CISTER
Research Center in
Real-Time & Embedded
Computing Systems

Technical Report

Multicore in Real-Time Systems – Temporal Isolation Challenges due to Shared Resources

Ondrej Kotaba

Jan Nowotsch

Michael Paulitsch

Stefan M. Petters

Henrik Theiling

CISTER-TR-140302

Version:

Date: 3/10/2014

Multicore in Real-Time Systems – Temporal Isolation Challenges due to Shared Resources

Ondrej Kotaba, Jan Nowotsch, Michael Paulitsch, Stefan M. Petters, Henrik Theiling

CISTER Research Unit

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail: , , , smp@isep.ipp.pt,

http://www.cister.isep.ipp.pt

Abstract

Recent advent of multicore platforms, along with associated desire for porting of aerospace applications to such platforms, pose a significant challenge for real-time execution. Sharing the on-chip resources introduces low-level temporal effects that impact the execution time determinism by a multitudeof mechanisms. We discuss such mechanisms in this paper and we outline possible approaches to mitigate them or bound their temporal impact. We assume a mixed-criticality real-time environment, as is typical in aerospace applications; it is however also applicable to similar use cases in automotive and industrial control.

Multicore In Real-Time Systems – Temporal Isolation Challenges Due To Shared Resources

Ondřej Kotaba*, Jan Nowotsch†, Michael Paulitsch†, Stefan M. Petters‡ and Henrik Theiling§
*Honeywell International, s.r.o., Aerospace Advanced Technologies, Czech Republic, ondrej.kotaba@honeywell.com
†EADS Innovation Works, Germany, [jan.nowotsch,michael.paulitsch]@eads.net
‡ISEP, Polytechnic Institute of Porto, Portugal, smp@isep.ipp.pt
§SYSGO AG, Germany, hth@sysgo.com

Abstract— **Recent advent of multicore platforms, along with associated desire for porting of aerospace applications to such platforms, pose a significant challenge for real-time execution. Sharing the on-chip resources introduces low-level temporal effects that impact the execution time determinism by a multitude of mechanisms. We discuss such mechanisms in this paper and we outline possible approaches to mitigate them or bound their temporal impact. We assume a mixed-criticality real-time environment, as is typical in aerospace applications; it is however also applicable to similar use cases in automotive and industrial control.**

I. Introduction

In a single-core real-time domain, ensuring the temporal isolation and sufficient execution time determinism is a critical, but nevertheless a well solved task. Many mechanisms to ensure execution time determinism employed therein assume sequentiality of application execution. To account for non-deterministic disturbances, such as interrupts, with carefully analysed temporal impact, additional amount of slack execution time is added. Applications are not in fact executed in parallel, but frequent context switches are performed to provide each application sufficient share of resources per the allocated time unit. Effectively, a Time Division Multiple Access (TDMA) partitioning of entire processor, with exceptions, is achieved. While those exceptions have temporal effects, such as Direct Memory Access (DMA) transfers and associated contention for memory bus, there typically exist well-known solutions for those and their deployment is very limited. For the mentioned case of DMA transfer, it can for example be achieved by replacing the DMA transfer by CPU-controlled approach, often without significant effect on the performance.

Transitioning to multicore, the assumption of sequential execution cannot be safely made any more - the applications actually run in parallel and the access of applications to resources is not exclusive anymore. In multicore domain, the applications compete for resource access, typically arbitrated in a non-explicit manner by the specific hardware implementation. This causes non-deterministic temporal delays to the execution. It is the goal of this paper to systematically analyse the effects and suggest mitigation techniques, where possible. A subset of the effects described in this paper might be present in each particular processor architecture. It is the intent that anybody takes the paper as guideline and inspiration for their analysis of a specific multicore processor to achieve completeness of possible effects.

In some cases, we distinguish between the domain of microcontrollers and General Purpose Processors (GPPs), where different criteria drive the design.

In Section II of this paper we describe the specific mechanisms causing temporal delays in commercial-off-the-shelf (COTS) multicore processors.

In Section III, we mention mechanisms and methods that can be used to maintain the deterministic execution needed for real-time applications. Those either implement mechanisms to prevent or control the access to resources in a deterministic manner, or evaluate the maximum temporal impact of resource sharing. The latter, in most cases, results in extreme pessimism of the worst-case response time (WCRT), but for some effects it is perfectly feasible.

Some of the resources in a typical COTS processor are difficult to arbitrate by other means than by direct hardware implementation. Such mechanisms, however, are nowadays not found in typical high-performance COTS general purpose processors optimized for maximum average-case performance.

In Section IV, we list some promising methods for such cases, requiring further research.

II. Temporal Effects of Resource Sharing

Any typical high-performance multicore COTS processor on the market today will share some or all of the following on-chip resources, such as: System bus, Memory bus and controller, Memories, Caches, Intelligent on-chip resources (e.g. DMA controller, Graphics Processing Unit (GPU), Interrupt controller, etc.), Supporting on-chip logic (e.g. Coherency mechanism, Transaction Look-aside Buffers (TLBs), etc.), I/O devices, Pipeline stags, Logical units, and other devices.

The specific effects were identified, that affect the temporal determinism; see Table I for overview. Detailed description of the respective effects follows.

A. System bus

While definition of this term might vary depending on HW platform provider, we consider here as system bus the interconnect tying together the CPU cores in a multicore system, connecting them to the memory bus or shared caches, as well as other devices. Depending on the HW architecture it

TABLE I

UNDESIRED MECHANISMS AFFECTING THE TEMPORAL DETERMINISM

Shared resource	**Mechanism**
System bus	Contention by multiple cores Contention by other device - IO, DMA, etc. Contention by coherency mechanism traffic
Bridges	Contention by other connected busses
Memory bus and controller	Concurrent access
Memory (DRAM)	Interleaved access by multiple cores causes address set-up delay Delay by memory refresh
Shared cache	Cache line eviction Contention due to concurrent access Coherency: Read delayed due to invalidated entry Coherency: Delay due to contention by coherency mechanism read requested by lower level cache Coherency: Contention by coherency mechanism on this level
Local cache	Coherency: Read delayed due to invalidated entry Coherency: Contention by coherency mechanism read
TLBs	Coherency overhead
Addressable devices	Overhead of locking mechanism accessing the memory I/O Device state altered by other thread/application Interrupt routing overhead Contention on the addressable device - e.g. DMA, Interrupt controller, etc. Synchronous access of other bus by the addressable device (e.g. DMA)
Pipeline stages	Contention by parallel hyperthreads
Logical units	Contention by parallel applications
	Other platform-specific effects, e.g. BIOS Handlers, Automated task migration, Cache stashing, etc.

may also connect to further buses like, Peripheral Component Interconnect (PCI) or I/O interconnects. The connection of several buses requires the deployment of interconnect bridges.

The contention here is driven by a usually fine grained and – unless explicitly managed – in principle highly varied access pattern to this common resource. One source of these accesses include the different cores, where the accesses are largely a function of cache misses. Depending on the implementation of the cache coherence mechanisms of the core local caches, these might also be routed through the system bus. Asynchronous access through DMA traffic forms a further impact on the predictability of system bus usage. In general the access to the system bus is managed by a HW arbiter, which grants access to the shared resource. The policy employed is quite often based on FIFOs, where contending accesses are served in a round robin fashion.

B. The memory, memory bus and controller

In general a program requires some memory for its execution, either for reading its instructions, writing its results or both. Hence if multiple processor cores are organised in a physically shared memory architecture[1] they interfere on multiple units on the path between the processor core itself and the main memory, e.g. interconnect, shared caches and memory controller. The common issue for all of these units is concurrent access by multiple processor cores. Interleaved accesses and refresh delays are specific issues for main memory in general and dynamic RAM respectively.

Concurrent access by multiple cores to one unit causes additional delays, depending on the units capabilities for parallel service. For instance, if a Network-on-Chip (NoC) has enough channels to serve all of the connected processor cores concurrently, accesses might not be an issue. In either case an arbitration policy is required to decide which requesting core to serve first, if multiple requests are outstanding. Depending on the selected arbitration policy the influence on determinism is more or less problematic. For example TDMA arbitration is not an issue for determinism since maximal latency can be guaranteed. Other policies such as round robin can possibly be used in an deterministic manner, depending on how fine grained accesses to the resource can be controlled. Policies that allow starvation of processor cores are impossible to be analysed since no maximal latencies can be defined.

The problem of interleaved accesses to main memory might cause additional delays for cores if they operate on different memory pages, forcing the controller to continuously open/close new pages. Depending on the number of parallel open pages this might be a more or less critical problem. In terms of determinism one would need to account accesses with the theoretical maximum delay, as long as an exact knowledge of which accesses might collide at the memory controller is not available.

Additional delays due to memory refresh cycles are an issue for dynamic memory only. They are not a new topic and also needed to be considered for single core processors in the past. Hence we do not discuss them any further.

C. The cache

Due to the huge overhead of accessing memory on modern architectures, CPUs use cache memories to temporarily store data in faster, yet somewhat smaller memories, for quick access. Today, multi-layer cache hierarchies are well-established where for each additional layer, size and access time increase. In multicore architectures the first level (L1) cache is typically local to each core, i.e. not shared. Of course, the main memory is shared among all cores. Depending on the architecture, intermediate levels (e.g. L2) may be core-local or shared.

For cores to communicate via shared memory, cache contents should be kept coherent, i.e. if one core is writing to main memory, this change must be reflected in the other core's caches if the corresponding memory region is used for communication. Such shared memory is a fast way of communication, but such sharing also has an impact in the timing of each core's access to the shared memory region.

[1] compared to virtually shared, physically distributed memory systems

There are several different kinds of effects that shared memory and shared caches have on timing. The most obvious is the simple bottleneck effect of accessing one memory or cache from several cores at the same time. Typically, access will be serialized, i.e., one core will wait for the other to finish access to the shared memory or cache before accessing it by itself, thereby being slowed down compared to a non-concurrent access [1].

An indirect effect of using a shared cache is that by writing to a cache, one core may evict data that another core has previously stored in the cache, because each cache only has a limited size. The caching effect is thus reduced when more and more cores access a shared cache. The next access to the evicted data will thus be slow again, while it would have been quick if no other core had written to the same cache location.

Keeping cache content coherent induces another, potentially significant slowdown to the cache access: if one core writes to a memory location, read accesses from other cores need to take this into account, i.e. cache content of a core's local caches will have to be invalidated by a write operation from another core. Such enforcement of coherency has several effects. There is the direct slowdown because a cache entry has been invalidated, and data has to be updated. This access can, again, lead to contention of the system and/or memory busses due to concurrency from other cores. Also, the busses needed for the coherency protocol itself may be contended if many write operations occur.

In all of these situations, additional to the concrete slowdown when executing the application on the hardware, each of the mentioned effects will also make a static worst-case execution time (WCET) analysis more pessimistic, especially on complex architectures were it is not possible to statistically add an expected slowdown to result of the WCET analysis performed under a single core assumption. On complex architectures, taking into account potential pipeline domino effects (as described in [2]) may make the static WCET analysis very difficult and pessimistic. Since such static WCETs are the basis for high-criticality task scheduling, pessimism has the same effect as concrete slowdown, as the corresponding resource has to be reserved.

D. Logical units, pipeline stages

Modern MPSoC use hyperthreading, where seemingly different cores actually use the same execution units and/or share caches (like instruction caches). This introduces timing effects at the instruction level. This is because one virtual core blocks and delays the execution of another virtual core that it is paired with for the specific resource. Similarly, logical units, co-processors or Graphic Processing Units (GPUs) in a MPSoC can be shared. The sharing of such units leads to possible delays in the execution of different cores accessing the same unit. Depending on the implementation of the scheduling unit responsible for sharing the unit, the delay can be significant or even lead to starving of one core in case the scheduling unit does not implement some level of fairness.

E. Addressable devices

Apart from cores and memories, there may be addressable devices on the same shared interconnecting bus, like I/O devices, interrupt controllers, or DMA controllers. In multicore settings, access to these devices may be more complicated. Assuming that multiple cores have access to a shared addressable device, exclusive access has to be achieved in order to maintain a consistent state of the shared device.

To ensure exclusive access, locking mechanisms have to be used. In single core situations, highest priority (e.g. kernel level execution) is enough to ensure exclusive access. With truly parallel execution, this is no longer true, and typically, *spinlocks* are used to serialise access in device drivers. Spinlocks have an obvious overhead compared to single core execution when they lock, i.e., all but one core will have to wait. Additionally, spinlocks are shared memory data structures, so all the effects of shared memory, coherent caches, etc., apply here, too.

Another type of interference stems from DMA controllers that autonomously access a shared bus. Those act similar to another core with respect to bus usage and contention. This is however not fundamentally a multicore effect.

In multicore systems, interrupt controllers are typically more advanced than in single core systems, as the interrupt controller usually has the ability to map interrupts to different cores, depending on requesting device, preference and priority settings, and possibly load. As a rare example, such advanced interrupt routing techniques may reduce temporal influences on multicore systems compared to single core systems: with multiple cores, interrupts can be routed to other cores, so that temporal influence on a critical task may be minimised.

On the other hand, operating systems for multicore systems running in an Symmetric MultiProcessing (SMP) setting may need to create additional core to core interrupts (door bell interrupts) for managing the system like synchronising TLB entries, if the hardware lacks support for TLB synchronisation, or in order to wake up waiting tasks on other cores. This may again lead to additional temporal influence.

F. Other effects

Depending on the specific platform, there are other effects, some not multicore related, that influence system's temporal behaviour. Such effects might include BIOS handlers and microcode, such as emulation microcode, or automated frequency multiplier setting to minimize power consumption.

Other, new, additional effects might be however included to improve the average case performance, thermal efficiency, et cetera, that are clearly related to multicore. Such effects include, for example, any automated process migration capability, as being currently studied by various platform developers, cache stashing functions, and various others. Any temporal effect of such additional effects must however be analysed on the case-by-case basis.

III. Mitigation of Effects of Resource Sharing

This section outlines possible methods capable of mitigating the undesired effects described in Section II. It is not to be taken as exhaustive list but rather as a discussion on possible approaches to treat them.

A. System bus

So far the most common solution to solve the contention on the system bus in the avionics domain has been through disabling all but one core in the system and avoiding asynchronous DMA and I/O traffic. However, this obviously does not exploit the performance gains offered by multicore chips.

In terms of cache coherence protocols, some platforms provide a separate interconnect for this. However, while decoupling the cache coherence from general system bus usage is helpful, it only reduces the problem, but fundamentally does not solve the issue. The same is true for using different memory banks to parallelise the access to memory. Again, this reduces the problem in terms of magnitude of average interference, but the issue remain.

Some approaches propose to use more deterministic arbitration scheme to access the shared system bus. One example is the use of priorities in bus accesses. However, such arbitration schemes are not to be found in COTS hardware due their potential detrimental impact on average-case execution time. Even when implemented in custom hardware, they need to be complemented by work in the analysis process to demonstrate that the latencies are within certain limits.

Current ongoing work in the ARAMiS project [3], [4] aims to limit the interference of different bus masters on each other, by means of monitoring the number of accesses and denying access (i.e. stalling) when the number of accesses in a certain time window reaches a limit. Also coarse grained windows of access are a conceivable alternative, but so far have not been explored in detail. Recent work within the RECOMP project aims to provide approaches to bound the impact of this contention via the use of measurements to force worst-case scenario [5] or analytical means [6].

B. The memory, memory bus and controller

To deal with concurrency on shared resources we identified multiple different solutions. Beneath the obvious approaches to disable parallel acting cores and assuming the worst-case influence during timing analysis the following alternatives are known: (1) fully parallel design, (2) deterministic arbitration (3) execution models, and (4) resource limitation. The first two approaches are completely hardware dependent. Hence, they can not be added to a given COTS device if they are not already available. As an example Rosen et al. focuses on TDMA bus arbitration policies [7].

Approaches following the third alternative define deterministic schemes of how applications are allowed to use a shared resource. The basic concept of Schranzhofer et al. [8], Pellizzoni et al. [9], and Boniol et al. [10] is to define different application phases of local computation and communication using shared resources. For example Schranzhofer et al. [8] split applications in acquisition, execution and replication phases and compare different schemes, changing the permissions of the phases to communicate via shared resources. In [9] Pellizzoni et al. introduce the PRedictable Execution Model (*PREM*). They split a program into predictable and compatible intervals, requiring all code and data of predictable intervals to be preloaded into core-local caches, further prohibiting system calls and interrupts. Peripheral traffic is only allowed during the execution of predictable intervals. Although their work is based on single core processors, mainly addressing the problem of shared usage of central interconnect by processor core and peripheral devices, its concepts can be adopted for multicore systems. Boniol et al. [10] use comparable approach but are focused on multicore, studying tool support for realisation of such an architecture.

Resource limitation is a mechanism that somehow extends processor time scheduling by additional dimension for the shared resources, such as interconnect. In [11], [12] Bellosa proposes to use hardware performance counters to acquire additional application runtime informations, such as cache hits and misses. This idea has been adopted recently by Yun et al. [13] and Nowotsch et al. [3] for the use in real-time systems. Yun et al. use performance counters to get information on the memory accesses behaviour of threads to separate real-time and non-real-time threads. Their main focus is on scheduling threads such that the influence of limiting non-real-time threads by their memory accesses is minimal. Nowotsch et al. [3] propose to use the additional runtime information for WCET analysis focused on mixed-criticality, safety-critical application domains, as avionics. Since the WCET of most applications is memory bound, they split the accesses of an application depending on their latency, which generally depends on the number of concurrently acting masters on an interconnect. By analysing the worst-case number of accesses they are able to safely assume different delays for memory accesses, which reduces the WCET of an application, compared to naive approaches that assume the worst-case latency for every memory access.

Interleaved accesses by different processor cores and the resulting additional delays can be addressed either by pessimistic assumptions for WCET analysis or by the mapping of applications to memory. Of course pessimistic assumptions result in less tight WCETs which reduce overall system utilisation, since the assumed cases only happen very rarely during normal execution. Hence the second approach seems more attractive. The idea is to map application memory spaces such that they can not interfere, since they for example do not share physical pages in the main memory. The problem here is the granularity of such mappings and the fact that they might be very platform specific.

C. The cache

Ideally, cross-core cache effects would be predictable. There are some attempts to analyse the behaviour [14], sometimes proposing special hardware. [15]. Still, prediction is difficult.

To overcome timing effects introduced by shared caches in a multicore system, the following techniques may be applied.

The easiest, yet seldom feasible, approach is to disable caching. Due to severe slow-down of memory access, it is deemed unrealistic. More fine-grained mechanisms are needed.

To minimise cache coherency influences, many architectures allow cache coherence to be selectively switched off. It may be that weaker memory models are applicable to the software in use, so switching off coherence may be a feasible approach. In some cases, software coherence protocols can be used instead, and will lead to be more predictable behaviour. [16]

Cache coherence problems only occur with shared memory, i.e., when communication among cores is based on shared memory. Instead, other mechanisms may be available from the operating system or even in hardware, i.e., message passing. This will remove any coherence effect, yet requires rewriting algorithms, and may require memory copying or TLB synchronisation among cores, depending on implementation.

There are also techniques to minimise the cache eviction problem. One of them is cache partitioning. Software implementation can be feasible, depending on cache architecture and capabilities of the Memory Management Unit (MMU) [17], [18], or directly in hardware.

D. Logical units, pipeline stages

One example of dealing with hyperthreading is via analysis level, which is applicable for some hyperthreading architectures and when the source code is relatively simple, deterministic, and known. Popular solutions for analysing applications on multicore and multithreaded processors are

1) a joint analysis and
2) trying to analyse applications apart from each other, as previously done for single-core processors.

Joint analysis techniques analyse all tasks against each other to determine overlaps and influences in terms of accesses to shared resources [10], [19]–[21]. Since task sets can be very complex and timing variations within an analysed task set requires to analyse the new task set. Hence joint analysis techniques are of high complexity, requiring significant processing time, or make infeasible assumptions.

Another approach to use hyperthreading architectures is to simple disable one of the virtual cores so that the hardware is not shared any more. This is a simple but effective solution, at least until temporal effects of hyperthreading are better understood.

For shared external units (logical units, co-processors, GPUs), one possible approach is to implement a server in software. This server manages the accesses to the shared resource. Depending on the shared resource, the approach can be quick enough and feasible; one of the possible approaches to such software arbitration is using the time-slot partitioning. Hardware-implemented sharing approaches depend on the hardware available and are very processor-specific.

E. Addressable devices

Problems introduced by addressable devices as stated above can be avoided by software and hardware. First of all, access to such devices may be handled by a driver, i.e. by a software layer. However, in many situations, the impact of a solution on performance is too large.

Modern multicore hardware offers fine-grained access control via Input/Output Memory Management Units (IO-MMU), so for example a DMA controller can be directly accessed by an application, while still the effects on critical memory regions can be controlled.

Another approach is to use the virtualization, such as is typical for the high-end network interface devices. The shared device itself provides multiple parallel logical interfaces and is capable of routing the data to appropriate dedicated cores. Specific analysis and careful application design must be performed in such cases to ensure that assumptions taken by the device manufacturers are met.

One of the problems with arbitrating the access to such devices is the stateful nature of some such devices, often disallowing truly interleaved access. Usage of such devices in the realtime domain is typically limited to a single thread; otherwise the software driver mentioned above can be used.

F. Other effects

Any effects described in the Section II-F must be studied on the case-by-case basis. Typically, the platform developer makes it possible to disable such effects by means of configuration. Cache stashing might improve the overall average case performance, but the non-determinism introduced by its function would need to be precisely quantified to be useful in the multicore domain. Setting of any thermal-related capabilities, with exception of fuse-type protective functions, would probably be chosen fixed or pre-set, with any dynamic behaviour minimized.

IV. Opportunities for further research

Multicore shared resources, however, not only pose issues, but also provide new opportunities when the additional resources are used in novel ways. For example, multi-threading architectures are traditionally used to gain performance. In systems where the worst case and determinism is a major concern, replicated resources like pipelines can be used to preload values into cache for the other pipelines to execute deterministically. Such approaches are very computing architecture specific. Research on automatic approaches to leverage such features can help achieve better WCET.

The MultiProcessor System-on-Chip (MPSoC) devices are considered very complex by certification authorities [22] and contain a lot of complex parts where special approaches, like safety nets, are needed to compensate for the complexity. Despite of this, some key information for fully understanding the design is often still missing and hard to access by the avionics industry due to the protective behaviour of silicon vendors and designers to maintain their competitive advantage. The avionic industry joined forces and works with silicon vendors where it can to compensate this deficiency by better understanding respective MPSoC. One group addressing this is the Multi-Core For Avionics (MCFA) working group [23].

The MCFA has been meeting regularly to address concerns of avionics companies towards leveraging MPSoC devices in future aerospace products.

The methods that require further research before they can be practically implemented can be found in the areas of preventing the non-determinism caused by memory and system bus. As described in III-B, research opportunity lies in providing mechanisms to time-partition the access to the shared memory, either using cooperative scheme (similar to PREM model described therein) or using some sort of TDMA access arbitration on software level. Similar methods could then also be used for addressing the system bus sharing effects, as mentioned in III-A.

V. Conclusion

Many of the problems described in the Section II are well discussed by research community. There is however a lack of a unifying approach resulting in a combination of solutions that would, for a specific platform, solve all the outlined problems.

In the area of microcontrollers, typically designed for worst-case execution performance, many of the problems are solved on hardware level, at the cost of overall performance.

For GPPs, which care typically used in cases when higher performance is needed, several of the problems identified are not sufficiently addressed in hardware. Namely, that includes the effects of sharing the system bus, memory bus, memory controller and the memory itself.

For some specific usage domains, solutions are in quite mature stage. Most of them center around minimizing the use of shared resources, such as memory, and maximizing core-local resource usage, like caches, to avoid interference. Others only use selected cores to avoid contention. This is not always possible and not very efficient. For such cases, we propose that further research is done in the field of software arbitration of resource access. It is also likely that new programming paradigms will be needed in order to take into account the limited possibility of the applications to access such shared resources.

Acknowledgment

This paper has been supported by ARTEMIS JU as part of the RECOMP project under grant agreement number 100202, national contract number 01IS10001 of German Ministry of Research and Education, national contract number of 7H10007 of Czech Ministry of Education, Youth and Sports, and contract number ARTEMIS/0202/2009 of Portuguese Foundation for Science and Technology. We appreciate the fertilizing discussions with RECOMP partners.

References

[1] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele, "Worst-Case Delay Analysis for Memory Interference in Multicore Systems," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2010, pp. 741–746.

[2] C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. M. (Burguière), J. Reineke, B. Triquet, and R. Wilhelm, "Predictability Considerations in the Design of Multi-Core Embedded Systems," in *Proceedings of ERTSS*, 2010, pp. 36–42.

[3] J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, and M. Schmidt, "Monitoring-based shared resource separation for commercial multi-core system-on-chip," *unpublished, in submission to DSN 2013*, 2013.

[4] ARAMiS Project, "Automotive, Railway and Avionics Multicore Systems - ARAMiS," http://www.projekt-aramis.de/.

[5] J. Nowotsch and M. Paulitsch, "Leveraging multi-core computing architectures in avionics," in *Proceedings of the 9th European Dependable Computing Conference*, May 2012, pp. 132–143.

[6] D. Dasari, B. Andersson, S. M. P. Vincent Nelis, A. Easwaran, and J. Lee, "Response time analysis of cots-based multicores considering the contention on the shared memory bus," in *in IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, Nov. 2011, pp. 1068–1075.

[7] J. Rosen, A. Andrei, P. Eles, and Z. Peng, "Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip," *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pp. 49 – 60, 2007.

[8] A. Schranzhofer, J.-j. Chen, and L. Thiele, "Timing predictability on multi-processor systems with shared resources," *Embedded Systems Week - Workshop on Reconciling Performance with Predictability*, 2009.

[9] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for COTS-based embedded systems," *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 269 – 279, 2011.

[10] F. Boniol, H. Cassé, E. Noulard, and C. Pagetti, "Deterministic execution model on cots hardware," *Proceedings of the 25st international conference on Architecture of computing systems (ARCS)*, pp. 98–110, 2012.

[11] F. Bellosa, "Process cruise control: Throttling memory access in a soft real-time environment," University of Erlangen, Tech. Rep., 1997.

[12] ——, "Memory access - the third dimension of scheduling," University of Erlangen, Tech. Rep., 1997.

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

[14] J. Yan and W. Zhang, "WCET analysis for multi-core processors with shared L2 instruction caches," in *IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2008, pp. 80–89.

[15] D. Hardy, T. Piquet, and I. Puaut, "Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches," *30th IEEE Real-Time Systems Symposium*, pp. 68–77, 2009.

[16] W. J. Bolosky, "Software Coherence in Multiprocessor Memory Systems," *Ph.D. Thesis*, 1993.

[17] D. Kaseridis, J. Stuecheli, and L. K. John, "Bank-aware Dynamic Cache Partitioning for Multicore Architectures," in *Proceedings of the International Conference on Parallel Processing*, 2009, pp. 18–25.

[18] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, "Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems," in *Proceedings of the 14th Intl Symp. on High-Performance Computer Architecture (HPCA)*, 2008, pp. 367–378.

[19] Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury, "Timing analysis of concurrent programs running on shared cache multi-cores," *30th Real-Time Systems Symposium*, pp. 638–680, Dec. 2009.

[20] P. Crowley and J.-L. Baer, "Worst-case execution time estimation of hardware-assisted multithreaded processors," *2nd Workshop on Network Processors*, pp. 36–47, 2003.

[21] P. Radojković, S. Girbal, A. Grasset, E. Quiñones, S. Yehia, and F. J. Cazorla, "On the evaluation of the impact of shared resources in multithreaded COTS processors in time-critical environments," *ACM Transactions on Architecture and Code Optimization*, pp. 34:1–34:25, Jan. 2012.

[22] EASA, "Certification memorandum - development assurance of airborne electronic hardware (Chapter 9)," Software & Complex Electronic Hardware section, European Aviation Safety Agency, CM EASA CM - SWCEH - 001 Issue 01, 11th Aug. 2011.

[23] Freescale, "Freescale collaborates with avionics manufacturers to facilitate their certification of systems using multi-core processors: New working group focusing on commercial off-the-shelf multi-core processing used in commercial avionics," Press release, Sept. 2011.