A Survey of Techniques for Achieving Time-Predictability in Multi-cores

SRIRAM VASUDEVAN and ARVIND EASWARAN, Nanyang Technological University

The chip industry has overwhelmingly moved towards multi-core architectures due to the increased demand for processing power. However, these architectures share hardware resources among the processing cores that causes timing unpredictability. Hence, the real-time industry has not yet adopted these architectures in their systems. This survey paper summarizes the research done in obtaining predictable executions on multi-core processors. The survey paper is updated up to May 2016.

CCS Concepts:•**General and reference** → **Surveys and overviews;**•**Computer systems organization** → **Real-time systems;**

General Terms: Design, Algorithms, Performance, Experimentation, Reliability

Additional Key Words and Phrases: worst-case execution time, shared caches, shared interconnect, memory controller, network on chip, static analysis, measurement-based analysis

ACM Reference Format:
Sriram Vasudevan, Arvind Easwaran, 2016. Techniques for Achieving Time-Predictability in Multi-cores - A Survey. *ACM Comput. Surv.* V, N, Article A (January YYYY), 36 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Cyber-physical systems (CPS) are embedded computing systems in which the cyber world of computation and communication closely interacts with the physical world of sensors and actuators. These systems are abundant in many industries including avionics, automotive, healthcare, consumer applications, etc. Many applications in CPS are real-time, meaning its correctness not only depends on its output but also the time within which the operation is performed. Therefore, these systems have to meet their timeliness requirements. Real-time Applications (RTA) are classified based on the consequence of not meeting their deadlines as hard, soft and firm. Hard RTA are those in which missing a deadline will result in the failure of the entire system. Hence, these applications are safety-critical in nature. Examples of such RTA are anti-lock braking, adaptive cruise control, and inflation of airbags. Firm RTA are those in which infrequent deadline misses are tolerable but will degrade the system performance. The utility of the result after its deadline is zero. An example of firm RTA is storm forecast systems. Soft RTA are similar to firm RTA, in that infrequent deadline misses are tolerable, but the utility of the result after the deadline is not zero. Instead, it degrades as time passes after the deadline. An example of soft RTA is video conferencing.

The hardware commonly used to realize Hard Real-Time Cyber-Physical Systems (RT-CPS) are Application Specific Integrated Circuits (ASICs), which are electronic circuits customized for specific purposes. These ICs typically contain a processing ele-

Author's addresses: Sriram Vasudevan, Interdisciplinary Graduate Studies, Nanyang Technological University; Arvind Easwaran, School of Computer Engineering, Nanyang Technological University;

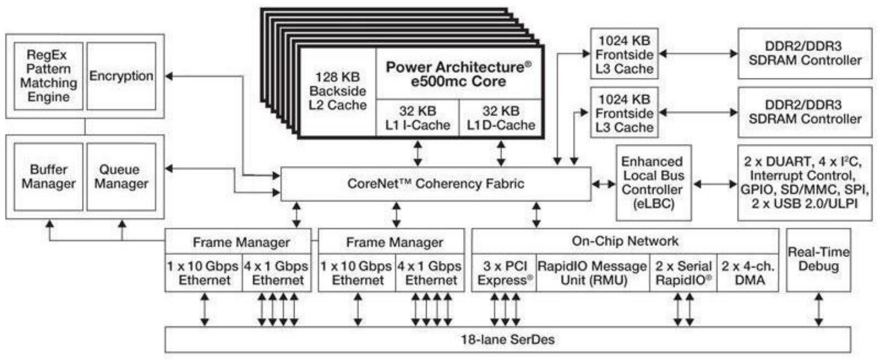
RegEx Pattern Matching Engine
Encryption
Buffer Manager
Queue Manager
128 KB Backside L2 Cache
Power Architecture® e500mc Core
32 KB L1 I-Cache
32 KB L1 D-Cache
1024 KB Frontside L3 Cache
1024 KB Frontside L3 Cache
DDR2/DDR3 SDRAM Controller
DDR2/DDR3 SDRAM Controller
CoreNet™ Coherency Fabric
Enhanced Local Bus Controller (eLBC)
2 x DUART, 4 x I²C, Interrupt Control, GPIO, SD/MMC, SPI, 2 x USB 2.0/ULPI
Frame Manager
1 x 10 Gbps Ethernet
4 x 1 Gbps Ethernet
Frame Manager
1 x 10 Gbps Ethernet
4 x 1 Gbps Ethernet
On-Chip Network
3 x PCI Express®
RapidIO Message Unit (RMU)
2 x Serial RapidIO®
2 x 4-ch. DMA
Real-Time Debug
18-lane SerDes

Fig. 1. Architecture of P4080 [p40 2010]

ment having a single core, and several peripheral management units to access I/O devices. But the complexity and number of applications in RT-CPS is always on a steady rise, as a result of which there is a growing demand for ASICs that are more powerful. For example, in a modern day car there are around 200 ASICs (called Electronic Control Units) networked together, serving a host of applications such as adaptive cruise control, anti-lock braking, battery management, collision detection and avoidance, parking assist, etc. [ecu 2010]. More powerful ASICs will help to mitigate scalability issues arising from this trend of increasing number of applications by way of consolidation. Multi-core ASICs are ICs where several processing cores are packaged into one die, and they seem like a natural choice to satisfy this demand. One of the primary advantages of using multi-cores is that the die can fit into a single package, thus providing better Size Weight and Power (SWaP) characteristics than a system with multiple single-core processors.

The architecture of P4080, a typical homogeneous Commercial-off-the-shelf (COTS) multi-core processor from Freescale Semiconductors, is shown in Figure 1. It comprises 8 cores that use shared interconnect and memory for inter-core communications (described under section 2 in reference manual [p40 2010]). The on-chip interconnect in P4080 is termed as “corenet coherency fabric”. The SWaP benefits of this processor arise from the fact that it shares a significant amount of hardware resources between the cores (caches, memory controllers, main memory, interconnect and access to input/outputs (I/O) and other peripherals).

COTS multi-cores offer significant advantages, but they are by design not suitable for use in RT-CPS. This is mainly due to the unpredictable nature of application execution on these processors. An application in RT-CPS typically comprises several tasks with timeliness requirements. To be able to meet those requirements and provide guarantees, it is then essential to know apriori the maximum amount of processing demand for each task. This maximum amount, denoted as the Worst-Case Execution Time (WCET) depends on various factors including application inputs, hardware architecture, tasks that are concurrently scheduled on the other cores in a multi-core processor, etc.

The Certification Authorities Software Team (CAST) examined multi-cores (specifically dual-core processors) and identified issues that could affect the safety and performance of any software related to aircraft systems when executed on such processors [Fa1 2014]. The paper clearly identifies shared resources as a source of interference which leads to unpredictability in estimating WCET. It also presents a list of experiments that must be performed and results that need to be shown before using a multi-core hardware for safety-critical applications. Thus, estimation of WCET on multi-cores is a major challenge for adoption of these platforms for RT-CPS.

The goal of this paper is to provide an insightful summary of the state-of-art techniques that have been developed to obtain WCET estimates on multi-cores. There have been several research efforts in the past to address this problem. These can be broadly classified into two main categories based on the aspect of the system they modify: hardware and middleware. Hardware refers to physical components that make up the processor. Middleware serves as a glue logic between the hardware and software (e.g., operating system, compilers, hypervisor, etc.). Depending on the modifications done to these aspects for WCET estimation, we classify the research efforts in this area as follows.

1. Custom hardware and unmodified middleware,
2. COTS hardware and unmodified middleware, and
3. COTS hardware and modified middleware.

The structure of the remainder of the paper is as follows. A brief introduction to COTS multi-cores, WCET analysis and challenges in estimating WCET is presented in Section 2. Sections 3, 4 and 5 discuss in detail, contributions to each research category listed above. Each section begins with an introduction to the respective research category, followed by various contributions to that category, including a description of implementation and analysis of the results. Finally, Section 6 concludes this paper and discusses the open problems and future directions.

2. BACKGROUND - MULTI-CORES

2.1. Generic COTS Multi-core Architecture

“Multi-core is a design where a single processor has the core logic of more than one processor” [Binstock 2012]. The generic architecture of a multi-core is shown in Figure 2 which has ‘n’ cores with each core having one level of private cache (local to the core and represented as private L1 cache in Figure 2). All the cores share at least one level of cache (accessible to all the cores and represented as Shared L2 cache in Figure 2). Finally, there is a system interconnect which helps in the communication between the cores and the Double Data Rate (DDR) main memory. Let us consider an example COTS multi-core, in particular, the Freescale P4080 shown in Figure 1, to understand the architectural features. The P4080 processor has eight e500mc cores, each of which is pipelined and superscalar.

The e500mc core integrates two **instruction units**, one **floating point unit** and one **load/store unit** to process data and instructions. The core also includes on-chip first-level data and instruction Memory Management Unit (MMU), a second-level unified MMU and two levels of on-chip cache memories. The instruction unit interfaces the system interconnect and L1 instruction cache and is used to buffer instructions that need to be executed. Memory unit consists of the MMU and a cache hierarchy. MMU consists of Translation Lookaside Buffers (TLB) that accelerate the virtual to physical addresses translation.

Caches are on-chip memories used by the processor to reduce the average time to access data and instructions from DDR memory. Caches may be implemented as independent or unified for handling data and instructions. Data caches are usually arranged as a hierarchy of several levels (L1, L2 and so on). The e500mc core has separate 32-KB, 8-way set associative L1 data and instruction caches. It also includes a 128-KB unified, 8-way set associative L2 cache. Whenever there is a write or read request to a location in the DDR memory, the initial step is to check the caches. If the requested content is already present in the cache, then the processor directly reads

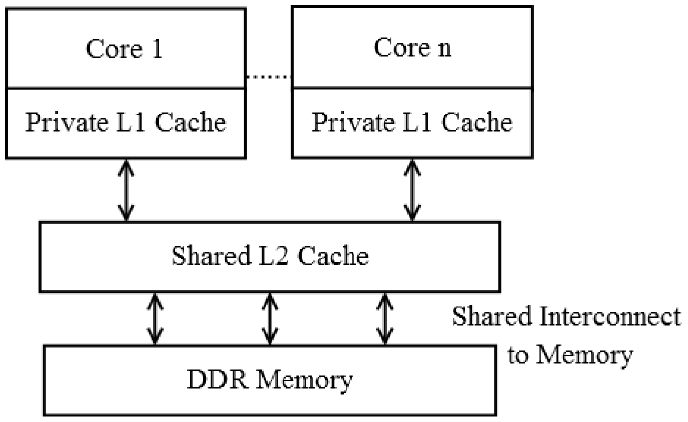

Fig. 2. Generic multi-core processor architecture

or writes to the cache. This is termed as a “cache hit”. On the other hand, when the cache does not have the data it is termed as a “cache miss”, and the requested data is brought into the cache from either the next level of caches or DDR memory. To create space for new data upon a cache miss, the caches may need to evict some existing data especially when they are full. This is referred to as the “cache replacement policy” that guides the eviction process. Many cache replacement policies have been developed such as First In First Out (FIFO), Most Recently Used (MRU), Least Recently Used (LRU), etc. [Podlipnig and Böszörmenyi 2003].

System interconnect forms the infrastructure for connectivity and enables to implement multi-core systems that are coherent. For example in P4080, it is termed as “Corenet Coherency Fabric” (CCF), which is a bus infrastructure to interconnect cores, caches, memory subsystem, I/O devices and other peripheral devices.

Multi-cores possess another level of cache beyond the system interconnect that facilitates sharing of data across the cores. For example, in P4080 this is termed as “Corenet Platform Cache” (CPC). The CPC functions as a general purpose cache and connects CCF to the memory controller. It is 1-MB and 32-way set associative. When data is shared between the cores and as a consequence there are multiple copies of data in the core specific caches, then the coherency protocol maintains consistency between these copies of data. P4080 supports MESI (Modified/Exclusive/Shared/Invalid) based coherency protocol [Papamarcos and Patel 1984].

Caches can be configured to handle data in two ways: inclusive and exclusive. For an inclusive cache, data in any lower cache level must also be present in every other higher level. On the other hand, exclusive caches do not require the data in L1 to be present in the other levels.

The **memory complex** of COTS multi-cores typically consists of DDR main memory and its controller. The purpose of a memory controller is to manage the flow of data to and from the DDR memory. It has digital logic that is required to read, write and refresh data in the DDR memory. As the number of cores increase, the controller should also manage requests from different cores by arbitrating among them. For example in P4080 two independent DDR memory controllers are connected to two DDRs each and handle requests from the eight cores.

DDR memory typically consists of a set of banks that are independent of each other. Each bank is made up of a 2-D array of rows and columns with buffers for each row and column. The basic commands that are issued to any DDR device are read, write, precharge and refresh. The time to service these requests depends on the status of the row that is intended to be accessed. If the row to be accessed is open then the request is served immediately; otherwise, that particular row has to be opened before serving the request. Requests can transfer data in bursts and the minimum size of the burst that can be transferred is called as burst-length. DDR devices have the capacity of transferring data both in the positive as well as negative clock edges, and hence a single burst of data transfer consumes time equivalent to half the burst length. If a memory request arrives and if it does not target the row that is opened, a precharge (PRE) is issued which closes the currently active row to open the row that is requested. A row in the DDR is also referred as a page. Every row has a buffer associated with it which can function with two different policies: close-page and open-page. Open-page policy leaves the row open after access to it, whereas close-page policy closes the row after completion of the access. DDRs work by holding the memory value as charge in capacitors and hence the charge drains over a period of time. This discharge is compensated by periodic refresh (REF) operations that restore it. A memory rank is a collection of DDR chips that are connected to the same chip select such that they can be accessed at the same time.

2.2. WCET Analysis

Consider the application that inflates airbags when a car is involved in an high-impact accident. For safety during a car accident, the airbags need to inflate within a specific time interval. The airbag crash sensor needs to pass information to the controller about a crash, and the controller, in turn, needs to respond by inflating the airbags. Thus this application comprises several smaller tasks that perform specific activities like sensing, control, and actuation. One of the key components in the response time of an application is the time the software program takes to execute on the allocated hardware and perform various tasks. In our example, this software program may for instance monitor the values from the crash sensors, execute a control loop to determine whether a crash has occurred, and eventually send control actions to the actuators. Further, since this application is safety-critical in nature, it is essential to have knowledge on the WCET of these tasks at design time so that the designer can allocate sufficient resources to the application. If the WCET of tasks can be accurately estimated at design time, then the worst-case response time of the safety-critical application can also be estimated.

System developers rely on either executing the task and measuring the execution time under various operating contexts or performing static code and hardware analysis, to obtain a WCET estimate at design time. Static analysis typically results in estimates that are pessimistic when compared to the actual WCET, mainly because of the complex nature of modern hardware and software. On the other hand, measurement-based approaches may have a scenario wherein the maximal observed execution time is less than the actual WCET. This is because in a real world scenario it is practically impossible to verify the entire input space and all possible hardware states for every task. Different inputs may cause a task to execute differently and hence may alter its execution time. Also, the hardware states of caches, processor pipelines and interrupts may also affect the execution time of the task. In the following two subsections, we will briefly summarize the techniques used in these two approaches for WCET estimation.

2.2.1. Static Analysis Methods. Static code and hardware analysis is a classical approach for estimating the WCET of a task that does not require the task to be executed on the hardware. Instead, it takes the task as input along with annotations from the user, finds all possible control flow paths, combines control flow and an abstract hardware model and estimates an upper bound to the WCET of the task. A typical example of this method is static cache analysis, wherein the process tries to predict the contents of cache at every instant of task execution and arrives at a particular solution for determining the statistics of cache hits and misses. This analysis is usually hard in the sense that it requires accurate knowledge of all possible task inputs that can result in a different number of cache hits and misses.

Static analysis depends on an abstract hardware architecture model to estimate hardware states during task execution. The most common hardware model for multi-core static analysis is as shown in Figure 2. Each core has at least one level of private cache; there is at least one level of shared cache and finally, a shared bus connects the processor to the DDR memory. Further, these analyses also make some assumptions on every component in the architecture. For instance, the shared bus can be assumed to use Time Division Multiple Access (TDMA) arbitration scheme or caches can be assumed to use LRU replacement policy. These assumptions are mainly for the purpose of simplifying the analysis, although some of them may not be very realistic. For instance, most COTS multi-cores do not use LRU cache replacement policy because it is very expensive (time and memory) to implement.

The steps in static analysis techniques begin by defining the processor model. After defining the processor model, the task to be executed is analyzed. A control flow analysis is performed with the help of annotations from the user. Annotations are general descriptions of memory layouts, nature of the input to programs, bounds on loops, etc. Control flow analysis determines how the control flows in the task by excluding infeasible paths or finding the frequency of execution. After analyzing the task the processor state is analyzed by performing cache analysis, pipeline analysis, branch analysis and loop analysis. Pipeline analysis tries to anticipate the task's behavior in the processor pipelines. Cache analysis tries to predict the instructions and data that would be cached when the task is executed. Loop analysis bounds the number of times a loop might get executed. Finally, branch analysis tries to predict the branch that would possibly be selected when the task is executed. When all the aspects of the analysis are completed the values are input to an Integer Linear Program (ILP) solver. This last stage of processing is called Implicit Path Enumeration Technique (IPET) analysis in which the results of all the previous stages are combined together with information about infeasible paths in the task to arrive at a WCET estimate.

2.2.2. Measurement-Based Methods. Measurement-based approaches execute tasks on the given hardware for an input state space and determine the distribution of the observed execution times. The most common approach to obtaining measurements is to execute the task repeatedly with many possible input states and instrument the code to collect timestamps for determining the execution time. Measuring execution times is also done externally by using hardware debuggers and tracer units that can collect the timestamps with minimal intrusion on the executing task. Measurements alone are not useful for safety-critical applications that require timing guarantees, as they do not consider the complete operating context such as all possible hardware states or the entire input domain. Therefore they are combined with some code analysis techniques such as value, loop bound, and path analyses to improve the confidence in the resulting WCET estimates. The main advantage of measurement-based methods over the static analysis methods is that they do not need a hardware model for analysis. This is a significant advantage because modern hardware architectures are too complex to be amenable to precise modeling. Another advantage of measurement-based methods is that the resulting WCET estimates are less pessimistic when compared to the estimates obtained by static analysis methods. However, because measurement-based methods are not guaranteed to consider the complete operating context of the task, the resulting estimates may be incorrect. Few examples of tools that implement measurement-based methods are SymTA/P [sym 2013], Rapitime [rap 2004], etc. The Rapitime tool, for instance, provides several features such as WCET estimation, identification of program code that is on the worst-case path, generation of execution time profiles to illustrate variability in code executions, and path analysis to provide an exact model of the software code structure.

2.3. Challenges in Estimating WCET on multi-cores

The challenges in estimating WCET on multi-cores have been discussed in several studies [Shekhar et al. 2012], [Shah et al. 2014], [Dasari and Nelis 2012], [Pellizzoni and Caccamo 2010], [Cullmann et al. 2010], [Shah et al. 2013], [Radojković et al. 2012], [Pellizzoni et al. 2010a] and [Bate et al. 2001]. We now summarize these challenges below:

1. Shared caches and their controllers,
2. Shared interconnect,
3. Shared memory controller,
4. Power saving strategies,
5. System interrupts,
6. TLB misses, and
7. Hardware prefetching.

In the following paragraphs we discuss these challenges briefly.

Shared caches and their controllers. All cores share at least one level of cache in a COTS multi-core as shown in Figure 1, allowing efficient sharing of data when interdependent tasks are running on different cores. Consider an example of two tasks executing in parallel on two different cores of P4080 (co-running tasks). There can exist a scenario when both contend for the same line in the shared cache, even though they share no instructions or data. This is because multiple locations in memory can map to a single cache line. In such a scenario, WCET can depend on how frequently the task accesses the cache, as every access can potentially evict useful data of the other task. If data is removed from the cache, it needs to be fetched again from the DDR memory which increases the task's execution time. Further, even if the two tasks do not access the same cache line, the cache controller which is shared between the cores can delay one of the tasks while serving the other. This also increases the task's execution time. Thus interference arising from co-running tasks due to shared caches has the potential to impact a task's WCET on a multi-core platform.

A simple strategy to overcome this interference is to partition the cache such that no two cores can access the same cache line. This approach is effective only when independent tasks are running on the different cores. But as more applications are parallelized to exploit the multi-cores better, sharing data through caches will significantly reduce the response times as the number of memory accesses decrease. Hence shared caches are important to exploit parallelism offered by multi-cores although they affect timing-predictability. Also, the contention at the shared cache controller cannot be solved by partitioning. The work by [Gracioli et al. 2015] provides a survey on all cache management techniques for real-time systems. In our survey we focus only on those techniques relevant for multi-cores.

Shared Interconnect. When there are multiple cores with a shared memory unit, there must be a common communication link that connects all these cores to the memory. This is the system interconnect (CCF in Figure 1). Although most multi-cores have high bandwidth for the shared interconnect, there exists contention as the access to this interconnect is not only dependent on the cores but also on other architectural features such as the caches and the traffic due to coherency protocol. Also, bounding this contention may be hard as the chip manufacturers do not provide extensive details on the architecture of the interconnect or the arbitration mechanism it uses.

Analysis of the shared interconnect depends on accurate knowledge of the interconnect architecture. When there is a lack of information, WCET estimation is performed using assumptions that may or may not be realistic. For instance, the arbitration scheme for the shared interconnect is assumed to be Time Division Multiple Access (TDMA) in many methods.

When a system maintains both private and shared memory resource such as caches, any change of data at any location needs to be updated in every other memory resource where a copy of this data is held. The coherency mechanism keeps all these copies in harmony with each other by continuously updating every change. Multiple cores updating shared data in their respective local caches can cause heavy coherency traffic on the interconnect, which in turn can cause a delay in the execution of applications.

Shared Memory Controller. The DDR memory controller (two 64-bit DDR2/DDR3 memory controllers shown in Figure 1) is essential for controlling the interaction of the cores with the memory units. The number of memory controllers is usually lesser than the number of cores, primarily because of packaging constraints that is limited by the number of pins available [Abts et al. 2009]. For example, in the P4080 processor, there are 8 processing cores. This uneven distribution of the number of cores to the controller itself is a reason for contention. Whenever there are multiple cores simultaneously trying to access the memory, there may be contention. Also the arbitration scheme employed by COTS controllers is mostly unknown, which increases the difficulty in the analysis of such controllers for WCET estimation.

Power Saving Strategies. Most modern processors have reached operating speeds that are almost saturated and are now primarily focused on reducing the power consumption. Hence a separate system is dedicated to implementing power saving strategies for the processor. One of the simplest strategies used to save power is to change the frequency of the processor. Reducing the frequency naturally reduces the operating voltage and hence the total power consumed. Such optimization techniques may cause unpredictability in the execution of tasks on the cores, and therefore can impact WCET estimation.

System Interrupts. A system might have interrupts that cannot be prevented and these interrupts can impact task executions. A simple scenario is when a processor heats up, it may either be turned off or taken to a system management mode that would bring it back to normal execution. This consumes processor cycles and hence must also be taken into account when estimating the WCET.

TLB misses. Virtual memory is an important aspect of most multi-cores. A dedicated Memory Management Unit (MMU) is assigned to translate these virtual memory addresses to physical ones. The cores refer to virtual memory addresses which in turn are translated to physical addresses by the MMU. MMU has a cache called Translation Lookaside Buffer (TLB) that stores the recent translations. The flip side of using this buffer is that if there is a TLB miss, the translation request needs to be sent to the DDR memory and then mapped back. This would incur additional cycles in the execution of a task. Since this cache is shared, it becomes hard to predict the eviction of translations from different cores. These evictions cause unpredictability and therefore also impact WCET.

Hardware Prefetching. Prefetching allows the processor to fetch blocks of instructions and data before the task actually requests for them. This is generally due to speculative techniques such as branch prediction and cache block prefetching. Speculation may on an average improve the execution time of tasks if it manages to predict the future correctly on a regular basis. But it also negatively affects WCET of tasks,

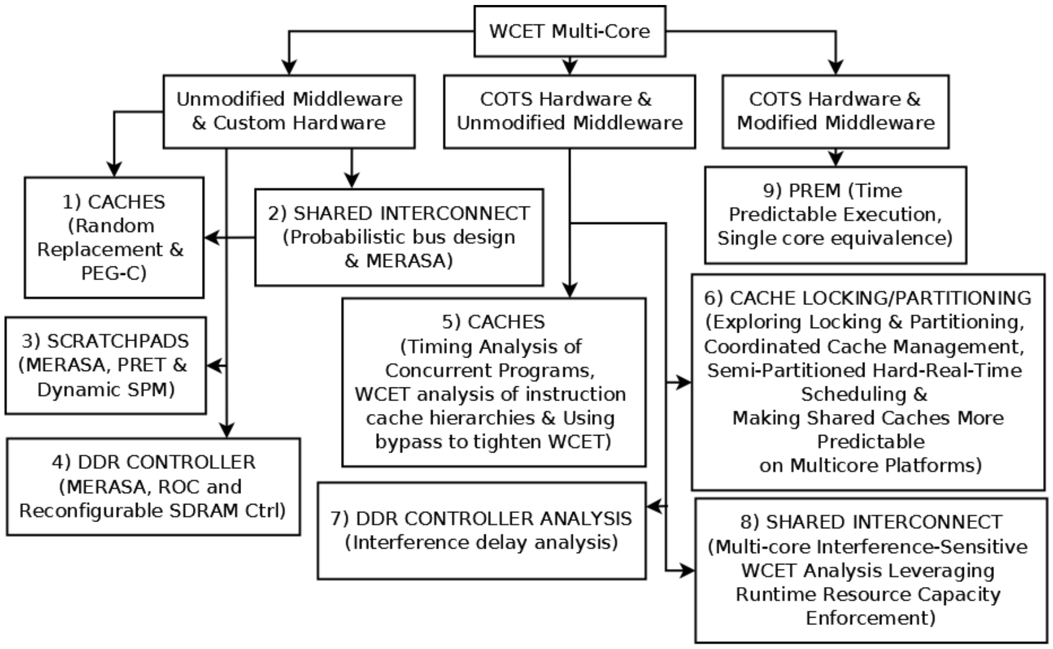
WCET Multi-Core
Unmodified Middleware & Custom Hardware
COTS Hardware & Unmodified Middleware
COTS Hardware & Modified Middleware
1) CACHES (Random Replacement & PEG-C)
2) SHARED INTERCONNECT (Probabilistic bus design & MERASA)
3) SCRATCHPADS (MERASA, PRET & Dynamic SPM)
4) DDR CONTROLLER (MERASA, ROC and Reconfigurable SDRAM Ctrl)
5) CACHES (Timing Analysis of Concurrent Programs, WCET analysis of instruction cache hierarchies & Using bypass to tighten WCET)
6) CACHE LOCKING/PARTITIONING (Exploring Locking & Partitioning, Coordinated Cache Management, Semi-Partitioned Hard-Real-Time Scheduling & Making Shared Caches More Predictable on Multicore Platforms)
7) DDR CONTROLLER ANALYSIS (Interference delay analysis)
8) SHARED INTERCONNECT (Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement)
9) PREM (Time Predictable Execution, Single core equivalence)

Fig. 3. Work towards predictable WCET

because for scenarios when the speculation is wrong, the resulting execution time also includes the misprediction penalty.

2.4. Research towards Time-Predictability on Multi-cores

As mentioned in Section 1, we classify the existing research towards obtaining a WCET estimate for tasks on multi-cores into three categories (described in Figure 3) based on the aspect of the system that it intends to modify. These categories are as follows:

1. Custom hardware and unmodified middleware,
2. COTS hardware and unmodified middleware, and
3. COTS hardware and modified middleware.

The first category, *custom hardware and unmodified middleware* focuses on building custom components of the multi-core architecture on Field Programmable Gate Arrays (FPGAs), without making any changes to the middleware components. Specifically, this category includes research on designing predictable caches, memory controllers and system interconnects on FPGAs. Projects such as MERASA [Paolieri et al. 2013], CoMPSOC [Hansson et al. 2009] and parMERASA [par 2012] fall into this category. The primary research problem this category addresses is the modifications required at the hardware level to estimate WCET with minimum pessimism, while assuming that the application and the middleware components remain unchanged. Note that although these studies use FPGAs to illustrate their hardware designs, the final designs could always be transformed into ASICs. Within this category, we classify the contributions into four subcategories based on the hardware component that is designed. These subcategories are caches, scratchpads, memory controllers, and system interconnects.

The second category, *COTS hardware and unmodified middleware* focuses on research that develops WCET estimation techniques for COTS multi-cores. This category of research includes studies on cache analysis, interconnect analysis, pipeline analysis, etc. Although these methods do not require any modifications to the hardware or middleware components, they make simplifying assumptions on the architecture that may or may not be valid for COTS platforms. The primary studies of interest concerning this survey are those that provide specific solutions to overcome the challenges in analyzing multi-cores. We further subdivide this category of research into four subcategories based on the aspect of hardware that the work analyzes. These subcategories are cache analysis, cache locking and partitioning, DDR controller analysis and shared interconnect analysis.

The final category of research *COTS hardware and modified middleware* does not require any modifications to the hardware (i.e., considers COTS multi-cores), and instead focuses on modifying the middleware components to obtain predictable WCET estimates. An example of work in this category is the implementation of fork and join operation within the OS where a task is segmented into smaller subtasks and executed in components to obtain predictable execution pattern.

In the following three sections, we discuss each category in detail. For each study, we present a brief introduction to the work and explain how it addresses the WCET estimation challenges. Thereafter we present some implementation details of the study and discuss experimental results.

3. CUSTOM HARDWARE AND UNMODIFIED MIDDLEWARE

This section presents the state-of-art in building custom hardware to provide predictable task execution. It presents the various techniques employed in designing hardware architectural features of the multi-core specifically caches, memory controller, and system interconnect.

3.1. Caches

In this subsection we discuss the various solutions presented for designing predictable caches in hardware (box 1 in Figure 3).

3.1.1. Probabilistic Caches. The source of unpredictability in shared caches on multi-cores is its replacement policy. This work introduces the Random Placement (RP) policy along with the existing Random Replacement (RR) to estimate the WCET of a task which is associated with an occurrence probability [Kosmidis et al. 2013]. The advantage of this policy is that every memory access is associated with a probability of hit or miss independent of the history of memory accesses. This probability depends on the total number of cache lines and also its associativity. By randomizing the placement of data in the cache, this approach computes the probability of hits and misses for every memory access in the task and obtains a probabilistic WCET estimate [Kosmidis et al. 2013].

Implementation: Hardware implementation of RP requires a Pseudo-Random Number Generator (PRNG) and a hash function. The hash function takes a memory address and a random number (provided by PRNG) and provides a cache mapping. To reduce the complexity of the hardware, the traditional cache design is kept unmodified with the PRNG and hash function implemented as separate units outside the cache.

Experiments and results: Experiments are performed using SoCLib simulation framework with binaries of powerpc [Consortium et al. 2008]. A processor with 4-stage pipeline and a memory hierarchy with separate L1 data and instruction cache and a DDR memory is selected for experiments. Both caches are set associative and implement the RR and RP policies. The experiments are run with EEMBC benchmark suite [Poovey 2007]. The performance of RR+RP cache is compared against modulo placement + LRU replacement cache. For 1 way and 256 set cache (direct mapped), the instructions per cycle for RR+RP cache is 0.234 and LRU+modulo cache is 0.613. As the associativity increases, the performance of RR+RP cache matches that of LRU+modulo cache. The WCET of set associative cache is also compared to that of a cache which has direct mapping. Although a fully associative random replacement cache provides the lowest WCET values, the RP+RR design offers the best trade-off between hardware complexity and WCET.

3.1.2. Performance Enhancement Guaranteed Cache (PEG-C). A hardware cache implementation designed by [Huangfu and Zhang 2014] improves the average-case performance while providing predictable access to caches. This is achieved by counting the number of cache hits and misses along with a cache preloading algorithm. The cache preloading algorithm loads the cache with instructions to avoid the initial cold misses. The counters record at run-time the statistics of hits and misses and provide access to the cache only when the difference between hit and miss counters is positive. Whenever the difference is not a positive value, the memory accesses are taken to the DDR while still loading the cache until the difference becomes positive.

Implementation: The architecture uses existing cache design with two counters to monitor the statistics of cache hits and misses. To avoid cold misses the cache blocks are preloaded with data by exploring all possible paths in the task. Once a task starts to execute the counters are incremented based on hits and misses that occur. When the value of the miss counter is greater than the hit counter, the cache is disabled and every access is directed to DDR memory. Simultaneously, the cache is loaded with the data that is accessed. The hardware remains in this state until the difference between hits and misses becomes positive.

Experiments and Results: Trimaran simulator has been used to implement and test the architecture with Mediabench benchmark [Lee et al. 1997] [Chakrapani et al. 2004]. The experiment to show the performance benefit of using this cache design measures the minimum number of instructions required to prevent the cache access from being disabled. From the experiments performed it is observed that in most benchmarks the minimum number of instructions required is 16. Another experiment to compare the performance of the proposed architecture with locked cache is performed. It is observed that the proposed architecture provides significant performance benefit when compared to cache locking for all the benchmarks.

3.1.3. Discussions. This section discussed hardware solutions that address the unpredictability in the access to shared caches. Two studies on building cache architectures on FPGAs have been reviewed. One of them presents a probabilistic approach [Kosmidis et al. 2013], whereas the other study proposes a cache architecture with hit and miss counters [Huangfu and Zhang 2014]. The drawback of using RR and RP policies in RTA have been identified by Reineke [Reineke 2014]. It shows that the probability of hits that are computed for the analysis are not independent. This means that the convolution of different execution time values obtained is not possible with randomized caches and hence is not favorable to be used in RTA. Although these studies contribute to building a predictable cache architecture, their implementations are limited to only a few aspects of the cache architecture. From the challenges described in Section 2.3, we can see that the unpredictability from the shared cache controller and coherency mechanism have not been addressed by any of these studies. To obtain predictable accesses to shared caches, these problems must be resolved.

3.2. ScratchPad Memories (SPM)

To avoid the unpredictability arising from using shared caches, researchers have proposed not to use them and instead have local faster memory termed as scratchpads. These are lightweight, low-power and high-speed memory units that reside close to the processor. There are two methods by which functions are loaded into the scratchpads. They are static and dynamic. This subsection discusses the various contributions towards predictable data loading into scratchpads (box 3 in Figure 3).

SPMs were identified as a hardware alternative to caches for embedded systems by Banakar et.al. [Banakar et al. 2002]. This work motivates the use of SPMs by providing a basic architectural implementation and comparing caches and SPMs of different sizes with respect to area, energy and performance (SPMs performs better than caches on all comparisons). However, this work does not explicitly address the major challenge in using SPMs, which is to decide what content will be loaded into this memory and when will it be loaded. A comparison of scratchpads and locked caches was provided by Puaut et.al. [Puaut and Pais 2007] to identify the scenarios that would degrade the performance of each of these memories. The study concludes that SPM performance degrades whenever there are larger functions (function size still less than the size of the scratchpad) to be loaded and hence there is not enough space to load more functions or the function itself is larger than the scratchpad's size. On the other hand, caches with larger line size suffer from a problem where many unused data gets locked in the cache and hence the space for useful data reduces.

3.2.1. Static SPMs. Static SPMs have fixed data allocation, i.e., their contents cannot be changed at run-time. The work by Suhendra et. al. [Suhendra et al. 2005] proposes three techniques to allocate data statically to the SPM. First is to formulate an ILP which considers all possible program paths and obtains the data to be loaded. Second is to use branch-and-bound search algorithm to find the best possible allocation by eliminating those paths which are infeasible in a task execution (this method is expensive due to the exhaustive search involved as the number of tasks increase). Finally, the third method is to reduce the complexity by using a heuristic where infeasible paths in the program are removed and then from the remaining paths, the one which leads to WCET (technique is discussed in [Chen et al. 2007]) is considered and data is allocated from it. This step is repeated till the SPM is fully filled. Another study by [Takase et al. 2010] proposes three different static loading scenarios. First is where every task has a partitioned memory space in the SPM. Second is where an executing task uses the entire SPM, and finally, the third where hard real-time tasks can use the space of non-real-time tasks. An ILP is formulated for each case which decides both on partitioning the SPM and also loading data into it.

3.2.2. Dynamic SPMs. Dynamic SPMs have data allocation that is managed at run-time. A Dynamic Instruction SPM for multi-cores was proposed by [Metzlaff et al. 2011]. The primary idea of this work is that whenever there is a function call, the pipeline is stalled and all the instructions of that function are loaded into the scratchpad before execution. This idea makes sure that there are no more DDR memory accesses performed during the function execution which avoids any unpredictability arising due to other architectural features of the processor.

Another study targets a dynamic SPM that uses a memory management unit to manage the loading of data into the dynamic partitions in the SPM [Wasly and Pellizzoni 2013]. The OS partitions the SPM into three segments. The first segment is of fixed size and is used to hold the data related to the OS itself. The second and third partition sizes can be dynamically varied by the OS based on the size of the function being loaded into the SPM.

The work by [Liu and Zhang 2012] explores different configurations of the SPM with regards to using it with multi-core architectures. In particular, it compares single against two-level SPM, unified against separate, and dynamic against static loading and obtains the merits and demerits of each scenario. Conclusions drawn from this work are that multi-level SPMs reduce WCET in most cases, separate SPMs provide

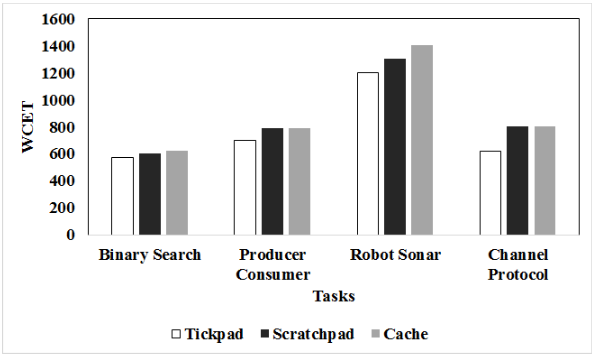

Fig. 4. WCET comparison of memory modules [Kuo et al. 2013]

better performance than unified and finally, dynamic loading helps in using the SPMs more efficiently to manage run-time demands.

The study by [Kim et al. 2014a] proposes two dynamic loading techniques for scratchpads in multi-core architectures where the entire execution depends on contents in the SPM (DMA is used to load the SPM with functions from DDR whenever they are requested). The first technique is to find the optimal method to map functions to different portions of the SPM. An ILP is formulated which iteratively determines the best possible function mapping using a control flow graph of the program to be executed. The disadvantage of this method is the exponential complexity as the number of functions and size of SPM increases. A simpler heuristic (reduced complexity) has been proposed where two different strategies are used to obtain the mapping. First is by mapping different functions to separate spaces on the SPM and then merging them to fit inside the SPM. Second is to map only one function to the SPM and then follow an iterative process to partition the SPM space to different functions.

3.2.3. TickPad Memories (TPM). Another solution presented to address the unpredictability in shared caches is to use tickpad memories [Kuo et al. 2013]. Tickpads are identical to caches and scratchpads as they provide quick access to memory, but are only used with synchronous programming languages such as Esterel [Berry and Gonthier 1992] or Pret-C [Roop et al. 2009].

Implementation: A synchronous language program is analyzed and a graph is generated which provides information on concurrent control flow and context switching points [Kuo et al. 2013]. Using this graph, data allocation decision to the tickpad is obtained. With this information, the memory access time is computed and the execution time of the program in the worst-case is estimated.

Experiments and results: A hardware synthesis of tickpad memory is provided and its performance is compared to scratchpads and caches. The benchmarks selected for experiments are both synthetic and standard (crc and binary search). The work presents a comparison of worst-case time to access the tickpads against scratchpads and direct mapped caches. From the results shown in Figure 4, it can be seen that the WCET for a tickpad is always lower than either of these alternatives due to its simpler architecture.

3.2.4. Comparison of Memory Modules. The inequality between the clock speed of a processor and the time to access DDR memory requires the presence of a memory unit that is not just faster, but also closer to the processor. The initial solution was to have a cache architecture that is on-chip so that the delay in accessing data/instructions is reduced. Caches are designed having multiple levels with at least one of them shared among the cores. This required coherency protocol to maintain the consistency of data in the caches, that lead to an increased processing overhead. Also, sharing of caches created the possibility of useful data of one core being evicted by another. To avoid these complexities SPMs were introduced. Data allocation and its replacement on SPMs are software guided and decisions are taken at compile-time. This improved their average-case and worst-case performance when compared to caches. Unfortunately, SPMs do not allow sharing of data between the cores, which is a key requirement to exploit the parallelism of multi-core architectures in many applications. In the absence of efficient sharing mechanism, all data sharing must happen through DDR memory, which can be very time-consuming. Tickpads offer a trade-off between caches and SPMs by providing dynamic loading capabilities that are statically controlled. This allowed them to perform better than SPMs, but they still do not provide support for efficient sharing of data between the cores. Thus, the open problem in this category would be to develop memory units that would be predictable and still offer efficient data sharing capabilities.

3.3. Memory Controllers

Another hardware resource that is shared in multi-cores is the memory controller. This unit is a digital circuit that controls the data flow in and out of the DDR memory. The bottleneck in obtaining predictable access to memory is hampered due to lack of information about the architecture of COTS memory controllers and multiple cores sharing fewer controllers. This poses a challenge in the estimation of WCET. In the following sub-sections, we shall discuss various contributions towards designing predictable memory controllers (box 4 in Figure 3). Note that the Joint Electron Device Engineering Council (JEDEC) provides a set of operational timing standards that device manufacturers need to adhere while manufacturing DDRs. Hence the controllers designed for these DDRs must also comply to these timing standards.

3.3.1. Analyzable DDR Controller. The work by [Paolieri et al. 2009] proposes a real-time controller implementation which performs round-robin arbitration between the different cores trying to access the DDR memory. This prevents the unpredictability in memory access due to inter-core interference. Interference delay is upper bounded using an exhaustive analysis based on the timing constraints provided by JEDEC standards. As an additional feature, to avoid interference between the different tasks executing on the same core, this implementation provides one buffer per task.

Analysis and Implementation: This implementation analyzes the tasks and obtains an upper bound for every memory request. To bound the delay experienced by a memory request, this work considers every other request that was previously issued (based on whether it is a read or a write request). For tasks executing on different cores, this work bounds the delay based on the round-robin arbitration policy. To account for the delay due to DDR memory refresh operation (predictable refresh), the start of the task execution is synchronized with the occurrence of a refresh. The work models a memory controller for a 4-bank, 256MB x 16 DDR memory and simulated using DRAMsim framework [Wang et al. 2005].

Results: WCET estimates are determined for different scenarios such as 1) shared resource interference with private memory controller, and 2) controller shared between real-time task with memory and on-chip interferences. The collision avoidance algorithm from Honeywell Corporation is used for testing the design. This design computes tight WCET estimates when compared to the estimate by Rapitime [rap 2004], when the algorithm is executed in its highest memory demanding workload.

3.3.2. Reconfigurable Real-time DDR Controller. A reconfigurable real-time DDR controller was proposed by Goossens et.al. [Goossens et al. 2013]. Since the architec-

Table I. Real-time Memory controllers

Controller	**Page policy**	**Arbitration Policy**	**Rank Scheduling**	**Bank Mapping (Interleaving vs Private)**	**Critical/ Mixed critical**
Predator	Close row	CCSP	-	Interleaved	Critical
Reineke	Close row	TDM	-	Private bank	Critical
Wang	Open row	Round robin	Rank hopping	Interleaved	Non-critical
Yonghui Li	Close row	Round robin	-	Interleaved	Mixed critical
Goossens	Cons ervati ve open row	TDM	-	Interleaved	Mixed critical
Leonardo	Close row	Fixed priority	Rank switching	Private bank	Mixed critical
Akesson	Close row	CCSP	-	Interleaved	Mixed critical
ROC (pellizioni)	Open row	Round robin	Rank switching	Private	Mixed critical
Hokeun Kim	Close/Open	Fixed Priority	-	Private	Mixed critical
Analyzable Memory controller	Close row	Round robin	-	Interleaved bank	Critical

ture is reconfigurable, a trade-off has been obtained between the provided bandwidth, response time for a memory request and power consumption. The controller uses TDMA arbitration (this module is also reconfigurable at run-time). This protocol allows changing the allocation of TDMA slots at run-time without losing out on timing-predictability.

Implementation: A System-C level implementation of the controller is presented. The controller architecture front-end is divided into three blocks of which the first block splits the requests into smaller fixed size requests, the second one groups the smaller size requests and converts it to the size the back-end needs and finally buffers to hold the output from the front-end. A shared interconnect connects this front-end to the physical layer (PHY) which forms the back-end that physically connects the controller to the DDR.

Analysis: The proposed architecture is predictable due to the implementation of two important features. First is the grouping of memory requests into predefined memory command patterns to eliminate interference between memory requests. Second is the use of TDMA arbiter for predictably switching between the different cores. The controller in this implementation translates every memory request into a set of DDR command patterns. The different patterns considered here are read, write, refresh, idle and switch patterns. Switch patterns are no-operation commands that are required to balance read and write pattern lengths. This work fixes the pattern length to be constant and based on the scheduling mode, the maximum delay of every memory request is obtained.

3.3.3. PRET DDR Controller. Another controller design that not only focuses on predictable memory accesses but also improves the worst-case latencies was proposed by Reineke et.al. [Reineke et al. 2011]. The key aspect of this implementation is that it considers the DDR memory as multiple devices that may be shared among several clients individually. It exploits the parallelism in the DDR memory by segregating the physical address space of the device in accordance with ranks and banks. Refreshing the DDR is performed by row accesses rather than actually sending a refresh command.

Implementation: Banks of the controller are issued with independent commands to exploit bank-level parallelism. Closed-page policy is followed wherein the accessed row is closed (precharged) before performing the next access. The front-end has buffers that queues the requests from different cores. The back-end implements a modulo-13 counter that provides access to the request buffers every 13 cycles. Meanwhile, the back-end also has circuits that generate different DDR commands based on the request from the front-end. Isolating the back-end with parallel access to multiple banks of the DDR and controlling the refresh operation provides predictability in this implementation.

3.3.4. Open Row DDR Controller. A predictable controller design by Krishnapillai et.al. [Krishnapillai et al. 2014] focuses on using rank switching and partitioning mechanism. Rank switching increases the utilization of the DDR and avoids the problem of write-read transition latency. Rank partitioning provides isolation by assigning banks in the ranks to specific cores thereby providing independent accesses to the banks.

Implementation: The memory controller is partitioned into front and back-ends but the focus of this implementation is the back-end. The back-end of the controller has a three level arbitration scheme. Level 1 arbiter prioritizes reads and writes over precharge and activate commands. Level 2 arbiter alternates between the different ranks and level 3 arbitrates among the cores within a rank. This 3-level arbitration provides predictable access to the memory.

Results: The reads and writes are compared with the memory controller proposed in [Paolieri et al. 2009]. It is observed that this design of controller on an average provides 33% lower latency than the memory controller in [Paolieri et al. 2009]. The WCET of a task for a 4 rank DDR device produces 5-35% lower latency than the controller in [Paolieri et al. 2009].

3.3.5. PREDATOR. Another predictable controller design (PREDATOR), which combines the properties of both static and dynamic controllers was proposed by Akesson et.al. [Akesson et al. 2007]. The unique feature of this controller is that instead of sending independent commands to the DDR, this controller groups and sequences several DDR commands together. These are read, write and refresh groups. Predictability is obtained by scheduling these groups of commands with a known arbitration scheme at run-time.

Implementation: The memory controller consists of a network interface with request and response buffers. This is connected to another module which has the arbiter, command generator, and memory mapping. The arbiter uses predictable arbitration schemes such as weighted round-robin or fair queuing to arbitrate among different cores. The memory mapping unit converts the incoming logical addresses to physical DDR addresses. The command generator is used to generate the DDR commands in accordance with the access group that is being scheduled.

3.3.6. Comparison of Memory Controller Designs. The unpredictability in memory controllers described in Section 2.3 has been addressed by the studies presented in the above subsections. One of the common features in all these memory controller designs is the use of TDMA or round-robin arbitration among the real-time tasks and cores. A few of these works ([Goossens et al. 2013] and [Krishnapillai et al. 2014]) rely on either generating composable patterns (patterns of DDR commands) or analyzing the task to obtain the nature of memory accesses (open-row or closed-row) that can increase or decrease the overhead in the memory controller operation.

Considering the state-of-art multi-cores that are clocked at several GigaBytes per second, we not only need real-time guarantees with respect to timing behavior but also increased performance to satisfy the growing demands in processing. Further, most of these studies are simulations and hence there is a need for implemented designs. Finally, the designed memory controller should be scalable with respect to the number of cores, i.e., as the number of cores increase, the memory controller should be able to provide similar performance and real-time guarantees for the system. Another aspect to fulfill is the need for composable architectures which can reduce the pessimism at the hardware level in estimating the WCET. Composable architectures are those where the execution of an application remains unchanged irrespective of the presence of other applications (inter-core interference problem mitigated). This essentially means that when there are shared hardware resources, every shared resource must be able to guarantee similar service for all applications in the system such that every application can be analyzed independently [Rumpler 2006] [Hansson et al. 2009].

3.4. Interconnect Designs

Another source of interference in multi-cores is the system interconnect that is shared among all the cores. In this subsection, we discuss the various designs of interconnect (box 2 in Figure 3) and how they address the WCET estimation challenge that was discussed in Section 2.3.

3.4.1. Probabilistically Analyzable Interconnect. Probabilistically analyzable interconnect designs for multi-cores was proposed by Jalle et.al. [Jalle et al. 2014]. The design obtains probabilistic WCET estimates using two arbitration policies: 1) Lottery arbitration and 2) Randomized-permutation. By randomizing the access to the bus, this work associates a probability with every memory access and estimates WCET.

Implementation:

Lottery arbitration: In this arbitration technique, the Pseudo Random Number Generator (PRNG) generates a random number using which access is provided to one of the cores. *Random Permutation:* Unlike the previous arbitration technique, in this the PRNG generates a random permutation for all the cores that are contending for interconnect access.

Experiments and Results: For the proposed bus designs, execution times are observed for programs from the EEMBC benchmark [Poovey 2007]. From the WCET estimate it can be observed that the reduction in the WCET range with respect to deterministic arbitration (round-robin) is between 3.5% and 6.7% for lottery arbitration, and between 1.5% and 9.6% for random permutation arbitration.

3.4.2. Predictable System Interconnect. The architecture of MERASA described in Paolieri et.al. [Paolieri et al. 2013] proposes an interconnect design as shown in Figure 5, which follows a fixed pattern for cores to access the DDR memory. The design has a two-level arbitration scheme: inter-core (arbitrates between different cores) and intra-core (arbitrates within a core). The intra-core real-time arbiter has buffers for each bank, wherein the memory requests from each core are sorted based on the bank that it needs to access. For real-time tasks, First-In-First-Out (FIFO) arbitration scheme is used and for all the other tasks, parallel out-of-order execution that targets different banks is used. When there are multiple real-time tasks from different cores trying to access memory simultaneously, a round-robin arbitration (inter-core arbiter) is performed to provide predictable access to the interconnect.

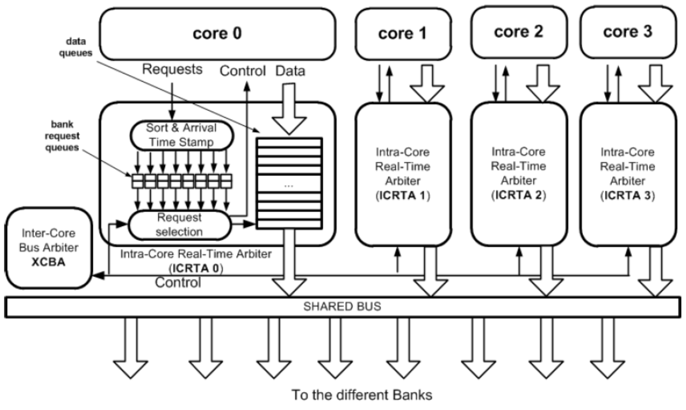
core 0
core 1
core 2
core 3
data queues
Requests
Control
Data
bank request queues
Sort & Arrival Time Stamp
Request selection
Intra-Core Real-Time Arbiter (ICRTA 0)
Intra-Core Real-Time Arbiter (ICRTA 1)
Intra-Core Real-Time Arbiter (ICRTA 2)
Intra-Core Real-Time Arbiter (ICRTA 3)
Inter-Core Bus Arbiter XCBA
Control
SHARED BUS
To the different Banks

Fig. 5. Merasa - Real Time Interconnect [Paolieri et al. 2013]

3.4.3. Predictable Network on Chip (NoC) Designs. The previous two subsections discussed the contributions towards a predictable system interconnect. In this subsection, we discuss various contributions to NoC designs, which can be regarded as generalized interconnect designs. System interconnects are synchronous and allow one transaction at a time, have a central arbitration scheme, limited bandwidth, and low cost. NoCs on the other hand forms the communication subsystem on an integrated circuit. It is different from a system interconnect from the fact that it applies networking principles and improves scalability and power efficiency of system designs. It allows pipelining of requests, different architectures, transaction re-ordering, etc.

A summary of different NoC designs and its scalability was studied by Goossens et.al. [Goossens and Hansson 2010]. NoCs have a constant performance to cost ratio, i.e., the costs linearly depend on bandwidth and latency. The study concluded that Time Division Multiplexed (TDM) router is the best-suited arbitration scheme that is predictable, composable and also low cost to implement for NoC designs. The potential future directions of work for NoCs are to improve its reliability and quality of service, provide guaranteed performance, reduce complexity and increase portability of the design.

A circuit switched NoC based on TDMA arbitration to be used in RTA was proposed by Schoeberl et.al. [Schoeberl et al. 2012]. This work introduces a fast and efficient (reduced complexity) hardware NoC design. It formulates an ILP to find a schedule that provides equal bandwidth between all the cores connected to the NoC.

TDMA static scheduling assures guaranteed bandwidth and a fixed latency to transmit data over the NoC. This simplifies WCET estimation. In this design, the routers keep the schedule of the packets that are sent and hence collisions are avoided. The router consists of multiplexers and registers that are compatible with a logic cell of the FPGA. The output of the routers can buffer a single packet and the input is fed by the multiplexer design.

A time-predictable TDMA memory arbiter for multi-cores was studied by Schoeberl et.al. [Schoeberl et al. 2014]. Each core has a local interface to the NoC and the NoC itself has an interface with the memory through a memory controller. The local interface at the core has a local TDMA schedule. Whenever there is a memory request from the core and the local interface obtains a TDMA slot, an acknowledgment is sent to the core and the request is sent to the memory controller interface. At this interface, the request is served using a handshaking policy. After the request is serviced, a broadcast message is sent with the core identity tag to return the data back to the core.

3.4.4. Summary of Shared Interconnect Designs. Above subsections presented solutions that address the unpredictability in the access to the shared interconnect. The interconnect designs focused on either adapting the existing interconnect architecture or generalizing this architecture to NoCs. The interconnect architecture based studies either focused on a probabilistic analysis where the access to the interconnect is randomized or used a two-level arbitration scheme that manages arbitration within and between the cores. With the introduction of System on Chip, NoC designs have been extensively favored. The advantages of NoCs over the system interconnect design are its bandwidth, cost, and arbitration. NoCs for real-time systems has primarily focused on TDMA arbitration. NoCs also provide a significant advantage in scalability of their design, along with different possible communication flows between the cores and memory.

3.5. Discussions

The previous section discussed various solutions to address the challenges in WCET estimation specifically in shared caches, controllers and interconnect. We now discuss the practicality of these solutions and their contribution towards providing effective solutions for WCET estimation.

Caches vs Scratchpads vs Tickpads: In general, caches are built to perform exceptionally well on an average-case for reducing memory access times. From the analysis of caches by Kosmidis et.al. [Kosmidis et al. 2013] and Zhang et.al. [Huangfu and Zhang 2014], it can be concluded that caches are predictable when they are built with Least Recently Used (LRU) replacement policy. Contradicting this fact, caches in most COTS processors do not employ LRU. Instead, the implementation is pseudo-LRU, which is less expensive to implement than LRU.

Table II outlines the functional differences in implementing the various memory architectures that were discussed in this category; namely, shared caches, scratchpads, and tickpads. Shared caches are always efficient for average case performance, while not providing worst-case predictability. On the other hand, implementing scratchpad memories provides fewer features with comparatively lower cost compared to caches, but offers very high worst-case predictability. The void is filled by tickpad memories that are loaded dynamically like caches, but the decision on what content to load is taken statically. However, tickpads are designed only for synchronous languages like Esterel and Pret-C.

Flexibility of the hardware designs: The proposed hardware modifications focus on either the shared cache, memory controller or the shared interconnect. In this paragraph, we discuss how these hardware designs could be combined to obtain a predictable system. The solution to shared cache was either the use of a benefit counter or a probabilistic cache. Both these designs had their drawbacks which were discussed in Section 3.1.3. Also, the lack of a predictable shared cache controller makes it difficult to combine these cache designs with other categories of work to obtain a fully predictable architecture. The alternative to caches was scratchpads and tickpads. Tickpads can be used only with synchronous programming languages. Hence, this restricts the use of tickpads for a predictable generic architecture. SPMs, on the other hand, form a viable option to serve not only as a fast memory but also offer predictability. Even SPMs have some limitations as they do not allow sharing of data. Thus, none of these designs adequately mitigate all the challenges when forming a predictable system. A robust and a predictable memory controller should be scalable and should be independent of the tasks being executed on the cores. In that sense, we can choose one of the several designs described in Section 3.3. Finally, the shared interconnect design mostly follows TDMA arbitration (which is predictable). The only difference is with the topology of this interconnect. Hence any design of the interconnect discussed in Section 3.4 can be

Table II. Caches vs Scratchpads vs Tickpads

Feature	Shared Caches	Scratchpads	Tickpads
Size	Small with hierarchies	Small and Fixed	Small
Control Nature	Hardware	Software	Hardware and Software
Allocation	Replacement Policies	Allocation Algorithms like ILP, Greedy	Tick Precise Allocation Device
WCET Estimation	Architecture model required with complex analysis	Simple to compute tight WCET estimates	Simple WCET estimates
Programming Language	All languages supported	All languages supported	Only for Synchronous languages

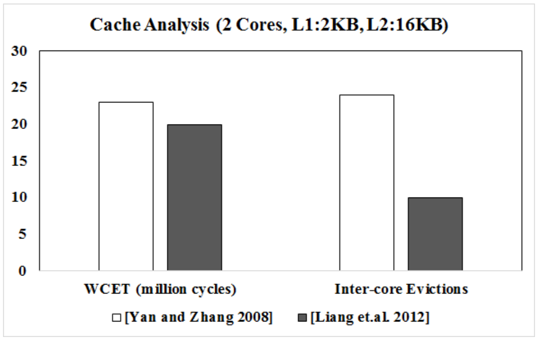

Fig. 6. Comparison of WCET estimates [Liang et al. 2012]

combined with the caches and memory controller. Thus, from the available solutions, we can use either SPMs or tickpads for high-speed memories, memory controller with TDMA or round robin arbitration scheme, and a shared interconnect with TDMA arbitration to obtain a predictable system. This system might not allow efficient sharing of memory as every access to shared data should be directed to the DDR. Also, another factor to keep in mind while combining such solutions is the composability of the system to keep the pessimism in WCET estimation as low as possible.

From the list of WCET estimation challenges on multi-cores listed under Section 2.3, we can see that there are contributions to build predictable caches, controllers and interconnect. However the other challenges mentioned under Section 2.3 have not been addressed.

4. ANALYSIS OF COTS HARDWARE AND UNMODIFIED MIDDLEWARE

This section discusses research that performs static analysis on COTS multi-cores and obtains WCET estimates for the tasks.

4.1. Cache Analysis

This subsection presents research that addresses the unpredictability in the access to shared caches by various analysis methods (box 5 in Figure 3).

4.1.1. Timing Analysis of concurrent programs. A timing analysis technique for a COTS multi-core with at least one level of shared cache was proposed by Liang et.al. [Liang et al. 2012]. The source of unpredictability addressed by this analysis is inter-core cache evictions. The analysis determines the lifetime of all the concurrently executing tasks and finds the potential interference in the access to the shared cache. Once this estimate is obtained, cache accesses are classified either as hits, misses, first access miss or unclassified. Using this information along with static analysis approaches discussed in Section 3, WCET of tasks are obtained. This work is an extension of the idea proposed by Yan et.al. [Yan and Zhang 2008].

Implementation: The following steps describe the proposed analysis framework.

1) Perform abstract cache analysis for L1 cache for every task [Ferdinand and Wilhelm 1999].
2) Memory requests that miss L1 cache are further analyzed at the next level (L2).
3) Once the L2 cache analysis is performed using techniques similar to L1, an iterative process begins. The iteration initially estimates the inter-core conflicts and computes additional cache misses at the L2 level. It then repeats steps 1 to 3 until there is no further change in the inter-core conflicts for all the tasks in the system. The resulting task lifetimes are the WCET estimates which include the additional delay due to inter-core interferences on the shared L2 cache.

This work advances the analysis by Yan et.al. [Yan and Zhang 2008] in the following ways. Firstly, this analysis considers only those tasks that have overlapping execution periods for analyzing the interference, whereas the work by [Yan and Zhang 2008] assumes all tasks scheduled on every other core to be interfering. Further, this work also analyses set associative caches that provide better estimates when compared with direct mapped caches.

Experiments and Results: Experiments are performed on a quad-core processor with 2GB RAM, and the benchmark used for analysis is DEBIE [deb 2000]. The work compares the WCET estimates by progressively increasing the number of cores and L2 cache size. In Figure 6, we present the comparison of the results between the two methods ([Yan and Zhang 2008] and [Liang et al. 2012]) for a particular cache configuration. We can observe that there is a significant reduction in the WCET estimate and inter-core evictions in the method proposed by Liang et.al. [Liang et al. 2012].

4.1.2. Using Bypass to tighten WCET estimate. Another study proposes a method to mitigate the effect of shared cache interferences [Puaut 2008], [Hardy et al. 2009]. The major contribution of this work is that it offers a method to reduce the pessimism in the estimated WCET while analyzing the inter-core interference. This is achieved by identifying code blocks that are used only once during the execution and avoid caching them. This work assumes that every core has at least one level private cache and all cores share another level of cache. Also, the caches use LRU replacement policy.

Implementation: This study uses Abstract Interpretation (AI) to perform cache analysis as shown in Figure 7. The hits and misses at every cache level are computed and the cache accesses are classified as hits, misses, first-time miss and unclassified.

Extending this analysis for multi-cores and the shared cache, interference is estimated using a heuristic. From cache analysis, information about the memory accesses that can occur at the shared cache level from every task is obtained. Using this information, all memory accesses that may potentially interfere in the shared cache are identified. An interference bound for all the tasks and every cache set is obtained.

This work also identifies blocks of code that are used only once during task execution. This is statically determined based on the access classification performed before. These blocks of code are cache inhibited. The cache analysis performed after removing these blocks provides tighter WCET estimates.

Experiments and Results: Experiments are conducted using MIPS binary code. The experiments are done using Malardalen WCET benchmarks [Gustafsson et al. 2010]. We consider two applications, ‘crc’ with lowest program size (1432 bytes) and ‘statem-

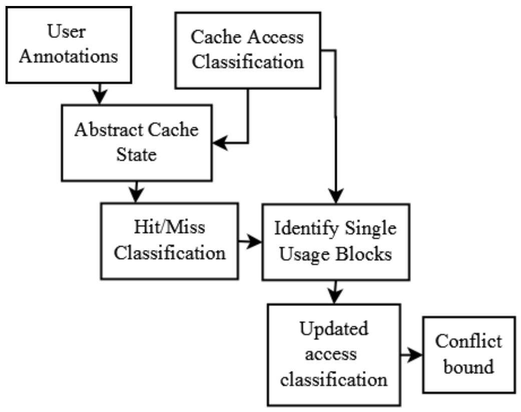
User Annotations
Cache Access Classification
Abstract Cache State
Hit/Miss Classification
Identify Single Usage Blocks
Updated access classification
Conflict bound

Fig. 7. Bypass technique to reduce pessimism in cache analysis [Puaut 2008], [Hardy et al. 2009]

ate' with the largest program size (8900 bytes), and compare across the experiments that estimates worst-case L2 hit ratio. For 'crc', the hit ratio in the worst-case without bypass is 61.85%, and with bypass it is 98.97%. Similarly, for 'statemate' the hit ratio in the worst-case without bypass is 0.19%, and with bypass it is 1.21%. From this, we can conclude that the inter-core interferences are reduced when bypass scheme is used.

Discussions: The static analysis work on caches try to solve the problem of shared caches by analyzing them off-line and without making any changes to the hardware architecture. The classical cache analysis is the concrete analysis that provides information about every cache state [Ferdinand and Wilhelm 1999]. The drawback of using this approach was that the number of cache states increases exponentially. To mitigate this problem, the concept of Abstract Interpretation (AI) was introduced, which reduces the number of cache states. With the advent of multi-cores, shared caches were introduced in the architecture which caused inter-core data eviction. This was analyzed using inter-core cache analysis discussed in this section. Both studies discussed in this section, Liang et.al. [Liang et al. 2012] and Hardy et.al. [Hardy et al. 2009], use AI to perform their single-core analysis. The difference between them is how inter-core analysis is performed. The study by Liang et.al. [Liang et al. 2012] uses task graph and an iterative mechanism to estimate the cache misses that occur in the shared cache due to inter-core cache conflicts. On the other hand, the study by Hardy et.al. [Hardy et al. 2009] uses a conflict bound to identify the blocks of program that interfere with each other. Also, the study by Hardy et.al. [Hardy et al. 2009] improves upon the existing analysis technique by identifying blocks of program that are used only once (single used blocks), and removing them from cache analysis thereby reducing the pessimism in WCET estimation. It is also essential to note that the evaluation by Hardy et.al. [Hardy et al. 2009] ignores task lifetimes and scheduling, whereas Liang et.al. [Liang et al. 2012] uses this information in the analysis.

One of the major drawbacks of both these analyses is that they rely on assumptions such as LRU cache replacement policy and non-preemptive processing. These unrealistic assumptions pose a serious question on the applicability of these analyses for COTS processors. On the contrary, robust analysis techniques without these assumptions is an open problem in this domain of research.

$CHMC_{r,\ell-1}$ / $CAC_{r,\ell-1}$	AM	AH	FM	NC
A	A	N	U-N	U
N	N	N	N	N
U-N	U-N	N	U-N	U-N
U	U	N	U-N	U

Fig. 8. Cache Hit Miss Classifications [Hardy et al. 2009]

4.2. Interconnect Analysis

This subsection presents research that addresses the unpredictability in access to the shared interconnect (box 8 in Figure 3) by analysis methods.

4.2.1. Interconnect Analysis using TDMA Offsets. A technique to analyze the delay in access to the shared interconnect was proposed by [Kelter et al. 2011]. The assumed system model is a multi-core architecture with every core having one level private cache and an in-order pipeline. The cores connect to a shared interconnect which performs TDMA arbitration. This interconnect is connected to another level of cache which is also shared and finally to the DDR memory via a controller. The cache in this analysis is exclusive and uses LRU replacement policy.

Implementation: To analyze the shared interconnect, this study assumes a TDMA scheme where a fixed length time slot is assigned to every core. Thus, each core can obtain access to the interconnect only during its slot. Otherwise, it needs to wait to obtain a slot. Depending on the time when a task requests access to the interconnect, the amount of time that this request would have to wait before getting access to the interconnect is determined. This approach helps to bound the interconnect delay and is used to estimate the WCET of the task. The analysis is embedded into the CHRONOS WCET analysis framework [Li et al. 2007].

Experiments and Results: Experiments are performed on Intel Xenon processor with Debian Linux OS. CPLEX ILP solver has been used to obtain WCET estimates [cpl 1993]. Tasks from several benchmarks such as DEBIE, MRTC and Papabench were used to test the technique. From the experiments, it is concluded that using offset analysis with the assumption of multi-core architectures without timing anomalies would provide the tightest WCET estimate.

4.2.2. Interference Sensitive WCET estimation. Another study proposes an approach to estimate WCET by taking into account different access times due to the concurrent usage of shared resources in multi-cores [Nowotsch et al. 2014]. This work is an offline analysis which is supported by monitoring technology that enforces guarantees on shared resource usage at run-time. The timing analysis is performed in two steps where the first is to quantify delay within a core and the second is to quantify delay due to sharing of resources. Every shared resource is abstracted using its bandwidth, and every requesting task is designated a certain portion of it. Using this allocated resource share, the technique ensures that none of its resources are over-utilized. Run-time monitoring is used continuously to update the resource usage of a task. Monitoring identifies if a particular resource is over-utilized by a task and provides a trigger for suspending that task.

Implementation: WCET estimation has been dealt with in two phases: 1) Single-core resource and timing analysis for every task, and 2) Bounds for multi-cores by using the single-core results for co-running tasks. For single-core timing analysis, cache and pipeline analysis are performed as described in Section 2.2, and is further supported by a resource analysis (for every shared resource and every task). Thus, an upper bound

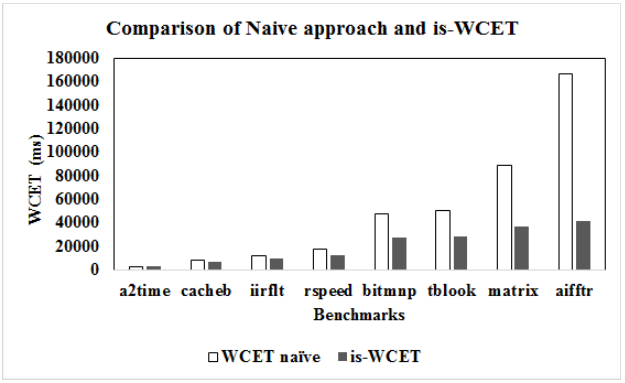

Fig. 9. is-WCET vs Naive approach (Refer Table III in [Nowotsch et al. 2014])

for using the shared resource is obtained. Using these analyses, a multi-core WCET estimate is obtained for each task along with the runtime resource usage capacity.

Experiments and Results: This work is evaluated on the octa-core Freescale p4080 processor with SYSGO's commercial RTOS (PikeOS) [Kaiser and Wagner 2007] and AbsInt timing analysis framework [Ferdinand and Heckmann 2004]. The WCET computed is compared to the naive approach where every memory access is accounted with its maximum delay as shown in Figure 9. It can be observed that as the code size increases, the proposed technique estimates tighter bounds when compared to the naive approach.

4.2.3. Model Checking for Bus Analysis. Another study proposes a model checking based approach for timing analysis of multi-cores [Lv et al. 2010]. This work tries to solve the unpredictability in multi-cores by modeling the cache and shared bus using Timed Automata (TA). Tasks are bounded using local cache analysis and assuming a predictable arbitration on the shared interconnect.

Implementation: The timing analysis of the local cache is performed using Abstract Interpretation (AI) [Ferdinand and Wilhelm 1999]. The shared interconnect accesses usually depends on the nature of cache accesses. We can infer that all accesses that are cache hits will not access the shared interconnect. On the other hand, those accesses that are cache misses will access the shared interconnect. Instructions that are not classified can either be a hit or a miss in the cache. Hence both these cases are considered while constructing the automata. The arbitration in the interconnect is assumed to be either First-Come First-Served or TDMA and is also modeled based on the cache analysis. After building the model based on the states discussed, the task for which the timing bound is required along with the model is given as input to UPPAAL model checker, which estimates the WCET of the task.

Results: The Malardalen benchmark suite is used to perform experiments with the constructed models [Gustafsson et al. 2010]. For each task the WCET from the model checker is compared with the worst-case delay (by analysis) and the improvement in the WCET estimate is determined. For instance, 'bs' task from Malardalen benchmark has an improvement of 77% over the worst-case interconnect delay for TDMA arbitration scheme.

4.2.4. Impact of Interconnect Contention on the WCET. This method [Dasari and Nelis 2012] computes a bound on the maximum memory access requests that a core can generate and also proposes a methodology to find the delay that a task incurs when multiple tasks execute on a multi-core. This improves the work proposed in [Dasari et al. 2011]

Implementation: The system model is a multi-core architecture with all caches either partitioned or disabled. The prefetch algorithm and request buffering in the memory controller is also disabled. The basic idea of this work is to reduce the pessimism in computing the bound on the number of accesses to the interconnect in any time interval. The work [Dasari et al. 2011] computes the upper bound in two steps. In the first step, the number of memory accesses of a task within a time interval is computed. The next step estimates the upper bound by the addition of all the accesses from every other task within that interval. To avoid this overestimation, this method [Dasari and Nelis 2012] splits the time interval into three sectors which are as follows: a) tasks that have already started to execute and enter into this time interval, b) tasks that begin and end within this interval and c) tasks that commence in this interval and end outside it. Once these sectors are classified, the total number of accesses to the interconnect reduces (when compared to the case where every request from all tasks within the time interval is considered) which mitigates the pessimism.

Results: For evaluation of this work, a set of 25 tasks were generated and randomly assigned to the four cores and was compared to the work in [Dasari et al. 2011]. It was then verified that this work provides tighter WCET bound when compared to [Dasari et al. 2011].

Discussions: Static analysis techniques for the system interconnect try to address the problem of unpredictability arising due to inter-core interference on the shared interconnect (described under Section 2.3). The studies on shared interconnect presented in this section view the problem differently, wherein one of them assumes TDMA arbitration for access to the shared interconnect, while another computes the total interference on the interconnect and monitors them at run-time, another work develops a timed automata for the functioning of the shared interconnect and the last contribution bounds the number of memory accesses to the interconnect. All these works primarily assume the arbitration to be TDMA or round-robin. This assumption may not be valid for a COTS hardware since most of the processor manufacturers do not disclose their arbitration schemes. Also, none of the analysis techniques presented in this section include the effect of shared caches on the interconnect, or consider the effect of running a coherency protocol.

4.3. DDR controller Analysis

This subsection presents research that addresses the unpredictability in access to the DDR memory by analysis methods (box 7 in Figure 3).

4.3.1. Memory Interference Delay Analysis. This work [Kim et al. 2014b] models the DDR memory resource and obtains a bound for the delay experienced by a task due to other tasks that run in parallel on a multi-core.

This work has three main contributions. First is considering a model which takes into account the timing details of DDR memory in accordance with JEDEC standards. Second is to bound the memory access interference of a task by considering its accesses and the interfering accesses during its execution. Finally, it also evaluates the consequence of using private against shared DDR banks.

Implementation: For this analysis, the memory controller is assumed to have a buffer for each bank and a two-level scheduler (within a bank and between banks). Also, the arbitration is assumed to be First-Ready First-Come First-Serve and cache interferences are assumed to be non-existent. The two levels of interference to be analyzed are intra-bank and inter-bank. The system model for analysis assumes a task with four parameters (WCET, Period, Relative deadline and Maximum DDR requests generated).

The first approach bounds the maximum delay experienced by every request by focusing on the number of memory accesses generated by the task itself. According to this method, the delay within a bank is bounded based on the timing constraints in the controller. If consecutive memory requests are to the same row, then the cost of opening a row is avoided which reduces the latency for multiple access to the same row. This delay is zero if the banks are privatized. The delay between banks is considered by computing the channel timing constraints.

The second approach bounds the maximum delay due to all the other memory accesses from the interfering task over a period. It assumes that all these accesses are processed before the access being analyzed. A response time test is performed for these approaches.

Results: PARSEC benchmarks are used with Linux/RK running on a quad-core processor for experiments. When the DDR banks are shared, the WCET estimation without re-ordering of commands is overly optimistic, but the one with command re-ordering is close to the observed values. When the banks are partitioned, the observed and estimated values are very similar.

4.3.2. Parallelism-Aware Memory Interference Delay Analysis. This work [Yun et al. 2015] improves the basic DDR analysis performed in [Pellizzoni et al. 2010b] by considering a realistic model of interference of a multi-core.

Implementation: This work also follows the two approaches as mentioned in [Kim et al. 2014b]. One is to bound the delay based on the memory demand of the task under analysis and the second is to bound the delay due to the interference from the other tasks. The key factors which form the basis of this implementation are that the number of contending memory accesses from other cores are bounded and overlap of DDR commands at command scheduling level are considered. These factors help to reduce the pessimism in the memory interference analysis to provide a tighter bound.

Results: The evaluation platform consisted of a quad-core processor with a DDR model and partitioned at the bank level. The benchmarks are from SPEC2006. From the results, it can be seen that this technique provides WCET estimates that are closer to measured values.

Discussions. This section focuses on the unpredictability in the access to the shared DDR memory through its controller. The studies presented ([Kim et al. 2014b] and [Yun et al. 2015]) consider the effect of interference when a task executes with respect to the memory controller. However, these analyses are still pessimistic in the worst-case (as presented in [Yun et al. 2015]). Hence they require more architectural knowledge/assumptions that can reduce the pessimism.

4.4. Cache Locking and Partitioning

Locking and partitioning of caches are two hardware techniques that facilitate predictable access to the shared cache on a COTS multi-core platform. When a core locks a line in the cache, the data in the locked line cannot be evicted by tasks running on the other cores. Orthogonally, caches can also be partitioned in a way that the different partitions are mapped to the distinct cores. In the following subsections, we present three studies that focus on achieving predictability on the shared cache through locking or partitioning (box 6 in Figure 3).

4.4.1. Semi-Partitioned Scheduling with Cache Locking. The work by Shekhar et.al. [Shekhar et al. 2012] proposes to use cache locking and locked cache mi-

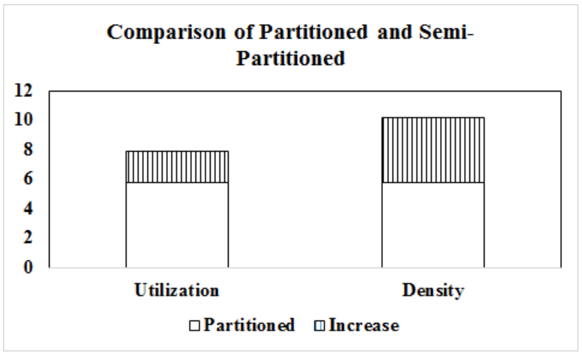

Fig. 10. Comparing semi-partitioned and partitioned approach [Shekhar et al. 2012]

gration to schedule a set of independent tasks using semi-partitioned scheduling strategy (strategy discussed in [Bletsas et al. 2008]).

Implementation: Given a set of tasks and cores, this work statically partitions as many tasks as possible on to the different cores. Those tasks that are not statically assigned to a core can migrate from one core to another among a predetermined set of cores. Within each core the task that has the earliest deadline is scheduled first for execution. The tasks within a single core can statically select and lock memory lines within the private cache of that core. While migrating, memory lines of that particular task are first unlocked and then moved to the target core and then locked again within the target core's private cache.

Results: Figure 10 shows a comparison of schedulable utilization and density ("ratio of execution time of a task to its relative deadline" [Shekhar et al. 2012]) of the proposed approach against partitioned strategies. The shaded portion in the graph indicates an increase in the density and utilization for the proposed algorithm.

4.4.2. Exploring Cache Locking and Partitioning. Another work explores locking and partitioning techniques for predictability in shared caches [Suhendra and Mitra 2008]. It uses different combinations of cache partitioning and locking to analyze the impact on the worst-case performance of an application. The various methods evaluated are static and dynamic locking techniques with and without partitions in the shared cache.

Implementation: The different locking and partitioning scenarios explored are as follows. The first scenario is where the contents of the cache are locked and kept unchanged during a task execution. A cache block is associated with a task regardless of the core on which it has been scheduled. This is limited in flexibility if the total code size exceeds the size of the cache. The second scenario is similar to the first except that the cache is partitioned such that each core can lock its task contents in some portion of the cache. The third scenario is where the entire cache is partitioned based on all tasks in the system. After partitioning, every task dynamically performs locking during run-time. The final one is where the cache is partitioned based on the total number of cores in the system and the tasks dynamically lock regions of the cache during run-time.

Experiments and Results: These scenarios are implemented on a dual-core processor with two cache levels and two-way associativity. The conclusions of these experiments are as follows. For core based partition, dynamic locking utilizes the cache better than static when the cache is limited and there are many cacheable portions of the task. On the other hand, when the size of cache increases and there are very few cacheable portions, then static locking performs better. The second category is the task based partition. The trend for this partition is also similar to the core based strategies.

Discussions. Cache locking and partitioning try to address the challenge of predictably accessing the shared cache (described under Section 2.3). These are hardware features that can be used at the software level to perform cache accesses. Two central questions that arise while using these techniques are "What data to lock and for how long?" and "How much of the cache is to be partitioned?". The studies discussed in this section try to address these issues by presenting various techniques for locking and partitioning. All these techniques, however, lock/partition the entire shared cache. But multi-core as an architecture is suited to share data across the cores. None of these techniques address this issue of efficient sharing of data between the cores when the cache is partitioned/locked.

5. UNMODIFIED COTS HARDWARE AND MODIFIED MIDDLEWARE

This section presents research that makes no changes to a multi-core hardware but modifies the middleware to obtain predictable execution of tasks in multi-cores (box 9 in Figure 3).

5.1. PREM (PRedictable Execution Model for multi-cores)

The work by Alhammad et.al. [Alhammad and Pellizzoni 2014] proposes a method to execute tasks such that the accesses to memory can be completely independent of the policies employed by the hardware. This is achieved using an algorithm that maps and schedules tasks on the multi-core using fork-join model. In this work, every sequential task is split into multiple segments and every segment is processed in two phases. In the first phase, data that is required for task execution is brought to that particular core's private cache. This step ensures that the execution does not suffer any cache misses and hence would also not access DDR memory. In the second phase, the task is executed.

5.2. OS-level Shared Cache Management Technique

An OS-level technique to allocate shared cache partitions to different tasks was proposed by Kim et.al. [Kim et al. 2013]. To predictably share caches, this work uses page coloring mechanism. Rows in the DDR memory are termed as pages. Page coloring algorithm allocates different colors to each page with every color physically mapped to different locations in the DDR memory.

Implementation: The approach presents a strategy that combines partitioning and sharing of caches to use shared cache efficiently. The unique feature of this approach is that while partitioning the shared cache, every partition is allocated to a core rather than to a task. Also, the tasks within a core can share the assigned partition. The disadvantage of sharing a partition by multiple tasks is that it creates preemption and warm-up delays. The work provides bounds for these delays by using a response time test that accounts for these delays. Using these two strategies, cache partitions are allocated to cores and tasks are allocated to these partitions based on best-fit decreasing bin packing heuristic.

Experiments and Results: The work has been implemented on Linux/RK and evaluated on Intel i7-2600, a quad-core processor. The evaluation system has 8MB of shared L3 cache that has been divided into four partitions each with 2MB. To test the allocation technique, periodic tasks from the PARSEC benchmark suite was used [Bienia et al. 2008]. For this experiment, best-fit and worst-fit bin packing algorithms along with normal cache partitioning was compared against the proposed strategy. It was observed that this technique not only reduced memory usage but also provided better processor utilization.

5.3. Page Coloring technique to manage Shared Caches

Another study proposes the use of page coloring mechanism and real-time resource locking protocols to handle inter-task interferences on the shared cache [Ward et al. 2013]. The challenge with coloring mechanism is to find an optimal strategy to assign colors to tasks. The idea of this work is that it utilizes page colors as resources that can be shared, and the access to these resources could occur through arbitration using any real-time locking protocol.

Implementation: For cache locking, every task that executes on the core must lock the required lines in the cache before the start of its execution, and hold the lock till its completion. This work combines cache locking protocols with cache coloring mechanism. The locking protocol manages the access to all the colors. The drawback of using this technique is that the task execution cannot be preempted. The work also provides solutions to mitigate this problem. It suggests to either split the period of execution (where all tasks will have the period of the shortest task) or split a larger task into smaller subtasks.

To reduce the effect of blocking due to the non-preemptive execution of tasks, this work views cache colors as a resource that can be preemptively scheduled. Thus when a task is ready for execution, it is scheduled along with all the required cache colors and hence it has exclusive access to those cache partitions. The advantage of this method is that when there is a higher priority task which needs to be executed, this method can preempt the current task with the higher priority one. However, the delay due to the reloading of tasks needs to be taken into account which is described in Brandenburg et.al. [Brandenburg 2011].

Results: The techniques described are implemented as part of LITMUS^{RT} [Calandrino et al. 2006]. The proposed method is compared against caches that do not have any particular management techniques. From the results, it can be observed that locking caches and splitting the period of task execution always provides better processor utilization with an improvement of at least 50% over the unmanaged cache.

5.4. PALLOC: Bank Aware Memory Allocator

Another work uses a virtual memory model that allocates memory pages of every task to only specific banks [Yun et al. 2014]. Operating systems lack details on the mapping of memory accesses to DDR banks. Most memory accesses are spread across several banks of memory. Consider a simple scenario when there are 2 cores and 2 banks. The best case memory access would occur when each core accesses only one bank or when only one of the cores requires memory accesses to both the banks. In these cases, there are no inter-core conflicts in the access to memory. But most memory accesses are such that both cores could potentially access data in both the banks. This causes unpredictability in the memory access times. This work offers a predictable DDR bank allocation scheme to address this issue.

Implementation: PALLOC has been implemented as an OS level memory allocator for tasks. It uses the memory translation unit to allocate memory pages for tasks in the DDR. The OS has a data structure that maintains a list of pages that are free. An allocator function is executed whenever there is a page fault. This function keeps a separate list for every bank in the DDR and finds a page from one of these banks. If a matching page is found, it returns it. Else, the free list is checked iteratively to obtain a match.

Results: PALLOC has been evaluated on two separate platforms. One is the Intel Xeon 3530 which has 4 X86-64 cores with Linux OS and 4GB DDR with 16 banks, and another is the Freescale P4080, which has 8 PowerPC cores with Linux OS and 2 x 2GB DDR with 32 banks. Two types of experiments are performed. In the first experiment, all the cores access the same bank and is compared with the case where core 0 accesses only bank 0 and the other cores access banks 1 to 3. It can be observed from the results that the average latency to access the same bank (bank 0) increases as the number of cores increase. This is evident from the fact that the memory accesses are serialized. On the other hand, when there are accesses to different banks, it can be seen that the access time for core 0 remains constant at approximately 65ms. This shows the effectiveness of mapping cores to banks.

A second experiment is performed to determine the impact in performance when DDR banks are partitioned among the cores. The testing unit is a data acquisition system wherein one of the cores would have real-time tasks while the other cores would have background tasks. A similar performance as the first experiment was observed, which highlights the importance of DDR bank partitioning.

5.5. Memguard: Memory Bandwidth Reservation System

Another study implements a memory bandwidth reservation system that allocates, recovers and maximizes the utilization of the memory bandwidth [Yun et al. 2013]. This work provides an OS-level approach not only maximizes the performance of each core but also provides predictable memory access.

Implementation: Memguard works by regulating every core's request rate to the memory (the rate at which every core can generate request to the DDR memory). The main principle of regulating this bandwidth is that the total memory request rate is always maintained below the DDR's total capacity. The architecture of Memguard has two functions. First is to assign memory bandwidth to cores by reading hardware performance counters and the second is to recover the unused bandwidth and distribute to all cores.

Results: This work has been evaluated on Intel Core 2 Quad 8400 processor with 4MB cache and Linux OS. The experiments were performed with SPEC2006 benchmarks [Henning 2006]. In the first experiment, one of the two cores has a real-time task while the other core executes background task. The benchmark is executed with and without Memguard. It can be observed that the real-time task achieves maximum utilization when there are no background tasks. On the other hand, its performance decreases when there are background tasks. In the second case, when executed with Memguard all tasks achieve a guaranteed performance level, irrespective of whether it is executed with a background task or not.

5.6. Real-time Cache Management Framework

This OS-level study develops a framework to obtain memory patterns of a task, and also a cache maintenance method for the most frequently accessed memory locations [Mancuso et al. 2013]. Shared caches are always a source of unpredictability due to the interference in the access to it by multiple cores. This study uses cache coloring and locking to provide predictable access to the memory.

Implementation: The first step in this framework is to profile all the real-time tasks. This is done by executing every task within a controlled environment and finding all memory requests that are generated. After obtaining this profile data, the most frequently used memory accesses are identified. The profile data is used for real-time shared cache management which works in two stages. Initially using cache coloring mechanism, the frequently used memory accesses of all tasks are provided a particular color exclusively. In the second stage, those frequent accesses are prefetched and locked in the cache.

Results: Experiments are performed on Pandaboard development board which has an ARM Cortex A9 processor [Instruments 2012]. Tasks used for testing are from the EEMBC benchmark [Poovey 2007]. The memory profiles of the tasks in this benchmark are obtained and the most frequently accessed pages are sorted. Although there are a number of pages that are accessed by the tasks, only those that are most frequently used are locked in the cache. Locking of frequently accessed pages reduces the WCET by as much as 2.5%.

A recent study combines multiple OS-level techniques to manage interference in shared resources [Sha et al. 2014]. This work combines 5.4, 5.5 and 5.6 techniques described before to address the challenges in shared hardware resources on COTS multi-cores.

Discussions. This category of work (Unmodified COTS hardware and modified middleware) tries to address the challenges of shared hardware resources (specifically caches and memory controllers) described under Section 2.3 by making modifications to the OS. The first contribution is a predictable execution model that tries to schedule every resource access and concurrent threads in two phases, prefetching and computing. This method of execution would suffer in throughput when there are a large number of cores that are executing real-time tasks. For predictable cache accesses, there are two contributions, both of which utilize the concept of page coloring mechanism. The study by Ward et.al. [Ward et al. 2013] proposes a more dynamic method of scheduling the tasks to access the cache, whereas the work by Mancuso et.al. [Mancuso et al. 2013] statically analyzes the tasks to obtain all possible memory requests. Finally, the work by Yun et.al. [Yun et al. 2014] proposes PALLOC that provides predictable DDR accesses. Again the drawback of this method is the use of list data structure that can cause a significant overhead in real-time.

6. DISCUSSIONS AND CONCLUSIONS

The relevance of time criticality in real-time systems has motivated the need for architectures and analysis that are amenable to WCET estimation. However, the advancements in hardware architectures have posed questions about the feasibility of these hardware to be used in real-time systems. Initially, a single-core processor was used as the primary hardware for implementing a real-time system. A single-core processor inherently has several design features that are difficult to analyze. This causes unpredictability in measuring the execution times of the applications. These include preemptions, hardware prefetching, timing anomalies, data-dependent control flows and context dependence of execution times. These challenges themselves have not been fully addressed, but by then the chip industry had overwhelmingly moved towards multi-core processors owing to the increasing demand in computational requirements. The introduction of these multi-cores has made single-cores obsolete, as most chip manufacturers have moved towards multi-cores owing to their exceptional SWaP characteristics. COTS multi-core has multiple single-cores that are fabricated on a single die and hence share hardware resources within the die. The increased complexity in the architecture has introduced more challenges in estimating the WCET of a task on multi-cores. Sharing of hardware resources such as caches, interconnects, cache and memory controllers, is the primary reason that has caused unpredictability in the estimation of WCET. Hence, most of the research in WCET estimation on multi-cores have focused on techniques to deal with shared hardware resources. Apart from these, secondary challenges that also added to the unpredictability are power saving strategies, system interrupts, and TLB misses. These challenges mostly remain unaddressed.

As discussed in Sections 3, 4 and 5, the research towards predictable WCET estimation has been broadly classified into three categories. The work on improving the hardware focused on either obtaining predictable multi-core components (memory controllers, caches and interconnect designs) or developing an entire hardware architecture suited for real-time systems (for example MERASA, parMERASA, etc.). Static analysis techniques derive WCET estimates by defining processor models and assuming the behavior of different components in its architecture. However, many of these analyses have been derived from assumptions that do not necessarily hold for COTS multi-core platforms (such as TDMA arbitration scheme in the interconnect). Finally, another section of research focused on modifying the operating system or utilizing hardware features (such as cache locking or partitioning) to obtain predictability in the execution of applications. Even these have their challenges of efficiently sharing data between the cores.

An interesting aspect to be discussed here is the interdependence among the solutions presented under different sections. The advantage of designing hardware components (Section 3) is to have a fine grained control over the execution of tasks. However, with any hardware it is also important to have a strong analysis to estimate WCET (Section 4). Also, in real-time applications, it is most likely that a middleware (Section 5) is also used to abstract the hardware architecture for software designers. Hence, it is important to resolve the challenges in all three categories to avoid any unpredictability in the system and to avoid pessimism in WCET estimation.

Since most research has focused on solving the problem of shared hardware resources (shared caches, controllers and interconnect) existing in multi-cores, none of them have focused on other challenges (power saving strategies, system interrupts, TLB misses, hardware prefetching) which are equally important to obtain a completely predictable system. They either require modifications at the hardware level, or sufficient pessimism must be added to the analysis to compensate for their effects. COTS hardware is known to be ‘unpredictable’ requires modifications at the architectural level. The solutions for shared hardware that were proposed have had considerable drawbacks in each of them. The solutions towards predictable memory controllers have issues of scalability, area, power consumption and integration with commercial processor architectures. System interconnects are limited in bandwidth and require efficient solutions when multiple cores try to access memory simultaneously. Shared caches have offered not only high performance and quick access to the main memory but also allowed sharing of data across the cores. There are no solutions to safely share data across the cores which is one of the key features to be exploited in this architecture. Finally, the unpredictability in the cache controller which is responsible for maintaining coherency and replacement of data has not been addressed in any of the research work. These issues need to be addressed to obtain a truly predictable COTS multi-core to be used in RTA.

REFERENCES

1993. CPLEX Solver. http://main.aimms.com/aimms/solvers/cplex/. (1993). [Online; accessed 15-June-2016].

2000. DEBIE Benchmark. http://www.mrtc.mdh.se/projects/WCC08/doku.php?id=bench:debie1. (2000). [Online; accessed 15-June-2016].

2004. Rapitime Analysis Tool. http://www.rapitasystems.com/products/rapitime. (2004). [Online; accessed 15-June-2016].

2010. P4080 Development System User’s Guide. http://cache.freescale.com/files/32bit/doc/user_guide/P4080DSUG.pdf. (2010). [Online; accessed 15-June-2016].

2010. SymTA/P Project. https://www.ida.ing.tu-bs.de/en/research/projects/symtap/. (2010). [Online; accessed 15-June-2016].

2012. Multi-Core Execution of Parallelised Hard Real-Time Applications Supporting Analysability. http://www.parmerasa.eu/. (2012). [Online; accessed 15-June-2016].

2013. Future Advances in Body Electronics. http://cache.freescale.com/files/automotive/doc/white_paper/BODYDELECTRWP.pdf. (2013). [Online; accessed 15-June-2016].

2014. Certification Authorities Software Team (CAST). https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/. (2014). [Online; accessed 15-June-2016].

Dennis Abts, Natalie D. Enright Jerger, John Kim, Dan Gibson, and Mikko H. Lipasti. 2009. Achieving Predictable Performance Through Better Memory Controller Placement in Many-core CMPs. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*. ACM, 451–461.

Benny Akesson, Kees Goossens, and Markus Ringhofer. 2007. Predator: a predictable SDRAM memory controller. In *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. ACM, 251–256.

Ahmed Alhammad and Rodolfo Pellizzoni. 2014. Time-predictable execution of multithreaded applications on multicore systems. In *Proceedings of the conference on Design, Automation & Test in Europe*. European Design and Automation Association, 29.

Rajeshwari Banakar, Stefan Steinke, Bo sik Lee, M. Balakrishnan, and Peter Marwedel. 2002. Scratchpad Memory: A Design Alternative for Cache On-chip memory in Embedded Systems. In *In Tenth International Symposium on Hardware/Software Codesign (CODES)*. ACM, 73–78.

Iain Bate, Philippa Conmy, Tim Kelly, and John McDermid. 2001. Use of modern processors in safety-critical applications. *The Computer JournalFyoo* 44, 6 (2001), 531–543.

Gérard Berry and Georges Gonthier. 1992. The Esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming* 19, 2 (1992), 87–152.

Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 72–81.

Andrew Binstock. 2012. Multi-core processor architecture explained. (2012). https://software.intel.com/en-us/articles/frequently-asked-questions-intel-multi-core-processor-architecture [Online; accessed 15-June-2016].

Konstantinos Bletsas and others. 2008. Sporadic multiprocessor scheduling with few preemptions. In *Euromicro Conference on Real-Time Systems*. IEEE, 243–252.

Björn B Brandenburg. 2011. *Scheduling and locking in multiprocessor real-time operating systems*. Ph.D. Dissertation. University of North Carolina at Chapel Hill.

John M Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C Devi, and James H Anderson. 2006. LITMUSˆ RT: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*. IEEE, 111–126.

Lakshmi N Chakrapani, John Gyllenhaal, W Hwu Wen-mei, Scott A Mahlke, Krishna V Palem, and Rodric M Rabbah. 2004. Trimaran: An infrastructure for research in instruction-level parallelism. In *Languages and compilers for high performance computing*. Springer, 32–41.

Ting Chen, Tulika Mitra, Abhik Roychoudhury, and Vivy Suhendra. 2007. Exploiting branch constraints without exhaustive path enumeration. In *OASIcs-OpenAccess Series in Informatics*, Vol. 1. Schloss Dagstuhl-Leibniz-Zentrum fr Informatik.

Soclib Consortium and others. 2008. *Soclib: an open platform for virtual prototyping of multi-processors system on chip*. Technical Report. [Online]. Available: http://www. soclib. fr, 2008. 4.5, 6.6. 1.

Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza, Jan Reineke, Benoıt Triquet, and Reinhard Wilhelm. 2010. Predictability considerations in the design of multi-core embedded systems. *Proceedings of Embedded Real Time Software and Systems* (2010), 36–42.

Dakshina Dasari, Björn Andersson, Vincent Nelis, Stefan M Petters, Arvind Easwaran, and Jinkyu Lee. 2011. Response time analysis of COTS-based multicores considering the contention on the shared memory bus. In *International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 1068–1075.

Dakshina Dasari and Vincent Nelis. 2012. An Analysis of the Impact of Bus Contention on the WCET in Multicores. In *International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS)*. IEEE, 1450–1457.

Christian Ferdinand and Reinhold Heckmann. 2004. ait: Worst-case execution time prediction by static program analysis. In *Building the Information Society*. Springer, 377–383.

Christian Ferdinand and Reinhard Wilhelm. 1999. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems* 17, 2-3 (1999), 131–181.

Kees Goossens and Andreas Hansson. 2010. The aethereal network on chip after ten years: Goals, evolution, lessons, and future. In *Design Automation Conference (DAC)*. IEEE, 306–311.

Stijn Goossens, Jasper Kuijsten, Benny Akesson, and Kees Goossens. 2013. A reconfigurable real-time SDRAM controller for mixed time-criticality systems. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*. IEEE, 1–10.

Giovani Gracioli, Ahmed Alhammad, Renato Mancuso, Antônio Augusto Fröhlich, and Rodolfo Pellizzoni. 2015. A survey on cache management mechanisms for real-time embedded systems. *ACM Computing Surveys (CSUR)* 48, 2 (2015), 32.

Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. 2010. The Mälardalen WCET benchmarks: Past, present and future. In *OASIcs-OpenAccess Series in Informatics*, Vol. 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. 2009. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 14, 1 (2009), 2.

Damien Hardy, Thomas Piquet, and Isabelle Puaut. 2009. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *Real-Time Systems Symposium*. IEEE, 68–77.

John L Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17.

Yijie Huangfu and Wei Zhang. 2014. PEG-C: Performance Enhancement Guaranteed Cache for Hard Real-Time Systems. *IEEE Embedded Systems Letters* 6, 2 (June 2014), 17–20.

Texas Instruments. 2012. PandaBoard. *OMAP4430 SoC dev. board, revision A* 2 (2012).

Javier Jalle, Leonidas Kosmidis, Jaume Abella, Eduardo Quiñones, and Francisco J Cazorla. 2014. Bus designs for time-probabilistic multicore processors. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*. IEEE, 1–6.

Robert Kaiser and Stephan Wagner. 2007. Evolution of the PikeOS microkernel. In *International Workshop on Microkernels for Embedded Systems*. 50–57.

Timon Kelter, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay, and Abhik Roychoudhury. 2011. Bus-aware multicore WCET analysis through TDMA offset bounds. In *Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 3–12.

Hyoseung Kim, Dionisio de Niz, Bjorn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. 2014b. Bounding memory interference delay in COTS-based multi-core systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 145–154.

Hyoseung Kim, Arvind Kandhalu, and Ragunathan Rajkumar. 2013. A coordinated approach for practical OS-level cache management in multi-core real-time systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 80–89.

Yooseong Kim, David Broman, Jian Cai, and Aviral Shrivastava. 2014a. WCET-Aware Dynamic Code Management on Scratchpads for Software-Managed Multicores. In *Real-Time and Embedded Technology and Application Symposium (RTAS)*. IEEE.

Leonidas Kosmidis, Jaume Abella, Eduardo Quinones, and Francisco J. Cazorla. 2013. A cache design for probabilistically analysable real-time systems. In *Design, Automation Test in Europe Conference Exhibition (DATE)*. 513–518.

Yogen Krishnapillai, Zheng Pei Wu, and Rodolfo Pellizzoni. 2014. A rank-switching, open-row DRAM controller for time-predictable systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 27–38.

Ming-Feng Kuo, Partha Roop, Sidharta Andalam, and Naresh Patel. 2013. Precision Timed Embedded Systems Using TickPAD Memory. In *International Conference on Application of Concurrency to System Design (ACSD)*. IEEE, 206–215.

Chunho Lee, Miodrag Potkonjak, and William H Mangione-Smith. 1997. MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 330–335.

Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. 2007. Chronos: A timing analyzer for embedded software. *Science of Computer Programming* 69, 1 (2007), 56–67.

Yun Liang, Huping Ding, Tulika Mitra, Abhik Roychoudhury, Yan Li, and Vivy Suhendra. 2012. Timing analysis of concurrent programs running on shared cache multi-cores. *Real-Time Systems* 48, 6 (2012), 638–680.

Yu Liu and Wei Zhang. 2012. Exploiting multi-level scratchpad memories for time-predictable multicore computing. In *International Conference on Computer Design (ICCD)*. IEEE, 61–66.

Mingsong Lv, Wang Yi, Nan Guan, and Ge Yu. 2010. Combining abstract interpretation with model checking for timing analysis of multicore software. In *Real-Time Systems Symposium (RTSS)*. IEEE, 339–349.

Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. 2013. Real-time cache management framework for multi-core architectures. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 45–54.

Stefan Metzlaff, Irakli Guliashvili, Sascha Uhrig, and Theo Ungerer. 2011. A dynamic instruction scratchpad memory for embedded processors managed by hardware. In *Architecture of Computing Systems-ARCS 2011*. Springer, 122–134.

Jan Nowotsch, Michael Paulitsch, Daniel Buhler, Henrik Theiling, Simon Wegener, and Martin Schmidt. 2014. Multi-core interference-sensitive wcet analysis leveraging runtime resource capacity enforcement. In *Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 109–118.

Marco Paolieri, Jörg Mische, Stefan Metzlaff, Mike Gerdes, Eduardo Quinones, Sascha Uhrig, Theo Ungerer, and Francisco J Cazorla. 2013. A hard real-time capable multi-core SMT processor. *ACM Transactions on Embedded Computing Systems (TECS)* 12, 3 (2013), 79.

Marco Paolieri, Eduardo Quinones, Francisco J Cazorla, and Mateo Valero. 2009. An analyzable memory controller for hard real-time CMPs. *IEEE Embedded Systems Letters* 1, 4 (2009), 86–90.

Mark S Papamarcos and Janak H Patel. 1984. A low-overhead coherence solution for multiprocessors with private cache memories. In *ACM SIGARCH Computer Architecture News*, Vol. 12. ACM, 348–354.

Rodolfo Pellizzoni and Marco Caccamo. 2010. Impact of peripheral-processor interference on WCET analysis of real-time embedded systems. *IEEE Trans. Comput.* 59, 3 (2010), 400–415.

Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo, and Lothar Thiele. 2010a. Worst case delay analysis for memory interference in multicore systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 741–746.

Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo, and Lothar Thiele. 2010b. Worst case delay analysis for memory interference in multicore systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 741–746.

Stefan Podlipnig and Laszlo Böszörmenyi. 2003. A survey of web cache replacement strategies. *ACM Computing Surveys (CSUR)* 35, 4 (2003), 374–398.

Jason Poovey. 2007. Characterization of the eembc benchmark suite. *North Carolina State University* (2007).

Damien Hardy—Isabelle Puaut. 2008. WCET analysis of multi-level set-associative instruction caches. *arXiv preprint arXiv:0807.0993* (2008).

Isabelle Puaut and Christophe Pais. 2007. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 1–6.

Petar Radojković, Sylvain Girbal, Arnaud Grasset, Eduardo Quinones, Sami Yehia, and Francisco J Cazorla. 2012. On the evaluation of the impact of shared resources in multithreaded COTS processors in time-critical environments. *ACM Transactions on Architecture and Code Optimization (TACO)* 8, 4 (2012), 34.

Jan Reineke. 2014. Randomized Caches Considered Harmful in Hard Real-Time Systems. *Leibniz Transactions on Embedded Systems (LITES)* 1, 1 (2014), 03:1–03:13.

Jan Reineke, Isaac Liu, Hiren D Patel, Sungjun Kim, and Edward A Lee. 2011. PRET DRAM controller: Bank privatization for predictability and temporal isolation. In *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM, 99–108.

PS Roop, S Andalam, and A Girault. 2009. PRET-C: A new language for programming precision timed architectures. *INRIA Grenoble Rhone-Alpes* (2009).

Bernhard Rumpler. 2006. Complexity management for composable real-time systems. In *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)*. IEEE, 9–pp.

Martin Schoeberl, Florian Brandner, Jens Sparsø, and Evangelia Kasapaki. 2012. A statically scheduled time-division-multiplexed network-on-chip for real-time systems. In *Sixth IEEE/ACM International Symposium on Networks on Chip (NoCS)*. IEEE, 152–160.

Martin Schoeberl, David Vh Chong, Wolfgang Puffitsch, and Jens Sparsø. 2014. A time-predictable memory network-on-chip. In *OASIcs-OpenAccess Series in Informatics*, Vol. 39. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Lui Sha, Marco Caccamo, Renato Mancuso, Jung-Eun Kim, Man-Ki Yoon, Rodolfo Pellizzoni, Heechul Yun, Russel Kegley, Dennis Perlman, Greg Arundale, and others. 2014. Single Core Equivalent Virtual Machines for Hard Real—Time Computing on Multicore Processors. (2014).

Hardik Shah, Kai Huang, and Alois Knoll. 2014. Timing anomalies in multi-core architectures due to the interference on the shared resources.. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*. 708–713.

Hardik Shah, Andreas Raabe, and Alois Knoll. 2013. Challenges of WCET Analysis in COTS Multi-core due to Different Levels of Abstraction. In *High-performance and Real-time Embedded Systems (HiRES 2013)*. Berlin.

Mayank Shekhar, Abhik Sarkar, Harini Ramaprasad, and Frank Mueller. 2012. Semi-partitioned hard-real-time scheduling under locked cache migration in multicore systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 331–340.

Vivy Suhendra and Tulika Mitra. 2008. Exploring locking & partitioning for predictable shared caches on multi-cores. In *Proceedings of the 45th annual Design Automation Conference*. ACM, 300–303.

Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. 2005. WCET centric data allocation to scratchpad memory. In *Real-Time Systems Symposium*. IEEE, 10–pp.

Hideki Takase, Hiroyuki Tomiyama, and Hiroaki Takada. 2010. Partitioning and allocation of scratch-pad memory for priority-based preemptive multi-task systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1124–1129.

David Wang, Brinda Ganesh, Nuengwong Tuaycharoen, Kathleen Baynes, Aamer Jaleel, and Bruce Jacob. 2005. DRAMsim: a memory system simulator. *ACM SIGARCH Computer Architecture News* 33, 4 (2005), 100–107.

Bryan C Ward, Jonathan L Herman, Christopher J Kenna, and James H Anderson. 2013. Making shared caches more predictable on multicore platforms. In *Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 157–167.

Saud Wasly and Rodolfo Pellizzoni. 2013. A dynamic scratchpad memory unit for predictable real-time embedded systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 183–192.

Jun Yan and Wei Zhang. 2008. WCET analysis for multi-core processors with shared L2 instruction caches. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 80–89.

Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. 2014. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 155–166.

Heechul Yun, Rodolfo Pellizzon, and Prathap Kumar Valsan. 2015. Parallelism-Aware Memory Interference Delay Analysis for COTS Multicore Systems. In *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*. IEEE, 184–195.

Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. 2013. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 55–64.