for the data-bound routine, respectively.

Contention The red box plots in Figure 7 outline the dramatic detriment to the execution time predictability caused by the introduction of L2 cache stress activity. Both the average and the worst execution time, for both the instruction- and the data-bound configuration are now one order of magnitude greater than the baseline results. In the 2 ms period, instruction-bound case, for instance, we observe more than a 5.7 factor on the median value, and a 12.3 factor on the maximum one — 0.75 and 2.1 μs, respectively.

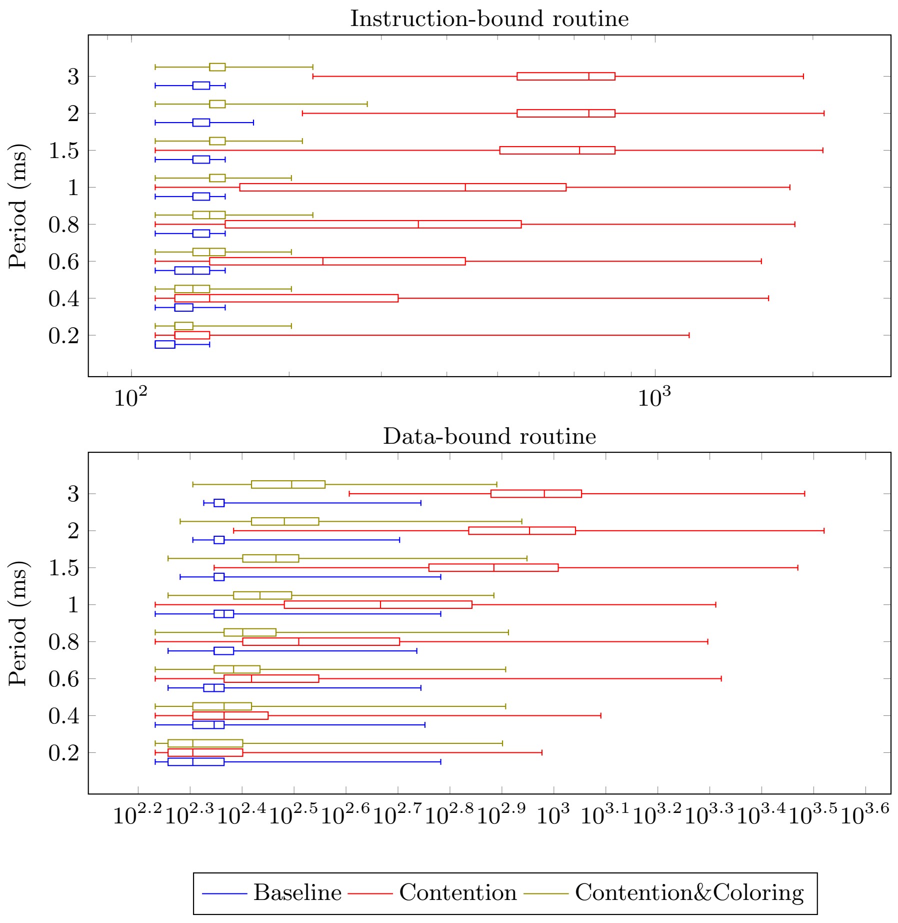

Fig. 7. Benchmark results: execution time (ns). Mixed linear and semi-logarithmic scales).

Contention with coloring Effects of the activation of the cache coloring configuration are visualized in the green box plots of Figure 7. It does not need to be much commented. The execution time is now only slightly higher than the baseline — in the worst case, the overhead amounts respectively at 0.11 and 0.22 μs for the instruction- and data-bound routines.

4 Conclusion

Discussion In this paper, we proposed a system partitioning strategy to overcome timing scalability issues for real-time military application running on heterogeneous embedded SoCs based on multi- and many-core. We introduced an extensive analysis on a realistic application, aimed at pointing out the potential performance bottlenecks. Then, we presented our approach to mitigate their effect on the timing behavior of the application — a partitioning hypervisor enhanced with page coloring support, which avoids the problematic contention on shared caches. The evaluations shows that the solution brings substantial improvements for the system predictability, since the jitter of tasks' execution time is reduced up to one order of magnitude. We believe that the proposed approach represents not just a solid framework for both the system development and the deployment engineering in the real-time military domain, but also a starting point for the application of memory-aware task co-scheduling policies.

Further works We plan to control the interference caused by conflicting memory requests from the FPGA by using a memory-arbitration mechanism inspired by the one proposed by Capodieci et al. in [17]. SiGAMMA is a server-based mechanism that behaves as a memory arbiter between CPU and GPU, balancing the penalties of the concurrent memory accesses performed by the GPU engine. Figure 5 depicts the proposed approach that is based on splitting the system onto isolated partitions, preventing in this way unwanted interference between the multiple cores and the FPGA accelerator.

Acknowledgment

This work was supported by the I-MECH (Intelligent Motion Control Platform for Smart Mechatronic Systems), funded by European Union's Horizon 2020 ECSEL JA 2016 research and innovation program under grant agreement No. 737453.

References

1. Courtney E. Howard. Its time: Avionics need to move to multicore processors, 2018. `https://www.intelligent-aerospace.com/articles/2018/01/it-s-time-avionics-needs-to-move-to-multicore-processors.html`.
2. Ignacio Sañudo, Roberto Cavicchioli, Nicola Capodieci, Paolo Valente, and Marko Bertogna. A survey on shared disk i/o management in virtualized environments under real time constraints. *SIGBED Rev.*, 15(1):57–63, March 2018.
3. ISO 26262-1:2011 - Road vehicles – Functional safety, 2011.
4. EUROCAE WG-12 RTCA SC-205. DO-178C, software considerations in airborne systems and equipment certification, 2011.
5. CAST 32 Superceded by CAST 32A Multi-core Processors. Standard, 2014.
6. Functional safety of electrical/electronic/programmable electronic safety-related systems. Standard, International Organization for Standardization, Geneva, CH, 2000.
7. ARM. Reference Manual Supplement Memory System Resource Partitioning and Monitoring (MPAM), for Armv8-A Documentation. `https://developer.arm.com/docs/ddi0598/latest`.
8. Sparsh Mittal. A survey of techniques for cache partitioning in multicore processors. *ACM Comput. Surv.*, 50(2):27:1–27:39, May 2017.

9. Tomasz Kloda, Marco Solieri, Renato Mancuso, Nicola Capodieci, Paolo Valente, and Marko Bertogna. Deterministic Memory Hierarchy and Virtualization for Modern Multi-Core Embedded Systems. In *25th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS, 2019)*, April 2019.
10. Jan Kiszka and Contributors. Jailhouse, 2018.
11. High-Performance Real time Architectures for Low-Power Embedded (HERCULES) project consortium (EU ID 688860 H2020 ICT 04-2015). The Hercules Project. `https://hercules2020.eu/`.
12. Nicola Capodieci, Roberto Cavicchioli, Pirmin Vogel, Andrea Marongiu, Claudio Scordino, and Paolo Gai. Deliverable 2.2 - Detailed characterization of platforms, 2017. High-Performance Real-time Architectures for Low-Power Embedded (HERCULES) project consortium (EU ID 688860 - H2020 - ICT 04-2015) `https://hercules2020.eu/wp-content/uploads/2017/03/D2.2_Detailed_Characterization_of_Platforms.pdf`.
13. Larry McVoy and Carl Staelin. Lmbench - tools for performance analysis, 2005. version 2 `http://www.bitmover.com/lmbench/`.
14. Marco Caccamo, Marco Cesati, Rodolfo Pellizzoni, Emiliano Betti, Roman Dudko, and Renato Mancuso. Real-time Cache Management Framework for Multi-core Architectures. In *Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, RTAS '13. IEEE Computer Society, 2013.
15. R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S. S. Phatak, R. Pellizzoni, and M. Caccamo. A Real-Time Scratchpad-Centric OS for Multi-Core Embedded Systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2016.
16. Giulio Corradi, Bill Klingler, Emmanuel Puillet, Peter Schillinger, Marko Bertogna, Marco Solieri, and Luca Miccio. Delivering real time and determinism of ZYNQ Ultrascale+ A53 clusters with Coloured Lockdown and Jailhouse hypervisor. Xilinx Inc whitepaper, to appear, November 2018.
17. Nicola Capodieci, Roberto Cavicchioli, Paolo Valente, and Marko Bertogna. Sigamma: server based integrated gpu arbitration mechanism for memory accesses. In *RTNS*, 2017.