Identifying the Sources of Unpredictability in COTS-based Multicore Systems

Dakshina Dasari, Benny Akesson, Vincent Nélis, Muhammad Ali Awan, Stefan M. Petters
CISTER-ISEP Research Centre, Polytechnic Institute of Porto, Portugal
{dndi, kbake, nelis, maan, smp}@isep.ipp.pt

***Abstract*—COTS-based multicores are now the preferred choice for hosting embedded applications owing to their immense computational capabilities, small form factor and low power consumption. Many of these embedded applications have real-time requirements and real-time system designers must be able assess them for their predictability and provide guarantees (at design time) that they deliver the correct functional behavior within predefined time bounds. However, the underlying architecture of commercially available multicores is extremely complex and non-amenable to straight-forward timing analysis. In this paper, we highlight the architectural features leading to temporal unpredictability, which mainly involve shared hardware resources, such as buses, caches, and memories. We explore some of the existing work in timing analysis with respect to these features, identify their limitations, and present some unaddressed issues that must be dealt with to ensure safe deployment of real-time systems.**

I. Introduction

Multicores developed using Commercially available Off-The-Shelf (COTS) components have become the preferred choice in the design of embedded systems. In-lieu of the strict timing requirements of hard real-time systems, real-time embedded systems were traditionally assembled from scratch using custom-built hardware and software components, specifically designed for them. The entire product development cycle was long and expensive, especially when used in massive and complex systems (e.g. aircrafts): Each of the individually developed components had to be designed, developed and unit-tested and then finally integrated with the rest of the system. Over the years, products have become more complex and there has been a strong push towards using COTS components for their development. The key driving factors for this shift are that readily available, cost-effective and pre-tested COTS components that can be seamlessly integrated with each other, result in shortening the end-to-end development time, giving them an edge over in-house developed products [1].

The adoption of COTS-based multicores in particular was also driven by the fact that functionality that was previously distributed across multiple chips is now available in a single chip [2], [3]. In earlier systems in which functionality was deployed in different chips, inefficiencies of working with multiple support environments and programming models led to a longer time-to-market and increased long-term support costs. Building and maintaining systems with multiple chips, power supply units, memories and I/O interfaces to support the different processors adversely impacted system component and manufacturing costs.

The use of COTS-based systems in the military domain is exemplified by the Arleigh Burke-class Aegis guided missile destroyers, such as the USS Stethem, which employ a 2.16 GHz Core2 Duo-based conduction-cooled CompactPCI boards [4]. Other military real-time applications like surveillance systems also benefit by the high computing power provided by multicore systems. In the medical health applications domain, researchers in Philips Healthcare have used a regularly available Intel Xeon E5345 processor to host X-Ray Imaging applications [5]. Although COTS components provide great opportunities for embedded system designers, they are not without their own demerits.

COTS components are already used in real-time systems with low criticality (also called soft real-time systems), but they are not yet typically employed in their hard real-time counterparts. The reason is that COTS components uphold the design mantra “Make the average case faster”; i.e., these systems are primarily designed towards increasing the average-case performance, which has been achieved by increasing the complexity in the design. In contrast, the key requirement for most hard real-time systems are platforms that are predictable and amenable to easier timing analysis. Real-time system analysts prefer components that can collaborate together to provide predictable and reliable behavior. Temporal and spatial isolation of components should ideally be provided by the hardware itself. Spatial partitioning ensures that an application in one partition is unable to change private data of another. Temporal partitioning, on the other hand, guarantees that the timing characteristics of an application, such as the worst-case execution time (WCET), are not affected by the execution of an application in another partition. If present, these features reduce the time, effort and cost involved in the analysis of these systems, since the temporal properties of each component can be validated independently. In the absence of these features, the vendors must provide documentation and mechanisms to be able analyze their timing parameters and derive safe upper bounds on key parameters like execution time. Unfortunately, the design of most of the current multicores contradicts these requirements. The presence of *shared hardware resources*, like the memory bus and the memory controller, governed by performance-oriented arbiters and schedulers facilitate neither composable analysis nor the computation of upper bounds on the key timing parameters on these systems. In existing COTS-

This work was partially supported by National Funds through FCT and ERDF (European Regional Development Fund) through COMPETE (Operational Programme ’Thematic Factors of Competitiveness’), within project Ref. FCOMP-01-0124-FEDER-022701; by FCT and COMPETE (ERDF), within REPOMUC project, ref. FCOMP-01-0124-FEDER-015050 and the REGAIN project, ref. FCOMP-01-0124-FEDER-020447; by FCT and ESF (European Social Fund) through POPH (Portuguese Human Potential Operational Program), under PhD grant SFRH/BD/71169/2010.

978-1-4799-0658-1/13/$31.00 © 2013 IEEE

based multicores, most often, only a brief description of its functionality is provided. Also, these components do not carry any guarantee of adequate testing for the intended hard real-time system environment. For example, a processor manual may report the average time to access main memory, but what is required is the corresponding worst-case estimates. In most cases, the designer does not have access to the source code of the component and this inhibits straight-forward modifications to the current design – Many COTS components are therefore black boxes without their source code or other means of introspection available.

The problem of timing analysis due to architectural complexities has been prevalent since the time of uniprocessors, which exploited instruction-level parallelism by employing techniques like pipelining, speculative fetching, branch prediction and out-of-order executions, and the research community successfully tackled these issues over the decades [6]. With multicores, the problems were exacerbated when the same (dedicated) interconnection networks, caches and memory systems were shared by multiple cores.

Contribution and importance of this work: This work does not present any novelty in terms of algorithms nor do we solve a particular problem. It has the sole objective to identify and discuss the architectural features that cause temporal variations in applications at run-time. The shared resources mentioned above and other features that are often overlooked (or barely mentioned) in timing analysis, such as hardware-prefetching mechanisms, power-saving strategies, system management interrupts, are discussed here. We believe that this work is important because until the industry shifts its current trend and starts building *predictable* systems, there is a strong need to fully analyze the existing COTS-based systems from the perspective of understanding the various factors that lead to temporal unpredictability. Although multicore systems have been around since 2004, there are no established approaches, as yet, to achieve certification for such platforms as time predictability and segregation of functionality cannot be guaranteed. Currently, even when multi-core systems are used to implement ARINC 653 standards, all but one of the cores in the multicore processors are typically disabled [7]. This work presents a compilation of the various causes of unpredictability and provides an insight into the existing work and their limitations.

II. Related Studies

The sources of unpredictability in *uniprocessor* platforms have been researched earlier, with the primary focus being on-chip features like speculative execution by branch prediction, pipelining and caches. As a result of the extensive research, tools adopted by the industry are already available for uniprocessors. Examples are aiT [8], SWEET [9] and RAPITime [10]. In [11], the impact of pipelines and caches on temporal behavior have been explored in detail. This paper also describes the architectural influence on static timing analysis and gives recommendations as to profitable and unacceptable architectural features (for multicores). Methods to analyze cache and pipeline behavior by abstract interpretation and pipeline modeling have been proposed in [12]. Another work [13] describes the threats to predictability considering the target architecture and the different system layers of the software. This paper focuses on the impact of shared hardware resources in *multicores* and also presents the currently available solutions, the limitations and a short survey of the same. Features like caches have been addressed for uniprocessors ([14], [15], [16], [17]), but in this work we present them from a multicore perspective. This work also provides a detailed discussion on memory controller scheduling policies, which is not present in related prior works. The impact of Translation Look-aside Buffer (TLB) misses, hardware prefetching and the thermal and power management strategies have also been incorporated here to re-iterate the importance of considering all aspects of the architecture in the timing analysis to ensure predictable behavior.

Outline of the document: Section III presents a COTS-based system. Next, the sources of unpredictability in this work have been broadly studied under two categories. (i) The *primary sources* that originate from architectural features like caches, the FSB, the memory and memory controller, which have been the highlights of most of the research and presented in Section IV (ii) The *secondary sources* that are generally not addressed in the research works and yet are non-negligible when safe estimates of timing parameters are desired. The impact of such features like hardware-prefetching, power-saving strategies, translation look-aside buffer (TLB) misses, system management interrupts generally tend to be overlooked in the timing analysis and are the focus of this category. This is presented in Section V. The document concludes with a summary of the work in Section VI.

III. Overview of a COTS-based system

The architecture of a general COTS-based multicore system is illustrated in Figure 1. All the cores of the multicore chip have an individual (private) Level-1 (L1) cache and share a Level-2 (L2) cache (or more levels of caches). The multicore chip is connected to the North-Bridge (NB) over an interconnection network, usually called Front-Side Bus (FSB) or the memory bus. The FSB is the electrical interface that connects the processor to the main chipset. All interrupt messages, memory and the cache coherency traffic flow between the cores and the chipset through the FSB. Requests to the memory subsystem are channeled from the cores via the FSB or from I/O devices via the I/O buses, as described below.

The North-Bridge (NB) typically handles the communications between the cores, the system main memory (RAM), the graphics controller and the South-Bridge (SB). The main memory is thus shared between multiple entities over the NB, including the cores residing on the chip, the graphics controller and the SB. The communication between the main memory and these entities is handled by a memory controller and a memory arbiter, both directly incorporated into the North-

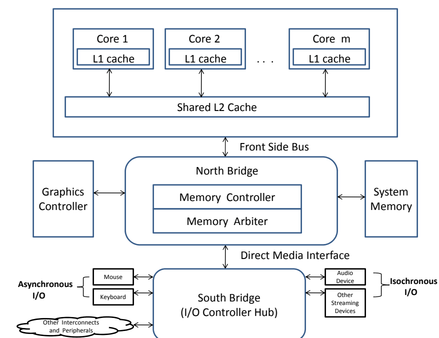

Fig. 1. A typical COTS-based multicores architecture.

Bridge. Generally, a graphics controller is connected to the NB (or is sometimes integrated into the NB as well depending on the chipset design). The South-Bridge (SB), often referred to as the I/O Controller Hub, handles communication with the peripherals, such as the hard-disk, keyboard, and printer, over a variety of buses (like PCI and PCI express). The peripherals can be connected in various ways depending on the chipset design. Typically, the SB is connected to the NB via a Direct Media Interface (DMI) channel. All the Direct Memory Access (DMA) traffic (arising from the peripherals) is also channeled through the SB.

Although the design of the architecture depicted in Figure 1 is aligned to (and borrows its terminology from) the Intel processor [18] and pertains to the PC architecture, the discussion applies to embedded multicores in general and the nomenclature may vary accordingly. Note that for the given model, the cores and the caches are *on-chip* elements while the rest (memory, peripherals and the controllers) are *off-chip*.

IV. Primary Sources of Unpredictability

The main focus of this section is on shared caches, shared interconnection networks and the shared main memory.

A. Shared Caches

In most of the existing multicores, the large gap between the core speed and the memory is bridged by keeping the most frequently accessed data closer to the cores. As seen in Figure 1, the cache hierarchy typically consists of a private L1 cache (closest) to the cores followed by a L2 cache of higher capacity, which is shared among cores. Modern processors feature large caches with high set-associativity (8-way and 16-way associative caches are not uncommon) and non-deterministic replacement algorithms, making cache analysis extremely challenging. Apart from the associativity and the write policies, the predictability of cache behavior is largely influenced by the replacement policy, which is usually pseudo least-recently used (PLRU) in many multicore platforms from Intel and Infineon. The impact of these replacement policies on predictability has been presented in [19].

In unicore systems, the problem of sharing the cache amongst tasks that could pre-empt each other (pre-emptive scheduling) on the same core is intricate [17] and the analysis to compute this extra cache-related pre-emption delay is already non-trivial. The problem is further exacerbated in multi-cores when co-executing tasks on other cores share and contend for the same cache lines, thereby increasing the possibility of cache-line evictions. Although the higher capacity of the caches was provided to decrease the accesses to main memory and thus reduce the stall time of executing tasks, non-ownership of these shared cache lines by the cores can lead to unregulated cache evictions and cache thrashing. This defeats the very purpose of providing a larger cache as it leads to increased memory requests. Additionally, bounding the number of memory requests that a particular task may generate in an interval is challenging at design time, since memory requests from tasks do not arrive periodically and the order in which tasks are executed is dependent on the scheduling policy. In fact, the number of varying patterns of task arrivals on other cores, replaceable cache lines and memory request patterns result in a combinatorial explosion of possibilities.

TABLE I. Latency in cycles for L1, L2, and L3 caches

Processor	L1	L2	L3
AMD Phenom II X4 920 (2.80 GHz)	3	15	No info
Athlon X2 5400 (2.80 GHz)	3	20	NA
Intel Core 2 Quad QX9770 (3.2 GHz)	3	15	NA
Intel Core 2 Quad Q9400 (2.66 GHz)	3	15	NA
Intel Core i7-965 (3.2 GHz)	4	11	42

Cache latency timings: To compute the total time to serve a cache-line request, it is necessary to determine the cache latencies. Manuals may provide certain data regarding the latency for cache accesses as provided in Table I, but these numbers do not necessarily capture the worst-case behavior and are best- or average-case latencies and hence cannot be used to guarantee safe WCET estimates. Also, there are no mechanisms provided directly by the underlying sub-systems (like the memory controllers) to carry out a confident evaluation of these timing parameters.

Existing solutions and their limitations: To a limited extent, in the multicore area, the impact of shared caches on the WCET has been addressed in ([20],[21],[22],[23]), amongst others. Given the complexity of the problem, researchers have made assumptions that limit the applicability of their solution. For example, detailed research in multicores has been limited to analysis considering: i) direct mapped caches or caches with low set associativity, ii) assumptions that data caches are perfect (the accessed data is always available) and thereby analyzing instruction caches only, iii) assuming that the underlying replacement policy is LRU, and iv) the complete impact of write back and write through mechanisms has not been considered. It is also important to note that no analysis has been verified on actual hardware.

Alternatives to the shared cache problem: Alternative ideas have also been proposed that circumvent the problem of shared cache contention by spatial isolation. A method to analyze shared instruction caches proposed in [24] is based on the combined use of cache locking and partitioning. Cache locking allows the user to load selected contents into the cache and subsequently prevents these contents from being replaced at run time. Cache partitioning assigns a portion of the cache to each task and restricts cache replacement to each individual partition. The objective of such a joint use of locking and partitioning is to completely avoid intra-task and inter-task conflicts and also provide spatial isolation.

Cache partitioning techniques have also been proposed by Guan et al. [25]. Their method employs page-coloring [26] combined with scheduling to isolate the cache lines of hard real-time tasks running simultaneously, thereby avoiding the interference between them. Page coloring is a software technique that controls the mapping of physical memory pages to a processors' cache blocks. Memory pages that map to the same cache blocks are assigned the same color. Cache bypass techniques for instruction caches, proposed by Hardy et al. [27], are based on the fact that many blocks stored in the cache after a miss may not be reused before their eviction. Such blocks, named *single-usage blocks*, contribute to cache pollution — a situation in which an executing program loads data into cache unnecessarily and causes other necessary data to be evicted from the cache into lower levels of the memory hierarchy, potentially all the way down to main memory, thus causing a performance hit. The paper proposes a method to identify such single-usage blocks and that does not store them in the shared cache(s), thereby tightening the WCET estimates.

A drawback of this method is that it requires special support in hardware to identify certain instructions as non-cacheable.

Shared caches and hard real-time systems: As seen above, given the complexity of the caches present in modern day processors, it is extremely challenging to derive tight estimates for shared caches. Hard real-time systems are more likely to be developed on processors with private caches or by either disabling or partitioning the shared cache, if present. Our stand of promoting private caches is also guided by the certification requirements imposed by the industry to ensure that the safety standards of critical real-time systems are met. Before a safety-critical system is deployed, a certification authority ensures that certain norms are met. All the components comprising that system (the software, hardware and the interfaces) are scrutinized to ensure conformance to safety standards. The certification process categorizes each task by its level of criticality and assigns it a corresponding Safety Integrity Level (SIL). When deployed on the same multicore system, tasks of different SILs can co-exist and share the same physical hardware resources. In order to limit the risk of failure of tasks with high SILs, systems must be designed to isolate the execution of the tasks, both in the spatial and temporal domains. If total isolation cannot be achieved then designers must be able to upper bound the impact that task executions have on each other. To cater to these requirements, international standards typically favor simple and safe designs, such as partitioned scheduling, *partitioned caches* [28], [29], time-triggered architectures and cyclic scheduling algorithms (CSA) as recommended in [30] (Annex F, page 103). Given these facts and norms, it is very unlikely that shared caches will ever be deployed in hard real-time systems. System designers hosting applications on multicores with shared caches will have to either reserve a part of the L2 cache for each core and think about *cache arbitration*, assign the L2 cache only to one core or turn the L2 cache off for all cores. Cache contention though widely studied, is just one of several factors that can cause performance variations — other factors like prefetching hardware contention, memory controller contention, and FSB contention account for as much as 60%- 80% and sometimes 100% of the performance variation that tasks experience on multicore processors [31] and hence a deeper insight into these off-chip factors is warranted.

B. Shared Front-Side Bus

In typical implementation of simple COTS-based systems (see Figure 1), multiple cores access the main memory via a shared bus. This often leads to contention on this shared channel, which results in an increase of the response time of the tasks. Analyzing this increased response time, considering the contention on the shared bus, is challenging on COTS-based systems mainly because: i) bus arbitration protocols are often undocumented and the implementation of arbitration protocol is hidden, ii) the exact instants at which the shared bus is accessed by tasks are not explicitly controlled by the operating system scheduler; they are instead a result of cache misses, TLB misses, coherency traffic, etc., and iii) requests are not tagged with any task priority information and thus, although the cores may enforce this prioritization and give preferential access to tasks with higher priorities, the bus may re-order the memory requests based on its internal prioritization and request scheduling mechanisms. As a consequence, requests issued by higher-priority tasks may be served later than those from lower-priority tasks. In principle, the extra overhead due to the FSB is attributed to two main factors: i) the communication delay on the bus, which depends on the speed and data width of the bus, and ii) the time until a free slot is available on the bus. If requests are served in-order, then the first overhead can be upper bounded, since the required parameters are generally documented. The second factor is largely dependent on the bus arbitration mechanism. If the bus is TDMA driven, an idle slot is guaranteed to each requester (e.g. a core) at fixed predictable time instants. In contrast, with other arbitration mechanisms, it is difficult to discern at design time the exact instants at which the bus is available, since it is largely dependent on the request patterns of the co-scheduled tasks. To complicate matters, the FSB in modern processors may be an out-of-order bus (e.g., the Intel Itanium Processor Family) and employ other performance-enhancing mechanisms, including split transactions and pipelining. If pipelined buses are employed, the time for "k" bus transactions is not tightly bounded by simply adding the execution times of the individual transactions, since the phases within a transaction (typically arbitration, request, error, snoop, response, optional data phase) may be overlapped. For example, the Intel 4 Chipset Family boards [18] have a 12-deep in-order queue to support up to twelve outstanding pipelined requests on the FSB.

The impact of these parameters must be accounted for, in the timing analysis to obtain tighter and safe worst-case estimates. Ideally, a point-to-point connection or switching mechanism between (private) main memories and the cores that provides a clean data-path separation would be highly approved by researchers and certification experts, but the physical realization of such a design may come at a cost of increase in the form-factor of the chip, excessive wiring, increased pin-count and power usage, which is not preferred by embedded system designers. Given that the hard real-time market does not drive the chip-design market, it is unlikely that chip vendors will invest much effort or time in designs that are real-time friendly. In lieu of these facts and the recognition of the problem of performance degradation due to bus contention, there have been notable inroads in the timing analysis area by the research community.

TDMA related research: Most of the research has been focused on the assumption that the underlying arbitration mechanism is Time Division Multiple Access (TDMA). Some noteworthy works in this direction amongst others are by ([32], [33], [34], [35]). Rosen et al. [36] describe a solution to implement predictable real-time applications on multiprocessors. Based on the tasks running on the cores, they pre-compute a bus schedule, such that all tasks meet their deadlines. This bus schedule is stored in a memory connected to the arbiter. However, this type of table-driven arbiter requires additional hardware modifications that are not provided by existing COTS buses.

Non-TDMA based research: Amongst the non-TDMA schemes, Schliecker et al. [37] have proposed a method to address the issue of bounding the shared resource load for multiprocessor systems using a general event-based model. They consider a system with a global shared resource in which the maximum and minimum numbers of accesses in particular time windows are assumed to be given. By assuming fixed-priority scheduling on the shared resource with statically assigned priorities to tasks, the worst-case interference is derived based on the access patterns of higher-priority tasks. In [38],

the authors compute an upper bound on the contention delay incurred by a task for time-triggered (periodic) tasks, using a restrictive pre-emption model. Tasks are split into superblocks in which each superblock can include branches and loops, but superblocks must be executed in sequence. Multiple tasks executed on the same core are scheduled according to fixed time slots, with a given set of superblocks assigned to each slot. Peripherals are represented as buffered flows and an arrival curve is computed for each peripheral. The delay of the task under analysis is computed based on the delays caused by traffic streams from all other cores. Analysis for arbitration-agnostic work-conserving buses (implying that the bus cannot be idle if a request is pending) have been done in [39], [40]. These approaches assume the maximum interference that co-scheduled tasks can pose to the analyzed task to derive the response time when in contention. Given that no specific arbitration mechanism is considered, the approaches can yield pessimistic results. Approaches based on timed automata (TA) have been proposed by Mingsong et al. [41] and [42], but TA-based methods inherently suffer from the drawback of state-space explosion for complex architectures.

Researchers have recently proposed a software-based memory throttling mechanism to explicitly control the memory interference [43]. The basic idea of memory throttling is to periodically limit the number of memory accesses (last-level cache misses). In order to do so, the OS scheduler additionally, monitors the cache-misses of each core and dequeues/enqueues tasks based on the specified cache-miss budget and the period.

Summary: The TDMA bus arbitration is predictable and composable, allowing tasks to be analyzed in isolation, making it a real-time friendly protocol. It is non-work-conserving and hence the bus is idle when the core owning a time slot does not have any requests to be served. Unfortunately, although favored by the research community, existing COTS-based systems (which are designed for high performance) do not employ it. Therefore, the related research, can in essence be applied to a custom-built bus that employs TDMA, but cannot be applied to existing processors with other arbitration protocols. Thus software-based techniques that emulate a TDMA-like behavior by giving *phased accesses* to the bus to a single task at a time are warranted. The mechanisms proposed in [34], which divides the task-phases in exclusive data fetch and execution, should be extended to non-TDMA buses. Tighter bounds must be calculated by taking into account the pipelining and split-transaction mechanisms in current buses. In summary, the lack of predictability in the FSB will have to be compensated with a combination of simple table-based cyclic scheduling at the core level, application-level budgeting of the bus slots, use of private caches, application request restructuring to have predictable memory request patterns to facilitate a tighter analysis of the increased WCET due to bus contention.

C. Other Interconnection Networks

With the increasing number of cores and the obvious scalability issues posed by a single FSB, more recent multicore designers have moved to the Non-Uniform Memory Access Architecture (NUMA) model. Examples are the QuickPath Interconnect from Intel and HyperTransport architecture from AMD. The basic design paradigm has shifted from a shared bus to a point-to-point connection in these cases. In the NUMA model, every core has a local memory and hence the contention on the FSB is relatively reduced. The time to access memory is variable depending on whether the requested data is available in local memory or remote memory (local memory of another core which may be off-chip). Though the average performance is improved by reducing contention, the problem of unpredictability due to remote memory access still persists in these architectures. If complex task models are assumed in real-time applications in which tasks can migrate, the analysis can become more complex, as the task has to migrate its entire working set in local memory from the source core to the target core and the migration time must be also be upper bounded to reduce the number of remote memory transactions and hence the unpredictability remains.

After an insight into the interconnection network, the document next proceeds to study the contention at the next level: the memory sub-system.

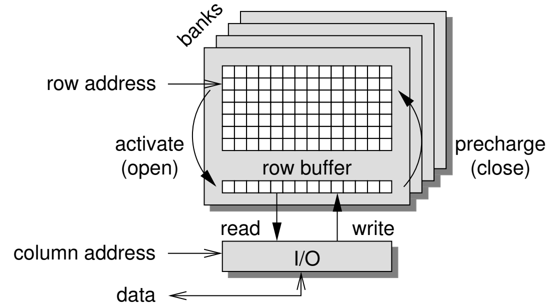

Fig. 2. Illustration of a DRAM chip.

D. The Memory Device

As described earlier, requests from the CPU and the peripherals (including DMA requests) are eventually directed to the memory via the memory controller (which is studied in the next section). The unpredictability in DRAMs stem from their internal architecture, which is designed to deliver high volume storage at low cost per bit. To reduce area and power, it additionally tries to minimize the number of off-chip pins by using a bi-directional data path. A contemporary COTS-system typically contains many DDR3 DRAM chips [44] connected in parallel on a dual in-line memory module (DIMM) to form a 64-bit data path to the memory. The chips may be organized in one or more ranks that share the same interface to increase its utilization without increasing the number of pins. As illustrated in Figure 2, each DRAM chip comprises several banks that can be accessed in parallel. Each bank contains a matrix-like memory array of rows (also called pages) and columns. In addition, each bank has a row buffer that can store the contents of one row. On a DRAM access, the target row must first be activated (opened) by copying its contents from the memory array to the row buffer before read or write operations can be issued to the word-sized column elements. Once there are no more read or write operations, the row is precharged (closed) and the contents of the row buffer are copied back to its original place in the memory array [45].

The DRAM architecture makes the response time of memory requests and the provided bandwidth highly variable for three reasons: i) a request targeting an open row can be served immediately, while it otherwise needs the current row to be closed and the required row to be opened, ii) the bi-directional data path requires several cycles to switch from read to write and vice versa, and iii) to prevent data loss, the memory must

occasionally be refreshed before executing the next request and the added refresh time may be longer than the time to serve the request itself. The impact of these three factors may cause the execution time of e.g. a 64 B memory request to vary by an order of magnitude from a few clock cycles to a few tens of cycles. DRAM memories can hence be considered highly unpredictable resources by nature and are challenging to work with in the context of real-time systems.

E. Memory Controller

The memory controller connects the system to the off-chip DRAM and is responsible for scheduling memory accesses according to the system requirements. In a COTS system, the memory controller achieves this by maximizing the average bandwidth and minimizing the average latency, while limiting power consumption. This typically implies maximizing the utilization of the data path, possibly subject to different priorities of memory streams, when there are pending requests and make efficient use of power-down modes in the memory device when there is idle time. Overall, there are three factors that affect the response time of memory requests in the memory controller: i) the page policy, ii) the scheduling algorithm, and iii) the power-management policy. We proceed by discussing each of these in turn.

Page policy: The page policy determines when precharge commands should be issued by the memory controller [46]. Currently, there are two prevalent page policies: open page and close page. The open-page policy tries to improve the average performance of the memory controller by exploiting locality among memory requests. This is achieved by speculatively keeping activated rows open after memory accesses, hoping that the following requests to the banks target the same rows, thereby eliminating the latency and power overhead of activating and precharging the banks [45]. The drawback of this approach is a latency penalty in case the following request requires different rows in the banks, as this results in precharging and activation while the request is stalling. The open-page policy works well in case there is sufficient locality in the memory stream to generate enough row hits to make a net gain in average performance despite this penalty. In contrast to the open-page policy, the close-page policy always closes the active rows immediately after each memory access to minimize the overhead of opening another row in the same bank. This policy is beneficial when there is not sufficient locality within the memory stream of an application, or when locality is destroyed when memory streams from different applications are multiplexed in the memory controller to access the single off-chip memory. This policy is typically favored by memory controllers for hard real-time systems [47], [48], [49], since they are unable to guarantee any locality in the worst case due to fine-grained sharing of the memory, and hence prefer to reduce the miss penalty.

Hybrid policies that combine properties of open- and close-page policies have also been proposed. To improve performance of their systems, Intel proposed an adaptive page policy [50] that dynamically switches between open- and close-page policies based on the locality in the memory streams. In the context of real-time systems, a conservative open-page policy [51] has been proposed. The key idea is to partially exploit locality by keeping active rows open as long as possible without negatively impacting the worst-case response time of memory requests. This approach works well if there is locality in the memory traffic and if requests arrive close enough together to enable row hits to be detected early.

Scheduling algorithm: The memory scheduler is responsible for ordering incoming memory requests and generating SDRAM commands that are scheduled according to the timing constraints of the memory. This may involve a two-level scheduler, one level for memory requests and a second one for SDRAM commands, although it is possible to integrate the two. The memory scheduler is often very dynamic and uses information about the memory state when scheduling to improve average bandwidth or reduce average latency. Optimizing bandwidth may involve preferring requests that target an open row in a bank [52], [46], [53], requests that fit with the current direction of the data path [54], [55], [56], or a combination of the two [57], [58], [59]. Example mechanisms that reduce average latencies is to prefer reads over writes [53], which is beneficial if reads are blocking while writes are posted, or let high-priority memory clients preempt lower priority clients [59]. Another technique to reduce latency is presented in [52] that schedules memory bursts belonging to the same requests simultaneously thereby unblocking the stalling processor earlier. It is also proposed in [60] to try to schedule refresh operations during idle cycle cycles when there are no requests pending or even executing multiple refresh operations in sequence when idle to amortize overhead [61]. The problem with these dynamic memory schedulers is that the interactions between the request and command schedulers are complex, especially in the presence of the aforementioned mechanisms. Thus, neither of the above memory controllers provide bounds on bandwidth or latency, making them difficult to use in the real-time context.

Power policies: DRAM memories have several power-down modes [44], e.g. power-down with fast exit, power-down with slow exit, and self-refresh. These modes have increasingly large transition times in and out of the low-power state, while the current through the memory is decreasing, thus offering different trade-offs depending on the length of the idle periods and the maximum tolerable wake-up penalty. This trade-off is done by the power-management policy in the memory controller, often implemented by transitioning to a low-power state after an idle interval that may be either constant [62], [63] or adaptive [63]. While the wake-up penalty is only a few cycles for fast-exit and slow-exit power down, it is hundreds of cycles for the self-refresh state in DDR3 memories. For this reason, methods are proposed to predict the length of idle periods to ensure that the memory is powered up before the arrival of the next request [64], [63], reducing the wake-up penalty. However, although these methods may reduce the average performance penalty of using power-down policies, they are not guaranteed to be helpful to reduce the worst-case penalty. A consequence of the sometimes substantial wake-up penalties is that the worst-case memory latency does not happen when the memory controller is maximally loaded, but when there are sudden bursts of memory requests while the memory is in self-refresh. Determining the critical instance for the memory controller may hence be difficult without information about the power-management policy, further complicating the process of estimating memory latencies with both analytical and measurement-based techniques.

Summary: The time to serve a memory request is *highly variable* and strongly depends on the architecture of the

memory itself, as well and the scheduling algorithm and page- and power policies used in the memory controller. All this information is generally not divulged for COTS systems, hence it is difficult to obtain an accurate estimate of the memory latency. The memory controller may offer configuration options to disable dynamic features, such as reordering mechanisms, which makes the scheduler easier to analyze. However, these options are not exposed to developers through the middleware (BIOS) in COTS systems. Instead, the only visible options are to reduce timing constraints of the memory to reduce latencies at the expense of reliability. These problems lead us to conclude that to improve the suitability of COTS systems in the context of real-time systems, more information is required about the scheduling algorithm and page- and power policies. The possibility to disable dynamic features of the controller must furthermore be exposed to developers through the middleware. This will enable researchers to accurately determine memory latencies using analytic or measurement-based approaches.

V. Secondary Sources of Unpredictability

In this section, we will in brief cover the secondary sources like the effects of hardware prefetching, power-saving strategies and TLB misses.

A. Power-saving Strategies and Thermal Effects

In many multicore systems, an Advanced Configuration and Power Interface (ACPI) specification [65] is commonly employed. This standard brings the power management under the control of the OS (OSPM) and encourages software/hardware vendors to develop ACPI-compliant implementations to: i) enhance the power-management functionality, ii) increase the robustness, iii) enable implementation of a variety of power-management solutions, varying from simple to aggressive approaches, allowing an appropriate cost/function trade-off, and iv) facilitate the industry-wide implementation of power management. These features are hard to implement in the traditional BIOS-central system that relies on the platform-specific firmware. Power-saving strategies and thermal management techniques in COTS-based systems can be categorized into two main groups based on whether they are completely or partially under the control of the OS.

Strategies under Total OS Control: Generally, processor power states in COTS-based multicores are compliant with ACPI. For example, processor power states (C-states) in the Intel's 3rd Generation Processor family of multicores built with 22-nanometer process technology ($i3, i5, i7$) [66] can be controlled through the OS. However, real-time system designers must consider the effects of these power-saving features. While clock gating can be considered instantaneous, the transition into and out of deeper sleep states requires time, which must be factored in the timing analysis. Similarly, in Dynamic Voltage and frequency Scaling (DVFS), the transition overheads of the processor speed-switch should be budgeted.

Temperature is another factor that may affect the temporal behavior of the real-time system and the long-term reliability of the chip. A processor typically has a safe operating temperature zone and if the processor is kept active for a prolonged duration of time, its temperature may exceed this safe threshold. Normally, some cooling systems are employed to tackle such situations – nevertheless, it may be necessary for some systems to suspend their execution (or even turn-off the processor) to protect the chip and to decrease the system's temperature. Such effects must be considered based on the processor's specifications. One common misconception is that these power-saving features can be disabled and the side-effects can be ignored safely. However, disabling the power-saving features (sleep states) and enabling the processor in an active state throughout the execution time can have an adverse effect on the reliability of the chip. Therefore, these effects should be accounted for in the thermal design to limit the peak temperature of the system. If these power or thermal management activities occur during the execution of safety-critical tasks and the related delays due to suspensions are not considered in the timing analysis, then the behavior of the tasks will be unpredictable.

Strategies Partially under OS Control: Apart from the aforementioned unpredictability sources, there are also some features of COTS-based multiprocessors that are partially controlled by the operating system in which the OS has the capability of initializing the feature, but *not managing it post-initialization*. One such feature is the Adaptive Thermal Monitor [67] in Intel's 3rd Generation Processor Family of multicore processors. Its purpose is to reduce the power consumption and temperature of the processor core by adjusting the operating frequency and the input voltages. This mechanism is activated when the temperature of the processor exceeds some threshold. It reduces the frequency and voltage adaptively, and modulates the internal processor core clocks. The threshold temperature is factory calibrated and is not user configurable. In Intel's chips, adaptive thermal monitor protection is always enabled. This mechanism should not be confused with the thermal design process, in which the system designer needs to maintain the temperature within a safe zone.

Similarly, the Freescale i.MX31 features a CPU Load Monitor Module, responsible for tracking the actual workload. This is paired with a DVFS controller implemented in hardware, which then executes the voltage and frequency switch sequence. A similar approach is adopted by Intel with its Enhanced SpeedStep Technology or by AMD with its TPM-based Opteron 6200. While the initial setup of these features is subject to user control, it can be an issue if the employed OS or board support package has configured the use of these features as a default setting, requiring active coding to switch these off.

Existing work: The majority of the works in the field of power management [68], [69], [70], [71], [72], [73], [74], [75] and thermal-aware design [76], [77], [78], [79] for multicores in the real-time scenario use simplistic assumptions, such as no communication overhead, independent tasks, frame-based task model and simple power/thermal models, due to the complex nature of the problem. The research in COTS-based multicores assuming different hardware performance features is in its niche stages and needs further exploration to facilitate reliable deployment of industrial real-time applications.

B. System Management Interrupts (SMI)

In this section, we shall, for completeness mention briefly the effects of system management interrupts [80] , which is widely known to affect real-time performance in processors, but is generally overlooked while computing the WCET. Modern processors (Intel, ARM) have a special operating mode called System Management Mode (SMM), which is used by the firmware to manage system-wide management functions. SMM has been used to detect chipset errors, handle system

failures, such as CPU overheating, and perform fan control. The firmware runs in a separate and isolated environment and is not under the control of the OS. To enter the SMM, an SMI has to be invoked. SMIs are the highest priority interrupts in the system, non-maskable interrupts included. SMI handlers take a non-negligible time to complete (hundreds of microseconds) and hence can pose an issue for real-time applications with strict timing constraints. The SMI steals CPU cycles and modifies the CPU state. The delay incurred, additionally involves the time to save this state in the System Management RAM, later restore it, and also the time to flush the write-back caches before entering the SMM. These features impact the execution of the scheduled real-time tasks by introducing hidden latencies. If these delays are not accounted for, the estimated timing properties can be unsafe.

C. Translation Look-aside Buffer Misses

This source of unpredictability is specifically for those embedded systems that employ virtual memory, managed by a memory management unit (MMU) to protect the memory spaces of different tasks. Every virtually addressed memory access from the task is intercepted by the MMU which then translates it into the corresponding physical address. This translation adds some overhead to the access time and to reduce this delay, the MMU employs a cache, called the TLB, where recent translations are stored. In case of a TLB miss, the translation is carried out by the interrupt handler that looks-up the address mapping in the page-table which resides in main memory; this extra look-up and page-walk time to retrieve the mapping must be factored in the timing analysis as it can be non-negligible if there are many TLB misses when a task executes [81]. The challenge is the same as with processor caches – the difficulty to predict in advance: i) the number of misses, and ii) the exact instant at which TLB misses may occur. Also, in many architectures, the TLB updates are completely done in hardware and thus are transparent to the programmer. Researchers have suggested mechanisms of pre-populating the TLB with all the translations required by a task, as a part of the context-switch routine, but doing so increases the context-switch overhead and requires knowledge of all the pages that will be accessed by the task. These overheads can add up to significant delays in the presence of a large number of context switches. Also, the upper bound on the translation time must be computed to integrate these misses in the timing analysis. Another suggested method is to let the programmer handle the TLB misses explicitly and write specific interrupt handlers to update the TLB cache, which requires additional features to be supplied by the instruction set. We do not enlist all the research work here, but wish to highlight that this source of unpredictability *must* be considered in the overall timing analysis. The MMU when investigated in detail, may have other sources of unpredictability and many embedded systems that do not need strict protection across applications therefore prefer to employ a flat memory addressing.

D. Hardware Prefetching

Modern processors also provide hardware pre-fetching as a memory-latency hiding mechanism. They predict the next memory addresses to be accessed and pro-actively fetch this data from the main memory to the last-level caches based on observing memory access patterns. Processors based on the Intel NetBurst micro-architecture provide two prefetch mechanisms through the BIOS: Automatic hardware prefetch and Adjacent Cache Line Prefetch [82]. The hardware prefetcher prefetches streams of data and instructions from memory into the unified L2 cache on detecting successive L2 cache misses and a stride in the access pattern, as in accessing successive elements in an array. The Adjacent Cache-Line Prefetch mechanism, when enabled through the BIOS, always fetches two 64-byte cache lines, irrespective of whether the additional cache line has been requested or not. However, there are two main problems when real-time tasks are concerned. Firstly, the prefetch requests consume bus bandwidth and may delay important demand requests issued by real-time tasks. Secondly they can lead to cache pollution by prefetching lines that are not required by the tasks and evicting re-usable cache lines belonging to real-time tasks. When enabled, this OS-transparent prefetching can run in the background at arbitrary times, resulting in variations experienced by the currently executing tasks. Many processors, e.g. from Intel, allow programmers to disable this feature (see [82]) and it is important to do so to minimize the variations in temporal behavior.

VI. Discussion and Conclusions

Although the computing capabilities of multicores are indisputable, they must be assessed for predictability before they can be reliably trusted to host and deploy hard real-time systems. On one hand, the presence of multiple cores provides a natural temporal isolation and fault-containment mechanism provided by the hardware, while on the other, the presence of shared hardware resources with complex features that are not predictable, pose serious hurdles in timing analysis. In this paper, we highlighted these sources of unpredictability, which included the caches, memory subsystem and the front-side bus. We also explored secondary sources that often tend to be overlooked, such as system-initiated power-saving strategies, hardware prefetching mechanisms, TLBs and system management interrupts. These sources can cumulatively lead to non-negligible temporal variations unless considered. This paper also highlighted some major research in timing analysis and some open issues that must be addressed to compute tighter bounds on the timing parameters. In addition to the software (the OS and the target application), all these architectural features must be taken into account to ensure a robust and reliable deployment of hard real-time applications.

References

[1] V. Tran, D.-B. Liu, and B. Hummel, "Component-based systems development: challenges and lessons learned," in *Software Technology and Engineering Practice, 1997. Proceedings., Eighth IEEE International Workshop on [incorporating Computer Aided Software Engineering]*, jul 1997, pp. 452 –462.

[2] C. Watkins and R. Walter, "Transitioning from federated avionics architectures to integrated modular avionics," in *26th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, October 2007, pp. 2.A.1–1–2.A.1–10.

[3] M. Di Natale and A. Sangiovanni-Vincentelli, "Moving from federated to integrated architectures in automotive: The role of standards, methods and tools," *Proceedings of the IEEE*, vol. 98, no. 4, pp. 603–620, April 2010.

[4] J. Child, *Multicore Processing Becomes the New Mainstream*. [Online]. Available: http://www.cotsjournalonline.com/articles/view/101319

[5] A. H. R. Albers, E. A. L. Suijs, and P. H. N. De, "Memory-communication model for low-latency x-ray video processing on multiple cores."

[6] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström, "The worst-case

execution time problem - overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 3, 2008.

[7] S. Mohan, M. Caccamo, L. Sha, R. Pellizzoni, G. Arundale, R. Kegley, and D. de Niz, "Using multicore architectures in cyber-physical systems," *Workshop on Developing Dependable and Secure Automotive Cyber-Physical Systems from Components.*, 2011.

[8] C. Ferdinand, R. Heckmann, and B. Franzen, "Static memory and timing analysis of embedded systems code," in *Proceedings of VVSS2007 - 3rd European Symposium on Verification and Validation of Software Systems, 23rd of March 2007, Eindhoven*, P. Groot, Ed., 2007. [Online]. Available: http://www-fp.cs.st-andrews.ac.uk/embounded/pubs/papers/VVSS07.pdf

[9] A. Ermedahl, *A Modular Tool Architecture for Worst-Case Execution-Time Analysis*. VDM Verlag, 2008.

[10] Rapita Systems Ltd., "Rapitime explained," Rapita Systems Ltd., http://www.rapitasystems.com/downloads/rapitime_explained_white_paper".

[11] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand, "Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, pp. 966–978, July 2009.

[12] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm, "The influence of processor architecture on the design and the results of wcet tools," *Proceedings of the IEEE*, vol. 91, no. 7, pp. 1038 – 1054, july 2003.

[13] L. Thiele and R. Wilhelm, "Design for time-predictability," in *Design of Systems with Predictable Behaviour*, 2004.

[14] S. Basumallick and K. Nilsen, "Cache issues in real-time systems," 1994.

[15] D. Grund, "Static cache analysis for real-time systems – LRU, FIFO, PLRU," Ph.D. dissertation, 2012. [Online]. Available: https://www.epubli.de/shop/buch/Static-Cache-Analysis-for-Real-Time-Systems-Daniel-Grund-9783844216998/13092

[16] H. Ramaprasad and F. Mueller, "Tightening the bounds on feasible preemptions," *ACM Trans. Embed. Comput. Syst.*, vol. 10, no. 2, pp. 27:1–27:34, Jan. 2011.

[17] C. Ferdinand and R. Wilhelm, "Efficient and precise cache behavior prediction for real-time systems," *Real-Time Systems*, vol. 17, pp. 131–181, 1999.

[18] *Intel 4 Series Chipset Family Datasheet, For the Intel 82Q45, 82Q43, 82B43, 82G45, 82G43, 82G41 Graphics and Memory Controller Hub (GMCH) and the Intel®82P45, 82P43 Memory Controller Hub (MCH)*, Intel Corporation, 2010.

[19] J. Reineke, D. Grund, C. Berg, and R. Wilhelm, "Timing predictability of cache replacement policies," *Real-Time Systems*, vol. 37, no. 2, pp. 99–122, Nov. 2007.

[20] J. Yan and W. Zhang, "WCET analysis for multi-core processors with shared L2 instruction caches," in *RTAS '08: Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, 2008, pp. 80–89.

[21] Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury, "Timing analysis of concurrent programs running on shared cache multi-cores," in *Proc. of IEEE Real-Time Systems Symposium*, 2009.

[22] J. M. Calandrino and J. H. Anderson, "Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study," in *ECRTS*, 2008, pp. 299–308.

[23] S. Cho, L. Jin, and K. Lee, "Achieving predictable performance with on-chip shared l2 caches for manycore-based real-time systems," in *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2007, pp. 3–11.

[24] V. Suhendra and T. Mitra, "Exploring locking & partitioning for predictable shared caches on multi-cores," in *DAC '08: Proceedings of the 45th annual Design Automation Conference*. ACM, 2008, pp. 300–303.

[25] N. Guan, M. Stigge, W. Yi, and G. Yu, "Cache-aware scheduling and analysis for multicores," in *EMSOFT '09: Proceedings of the seventh ACM international conference on Embedded software*, 2009, pp. 245–254.

[26] G. Taylor, P. Davies, and M. Farmwald, "The tlb slice—a low-cost high-speed address translation mechanism," in *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*. ACM, 1990, pp. 355–363.

[27] D. Hardy, T. Piquet, and I. Puaut, "Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches," in *RTSS '09: Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, 2009, pp. 68–77.

[28] ISO26262-4, *Road vehicles – Functional safety – Part 4: Product development at the system level*, 1st ed., 2011.

[29] ISO26262-1, *Road vehicles – Functional safety – Part 1: Vocabulary*, 1st ed., 2011.

[30] IEC 61508, *Functional safety of electrical/electronic/programmable electronic safety-related systems*, 2010.

[31] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," *SIGPLAN Not.*, vol. 45, no. 3, pp. 129–142, 2010.

[32] S. Chattopadhyay, A. Roychoudhury, and T. Mitra, "Modeling shared cache and bus in multicores for timing analysis," in *Proceedings of the 13th International Workshop on Software Compilers for Embedded Systems*, 2010, pp. 6:1–6:10.

[33] A. Schranzhofer, J.-J. Chen, and L. Thiele, "Timing analysis for TDMA arbitration in resource sharing systems," in *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2010, pp. 215–224.

[34] A. Schranzhofer, R. Pellizzoni, J.-J. Chen, L. Thiele, and M. Caccamo, "Timing analysis for resource access interference on adaptive resource arbiters," in *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2011, pp. 213–222.

[35] T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury, "Bus-Aware Multicore WCET Analysis through TDMA Offset Bounds," in *Proceedings of the 2011 Euromicro Conference on Real-Time Systems*, 2011, pp. 3 –12.

[36] J. Rosen, A. Andrei, P. Eles, and Z. Peng, "Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip," in *RTSS '07: Proceedings of the 28th IEEE International Real-Time Systems Symposium*, 2007, pp. 49–60.

[37] S. Schliecker, M. Negrean, and R. Ernst, "Bounding the shared resource load for the performance analysis of multiprocessor systems," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2010, pp. 759–764.

[38] 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 Design, Automation, and Test in Europe*, 2010, pp. 741–746.

[39] D. Dasari, B. Andersson, V. Nelis, S. M. Petters, A. Easwaran, and J. Lee, "Response time analysis of COTS-based multicores considering the contention on the shared memory bus," in *IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, nov. 2011, pp. 1068–1075.

[40] D. Dasari and V. Nelis, "An analysis of the impact of bus contention on the wcet in multicores," in *IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS)*, june 2012, pp. 1450 –1457.

[41] W. Yi, "Multicore Embedded Systems: The Timing Problem and Possible Solutions," in *ICFEM*, 2010, pp. 22–23.

[42] A. Gustavsson, A. Ermedahl, B. Lisper, and P. Pettersson, "Towards WCET analysis of multicore architectures using UPPAAL," in *WCET*, 2010, pp. 101–112.

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

[44] *DDR3 SDRAM Specification*, JESD79-3E ed., JEDEC Solid State Technology Association, Jul. 2010.

[45] B. Jacob, N. G. Spencer, and D. Wang, *Memory Systems Cache, DRAM, Disk*. Morgan Kaufmann, 2007, pp. 497–520.

[46] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, 2000, pp. 128–138.

[47] B. Akesson and K. Goossens, "Architectures and modeling of predictable memory controllers for improved system integration," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, 2011, pp. 1–6.

[48] M. Paolieri, E. Quinones, F. Cazorla, and M. Valero, "An Analyzable Memory Controller for Hard Real-Time CMPs," *Embedded Systems Letters, IEEE*, vol. 1, no. 4, pp. 86 –90, 2009.

[49] J. Reineke, I. Liu, H. Patel, S. Kim, and E. A. Lee, "PRET DRAM Controller: Bank Privatization for Predictability and Temporal Isolation," in *CODES+ISSS '11: Proceedings of the IEEE/ACM international conference on Hardware/software codesign and system synthesis*, Oct. 2011, pp. 99–108.

[50] J. Dodd, "Adaptive page management," Jul. 2006, uS Patent 7,076,617.

[51] S. Goossens, B. Akesson, and K. Goossens, "Conservative Open-page

Policy for Mixed Time-Criticality Memory Controllers," in *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2013, pp. 525–530.

[52] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enabling High-Performance and Fair Shared Memory Controllers," *IEEE Micro*, vol. 29, no. 1, pp. 22–32, 2009.

[53] J. Shao and B. Davis, "A burst scheduling access reordering mechanism," in *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, 2007, pp. 285–294.

[54] S. Heithecker and R. Ernst, "Traffic shaping for an FPGA based SDRAM controller with complex QoS requirements," 2005, pp. 575–578.

[55] S. Whitty and R. Ernst, "A bandwidth optimized SDRAM controller for the MORPHEUS reconfigurable architecture," in *Proceedings of the Parallel and Distributed Processing Symposium (IPDPS)*, 2008.

[56] A. Burchard, E. Hekstra-Nowacka, and A. Chauhan, "A real-time streaming memory controller," in *Proc. of Design, Automation and Test in Europe Conference*, 2005, pp. 20–25.

[57] C. Macian, S. Dharmapurikar, and J. Lockwood, "Beyond performance: Secure and fair memory management for multiple systems on a chip," in *IEEE International Conference on Field-Programmable Technology (FPT)*, 2003, pp. 348–351.

[58] W.-D. Weber, *Efficient Shared DRAM Subsystems for SOCs*, Sonics, Inc, 2001, white paper.

[59] K. Lee, T. Lin, and C. Jen, "An efficient quality-aware memory controller for multimedia platform SoC," vol. 15, no. 5, pp. 620–633, 2005.

[60] S. Novak, J. Peck Jr, and S. Waldron, "Method and apparatus for optimizing memory performance with opportunistic refreshing," Nov. 2000, uS Patent 6,147,921.

[61] S. Biswas, "Refresh-ahead and burst refresh preemption technique for managing dram in computer system," May 1999, uS Patent 5,907,857.

[62] E. Gerchman, M. Gildea, W. Hovis, R. Jensen, W. Maule, T. Osten, and A. Wottreng, "System and method for memory self-timed refresh for reduced power consumption," Dec. 2001, uS Patent 6,334,167.

[63] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. Irwin, "Hardware and software techniques for controlling dram power modes," *Computers, IEEE Transactions on*, vol. 50, no. 11, pp. 1154–1173, 2001.

[64] G. Thomas, K. Chandrasekar, B. Akesson, B. Juurlink, and K. Goossens, "A predictor-based power-saving policy for dram memories," in *Proceedings of 15th Euromicro Conference on Digital System Design (DSD)*, 2012.

[65] *http://www.acpi.info/*, Hewlett-Packard, Intel, Microsoft, Phoenix, and Toshiba.

[66] *Desktop 3rd Generation IntelCore Processor Family Datasheet Volume 1 of 2*, Intel Corp.

[67] *Thermal Mechanical Specifications and Design Guidelines (TMSDG)*, Intel Corp.

[68] F. Gruian, "System-level design methods for low-energy architectures containing variable voltage processors," in *Proceedings of the 1st International Workshop on Power-Aware Computer Systems-Revised Papers*. Springer-Verlag, 2001, pp. 1–12.

[69] Y. Zhang, X. S. Hu, and D. Z. Chen, "Task scheduling and voltage selection for energy minimization," in *Proceedings of the 39th annual Design Automation Conference*, ser. DAC '02. ACM, 2002, pp. 183–188.

[70] R. Mishra, N. Rastogi, D. Zhu, D. Mossé, and R. Melhem, "Energy aware scheduling for distributed real-time systems," in *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '03. IEEE Computer Society, 2003, pp. 21.2–.

[71] J.-J. Chen, H.-R. Hsu, K.-H. Chuang, C.-L. Yang, A.-C. Pang, and T.-W. Kuo, "Multiprocessor energy-efficient scheduling with task migration considerations," in *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, ser. ECRTS '04. IEEE Computer Society, 2004, pp. 101–108.

[72] J.-J. Chen, H.-R. Hsu, and T.-W. Kuo, "Leakage-aware energy-efficient scheduling of real-time tasks in multiprocessor systems," in *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, ser. RTAS '06. IEEE Computer Society, 2006, pp. 408–417.

[73] J.-J. Chen and C.-F. Kuo, "Energy-efficient scheduling for real-time systems on dynamic voltage scaling (dvs) platforms," in *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, ser. RTCSA '07. IEEE Computer Society, 2007, pp. 28–38.

[74] V. Nélis, J. Goossens, R. Devillers, D. Milojevic, and N. Navet, "Power-aware real-time scheduling upon identical multiprocessor platforms," in *SUTC '08: Proceedings of the 2008 IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing*, 2008, pp. 209–216.

[75] V. Nelis and J. Goossens, "Mora: An energy-aware slack reclamation scheme for scheduling sporadic real-time tasks upon multiprocessor platforms," in *Proceedings of the 2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, ser. RTCSA '09, 2009, pp. 210–215.

[76] T. Chantem, R. P. Dick, and X. S. Hu, "Temperature-aware scheduling and assignment for hard real-time applications on mpsocs," in *Proceedings of the conference on Design, automation and test in Europe*, ser. DATE '08. ACM, 2008, pp. 288–293.

[77] N. Fisher, J.-J. Chen, S. Wang, and L. Thiele, "Thermal-aware global real-time scheduling on multicore systems," in *Proceedings of the 2009 15th IEEE Symposium on Real-Time and Embedded Technology and Applications*, ser. RTAS '09. IEEE Computer Society, 2009, pp. 131–140.

[78] W.-L. Hung, Y. Xie, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "Thermal-aware task allocation and scheduling for embedded systems," in *Proceedings of the conference on Design, Automation and Test in Europe - Volume 2*, ser. DATE '05. IEEE Computer Society, 2005, pp. 898–899.

[79] S. Murali, A. Mutapcic, D. Atienza, R. Gupta, S. Boyd, and G. De Micheli, "Temperature-aware processor frequency assignment for mpsocs using convex optimization," in *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, 2007, pp. 111–116.

[80] B. B. Brandenburg, H. Leontyev, and J. H. Anderson, "Accounting for interrupts in multiprocessor real-time systems," 2009.

[81] M. Bennett and N. Audsley, "Predictable and efficient virtual addressing for safety-critical real-time systems," in *Real-Time Systems, 13th Euromicro Conference on*, 2001, pp. 183 –190.

[82] *Optimizing Application Performance on Intel Core Microarchitecture Using Hardware-Implemented Prefetchers*. [Online]. Available: http://software.intel.com/en-us/articles/optimizing-application-performance-on-intel-coret-microarchitecture-using-hardware-implemented-prefetchers