The Impact of Interference due to Resource Contention in Multicore Platform for Safety-critical Avionics Systems

[1]K. Nagalakshmi, [2]N. Gomathi
[1]Hindusthan Institute of Technology, Coimbatore, India. [2]Vel Tech Dr. RR & Dr. SR University, Chennai, India
[1]nagulaxmi@gmail.com, [2]gomathin@veltechuniv.edu.in

Abstract Multicore architectures are increasingly adopted as computing platforms for safety-critical avionics systems because of their superior performance and cost benefits. In the past two decades, the embedded systems research community has devoted significant attention to the impact of interference on execution timing determinism that arises mainly due to resource sharing. The interference issue reaches a new level of rigorousness in the context of a safety-critical platform that makes timing analysis becomes more and more challenging and lead to extremely multifaceted nondeterminism. We identify and assess the major sources of unpredictability of the system behavior and we discuss potential methods to alleviate them or limit their timing impact. For this study, we consider a mixed-criticality safety-critical domain, as is typical in aeronautics industry.

Keywords* — embedded system, *interferences, mixed-criticality, multicore processor, resource sharing, safety-critical system.

I. INTRODUCTION

With the relentless developments of high performance multicore processors, several functionalities with different criticality (i.e., importance) levels are assimilated together and performed concomitantly on a shared platform. Though typical multicore architectures are mostly intended to increase the system performance, the adoption of multicore architectures in safety-critical systems (SCS) have very different demands in terms of safety, reliability, quality of service, predictability and temporal correctness of the system. Most of the SCS are mixed-criticality (MC) systems that consolidate different tasks (workloads) with diverse criticality levels on a single, shared execution platform. Each criticality level reflects a degree of guarantee required against the subsystem's malfunction. The embedded system research community is interested in implementing multicore processors to realize the safety-critical avionics system.

Integrating various software components on a common platform brings many potential benefits to the electronics market, allowing us to schedule a large number of tasks hence maximizing the resource utilization while decreasing the cost and SWaP (Space, Weight, and Power consumption) demands of the system. Resource sharing implies sharing physical resources such as computational cores (i.e., processing elements (PE), buses, caches, main memory (usually a DRAM), and memory controllers between system components. However, using multicore architectures in the SCS imposes several challenges including designing of certifiable multicore architectures, the organization of the system resources and integration of parallel software to the computing industry has to face [1]. Effective utilization of system resources, which is conceived to increase the average performance, needs sharing. On the other hand, resource sharing breaks rigidity in timing analysis and jeopardizes the dependability of the system. A general property of these SCS is that malfunctions may have catastrophic consequences, such as the potential loss of human lives/equipment or severe financial ramifications.

Multicore processors are shaking the basis of traditional timing analysis methods of real-time applications, i.e. traditionally, the worst-case execution time (WCET) can be estimated on each task to calculate the schedulability of the entire system when tasks are executing concurrently. Over the last few decades, this basic notion has been widely accepted by conventional scheduling algorithms; unfortunately, while dealing with modern safety-critical systems, this notion is not true and causes a deficiency of composability. Common physical resources such as caches, global memory, and interconnects are all sources of complex interferences and timing dependencies between concurrent workloads. The

 © 2016, IJREAM All Rights Reserved.

adoption of a multicore processor in SCS creates the necessity to certify that systems operate in the way they are intended to and that consequences of a task's failure are absolutely tackled in a safe manner.

An important requirement of safety-critical applications is the necessity to deliver predictable timing behavior: the temporal correctness of the system should be analyzable during the certification process with a quantitative metric and assured in the implementation phase. Therefore, this requirement of predictability is imperative for many critical domains, including automotive, avionics, military, medical systems, manufacturing, and nuclear power stations. There are several methods to ensure predictability for serial applications. In order to consider unpredictable interferences, such as interrupts, with prudently analyzed timing characteristics, additional execution time (or penalty) is added.

In the multicore platform, a number of tasks may request a shared resource simultaneously, but the resource can only acknowledge one request at a time. A resource arbiter is responsible to allocate the resource bandwidth among the workloads. Consequently, the arbitration logics implemented in some resources (e.g., buses) will delay the request of all but one task, hence retarding the execution of other tasks. This type of resources is usually named bandwidth resources. Conversely, in some cases (e.g., shared caches), one task may alter the state of the common resource such that the co-running tasks incur additional execution latency. This type of shared resources is known as storage resources.

The interferences due to resource contention are fall into one of two camps: inherent interferences and virtual interferences. Virtual interferences create artificial nondeterminism whereas inherent interference eff ects introduce actual nondeterminism. However, both are harmful for temporal behavior of the system. The inherent interference effect is behavior generated by the accesses from the co-running accessors (resource-users) at random intervals. Hence, the running application incurs higher execution delays. In multicore platforms, memory and buses are deemed main sources of inherent interferences. Different accessors access the buses in uncontrolled manners. Main memory and caches are shared among different cores simultaneously. Hence, such interferences might increase the actual execution times of workloads and therefore, inherently, the WCET bounds of those workloads, too.

Virtual interferences are introduced by the inevitable abstraction of the system (i.e., loss of information about system behavior). Even though all the interferences might not ever occur in a tangible way, the investigations cannot verify these effects, as it can merely depend on its inadequate, static information. As an example, if the execution time analysis for task Γ abstracts from the parallel execution of multiple workloads, it has to assume interference by other workload Γ1 whenever Γ generates access request to a shared resource. Loss of information due to an abstraction from the concurrent workloads and the scheduling algorithm introduces non-determinism, which confine what can really take place in parallel computing. Limiting the loss of information about parallel workloads by appropriate abstractions is a challenging endeavor. It is the key objective of system designer and application developers to bound both kinds of interference effects. The basic intuition behind modern system architecture is to yield a decent tradeoff between performance, cost, and composable timing behavior where resource contention is considered.

Through this work, we target to investigate interferences due to resource sharing in multicore safety-critical embedded systems. We examine the major sources of such interferences which make timing predictability more challenging. We address the challenges to predictability imposed by the multicore architecture in consolidating several tasks with varying criticality levels on a common platform or porting the single core software onto multicore platforms. Subsequently, we describe state-of-the-art approaches to ensure time-predictable execution. Remainder sections of this article are structured as follows. First, Section II of this paper identifies some major sources of interferences owing to resource contention on multicore systems. Next, Section III discusses Hardware/software solutions to safely limit the interferences. This is followed by Section IV which discusses the future direction of research in the context of interference in integrated platforms. Finally, we conclude the paper in Section IV.

II. INTERFERENCES DUE TO RESOURCE CONTENTION

Modern high-performance commercial off-the-shelf (COTS) hardware will share the following built-in physical components for cost, energy, and communication reasons: System bus, Main memory (DRAM), Memory bus or interconnects, the DRAM access controller, shared cache memories, Intelligent built-in computational accelerators (e.g. DMA engine, General Purpose Graphics Processing Units (GPGPUs), Interrupt Service Routines (ISR), etc.), Supporting built-in logics (e.g. Cache Coherence techniques, Translation Look aside Buffers (TLB), etc.), Logical units, I/O buses, Pipelines, and other attached peripherals.

 © 2016, IJREAM All Rights Reserved.

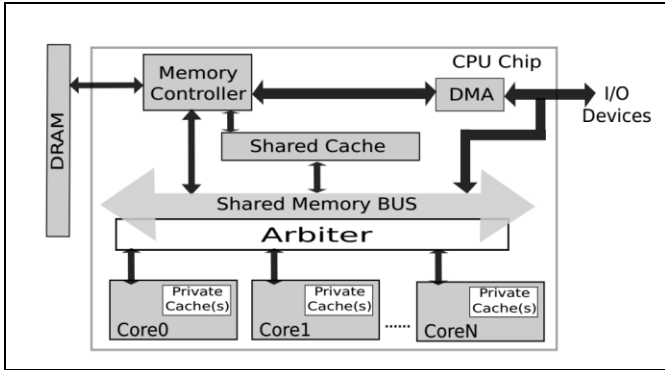

Figure1: Multicore Processor with the shared Resources

A multicore architecture with the shared resources is shown in Figure 1. A workload running on one processing element can access the shared DRAM through memory bus and memory controller. The shared resources are assumed to process only one request at a time. Access to these resources must be arbitrated either through TDMA arbitration [2] or a dynamic arbiter [3] or else an adaptive arbitration (as in FlexRay) [4].

A. System bus Review

One of the clearest facts of contention in the multicore platform is the shared system bus. In a multicore system, bus is an interconnect structure for transferring data among various subsystems within a processor, between a processor and its external devices, or among different processors. The components of the system that demand bus access are named as bus masters. To decrease the system complexity, only one master is permitted to access the bus at a time and the bounded bandwidth of that bus is used exclusively with respect to the system's timing. Multiple bus masters may contend for the same bus simultaneously and hence lead to bus conflicts. The system bus is tethering with the PEs, the memory bus, common caches, and other attached devices as shown Figure 2. Depending on the underlying hardware it may also connect to other internal/external buses, such as Peripheral Component Interconnect (PCI), CAN and FlexRay.

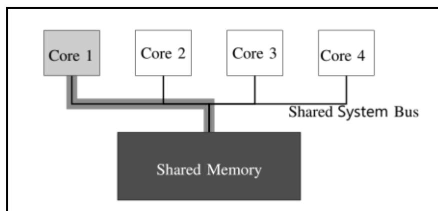

Figure 2: Multicore processor with shared memory and bus

The connection of multiple buses needs the implementation of interconnect bridges. In this case, a fine-grained mechanism is used to control the access conflicts. According to employed coherence protocols of private caches, these might also be directed over the system bus. Applications that only rely on asynchronous accesses mechanisms such as DMA traffic or DRAM refresh causes additional interference effects on the time determinism of the system. Typically, a hardware arbiter is used to control access to the system bus. This arbitration logic is depending on Round Robin (RR) and First-Come-First-Serve (FCFS) policies. For example, in earlier researches on limiting memory interference patterns, each memory request is processed for a time slot of fixed size and accesses originating from multiple PEs are granted in FCFS or round robin fashion [5], [6], [7].

B. Main memory and shared memory bus

Main memory is becoming an important source of unnecessary interferences that cause unpredictability owing to its nondeterministic access time (i.e., processing time of a memory request). Modern multicore platforms use Dynamic Random Access Memory (DRAM) as their main memory to satisfy high performance, low power and extremely low latency demands of safety-critical applications. In general, the main memory comprises multiple components such as ranks, banks, and buses as shown Figure 3. The typical problem for these components is simultaneous access by several cores in the processor.

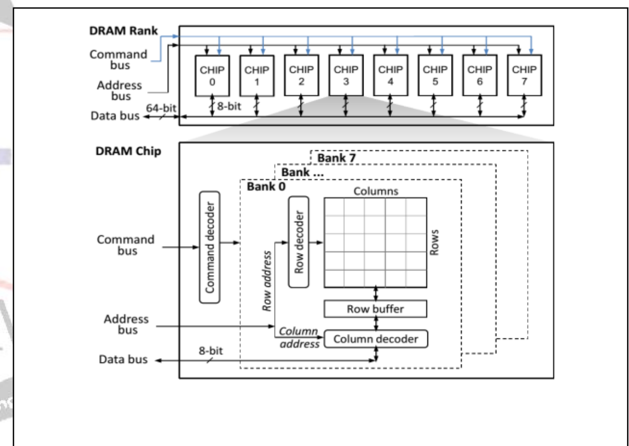

Figure 3: DRAM system organization

The memory access time is susceptible to variations according to the requested location and the timing constraints of rank/bank/bus. Memory requests from multiple banks can be processed concurrently. Conversely, simultaneous requests to one shared unit are typically serialized. Thus, multiple requests by different cores to the same resource lead to extra latencies, based on the resource's competencies for concurrent execution. For example, if a NoC (Network-on-Chip) has adequate logical communication links to serve all the PEs simultaneously; hence concurrent requests by multiple cores are usually not a problem. An arbiter is required to determine the order in which the pending access requests are serviced, which can lead to additional access latency. This additional latency can be circumvented by not assigning the same memory bank to different threads on

 © 2016, IJREAM All Rights Reserved.

ISSN : 2494-9150 Vol-02, Issue 08, Nov 2016

multiple cores. By applying this idea, modern researches propose many software elucidations to assign memory banks dynamically and evade bank sharing between different cores [7]. In addition to this, researchers have also probed approaches to limit the inherent interferences in memory [8]. According to the employed bus arbiter and a given accessing budget, the interference problem on predictability is more or less challenging. For instance, TDMA deterministic arbitration logic is not a problem for predictability since a constant accessing budget is assigned to each bus master. This arbitration logic is illustrated in Figure 4.

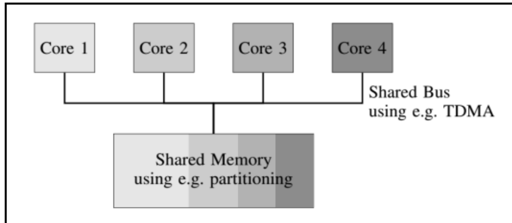

Figure 4: Multicore system with shared resources and Composability by timing isolation.

Other time-randomized mechanisms, like RR, can be implemented in a deterministic way, based on how fine-grained control of usage of these resources can be realized. Accessing mechanisms that allow starvation of PEs are very difficult to be evaluated since they do not provide a tight upper bound on latencies. The possible interleaved accesses from multiple PEs to DRAM may already exhibit extra latencies if they work with various data (i.e., memory pages), coercing the memory controller to constantly open and close memory pages. Based on the amount of concurrent active data, this may be a relatively serious concern. With respect to predictability, one would require considering accesses with the given theoretical upper bounds on latency, as far as precise information of which accesses might crash at the controller is missing. For dynamic memory systems, the additional latency owing to memory refreshes is also a serious dispute.

A modern DRAM system consists of a number of ranks; each rank contains many DRAM chips as depicted in Figure 3. The DRAM device has a narrow data bus size (typically 4, 8 or 16 bits), but chips are generally assembled to extend the size of the data bus (e.g. 8 chips X 8 bits = 64 bits wide data buses). Each DRAM device has several banks and accesses to multiple banks can be served concurrently. A DRAM bank contains a row-buffer (RB) and DRAM cells used to store data. These cells are arranged in a 2D memory structure of rows (i.e., pages) and columns. On a DRAM access, the target row having the required data needs to be transferred into the RB by means of the row decoder. The entire page that is transferred into row-buffer is called an open page. Hence, the width of the RB is the same as the width of a complete page. Consequently, the data is fetched from the particular column using the column decoder. Successive accesses to the same page are processed directly and the required data is fetched from the corresponding column without transferring the page into RB once again. The processing time of memory request varies according to which page is cached in the RB at present. A request to the page that is already stored in the RB (i.e., open page) is said to be page -hit, whereas a request to the page that is not stored in the RB presently (i.e., the closed page) is called as a page-conflict. If the required page is different than the page cached in the RB, then the active page should be is precharged (closed) and the required page has to be transferred to the RB. Subsequently, the required data item is fetched from the RB. However, moving data over the data bus experiences more delay. This latency is typically reduced by bursting and the amount of data transmitted during the execution of a read or write (R/W) operation is regulated by the burst rate.

C. DRAM Access Controller

On-chip DRAM access controller is one of the main sources of nondeterminism of the multicore processor regarding the timing estimation of real-time applications. The DRAM controller is an intermediary between the *last-level caches (LLC)* and the DRAM chips that schedules memory R/W requests generated by the core to the shared DRAM. For that reason, it is liable for the implementation of DRAM access control protocol. Furthermore, it interprets R/W operations into equivalent DRAM commands. Subsequently, these commands are scheduled according to the real-time requirements of an underlying memory subsystem. For this purpose, a DRAM controller contains a scheduler, an RB, and an R/W buffers. The RB contains certain metadata (i.e., address, R/W type, the status of the request, and timestamp) of the requestor. The R/W buffers store the data item read from or to be stored in the memory.

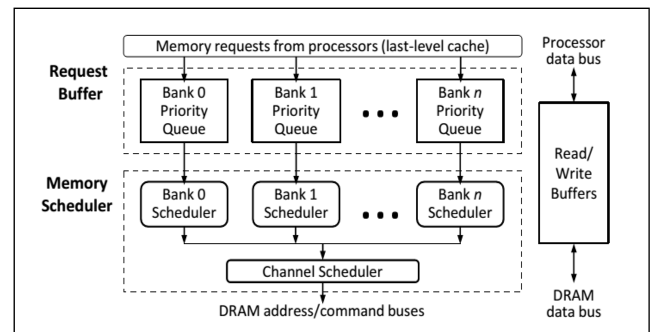

Figure 5. Two-level hierarchical structure of DRAM memory scheduler

The memory scheduler decides the access sequences of outstanding requests. This scheduler has a 2-level memory hierarchy as depicted in Figure 5. The first-level hierarchy is

 © 2016, IJREAM All Rights Reserved.

made up of priority run queues and bank schedulers (BS). Once an access request is made by the PE, first it is inserted into the priority queue. Then the BS is responsible for assigning priorities for outstanding requests and produces corresponding commands. Furthermore, the BS monitors the status of the bank. If a command with maximum priority guarantees the timeliness of the bank, then it is considered as a *ready command* and is transferred to the second level.

In the second level of this hierarchal architecture, a *channel scheduler* (CS) is responsible for monitoring commands from all BSs and keeps track the real-time guarantees of memory subsystems. Amongst all the ready commands regarding the desired channel real-time constraints, the CS selects the highest priority command for service. When the command is delivered, the CS sends an acknowledgment signal to the corresponding BS, and then it chooses the subsequent command to be processed.

D. Cache Memories

Caches are small, but fast built-in memories that temporarily store a subset of instructions and/or data of the memory for quick access. They can efficiently hide the huge access delay gap between pipelines and shared DRAM. In current and forthcoming multicore systems, more advanced levels of cache hierarchies are implemented where for every added level, size, and access time increase. The L1 (Level 1) cache is not shared among PEs, i.e. private (local). Obviously, the DRAM is common for each PE. The intermediary cache memories (e.g. Level 2 (L2) caches) may be private or common based on architecture. For a multicore platform with a shared memory model, cache data should be kept coherent. Cache coherency is a state where the change in the value of the shared data item in main memory ought to be reflected in the other processing unit's caches to maintain a consistent state of the memory hierarchy. In order to prohibit access to outdated data, extra bus transactions are required. This increases the amount of non-determinism, causing a deprived predictability of the entire processor. Though the shared memory model guarantees faster communication, sharing leads to an adverse impact on the access time of each PE.

Regrettably, the temporal behavior of shared cache memory is difficult to characterize statically. Sharing memory and caches will cause different side-effects in terms of time determinism. The major issue that we observe in a multicore platform is the bottleneck for building time-predictable software for accessing shared DRAM/caches. Traditionally, concurrent accesses will be serialized, thus a memory request incurs an additional latency due to the contention at the shared resources [9]. An unintended interference of employing a common cache is that by storing data into a cache, one PE can evict cache lines belonging to another PE since caches only have a bounded capacity. The consequence of caching is hence decreased while a large number of PE makes accesses to a common cache. Hence, the subsequent request to the evicted cache lines can be delayed again, however, it would have been fast if no other PE had stored to the same location.

Maintaining data coherency persuades a new, extensive slowdown to cache access: if one PE stores data item to a memory location, concurrent *read* requests from other PEs need to consider this, i.e. data item of a PE's private caches need to be invalidated by a *write* request from other PE. As a consequence, there is an additional resource access delay since the content has been invalidated, and it would be updated. This may again cause bus access conflict due to parallel operation from other PEs. Moreover, the bus transaction required for the coherency mechanism itself may be competed if several write requests happen.

In all of these scenarios, besides the tangible latencies when running the application on physical resources, each of the cited effects will also make a static WCET analysis more conservative, particularly on complex architectures where it is impossible to add a fixed delay to the result of the WCET analysis performed under the unicore assumption. On complex multicore processors, considering pipeline domino effects may introduce additional difficulties in the static WCET analysis. Since static worst-case temporal analysis is the foundation for task scheduling approaches, pessimism has the similar result as a tangible delay, as the sufficient resource needs to be reserved.

E. Logical units and pipelines

Current MPSoC exploits hyperthreading principles, where multiple PEs actually use the same execution units and common caches such as instruction caches. Since one virtual PE blocks and delays the execution of another PE associated with the shared unit, this cause instruction level interferences. Likewise, logical units, coprocessors and Graphic Processing Units (GPUs) can be shared. The access conflicts in such resources lead to potential latencies in the execution of multiple PEs accessing the shared resource. Based on the deployment of the resource scheduler, the delay can be imperative or even cause starving of one PE if the scheduler does not implement some level of fairness.

F. Addressable peripherals

In addition to processing elements and memories, there may be addressable devices (e.g., I/O devices, interrupt controllers and DMA engines) on the shared system bus. In a multicore platform, access to these peripherals may be more difficult. In

 © 2016, IJREAM All Rights Reserved.

order to achieve predictable use of shared addressable devices in multicore platforms, we need to enable exclusive access. Typically, locking mechanisms are required to avoid critical interferences amongst transactions accessing the same resource. In unicore systems, kernel-level scheduling is sufficient to ensure exclusive access. Alternatively, in a concurrent multicore environment, spinlocks are used to serialize access requests. Conversely, they have a noticeable overhead related to unicore implementation, i.e., all but one PE will be obliged to wait. Since spinlocks follow the shared memory model, all the features of this memory model, cache coherency mechanism, etc., apply here, also.

A different kind of interference arises from DMA engines that independently access a common bus. The addressable devices are analogous to processor cores regarding bus utilization and contention. Interrupt controllers used in multicore architectures are more advanced than in unicore processor environment, as the interrupt controller has the capacity to allocate interrupts to multiple PEs, based on requestor, preference settings, and load. The well-established interrupt routing methods may decrease timing impacts on multicore platforms compared to unicore processors: with different PEs, interrupts can be moved to other PEs, so that timing impacts on a highly critical workload may be reduced. Conversely, operating systems for multicore processors running in a Symmetric Multi-Processing (SMP) model needs to trigger other core-to-core interrupts (i.e., doorbell interrupts) for realizing TLB synchronization, or so as to assign pending workloads on other PEs. This may again cause more timing impacts.

G. *Other sources of unpredictability*

According to the given platform, there are some impacts, not associated with multiple cores that affect timing behavior of the system. These impacts might include power and thermal control strategies (e.g., Dynamic Voltage and frequency Scaling (DVFS)) and BIOS handlers and microcode (i.e., emulation microcode). Modern SCS application developers must take the consequences of power-saving techniques into account, which are evidently associated with average-case processor performance. Even though the clock-gating strategy is instant, the switching time of going into and out of sleep states is significant and must be considered in the analysis of temporal behavior. In the same way, in DVFS technique, speed-switch must be planned to reduce switching overheads.

Table 1: Undesired Mechanisms Affecting the timing predictability

Common Resource	Mechanism	Impact Level
System bus	• Conflicts generated by multiple cores • Conflicts generated by I/O devices, DMA and other devices • Conflicts generated by coherency mechanism traffic	High
Main memory	• Interleaved access by different PEs leads to latency • Latency due to memory refresh cycles	High
Memory bus and controller	• Simultaneous access • Timing anomalies	High
Common cache memories	• Conflicts due to simultaneous access • Delay due to cache line eviction • Deferred read due to invalidated data • Latency owing to congestion of read request generated by lower level cache • Conflicts due to coherency	High
Private cache memories	• Deferred read owing to invalidated data • Congestion by read requests generated by coherency	High
TLBs	• Overhead due to coherency	Medium/ High
Pipeline components	• Conflicts generated by concurrent hyper threads	High
Logical units	• Conflicts generated by concurrent applications • Some platform-specific consequences such as BIOS Handlers, Cache stashing, etc.	Medium/ Low
Bridges	• Conflicts due to other connected busses.	Medium/ Low
I/O devices	• Complexity due to locking mechanism • Status of addressable devices changed by other thread/application • Overhead due to interrupt routing • Conflicts due to the addressable peripherals like DMA engine, ISR, etc.	Medium/ High

Temperature is one more issue that may distress the timing behavior and the reliability of the SCS. Usually, some cooling methods are implemented to reduce the core's temperature to a safe threshold. If such power/ thermal controlling activities are performed during the execution of high-critical workloads and the associated latencies owing to suspensions are not taken into account in the temporal analysis, then the execution of the workloads will be nondeterministic. Any timing impact of such additional effects must, however, be examined in the case-by-case fashion. The timing effects of contention for these shared resources, which affect the predictability of the system, were identified and given in Table 1.

 © 2016, IJREAM All Rights Reserved.

III. Mitigation of Interference Effects

In the following discussion, we summarize possible state-of-the-art approaches, which are able to alleviate the unintended interferences mentioned in the preceding section. It is not to be deemed complete list but rather as a description of potential tactics to handle the effect of such interferences.

A. System bus

Up to now the typical resolution to the problem of congestion at the shared bus has been derived by deactivating parallel acting cores in the processor and circumventing asynchronous aggressive accesses such as DMA, and I/O traffic. On the other hand, this does not utilize the performance benefits delivered by multicore architectures completely. For implementing coherency, some architectures provide a distinct interconnection network. However, while deactivating the coherency from shared bus utilization is useful, it only decreases the effect of interferences, but it does not resolve the problem. The same is true for exploiting multiple banks to parallelize the concurrent accesses to memory. Once again, this decreases the consequences with respect to the level of average contention, but the problems continue.

Some methods motivate us to implement a deterministic arbitration mechanism to alleviate shared bus contention (e.g., usage of priorities in accesses). Nevertheless, these deterministic arbiters cannot be implemented in COTS architectures owing to their negative influence on average-case performance. The temporal correctness of the operations has to be assured even when employed with traditional hardware. By controlling the number of accesses, some ongoing projects, like ARAMiS [10], [11], target to reduce the highly complex interactions of multiple bus masters. Once the maximum number of accesses in a particular time window reaches a threshold, all the accesses are stalled. Although coarse-grained access windows are considered as a viable alternative, up to now, those approaches have not been examined completely. Recently, RECOMP project aims at providing methods to reduce the effect of contention through the usage of system parameters to facilitate the worst case scenario [12] or analytical techniques [5].

B. Main memory, shared memory bus and controller

In order to tackle the timing effects of resource contention, different solutions are proposed in the literature. Beneath the obvious methods to disable all but one core in the system and considering the worst-case estimates for temporal behavior, the following approaches are used: (i) complete concurrent architectures, (ii) more deterministic arbiters (iii) execution patterns, and (iv) resource limitation mechanisms. The first two methods are absolutely platform-dependent. Therefore, they cannot be implanted to the particular COTS processor if they are not already available.

Methods following the third alternative provide deterministic techniques of in what way the tasks are allowed to utilize common resources. The key idea of many state-of-the-art solutions is to define various application phases of calculation and communication by means of common resources [13], [14], [15]. As an example, Schranzhofer et al. [13] decompose tasks in various phases such as acquisition, execution, and replication. They evaluate many patterns by varying the permissions of the phases to achieve communication through common resources. Conversely, Pellizzoni et al. [14] develop the Predictable Execution Model (PREM) to decompose a task into several time-predictable intervals. Each interval is again divided into memory phase (where executable instructions and data are preloaded into private caches) and execution phase (where the workload does not incur any LLC conflict and it does not produce any access request to the common bus). Accordingly, in the execution phase, workloads will not incur any latency since there is no additional congestion owing to the bus sharing.

Boniol et al. [15] employ a similar deterministic execution model for SCS but is emphasized on a multicore system. Bellosa suggests to implements hardware means to obtain runtime information, like cache hits and misses [16]. This notion has been accepted in recent times by Nowotsch et al. [10] and Yun et al. [7] for safety-critical applications. Nowotsch et al. [10] present their approach to exploit timing statistics for WCET analysis in SCS domains. As the WCET of real-time tasks is memory bound, they decompose the accesses of a task based on their delay that is based on the number of parallel bus masters. By considering the number of worst-case accesses, they can adopt various latencies for accesses. Consequently, it decreases the WCET of a task, compared to traditional methods, which consider the worst-case access time for each memory request. Yun et al. exploit certain applications behave counters to acquire statistics about the memory accesses to distinguish highly-critical and non-critical tasks. Their objective is to schedule tasks in such a way that the impact of bounding non-critical tasks by their accesses is minimal.

All potential inter leavings of bus accesses and the ensuing extra latencies can be handled either by conservative assumptions for the estimation of worst-case execution time or by the allocation of tasks to memory. Evidently, conservative assumptions cause flexible WCETs which decrease processor usage, since assumed scenarios only occur hardly in reality. Therefore, the allocation of applications to

 © 2016, IJREAM All Rights Reserved.

memory is a more efficient one. The basic idea behind this is to allocate the task to distinct memory locations such that they cannot collide. The main issue in this method is the granularity of such allocations and they are typically platform-dependent.

C. *Cache Memories*

Unsurprisingly, cross-core cache interactions are predictable. There are several research efforts that investigate the system execution behavior [17], sometimes offering distinct hardware means [18]. However, the prediction is challenging. In order to precisely overcome unintended temporal impacts generated by common caches in a multiple core processor, the following methods can be used. The simplest, yet rarely practicable, method is to turn off caching. Owing to large access latency, it is considered as impractical. Hence, more fine-grained techniques are required. To diminish the impact of cache coherence mechanism, several configurations enable coherence mechanism to be selectively disabled. The relaxed memory models are appropriate for the software in use, thus disabling coherence mechanism may be a possible method. In some scenarios, software coherence mechanisms may be implemented as an alternative and will make more predictable execution [19].

Typically, cache coherence issues only ensue with a shared memory model, where interaction between cores is based on shared memory. As an alternative, new techniques may be employed by OS or even in hardware. This will eliminate the possible effect on coherence traffic, however, needs rephrasing algorithms, and may need TLB synchronization between PEs. There are also some methods such as cache partitioning to reduce the cache eviction issue. Software implementation can be possible [20] [21], or directly implanted in hardware.

D. *Logical units and pipelines*

In hyper threading architecture, all resources including the pipeline are shared. Hence, the applications running on the multicore platform have more influences of resource contention on their execution times. This makes hyperthreading unfeasible for hard real-time SCS. To evaluate tasks on multithreaded multicore systems the following methods are used: (i) attempting to analyze individual tasks, and (ii) hybrid analysis, which evaluates all workloads against each other to identify the potential interferences and impacts regarding resource contention [15], [22] [23] [24]. These schemes are of highly complex and demanding additional execution time or make pessimistic assumptions. One more method to employ hyper threading principles is to switch off all but one logical PEs so that the platform is not shared at all. It is a simple but efficient solution, at least until the timing impacts of hyper threading principle is better understood. One possible approach for sharing external resources is to use a server in software. The server is responsible for tackling concurrent accesses to the common resources.

E. *Addressable peripherals*

Interference issues generated by peripherals as mentioned earlier can be evaded by software/hardware solutions. First of all, the requests to these peripherals can be controlled by a software device driver. In some scenarios, the effect of an interference solution on performance is tremendous. Contemporary multicore architectures provide fine-grained access control through I/O Memory Management Units (IO-MMU). Another method is to employ the virtualization. The shared device itself delivers different concurrent virtual interfaces and is able to route the data to suitable dedicated PEs. Achieving determinism through cautious application design and temporal analysis are essential features to guarantee that assumptions made by the device manufacturers are satisfied. The important issue with arbitration to these peripherals is its stateful nature, often prohibiting interleaved access.

F. *Other sources of unpredictability*

The effects include any automated application migration ability, as being currently investigated by an embedded system community, cache hoarding activities, and many others. By implementing special configurations, the system designer is able to restrict such effects. Although cache hoarding might enhance the average-case performance, its unpredictable timing behavior needs to be estimated accurately. Implementation of efficient thermal management and power saving techniques reduce non-deterministic temporal delays.

IV. OPPORTUNITIES FOR FUTURE RESEARCH

In safety-critical embedded systems, resource sharing not only enacts challenges but also opens many avenues for exciting opportunities when the system resources are utilized in innovative ways. As an example, multi-threaded COTS processors are most frequently used in integrated architectures due to their good performance as well as energy efficiency. In a safety-critical domain where the worst-case performance and predictability are most important concerns, micro architectural components such as pipelines can be employed for prefetching data into cache to make the other pipeline states more predictable. These approaches are most appropriate for application-specific computing architecture. Investigation of automatic mechanisms to leverage such features can aid to realize improved WCET.

 © 2016, IJREAM All Rights Reserved.

The Multi-Processor System-on-Chip (MPSoC) architectures are categorized as highly complex embedded systems [25]. They encompass several complex subsystems and special tactics are required to reimburse the system complexity. In spite of this, some vital information required to have an in-depth understanding of application behaviors and trends on multicore architecture is still missing and very difficult to use in the aerospace system owing to the protective conduct of chip manufacturers to preserve their competitive benefit. Safety-critical engineering united forces and works with chip manufacturers where the potential difficulty associated with a shared platform is reduced by an enhanced understanding of MPSoC. Multi-Core for Avionics (MCFA) is one such industry working group established on the initiative of silicon experts to address the problems regarding implementing the multicore architecture in forthcoming safety aeronautic products [26]. The approaches that involve extensive investigation before they can be employed can be established in the field of hindering the nondeterminism produced by shared resources. As mentioned earlier, research opening lies in providing techniques for scheduling the concurrent request from multiple accessors to the shared resources, either by implementing cooperative method or by means of TDMA arbiter. Similar approaches could then also be established to address the interference problems in a shared system bus.

V. CONCLUSION

In this paper, we address the interference problems due to resource contention in multicore safety-critical systems. Most of the challenges and open problems explored in this paper are already well studied by research communities. Conversely, there is an evident lack of clarity in the meaning of integrated approaches and of the assumptions made to resolve it. In the field of application-specific microcontrollers, widely used in worst-case execution scenario, most of the issues are resolved at the hardware level but may have a detrimental effect on overall system performance. In the arena of general-purpose computing, typically designed to meet higher performance and greater computing capacity requirements of real-time applications, many of the problems cannot be tackled at the hardware level.

For some safety-critical computing realms, interference solutions are at a relatively immature stage. Several solutions focus on reducing the utilization of common system resources (e.g., shared DRAM), and increasing the utilization of private resources (e.g., caches) of the core, to thwart access collisions. Few approaches only exploit designated PEs to circumvent conflicts. However, both solutions are not always feasible in practice, either technically or economically and not at all efficient. For such a scenario, an extensive amount of investigation on software arbiters is required before they can be applied. Moreover, innovative programming models will be strictly required to consider the constrained accessing capacity of applications to use a shared platform.

REFERENCES

[1] Nagalakshmi, K., and Gomathi, N., "An Irreversible Transition towards Multicore Platform in Safety-critical Domain for the Aviation Industries", *International Journal of Scientific Research in Science, Engineering and Technology (IJSRSET)*, vol. 2, Issue 5, pp.345-359, 2016.

[2] Schranzhofer, A., Chen, J., and Tiele, L., “Timing analysis for TDMA arbitration in resource sharing systems,” in *Proc. 16th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 215–224, April 2010.

[3] Simon Schliecker, Mircea Negrean and Rolf Ernst, "Bounding the shared resource load for the performance analysis of multiprocessor systems", in *Proc. Automation & Test in Europe Conference & Exhibition (DATE 2010)*, IEEE, pp. 759-764, 2010.

[4] Schranzhofer, A., Pellizzoni, R., Jian-Jia Chen, and Thiele, L., " Marco CaccamoA.Schranzhofer et al. Timing analysis for resource access interference on adaptive resource arbiters", in *Proc. 17th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 213–222, 2011.

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

[6] B. Andersson, A. Easwaran, and J. Lee, "Finding an upper bound on the increase in execution time due to contention on the memory bus in COTS-based multicore systems", in *Proc. Special Issue on the Work-in-Progress (WIP) Session at the 2009 IEEE Real-Time Systems Symposium (RTSS)*, ACM, SIGBED Review, vol.7(1), pp.4, 2010.

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

[8] Kim, H., de Niz, D., Andersson, B., Klein, M., Mutlu, O., and Rajkumar, R., "Bounding memory interference delay in COTS-based multi-core systems", Technical Report CMU/SEI-2014-TR-003, Software Engineering Institute, Carnegie Mellon University, 2014

[9] Pellizzoni, R., Schranzhofer, A., Chen, J., Caccamo, M., and Thiele, L., "Worst case delay analysis for memory interference in multicore systems", in *Proc. Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pp. 741- 746, 2010.

 © 2016, IJREAM All Rights Reserved.

[10] Nowotsch, J., Paulitsch, M., B¨uhler, D., Theiling, H. Wegener, S., and Schmidt, M., "Monitoring-based shared resource separation for commercial multi-core system-on-chip," in *Proc. 26th EUROMICRO Conference on Real-Time Systems (ECRTS)*, July 2014.

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

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

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

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

[15] Boniol, F., Cass´e, H., Noulard, E., and Pagetti, C., "Deterministic execution model on cots hardware," in *Proc. 25st international conference on Architecture of computing systems (ARCS)*, pp. 98–110, 2012.

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

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

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

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

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

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

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

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

[24] Radojkovi´c, P., Girbal, S., Grasset, A., Qui˜nones, E., Yehia, S., and Cazorla, F. J., "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.

[25] 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.

[26] 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, Sep. 2011

 © 2016, IJREAM All Rights Reserved.