Real-Time Requirements for ADAS Platforms Featuring Shared Memory Hierarchies

Nicola Capodieci, Paolo Burgio, Roberto Cavicchioli, Ignacio Sañudo Olmedo, Marco Solieri, and Marko Bertogna
University of Modena and Reggio Emilia, Modena 41121, Italy

Editor’s notes: Automated driving applications combine short computation latencies and high memory throughput requirements. This article makes the case for hardware-based techniques for mitigating the impact of interference on complex memory hierarchies on embedded high-performance platforms.
—Selma Saidi, TU Dortmund

A MANDATORY SET of requirements for automotive applications are those concerning the real-time capabilities of embedded systems. Such requirements imply timing predictability, meaning that all tasks running in the considered system are analyzed with respect to their schedulability, i.e., the formal assessment of whether the timing characteristics for each application in the taskset can be satisfied with the available computing resources. In this domain, each application is generally characterized by a period (minimum time span between two instances of the same task), a worst case execution time (WCET) of the task when executed in a specific computing device and its deadline. A taskset is defined as schedulable if all of its tasks can be mapped to an execution engine (EE) with no deadline misses at runtime. In an automotive scenario, this translates as having tasks such as obstacle detection, path planning, lane centering, pedestrian crash avoidance mitigation, and everything related to rendering in virtual cockpits and infotainment systems to execute within a single embedded platform in a timely manner.

As commercial embedded platforms get more and more complex at every generation, so to ensure a suitable performance-per-watt ratio, satisfying real-time requirements in such complex platforms is not a trivial task. This is mostly due to how memory hierarchies are implemented in modern systems-on-chips (SoCs) in which system memory (e.g., DDR RAM) is shared between all CPU cores and surrounding compute accelerators: these memory clients access memory banks in overlapping time periods causing contention in the memory controller when multiple clients perform transactions.

Memory controllers’ arbitration policies usually favor performance rather than real-time requirements causing significant deterioration in the applications’ response time, consequently introducing pessimism in the WCET estimation.

The effect of memory contention might also occur within the cache hierarchy of the compute engines, as applications running in different computing devices that share the same cache (e.g., two cores within a single CPU island) can cause mutual...
and uncontrolled eviction of useful cache lines. In this contribution, we discuss the magnitude in which memory contention can threaten timing predictability of embedded architectures commonly used in automotive scenarios to then investigate currently available mitigation strategies and propose how future embedded architecture should be designed to properly account for memory interference in the presence of heterogeneous compute engines.

Reference platforms for automotive

State-of-the-art embedded devices are heterogeneous SoCs, composed of a multicore CPU host and one or more compute accelerators. Depicted in Figures 1 and 2, we show two examples of the most commonly used heterogeneous SoCs in automotive-grade platforms.

The NVIDIA Tegra X2 (TX2) host features two CPU islands, a quad-core ARMv8 A57 and a dual-core ARM compatible NVIDIA proprietary CPU design (Denver). Each CPU island shares a 2MiB L2 cache. The accelerator is an integrated GPU (gp10b iGPU) featuring 256 CUDA cores grouped in two streaming multiprocessors (SMs). The SMs are the core components of the EE, which is responsible for massively parallel workloads. Integrated with the iGPU, a high-bandwidth direct memory access (DMA) transfer engine is present [copy engine (CE)]. All these engines and each CPU core might act in parallel not only for computing but also for transferring data. Therefore, the memory subsystem is exposed to several contention points. When CPU and iGPU share the same address space, the coherency mechanisms between CPU and iGPU share a communication channel (point 3 in Figure 1). Moreover, the last-level cache (LLC) in both CPU and iGPU is a source of contention when multiple cores/SMs are running different tasks simultaneously (points 1 and 2 of Figure 1 for A57 and Denver CPU cluster, respectively; point 4 for GPU). Between the caches within the CPU islands, cache coherence mechanisms are also in place. In addition, also the access to the memory buses (points 5–7) and the embedded memory controller (point 8) are contention points for accessing the system RAM [1].

Wang et al. [2] introduce CAVBench, a collection of applications used in the context of autonomous vehicles. Through a detailed experimental evaluation, the authors show that these applications...
are extremely memory bound. Their conclusion is in line with our findings, i.e., memory bandwidth will become the major performance bottleneck and predictability threat in these systems. This is due to the fact that latency-sensitive applications can be severely impacted by memory intensive corunning tasks.

Memory interference characterization

In this section, we report the magnitude of the interference on memory accesses on each of the different contention points detailed in the previous section. We start from Table 1 in which latencies from a single CPU core are measured while all the other computational units (accelerator and remaining CPU cores) are accessing large buffers contiguously. The indicated latencies refer to the time needed to read a single integer value from L2 or RAM: such a value is extracted from a large buffer in a series of sequential memory accesses (abbreviated as seq. in the table) or randomly strided accesses (strided).

Experimenting on both sequential and strided accesses is useful to understand how tasks with different memory access patterns will behave in the presence of interference. The observed CPU core latencies are measured using the `lat mem rd` utility.

The latencies measured in Table 1 for the NVIDIA platform only refer to the Denver CPU island, as the A57 related experiments have been already performed for the TX2 predecessor platform [1]. The impact of memory contention is dramatic in both the tested platforms: the latency increase is almost 2× in all the experimental settings and the worst possible combination is having a CPU application characterized by sparse access to memory being interfered by the DMA engines of the accelerators (GPU or FPGA fabric). Accelerators’ DMA engines create contention only on system RAM and, in the GPU case, on the accelerator’s LLC.

When measuring latencies for the accelerators, heavily memory-bound CPU activity from all the available cores have an impact of latency deterioration on accelerators’ memory accesses that ranges from 1.5× for the FPGA to 1.8× for the GPU (not shown in Table I).

Moving toward the more recently released NVIDIA embedded boards, we performed a set of preliminary experiments regarding memory interference on the NVIDIA Jetson Xavier. The NVIDIA Jetson Xavier features an 8-core ARM v8.2 compliant CPU in which each couple of cores shares a L2 cache, whereas the entire CPU complex is connected to an L3 victim cache. The main memory capabilities have been dramatically improved compared to the older generation Jetson as it now features a theoretical peak bandwidth of 137 GB/s. Such bandwidth, however, is not only consumed by the CPU complex and the iGPU, but also by a collection of ASIC engines as two Deep Learning Accelerators (DLAs) and two programmable vision accelerators (PVAs) are present. With respect to memory latency increase on CPU accesses, Xavier’s memory subsystem configuration is able to bound memory interference to negligible values if one of few more engines are co-running (e.g., from 1.25× to 1.5× in latency increase for sequential or sparse accesses when only the iGPU is interfering). However, if all the compute engines or all the other CPU cores act as interference, memory latency increase ranges from a 5× to 15× depending on the sparsity of memory accesses and the location of the contention point within the memory hierarchy.

Specific to GPUs, there is another contention point to explore and it is related to GPU LLC. As seen in the previous section, the GPU EE in the TX2 is composed of two distinct SMs: this allows the developer to map different GPU tasks to different portions of the execution units. Such units have to share access to GPU L2. A recent contribution presented by Brilli et al. at the latest Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS) challenge detailed a preliminary investi

---

Table 1. Latencies (ns) of CPU access times in NVIDIA Tegra X2 and Xilinx Ultra Scale.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Baseline Sequential access</th>
<th>Baseline Random Strided access</th>
<th>Max Interf. From CPU L2 (seq.)</th>
<th>Max Interf. From CPU L2 (strided)</th>
<th>Max Interf. From CPU to RAM (seq.)</th>
<th>Max Interf. From CPU to RAM (strided)</th>
<th>Max Interf. From Accel. (seq.)</th>
<th>Max Interf. From Accel. (strided)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA TX2</td>
<td>7</td>
<td>13</td>
<td>12</td>
<td>250</td>
<td>13</td>
<td>&gt;350</td>
<td>14</td>
<td>&gt;350</td>
</tr>
<tr>
<td>Xilinx US+ XCU9EG</td>
<td>20</td>
<td>200</td>
<td>72</td>
<td>330</td>
<td>75</td>
<td>&gt;350</td>
<td>89</td>
<td>&gt;350</td>
</tr>
</tbody>
</table>

---


2. [https://www.ecrts.org/forum/viewtopic.php?f=44&t=130&sid=4cb1cd2b62c3b7b76722e1ae80a9625](https://www.ecrts.org/forum/viewtopic.php?f=44&t=130&sid=4cb1cd2b62c3b7b76722e1ae80a9625)
igation on the magnitude of the interference that a task mapped on an SM experience from a cache-bombing task mapped on the remaining GPU portion. Depending on the cache line access stride of the measured task and its memory-to-compute ratio, deterioration of latencies for execution times range from 3 to 6× with respect to a noninterfered baseline. If during the EE computation the GPU initiates a DMA transfer by using the CE, latencies rise from 1.1 to 2.5×.

These measurements call for mandatory mitigation strategies for memory interference: every memory access of a task is susceptible to dramatic latency increase, hence forcing the system engineer to account for significantly larger WCETs during schedulability analysis. We discuss these mitigation strategies in the following sections.

Mitigation strategies for CPU cores

First, we focus on mitigation strategies on CPU cores accessing shared caches. This is an old problem usually solved with hardware support for cache locking or partitioning, in which hardwired mechanisms are able to statically assign a portion of caches to each core in a multicore host. Unfortunately, such mechanisms are not available to the commercial SoCs like the ones we described earlier. As a workaround, cache coloring is used. Cache coloring is a software mechanism that usually acts on the OS/application allocator to force the memory reservations for each application or entire domain to only map to specific portions (called colors) of the shared cache; trivially, the mapping between system memory addresses and cache lines/sets must be known. Modifying the OS allocator or each application within the taskset is not practical nor a scalable solution; for this purpose, a recent contribution [3] shows how it is possible to apply coloring at the hypervisor level, hence proposing a scalable and suitable solution to cache partitioning in a Tegra platform. The solution in [3] manages to keep latencies in CPU L2 free from interference from the other CPU cores, at a negligible overhead (<5%).

Experiments on TX2: We evaluated the cache coloring support [3] on the Tegra X2. A bare-metal application hosted by the Jailhouse hypervisor measures the average latency of a read instruction in a variable-length array. We consider both sequential and random access, and we run the test both on a Denver core and on an A57. Most importantly, we compare three setups: “solo,” where all other cores are off; “int,” where all cores in the test cluster are generating memory stress; “int+col,” where the cache is partitioned so to dedicate 1 MB to the test application.

Results in Figure 3 show how eliminating the contention on L2 usage brings dramatic benefits. With respect to isolation (in blue), interference produces a linearly growing latency (in orange), that is reduced to a mere constant and predictable overhead by cache coloring (in green). The behavior is more evident on A57 rather than Denver cores due to their numerosity in the cluster. The performance gain peaks range from 6.4× for sequential access pattern up to 10.9× for random access.

**Figure 3. Read latency across the memory hierarchy—the impact of CPU interference on L2 and mitigation by coloring.**
Isolating cache partitions is not enough for guaranteeing predictable execution times to latency-sensitive workloads, as contention also occurs at the system memory level. For this problem, the PRedictable Execution Model (PREM) has been proposed [4]. In such a model, system memory is seen as a resource to be acquired and released at specific time intervals (called phases) during the task execution: we have an initial memory load phase, in which data is prefetched into a specific CPU-core exclusively accessible memory (a scratchpad or a portion of a partitioned cache); then follows an execution phase on the prefetched data to then conclude with a memory unload phase that write-backs the modified content to the isolated memory to the shared system memory. By treating memory as a computation engine in which accesses are arbitrated via software greatly simplifies the investigation of suitable system scheduling algorithm (memory-centric scheduling). This is explained in Figure 4; on the left side, we can see how a non-PREM application accesses memory during its flow of execution: accesses are sparse, sporadic, and too fine-grained to be scheduled, whereas the PREM model forces memory accesses to be batched in loading/unloading procedures: memory phases can be therefore thought as jobs to be arbitrated at a software level, making this solution feasible for commonly available embedded systems.

Isolating cache partitions and applying the PREM execution model has a significant drawback: applications have to be refactored to be compliant to the memory and compute phases alternation. To lighten the development effort for implementing such a requirement, compilers support has been proposed [5]. Such compilers are able to exploit the constructs employed by the most commonly used heterogeneous programming frameworks (e.g., CUDA, OpenMP, OpenAcc, and OpenCL) to identify shared data envelopes. Moreover, these programming models natively support explicit/manual host-to-accelerator data transfers, hence being amenable to predictable memory-centric execution models.

Mitigation strategies for accelerators

In the “Memory interference characterization” section, we highlighted that the highest magnitude of interference to CPU memory accesses is caused by heavy memory-bound tasks in execution on the compute accelerators: this is due to the wider buses that usually connects the accelerator to the platform main memory.

To mitigate such a threat in Tegra-based systems, Capodieci et al. [6] propose a software-based solution in which a centralized server intercepts GPU’s DMA transfer and splits them in smaller chunks by giving priority to CPU memory phases, assuming CPU jobs modeled as PREM tasks. Preemption at the level of blocks of GPU threads is leveraged to spin the GPU’s EE, thus preventing interference to CPU memory accesses. A complementary work that protects GPU kernels from CPU interference is presented in [7].

Authors of GPUguard [8] propose a model similar to PREM to be applied to a Tegra GPU: the minimal schedulable entity in the GPU EE is called the warp, that is the minimum execution unit which is a group of 32 threads executing in lockstep; usually, a warp performs both memory and compute instructions: in GPUguard a single warp only performs memory transfers from shared system memory to the internal SM memory or vice versa, whereas a compute warp can only act on data stored in the internal SM memory. Between a memory warp and a compute warp the GPU signals the rest of the system which phase is about to start, allowing a central software module (e.g., a server implemented at hypervisor level) to arbitrate both CPU and GPU memory accesses. Internal SM memory however is typically very small (<100 kB), hence the overhead of SM to arbitration system synchronization might cause significant performance deterioration for GPU tasks with a low
count of computing instructions. To overcome this limitation, a preliminary investigation on applying GPUguard on GPU L2 has been proposed [9].

Arbitration acts by limiting data requests from lower priority clients (CPU cores or accelerators). As shown in [10], memory accesses to system RAM can be throttled at the memory controller level, rather than a purely software-based solution based on scheduling active wait or spin tasks on the lower priority clients.

As far as self-interference on shared GPU caches are concerned, an equivalent version of CPU-cache coloring is proposed in [11] reducing the average variation of execution times up to two orders of magnitude. Feasible hardware mechanisms were investigated in [12], in which GPU L2 cache locking was simulated using GPGPU-sim. Although today’s integrated GPUs feature a rather small SM count, being able to split the GPU into multiple logical units and assign different workloads to each partition is still a desirable feature for both future iGPUs and today’s discrete GPUs, as these latter ones are accessible in recently released automotive-grade platforms (e.g., NVIDIA Drive PX-2 and Pegasus).

**The Lesson Learned** from the previous sections is that memory interference acts at all the levels of the memory hierarchy causing significant deterioration on memory-bound task latencies. Accelerators (such as GPUs and FPGAs) feature a wider bandwidth with respect to CPU cores, thus becoming responsible for the highest system performance deterioration. Most of the mitigation strategies discussed in this article are purely software or involve simulated hardware; although the presented software solutions are applicable to all the commercial SoCs, they come at the cost of a significant effort for the system engineer. Moreover, overheads might be notable in some specific applications. Optimized hardware solutions would significantly relieve the developers’ effort within negligible overheads.

As far as CPU/accelerator latencies on shared caches are concerned, being able to partition caches for each CPU core or accelerator’s compute unit is a must. While we are not aware of similar mechanisms for GPUs, CPU cache locking or partitioning is not new in the field of processor design: Intel CAT (Cache Allocation Technology) in Xeon servers allows the system engineer to reserve cache portions to specific CPU cores for enhanced isolation. AMD equivalent is the recently announced cache quality of service (QoS) support for desktop Zen 2 processors: this feature enables L3 cache allocation enforcement, L3 cache occupancy monitoring, L3 code-data prioritization, and memory bandwidth reservation. We argue that similar features must be also implemented in high-performance low-power ARM Cortex-A-based architectures like the ones we discussed in this article and should account for the larger bandwidth requirements coming from the accelerators; however, those features were only recently announced with ARM MPAM (Memory System Resource Partitioning and Monitoring) for Armv8-A compliant architectures. This latter design allows the system engineer to operate both partitioning and monitoring of all the memory components of each processing element. By exploiting such hardware support, the system engineer should be able to create ad hoc configurations for latency-sensitive scenarios such as automotive. Ideally, controlling memory bandwidth and cache partitions should support dynamic reconfigurations for accommodating real-time scheduling algorithms based on event-driven and dynamic task priorities, as opposed to static scheduling approaches that characterize today’s real-time systems. This might be accomplished with a properly designed virtualized environment able to act as an abstraction layer between the system domains and the hardware register interface to these memory monitoring features.

Moreover, memory controller policies for system RAM access must be designed with real-time requirements in mind: the significant interference caused by sequential reads toward the observed compute unit while it is performing sparse memory access indicate that memory controllers’ arbitration policies favor average performance rather than predictable latencies. While this design choice makes perfect sense for general-purpose platforms, in latency-sensitive applications this approach leads to a significant number of deadline misses. We argue that the system designer should have a clear interface toward the memory controller for preventing occasional data request starvation: this can be accomplished by decorating memory requests to the controller with contextual information such as priority of the request and ID of the request submitter. This information will help the memory controller in implementing hardware mechanisms for predictable response times.

**Acknowledgments**

This work was supported by the Prystine (Programmable Systems for Intelligence in Automobiles), funded by the European Union Horizon 2020 ECSEL research and innovation program under Grant 783190.
References


Nicola Capodieci is an Associate Researcher with the HiPeRT Lab, University of Modena and Reggio Emilia, Modena, Italy. His research interests include bio-inspired artificial intelligence, and more recently languages, architectures and programming models for graphic processing units. Capodieci has an MSc and a PhD in computer engineering from the University of Modena and Reggio Emilia. He is a member of IEEE.

Paolo Burgio has been with HiPeRT Lab, University of Modena, Modena, Italy, since 2014. His research interests include next-generation predictable systems based on heterogeneous many-cores and GP-GPUs, with an eye on compilers, and parallel programming models. Burgio has a PhD in electronics engineering from the University of Bologna, Bologna, Italy, and the University of Southern-Brittany, Lorient, France. He is a member of IEEE.

Roberto Cavicchioli is a Postdoctoral Researcher with the HiPeRT Lab, University of Modena and Reggio Emilia, Modena, Italy. His research interests include parallel computing for heterogeneous systems, optimization algorithms, machine learning, and real-time scheduling. Cavicchioli has a PhD from the University of Modena.

Ignacio Sañudo Olmedo is a Postdoctoral Researcher with the High-Performance Real-Time (HiPeRT) Lab, University of Modena, Modena, Italy. His research interests include safety, reliability in hard real-time systems, and self-driving vehicles. Sañudo Olmedo has a PhD in computer science from the University of Modena.

Marco Solieri is a Postdoctoral Researcher with the HiPeRT lab, University of Modena and Reggio Emilia, Modena, Italy. His recent research interests include operating systems and hypervisors for realtime applications and embedded systems. Solieri has a double PhD in computer science from Université Paris 13, Paris, France, and the Università di Bologna, Bologna, Italy. He is a member of IEEE.

Marko Bertogna is a Full Professor with the University of Modena, Modena, Italy, where he leads the HiPeRT Lab. His research interests include real-time systems for applications in autonomous driving and industrial automation. He is a Senior Member of IEEE.

Direct questions and comments about this article to Nicola Capodieci, University of Modena and Reggio Emilia, 41125 Modena, Italy; nicola.capodieci@unimore.it.