Measurement based WCET Analysis for Multi-core Architectures

Hardik Shah[1], Andrew Coombes[2], Andreas Raabe[3], Kai Huang[1,4] and Alois Knoll[1]
[1]Department of Informatics VI,Technical University Munich, 85748 Garching, Germany
{shah, huang, knoll}@in.tum.de
[2]Rapita Systems Ltd, UK
acoombes@rapitasystems.com
[3]fortiss GmbH, 80805 Munich, Germany
[4]School of Mobile Information Engineering, Sun Yat-Sen University

ABSTRACT

The interference on shared resources caused by concurrently executing applications unpredictably prolongs their execution. Hence, determination of the Worst Case Execution Time (WCET) of applications executing on shared memory multi-core processors is hard to estimate. This hinders the adoption of Commercial Off The Shelf (COTS) multi-core processors in hard real-time systems. The existing techniques opt for tailored multi-core architectures to provide high computation power at predictable execution time. However, this approach yields poor resource utilization and high costs. In this paper, we present a technique to measure the WCET of applications on multi-core architectures using existing measurement based timing analysis tools. Our technique has a minor area impact ($\approx$ 5%). However, this impact is limited to the *emulation devices* only and production chips remain unchanged. Thus, our technique does not impact performance of the COTS chips by any ways. The technique is demonstrated by measuring WCET of benchmark applications using the RapiTime timing analysis tool. The tests are conducted on a quad-core NIOS II processor on an Altera FPGA.

1. INTRODUCTION

Multi-core architectures can satisfy the computation demand of advanced Hard Real-Time (HRT) applications under manageable energy consumption. There is a growing interest of using COTS multi-core architectures in HRT applications due to their high computation power and cost benefits. However, they are optimized for average case performance and not for the worst case performance. For example, to reduce the packaging costs, they employ shared resources, e.g. shared memory. Depending on the shared resource access pattern of concurrently executing applications, shared resource accesses issued from different cores may or may not collide. This uncertainty increases execution time of applications unpredictably. Thus, the WCET estimation becomes hard to determine. The WCET estimate of an application on underlying hardware is a key requirement before industrial integration of the combined (hardware/software) HRT system.

Traditionally, the above mentioned problem is addressed by radically transforming multi-core architectures to make them more timing predictable. The simplest of these transformations is static time slicing to access shared resources, e.g. Time Division Multiple Access (TDMA). This approach yields poor shared resource utilization since the unused time slices are wasted. Moreover, only a fraction of digital chips produced today goes into the HRT systems. Due to this small market share, it may not be economically feasible to produce highly customized chips suitable to hard real-time systems only. The economic feasibility is yet to be investigated.

In this paper, we present a technique to measure the WCET of applications executing on multi-core architectures using existing timing analysis tools for single core architectures. Our technique inserts a cache observation and a time stamping module, e.g. Performance Counter [1], in a COTS multi-core architecture. The insertion is only limited to *emulation devices* (Sec. 4.4) of production chips. Hence, mass produced COTS chips stay unchanged preserving their economic benefits.

The interference on shared resources occurs deep inside the chip and only perceivable effect is unpredictable execution time of applications executing on them. Our time stamping module observes the activity on the shared resource and saves it in a trace. The trace is later processed offline to compensate for the worst case interference. The pre-processed trace is analyzed by the existing measurement based timing analysis tool, e.g. RapiTime, to determine the WCET on the underlying multi-core architecture.

The main contributions of the paper are as follows. i) We present a technique to measure the WCET of applications on multi-core architectures using existing tools for single-core architectures. This is demonstrated by measuring WCET of applications on a quad-core processor using RapiTime timing analyzer. ii) Our technique slightly modifies the COTS component, however, this modification is limited to *emulation devices* only. The production chips remain absolutely unchanged. Since the production chips are not modified, their performance is not impacted by any ways (resource utilization, heat dissipation, operating frequency,

RTNS 2014 , October 8 - 10 2014, Versailles, France

http://dx.doi.org/10.1145/2659787.2659819.

energy consumption etc.). iii) The presented technique is tested on a quad-core NIOS II processor on an Altera FPGA. The test applications are chosen from the Mälardalen WCET benchmark suit [10]. iv) In principle, the technique can be incorporated into a tool qualification argument under, for example, DO-178B/DO-330 [2, 3]. The main advantage of our technique is that it does not need any modification in existing COTS hardware or single-core WCET analysis tools. Only modification in an emulation device (test chip) of the production chip is required.

The paper is organized as follows. Sec. 2 presents the state-of-art related to the this paper. Sec. 3 provides necessary background to understand the core technique explained in Sec. 4. Sec. 5 tests the technique on a quad-core NIOS II platform using real applications from the Mälardalen WCET benchmark suit. Sec. 6 discusses the future extension and Sec. 8 concludes the paper.

2. RELATED WORK

In this section, we discuss the work related to measurement based WCET analysis and related to architectural reforms for predictable execution time.

Since our technique is not related to the static WCET analysis techniques, we only describe it superficially. The static approach uses abstract models of underlying hardware and the application executing on them. Later, mathematical formulation is created for Integer Linear Programming (ILP) with the optimization goal of maximizing the execution time. The static analysis formally guarantees the upper bound on the execution time. Additionally, it can analyze undesirable timing effects, e.g. timing anomalies and domino effects [9, 30]. The drawback of the static approach is the significant effort required to create abstract models of architectural components.

The hybrid measurement based WCET analysis approach [13] records traces of execution by inserting instrumentation points[1] at the beginning of each basic block[2]. Later, these traces are statically analyzed to construct the worst case path considering Maximum Observed Execution Time (MOET) of individual basic blocks. This approach is commercially utilized in the RapiTime[3] tool. The advantage of this approach is that the architecture is treated as a black box. Hence, WCET of applications executing on reasonably complex architectures can be measured. However, in its existing form, this approach cannot be used for multi-core architectures since the shared memory access latency depends on the interference on the shared resource. The interference could unpredictably increase the MOET of each basic block. Thus, the worst case path and the corresponding WCET are invalidated. Our approach complements this method by inserting statically analyzed shared memory interference information for each shared memory access which makes the hybrid approach multi-core capable.

Measurement under uninterrupted interference, as the name suggests, does measurements of execution time under synthetically created uninterrupted interference. Typically, this interference is created by executing a simple code in an infinite loop on each co-existing core. The code intentionally accesses memory such that each instruction produces a cache miss. Thus, uninterrupted traffic towards shared memory is created and it is assumed that this uninterrupted traffic produces maximum interference. The biggest advantage of this method is that it is absolutely free of charge. Additionally, the synthetic code to create the interference is trivial, change in architecture is not at all required and hybrid WCET analysis technique for single core architectures can be used for multi-cores without any modification. However, as shown in our previous work [26], this technique neither guarantees the worst case interference nor the worst case execution time except under the Priority Division [22] arbiter.

To consider the maximum possible interference due to the co-existing applications, Pellizzoni et al. [19] use memory access traces to build arrival curves of concurrently executing applications from each core. Using real-time calculus, interference bound for any access is derived. However, even a trivial bug fix in any concurrently executing application invalidates the derived bound and enforces re-analysis. Clearly, this approach restricts updates in all concurrently executing applications and task migration. Collision of memory accesses from concurrently executing applications are investigated by Lv et al. [15]. Here, model checking is used to determine maximum interference and corresponding WCET. Like [19], this approach also restricts not only application under test, but also the concurrently executing applications. In our approach, applications are analyzed in isolation. Hence, no restrictions on the co-existing applications are enforced.

In our previous works [23], [25], we analyzed applications in isolation. Here, the memory access traces are achieved from a cycle accurate simulation models and the worst case interference is computed for advanced budget based arbiters. These works have the following drawbacks, i) A cycle accurate simulator model is required for memory activity trace. ii) It could analyze only a single path applications due to the lack of tool integration. In this paper, we overcome these drawbacks.

The unpredictable interference on the shared resources is the biggest challenge of timing analysis on multi-core architectures. To avoid the interference on the shared resources all together, **tailored architectures**, built especially for predictable timing behavior, are proposed. PRET, CoMPSoC and MERASA are the examples of such architectures. Here, the predictable timing behavior is achieved by time triggered access to the shared resource - PRET [14], by intentionally delaying a shared resource access to the worst case latency - CoMPSoC [11] or by providing high priority to the accesses from the safety critical task - MERASA [29]. parMERASA is an extension of the MERASA project. It suggests a TDMA-like approach to analyzing timing behavior of multi-core systems [28]. The probabilistic execution time analysis provides an execution time distribution of the application instead of providing a single WCET value. The approach was first presented by Edgar et al. [8] and Bernat et al. [5] and recently investigated by PROARTIS [7] project. PROXIMA project extends the probabilistic approach to multi-cores. The approach employs customized components such as random cache [20] and customized bus design [12] etc. As explained

[1]Instrumentation points simply save the performance counter value in the trace to time stamp their execution.
[2]Basic block is defined as a linear code segment with single entry and single exit points.
[3]http://www.rapitasystems.com/

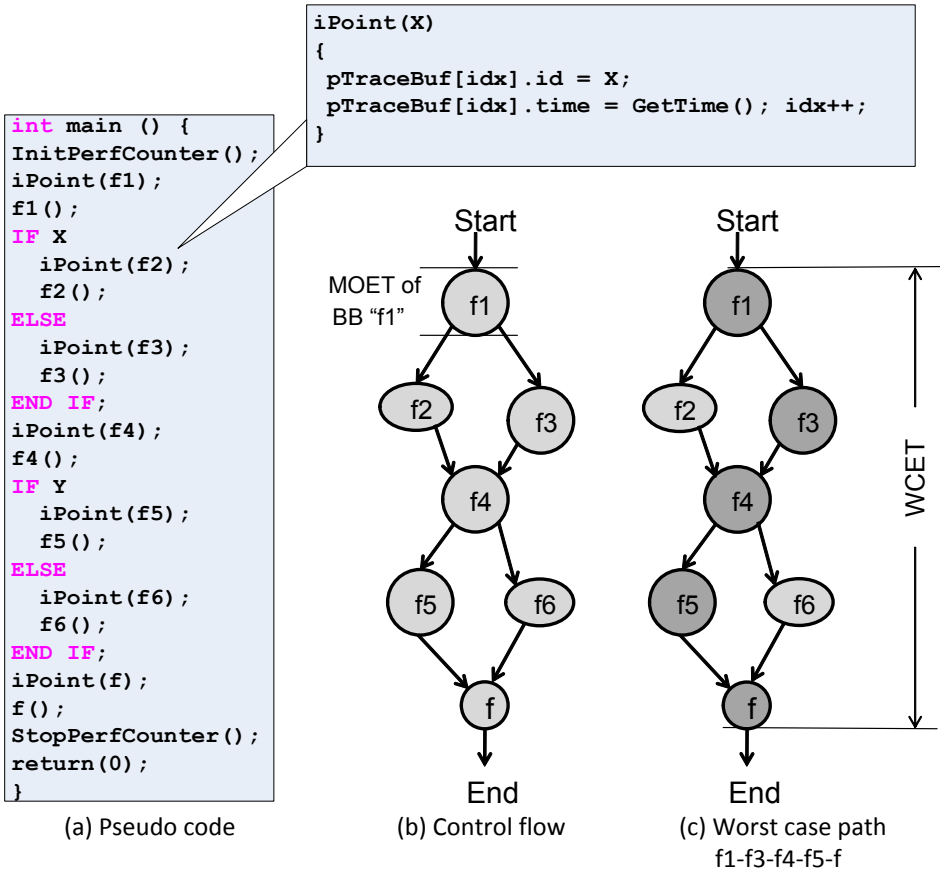

Figure 1: RapiTime basic work flow

in Sec. 1, due to the significantly small market share of hard real-time systems, economic feasibility of producing these customized architectures for safety critical systems is yet to be investigated.

The closest related work is from Nowotsch et al [16]. They propose monitoring and suspension mechanisms. The monitoring mechanism observes if a resource (shared resource in this case) is used under certain enforced limit during a unit time. If the limit is reached, the suspension mechanism is triggered which suspends the execution until the next unit time starts. The unit time (time frame) and the limits of co-existing applications are known. This information is used to determine the gradually decreasing number of interfering accesses within a unit time which helps in tightening the upper bound on WCET. Compared to their approach, our approach does not enforce any limits and time frames. Additionally, our approach, being measurement based, lets experiments directly on the emulation device of the targeted chips and does not require abstract model construction. We also provide experiment results using an existing industrial tool and a mutli-core architecture built on an FPGA.

Our approach minimally modifies the *emulation devices* of COTS components such that shared resource interference information is available to the tool. This information is traced and processed to consider the worst case interference. After the pre-processing, the measurement based WCET analysis tool is invoked. Hence, the existing measurement based WCET analysis tool does not require any modifications. Since our technique is measurement based, unlike static analysis techniques, it cannot analyze timing anomalies and domino effects.

3. BACKGROUND

This section provides the necessary background information about the hybrid measurement based WCET analysis on single core architectures. The section also explains why such measurement based approach is invalid in the case of multi-core architectures. The invalidity is explained by an example of a Round Robin (RR) arbiter, however, the explanation is valid for any dynamically scheduled arbiter.

3.1 Hybrid Measurement Based WCET analysis

The hybrid measurement based WCET analysis combines static code analysis and execution time measurements. This section explains the basic work flow of the RapiTime tool from Rapita systems Ltd.

The tool statically parses the source code to identify function boundaries, conditional structures and control flow. The Fig. 1 depicts the basic work flow. At first, the pseudo code of Fig. 1(a) is statically analyzed and control flow graph is created (Fig. 1(b)). In the next step, light weight instrumentation points (`iPoint()`) are inserted at the beginning of each basic block – BB. The instrumented code is then executed on the targeted platform multiple times using different test vectors. During each execution,

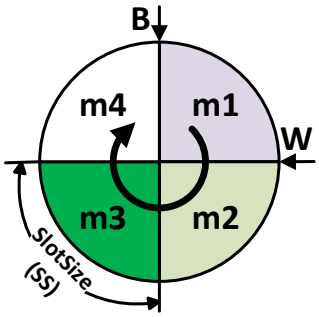

Figure 2: Graphical View of the Round Robin Operation

the `iPoint()` registers the starting time of each BB in a trace memory. The time is acquired from a hardware performance counter[4] in terms of number of clock cycles. Thus, depending on the test vector coverage, starting times of each basic block are recorded in a trace.

The recorded trace (aka `iPoint()` trace) is analyzed by the RapiTime and execution time of each BB is derived. The execution time of a BB is derived by subtracting its starting time from the starting time of the next BB in the control flow graph. Since a BB may experience different execution time depending on the input test vector, execution time profile of each BB is created. From the execution time profile, RapiTime selects the maximum of the recorded execution times, termed as MOET– Maximum Observed Execution Time. The control flow graph of Fig. 1(b) is then populated by the MOET of each BB. In the figure, the higher execution times are represented by bigger circles.

At last, MOET of BB on all paths from `Start` to `End` are considered to select the longest path which is termed as the worst case path. The WCET is the addition of MOET of all basic blocks falling on the worst case path.

There are many advantages of this technique. i) Since the targeted platform itself is used for measurements, costs of building precise hardware model of the target is avoided. ii) Due to the static structural analysis, test coverage information can be obtained together with the timing information. iii) Hotspots for optimizations, i.e. BB on the worst case path, are immediately identified. Despite all the advantages, this technique is limited to single-core architectures. As explained in the following subsection, the technique cannot be used in multi-core architectures with a dynamically scheduled shared resource arbiter.

3.2 Work Conserving Round Robin Arbitration

The Fig. 2 depicts the Round Robin (RR) arbitration graphically. The RR arbiter is a dynamically scheduled shared resource arbiter (a greedy version of the static TDMA arbiter). Under the RR scheme, the shared resource contenders are assigned a fixed number of slots in a virtual ring. The assigned number of slots depends on their bandwidth requirements. Although, our analysis is valid for any slot allotment, for simplicity, we consider one slot per contender without loss of generality. The figure shows four contenders (master1, master2, master3 and master4) in the ring. Here, we assume that these contenders are processor cores executing independent applications and the shared resource is a shared main memory. These cores access the shared main memory when a cache miss occurs. *Throughout the paper, we use master and core terms interchangeably. Similarly, we use memory and shared memory interchangeably.*

The arbiter continuously searches for a master which wants to access the memory in a clock-wise direction. We call this master an active master. As soon as an active master is encountered, it is granted the memory for a predefined maximum number of clock cycles (`SlotSize - SS`). The `SS` is big enough to accommodate the burst issued for one cache-line fill. After the granted master finishes its burst access, the search process resumes from the next slot in the ring. Thus, the memory is always occupied as long as there is at least one active master (hence the name, "*work conserving*").

Now, let us assume that the application-under-test is executing on `m1` and the `SS` is same for all masters. For this architecture, an access request from `m1` experiences the worst case completion latency ($W_L = 4 \times SS$) if it is issued when the arbiter pointer is at **W** in Fig. 2 *AND* all other masters utilize their slots. Similarly, an access request experiences the best case completion latency ($B_L = 1 \times SS$) if it is issued when the arbiter pointer is at **B** in the figure. In general, the completion latency could be any value in the range $[B_L, W_L]$.

In this paper, by latency of an access, we mean completion latency of an access (scheduling latency + time required to complete the access).

From the above explanation, it is clear that the latency of a cache miss depends on the arbiter pointer and the activity of other masters at the time of its occurrence. Thus, applications executing on these masters heavily influence the execution time of each other. Here, the execution time of a basic block deviates significantly and unpredictably depending on the experienced interference on the shared memory. Hence, the measured execution time of any basic block may not be the MOET of that basic block. Consequently, constructing the worst case path depending on the measured execution time of basic blocks (Sec. 3.1) is invalid.

4. WORST CASE INTERFERENCE AUGMENTED TRACING

This section explains how the hybrid measurement based WCET analysis (Sec. 3.1) can be used for multi-core architectures. Our technique inserts a tiny module to multi-core architectures which enables the hybrid measurement based WCET analysis for multi-core architectures. The module is used only during analysis and can be turned off during application deployment. Hence, emulation chips are the perfect target for our technology.

The first part introduces the basic technique and the second part presents an optimized version.

4.1 The Basic Technique

The Fig. 3 depicts a multi-core architecture with N number of cores and a *cache observer* module. The module incorporates a performance counter, which is a standard component, a cache observation unit and a bus master

[4]Trace capture devices like RTBx from Rapita systems Ltd or logic analyzers can also be used for time stamping and trace storage. The advantage of using a dedicated trace capture device is the availability of large trace memory.

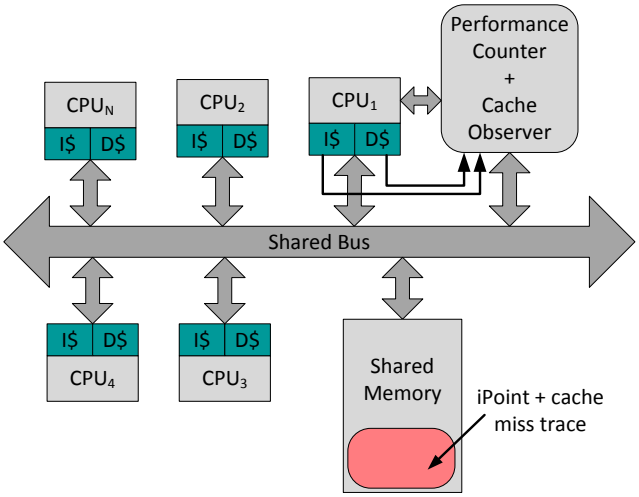

Figure 3: Cache Observer

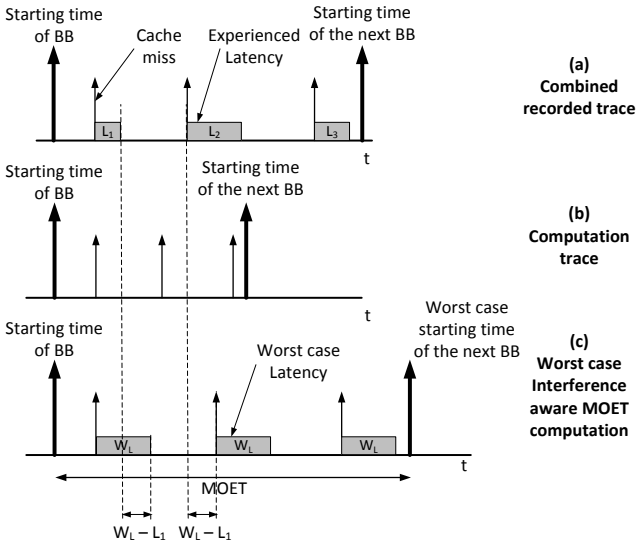

Figure 4: Trace Manipulation

interface.

The instrumented code of Sec. 3.1 is executed on this architecture (CPU_1 in this case) and the `iPoint()` trace is captured in the memory. The module observes the caches during instrumented code execution and as soon as a cache miss occurs, it records time of its occurrence and the experienced latency in a trace. Note that the experienced latency can be any value in the range $[B_L, W_L]$[5] depending on the activities of other masters and location of arbiter pointer (Sec. 3.2). Now, the memory contains two traces, an `iPoint()` trace and a cache miss trace. Due to the time stamping, this traces can be easily combined to form a single trace. The combined trace is depicted in Fig. 4(a). The figure depicts execution trace of a single basic block.

The combined trace is processed on the host machine before RapiTime analyzes it. At first, all the experienced latencies of the combined trace are removed. This results in a computation trace [26] of Fig. 4(b). The trace is called a computation trace since it incorporates timing of pure computation and cache hits. The main memory access latencies (due to cache misses) are absent in the computation trace. Note that latency of each cache miss delays the occurrence of subsequent cache miss and starting of the next basic block by the same amount. Therefore, when latency associated to the first cache miss is removed, placeholders of all subsequent cache misses and the starting time of the next basic block are shifted towards left in time. The process is repeated for all cache misses. Clearly, the place holder of the first cache miss remains at the same place.

In the next step, the theoretical worst case latency (W_L) is appended to each cache miss place holder. This shifts each subsequent cache miss placeholders and the starting time of the next basic block to the right. The process is repeated for all cache misses. The resulting trace is depicted in the Fig. 4(c). Thus, execution time of each basic block is *artificially inflated* to consider the theoretical worst case latency for each cache miss occurring inside the respected basic block.

The Fig. 5 represents the post processing of the combined trace on the control flow graph level. Had we considered only the combined trace, the control flow graph would have been as depicted in Fig. 5(a). Instead, the control flow graph is populated using the *artificially inflated* MOET of the basic blocks, as depicted in Fig. 5(b). Here, an MOET incorporates worst case interference. The RapiTime analyzes the control flow graph of Fig. 5(b) to construct the worst case path and the WCET. The estimated WCET is now compensated for the worst case latency for each cache miss. Notice that the worst case path considering the worst case latency is f1-f2-f4-f5-f instead of f1-f3-f4-f5-f as depicted in Fig. 1(c) for single-core architectures.

Advantages: There are many advantages of this technique i) Unlike [14] and [11], it does not alter the performance of the chip. ii) Algorithms for the worst case analysis under advanced arbitration techniques, e.g. Priority Based Budget Scheduler (PBS) [27] and Credit Controlled Static Priority (CCSP) [4] can be used. iii) Minimal modifications to the existing architecture is required. iv) Although the area overhead of the technique is negligible, the implementation is limited to emulation devices only. This avoids any alteration to the production chips. v) The existing tools, e.g. RapiTime, does not need any modifications.

Disadvantages: There are two drawbacks of the above mentioned technique. i) For some applications, large number of cache misses may overflow the trace memory. ii) An additional master interface of the cache observer increases capacitive loading on the shared bus which may result in slower operating frequencies.

To circumvent these drawbacks, the next subsection presents an optimized version of the cache observer. In the optimized version the master interface to the shared bus is removed. Hence, it does not generate any cache miss trace. Moreover, it has ultra low area footprint.

[5]In this set-up experienced latency for some cache misses could be more than the theoretical maximum. The W_L is calculated based on the maximum number of co-existing masters. During the instrumented code execution, there is one additional co-existing master – cache observer. However, the cache observer is turned off during real application deployment. Hence, the theoretical worst case latency, W_L, should not include interference from the cache observer.

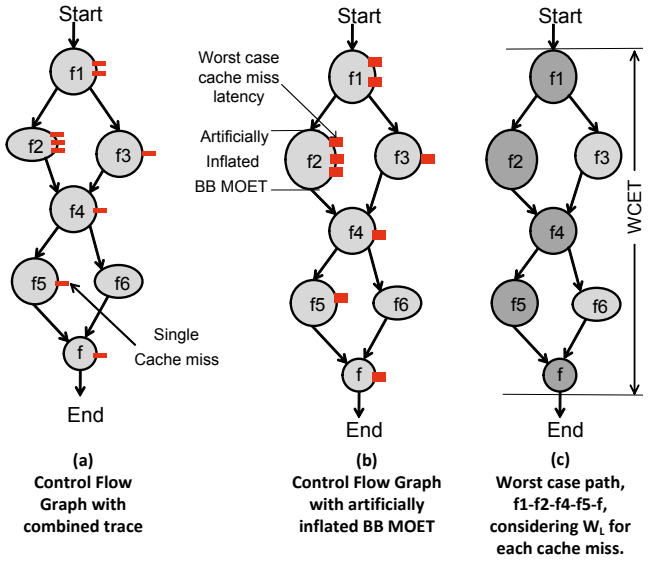

Figure 5: Construction of the worst case path considering worst case latency for each cache miss

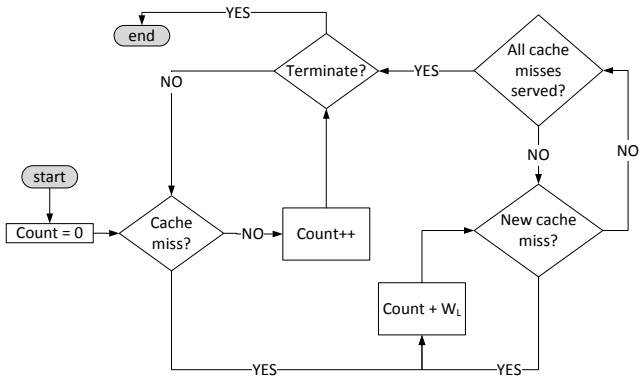

Figure 6: Modified performance counter logic

4.2 The Optimized Technique

The optimized version of the cache observer contains only a modified performance counter and the cache observation logic. The algorithm of the performance counter is depicted in Fig. 6. The performance counter, if enabled, continuously observes the cache. On every clock tick, if a cache miss does not occur and none of the cache miss is pending to be served, it increments the `count` by one. This is a standard performance counter operation. However, if a cache miss occurs, the `count` is immediately incremented by W_L and paused until the cache miss is served. During the paused state, if another cache miss occurs, the `count` is again incremented by W_L. As soon as all pending cache misses are served completely, the `count` resumes the standard performance counter operation.

The intuition behind the idea can be explained by an example. Let us assume at `count` value t, a cache miss occurs. This cache miss takes L number of clock cycles, including interference and data transfer, to be served. Hence, the experienced latency is L. In absolute time, the value of the `count` after the cache miss is served should be $t + L$. However, we add W_L to the `count` as soon as a cache miss occurs and pause until the cache miss is served completely. Thus, after the cache miss is served, the value of the `count` is $t+W_L$. Note that the difference between the artificially extended time and the absolute time is $W_L - L$.

The time is shifted by $W_L - L$ during the post processing step in the non-optimized version of Sec. 4.1. In the optimized version, the time is already shifted, therefore, the cache miss trace generation is not required. Here, when an instrumented code is executed on the architecture, the `GetTime()` method of an `iPoint()` returns the artificially forwarded time. Note that the `count` value is already compensated for the worst case interference. Hence, the MOET calculated based on it by RapiTime is also worst case interference compensated.

There is certainly a cost of the simple architecture in terms of accuracy. The optimized version can only add constant value, W_L, to the `count` for a single cache miss. As explained in Sec. 3.2, the consideration of the constant worst case latency for every cache miss is perfectly suitable for the round robin arbiter. However, for advanced arbiters, e.g. CCSP [23] and PBS [25] the worst case latency for any cache miss depends on the time of its occurrence. Assuming a constant worst case latency under these arbiters will significantly increase the WCET of application.

4.3 Over estimation

It is clear that, like hybrid WCET measurement technique, our technique is also intrusive. Here, the measured WCET contains execution time impact of instrumentation points. An intuitive way to compensate for instrumentation is to analyze the number of clock cycles (N_{ip}) required per instrumentation point. The instrumentation overhead, Nip, is then subtracted from every basic block in the post-processed trace data. However, this compensation is not enough due to the following reasons. **i)** The instrumentation code changes state of instruction cache. Hence, more number of cache misses occur compared to that of the non-instrumented code. **ii)** Cache misses originating due to the write-back of instrumentation data are treated indifferently by our technique. Hence, also for these cache misses, the worst case latencies are considered. However, these cache misses do not occur in the deployed application due to the absence of instrumentation. **iii)** The *always taken* branches of instrumentation code may impact the decision making of history based branch predictors. All the above mentioned factors contribute in yielding a pessimistic WCET.

Note that the assumption of the worst case latency, W_L, for every cache miss is a conservative assumption. As presented in [26], certain applications are highly vulnerable to worst case latencies due to their shared resource access pattern. If the analysis presented in [26] is not available then assuming W_L for every shared resource access is the only safe assumption.

In order to determine the pessimism originating due to the instrumentation[6], we determine the worst case path

[6]The instrumentation code occupies space in caches. Hence, the effective cache size available to the application is reduced. This increases number of cache misses during instrumented code execution, and consecutively its WCET, compared to that of the non-instrumented code execution. Therefore, measurement of execution time on instrumented code does not yield optimistic results.

Architecture	Logic Elements
Without the Cache observer	13555
With the Cache observer	14272

Table 1: Synthesis results

using an instrumented code of application. Once the worst case path is determined, we remove the instrumentation[7]. We then re-execute the non-instrumented worst case path on the target platform and measure start to end execution time using our optimized time stamping module. Thus, the measured execution time is not an absolute execution time, but worst case interference compensated execution time. The measured execution time does not have any impact of instrumentation, however, it is worst case interference compensated. We denote this execution time as $WCET_{ni}$ (ni stands for non-instrumented).

Irrespective of single core or multi-core, it could happen that after removal of instrumentation points, some other path is the worst case path. Hence, intensive end-to-end measurements must be performed on non-instrumented code.

4.4 Emulation devices

For the completeness of the paper, we briefly explain the design flow of the emulation devices. Semiconductor vendors (e.g. Freescale, Infinion etc) typically make few chips (test chips or emulation devices) with enhanced debugging facilities. These chips are sold to the product/application developers (Robert Bosch, Continental etc). The application developers test their code vigorously on the emulation devices and use their enhanced debugging facilities to be confident about the correctness (functional and non-functional) of their application code. Thus, the emulation devices go only in the test products (test vehicles in this case). After being confident about their applications, production chips in large quantity is ordered from semi-conductor vendors. These production chips go in the real product (production cars in this case).

5. TEST CASES

This section provides information about tests that were carried out to support our technique. The first subsection describes the test architecture and the second subsection discusses the results.

5.1 Test Architecture

We implemented a quad-core processor using Altera NIOS II F cores on Altera Cyclone III FPGA. Each core is equipped with 512 Bytes of instruction and data caches in the first experiment. In the second experiment, the cache size is increased to 4K Bytes. The cache-line size is 32 Bytes. An on-chip SRAM serves as a shared main memory and trace storage. In this paper, we used only L1 caches and a shared main memory. Measurement of WCET in the presence of shared L2 cache is explained in the Sec. 6. The test architecture is depicted in Fig. 7. We avoided use of

[7]The instrumentation is required to determine the MOET of basic blocks, test coverage and thereby, the worst case path. After obtaining the worst case path, the instrumentation is not required anymore.

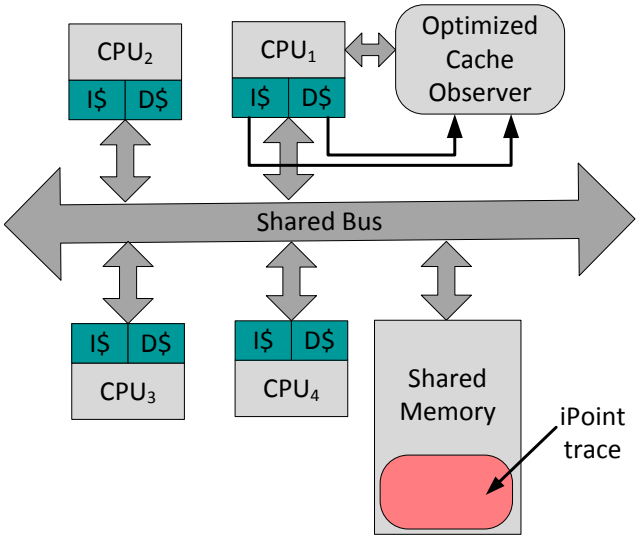

Figure 7: Test Architecture

wait states during burst transfers and arbiter lock signals as suggested in [24] to produce a valid WCET bound.

Here, core1 executes test applications while other cores execute dummy load to stress the shared main memory. Test applications are chosen from the Mälardalen WCET benchmark suit [10]. We typically chose the multi-path applications from the suit since the single path applications are already tested in our another work [22]. For testing purpose, we used the optimized version (Sec. 4.2) of our cache observer. The operating frequency is 125 MHz.

The area impact of our technique is presented in Table 1. The overhead of our technique is ≈5%. This overhead has a minuscule cost impact due to the following reasons. i) This increase is only limited to emulation devices. The production chips remain unchanged. ii) The increase is compared to our basic test architecture. COTS multi-core architectures contain hardware accelerators, DMAs, I/O controllers etc. in addition to the components shown in Fig. 7 which makes the COTS architecture much larger than our test architecture. However, our technique retains its size. Thus, increase in area compared to a COTS architecture is much less than 5%.

5.2 Results

Table 2 and Table 3 present results of the tests. In the tables, the results are divided into two parts: Instrumented execution and Non-instrumented execution. At first, an instrumented application is executed on the test architecture and its `iPoint` trace is collected. Remember that this `iPoint` trace is already compensated for the worst case interference (Sec. 4). The trace is then analyzed by the RapiTime timing analyzer. As explained in Sec. 3.1, RapiTime detects the worst case path and also calculates WCET. The calculated WCET is shown in the tables under instrumented execution column. The worst case path is re-executed on the architecture, this time – non-instrumented, and its start to end (worst case interference compensated) time is measured. This measured time is considered as the actual WCET of application since the deployed applications are non-instrumented.

Bench-mark	Instrumented Execution				Non-instrumented Execution			
	WCET	MOET	$WCET_s$	$\frac{WCET}{WCET_s}$	$WCET_{ni}$	$WCET_{nis}$	$\frac{WCET_{ni}}{WCET_{nis}}$	$\frac{WCET}{WCET_{ni}}$
Binary search	2857	2662	1729	1.65	775	415	1.86	3.68
BBSort	133570	111935	72896	1.83	20941	20362	1.02	6.37
cnt	54930	35182	28435	1.93	8900	7772	1.14	6.17
insertsort	20741	17310	10081	2.05	4096	3590	1.14	5.06
ludcmp	515703	423071	241266	2.13	456778	220309	2.07	1.12
ndes	1468191	1161950	697753	2.10	623355	295531	2.10	2.35
ns	194401	163983	119977	1.62	83165	36555	2.27	2.33
nsichneu	183104	152792	86492	2.11	64738	25996	2.49	2.82
qsort-exam	101364	82574	46289	2.18	74008	31887	2.32	1.36
select	153437	131381	60855	2.52	47025	21867	2.15	3.26
ST	59384913	45377728	26377862	2.25	51651564	23957615	2.15	1.14

Table 2: Test Results: Execution Times in Clock Cycles, 512 Bytes I$ and D$.

Bench-mark	Instrumented Execution				Non-instrumented Execution			
	WCET	MOET	$WCET_s$	$\frac{WCET}{WCET_s}$	$WCET_{ni}$	$WCET_{nis}$	$\frac{WCET_{ni}}{WCET_{nis}}$	$\frac{WCET}{WCET_{ni}}$
Binary search	2382	2093	1553	1.53	705	395	1.78	3.37
BBSort	71618	68254	60293	1.18	20704	20294	1.02	3.45
cnt	27405	25935	22701	1.20	7975	7612	1.04	3.43
insertsort	7789	7686	6923	1.12	4026	3570	1.12	1.93
ludcmp	205085	184567	159866	1.28	182309	146528	1.24	1.12
ndes	748654	655750	523350	1.43	357602	209319	1.70	2.09
ns	130930	122743	106831	1.22	37794	35266	1.07	3.46
nsichneu	126971	105534	73221	1.73	54383	23563	2.30	2.33
qsort-exam	33582	31957	29058	1.15	19189	17162	1.11	1.75
select	30938	29447	26769	1.15	15042	13383	1.12	2.05
ST	25334087	19541872	17578497	1.44	19209880	15164451	1.26	1.31

Table 3: Test Results: Execution Times in Clock Cycles, 4 KBytes I$ and D$.

MOET is the maximum observed execution time during several iteration of execution in the presence of varying dummy loads on co-existing cores. $WCET_s$ stands for the WCET of an application on a single-core (no-interference) architecture. Similarly, $WCET_{ni}$ and $WCET_{nis}$ stand for non-instrumented quad-core and non-instrumented single-core WCET, respectively. The factors $WCET/WCET_s$ and $WCET_{ni}/WCET_{nis}$ provide respective increase in the WCET of an application when the application is ported from a single-core architecture to a quad-core architecture.

The increase in the WCET is due to the interference on the shared resources (here, memory). Those applications which do less number of shared resource accesses per unit time, experience less increase in the WCET. This is demonstrated by the results of Table 3. Here, the cache size is increased to 4 KBytes. In Table 2, the average of $WCET_{ni}/WCET_{nis}$ is 1.89 while in Table 3, the average of $WCET_{ni}/WCET_{nis}$ is 1.35. When the cache size increases, less number of cache misses occur during application execution. Hence, the worst case penalty W_L for accessing the shared resource must be considered for less number of times. This decreases the $WCET_{ni}/WCET_{nis}$ factor.

The last column in the tables presents the increase in WCET due to the instrumentation. The increase is application and cache size dependent. If a test application has many branches (many basic blocks), then high number of instrumentation points impacts the cache state and increases the WCET. However, this increase is moderated if bigger caches are used. The number of instrumentation points does not change with the change in the underlying cache size. Hence, the impact of instrumentation is diluted in bigger caches. The evidence can be seen in the tables. In table 2, average value of $WCET/WCET_{ni}$ is 3.24 while in the table 3, the average value is 2.39. Remember that the results in table 3 are for a bigger cache.

6. FUTURE EXTENSION

COTS processors often use shared L2 cache. If a cache miss in L1 cache occurs, at first, the shared L2 cache is accessed. If the required data is unavailable in the L2 cache (L2 cache miss) then the main memory (mostly an SDRAM) is accessed. Typically, the penalty associated with the L2 cache miss is large since the data has to be fetched from a slow off-chip memory. If the L2 cache is used without any protection and partition, one core can evict the useful data of another core from the shared L2 cache. Hence, these cores, in the worst case, experience an L2 cache miss for every L1 cache miss. This tremendously increases the WCET of applications executing on architectures with unprotected shared L2 caches.

We propose a shared architecture as depicted in the Fig. 8. One of the cores, core1 in the figure, programs

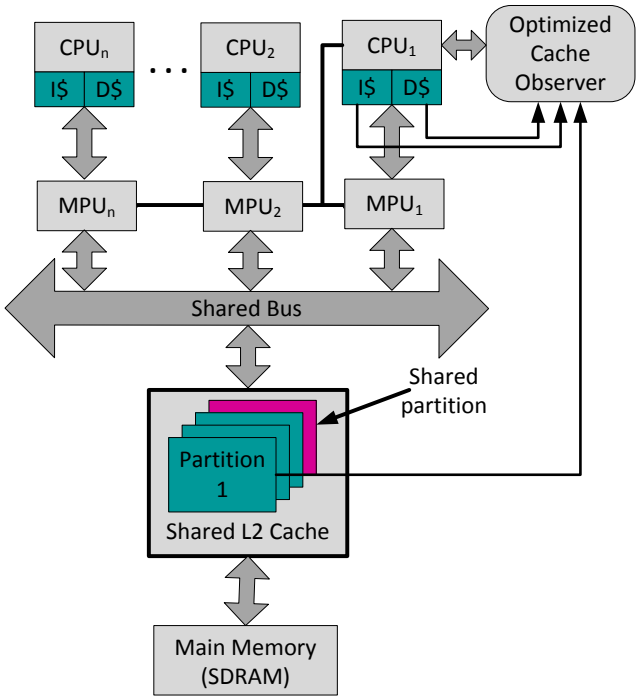

Figure 8: Proposed Architecture with a Shared L2 Cache

all Memory Protection Units (MPU). Here, the shared L2 cache is partitioned. The partitioning should be achieved by address space separation (logical partitioning) rather than having hardware partitions. This enables changes in partition size dedicated to each core according to the memory requirements of deployed applications. An MPU forwards only those requests which fall into the allocated address space. Thus, every core has a reserved space in the shared L2 cache and one core cannot evict data of another core. Our approach extends the MERASA [18] partitioned L2 cache by adding a shared partition and the cache observation module.

One of the partitions is termed as a shared partition where each core has an access right. The shared partition is required if applications need to exchange data (e.g. Symmetric Multi-processing). The data in the shared partition is unpredictable since each core can modify it. Hence, an access to the shared partition of the L2 cache should be considered as an L2 cache miss during the WCET analysis.

Our cache observer must observe, apart from L1 caches, the partition dedicated to the core1. As soon as an L2 cache miss originates from the partition dedicated to the core1, the worst case latency to access the off-chip memory is added to the performance counter. Similarly, the worst case penalty for accessing the off-chip memory is added to the performance counter as soon as an access to the shared partition of the L2 cache is detected.

It is obvious that the insertion of MPUs *must* retain in production chips making it an expensive modification. There are following two alternatives to avoid the expensive modification, however, both have their drawbacks. **i)** A similar logical partitioning can be achieved using Memory Management Unit (MMU) of each core. However, in this approach, it must be assumed that each core configures its own MMU correctly at boot-up. This may increase the certification costs since the boot-up code of each core must be certified at the highest level. **ii)** Avoid partitioning altogether (entire L2 cache is shared) and assume during the analysis that each L1 cache miss results in an L2 cache miss. Subsequently, for every L1 cache miss, the worst case latency of L1 miss and L2 miss must be considered for the WCET analysis. This results in an overly pessimistic WCET since the L2 cache miss latency is far greater than the L1 cache miss latency and during real execution, only a fraction of L1 cache misses results in L2 cache misses.

7. DISCUSSION

Porting applications from a single core architecture to a multi-core architecture increases its WCET. This observation has also been made in literature [17], [21] and more recently, in [6]. The increase in the WCET is often argued against the use of multi-cores in safety critical systems. We are rather optimistic because of the following reasons.

Free Lunch: For years, general purpose applications were running systematically faster with each new version of the processor due to the higher operating frequency of the new processors. This "free lunch" is over for general purpose computing. However, the free lunch is still available for HRT systems. Current safety critical applications execute on relatively simple cores operating at $\approx$ 100 MHz. The current multi-core COTS processors operate at $\approx$ 800 MHz. The higher operating frequencies compensate for the increase in WCET in terms of clock cycles and keep it under enforced limits in terms of milliseconds. However, more research is required to be confident about it.

Mixed critical systems: Note that the WCET of applications increase when ported to multi-core architecture, not the average case execution time (ACET). This can be exploited well by mixed critical systems. A Soft Real-time Task (SRT) of a mixed critical system should be provided resources considering the ACET of co-existing HRT, instead of considering WCET. Since, most of the time the HRT will complete its execution faster than its WCET, the freed resources can be used by the SRT. Only, when the HRT experiences the worst case circumstances, the SRT may miss a deadline which is harmless.

8. CONCLUSION

This paper has presented a novel technique to measure the WCET of applications on multi-core architectures. The technique empowers existing single-core WCET measurement tools to measure WCET on multi-core architectures without any modifications. This has been demonstrated by measuring the WCET of Mälardalen benchmark applications using RapiTime timing analyzer. The experimental quad-core processor is implemented on Altera Cyclone III FPGA using NIOS II cores. The technique inserts a cache observation and time stamping module in COTS multi-core processor. However, this modification of an existing COTS processor is limited to the emulation devices only. Thus, mass produced COTS devices stay unchanged preserving their economical and performance benefits. The technique stands out from existing work since it does not need any modification in either commercialized tools or commercialized architectures. The test applications are analyzed in isolation and the worst

possible interference on the shared resource is assumed from co-existing applications.

The technique presented in this paper is scalable for shared L2 cache architecture. Although the technique is demonstrated using a Round robin arbiter and a shared SRAM, it can be used for any real-time arbiter and any shared memory in multi-core processors.

9. ACKNOWLEDGMENTS

This work was funded by German BMBF projects ECU (13N11936) and Car2X (13N11933). The work is also partially supported by the EC FP7 project parMERASA under Grant Agreement No. 287519.

10. REFERENCES

[1] Profiling nios ii systems.

[2] Rtca sc-167, software considerations in airborne systems and equipment certification, december 1992.

[3] Rtca sc-167, software tool qualification considerations, may 2012.

[4] B. Akesson, K. Goossens, and M. Ringhofer. Predator: a predictable SDRAM memory controller. CODES+ISSS '07, NY, USA.

[5] G. Bernat, A. Colin, and S. Petters. Wcet analysis of probabilistic hard real-time systems. In *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*, pages 279–288, 2002.

[6] J. Bin *et al.* Studying co-running avionic real-time applications on multi-core cots architectures. In *ERTS '14*.

[7] F. Cazorla *et al.* Proartis: Probabilistically analyzable real-time systems. *ACM Trans. Embed. Comput. Syst.*, 2013.

[8] S. Edgar and A. Burns. Statistical analysis of wcet for scheduling. In *Real-Time Systems Symposium, 2001. (RTSS 2001). Proceedings. 22nd IEEE*, pages 215–224, 2001.

[9] G. Gebhard. Timing Anomalies Reloaded. In *WCET 2010*, Dagstuhl, Germany.

[10] J. Gustafsson *et al.* The Mälardalen WCET benchmarks – past, present and future.

[11] A. Hansson *et al.* Compsoc: A template for composable and predictable multi-processor system on chips. *ACM Trans. Des. Autom. Electron. Syst.*, 2009.

[12] J. Jalle, L. Kosmidis, J. Abella, E. Quiñones, and F. J. Cazorla. Bus designs for time-probabilistic multicore processors. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, pages 50:1–50:6, 3001 Leuven, Belgium, Belgium, 2014. European Design and Automation Association.

[13] R. Kirner, I. Wenzel, B. Rieder, and P. Puschner. Using measurements as a complement to static worst-case execution time analysis. In *Intelligent Systems at the Service of Mankind*, volume 2. UBooks Verlag, Dec. 2005.

[14] B. Lickly *et al.* Predictable programming on a precision timed architecture. CASES '08.

[15] M. Lv *et al.* Combining abstract interpretation with model checking for timing analysis of multicore software. In *Proc. RTSS*, 2010.

[16] J. Nowotsch, M. Paulitsch, A. Henrichsen, W. Pongratz, and A. Schacht. Monitoring and wcet analysis in cots multi-core-soc-based mixed-criticality systems. In *DATE '14*.

[17] J. Nowotsch *et al.* Leveraging multi-core computing architectures in avionics. In *EDCC '12*.

[18] M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero. Hardware support for wcet analysis of hard real-time multicore systems. *SIGARCH Comput. Archit. News*, 37:57–68, June 2009.

[19] R. Pellizzoni *et al.* Worst case delay analysis for memory interference in multicore systems. In *Proc. DATE*, 2010.

[20] E. Quinones, E. Berger, G. Bernat, and F. Cazorla. Using randomized caches in probabilistic real-time systems. In *Real-Time Systems. ECRTS '09.*

[21] P. Radojković *et al.* On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments. *TACO*, 2012.

[22] H. Shah, K. Huang, and A. Knoll. The priority division arbiter for low wcet and high resource utilization in multi-core architectures. In *RTNS 2014, Versailles, France.*

[23] H. Shah, A. Knoll, and B. Akesson. Bounding sdram interference: detailed analysis vs. latency-rate analysis. In *Date '13, Grenoble, France.*

[24] H. Shah, A. Raabe, and A. Knoll. Challenges of wcet analysis in cots multi-core due to different levels of abstraction. *Workshop on High-performance and Real-time Embedded Systems (HiRES 2013)*, 2013.

[25] H. Shah *et al.* Bounding WCET of Applications Using SDRAM with Priority Based Budget Scheduling in MPSoCs. In *Proc. DATE*, 2012.

[26] H. Shah *et al.* Timing Anomalies in Multi-core Architectures due to the Interference on the Shared Resources. 2014.

[27] M. Steine *et al.* A priority-based budget scheduler with conservative dataflow model. In *Proc. DSD*, 2009.

[28] T. Ungerer, C. Bradatsch, M. Gerdes, F. Kluge, R. Jahr, J. Mische, J. Fernandes, P. Zaykov, Z. Petrov, B. Boddeker, et al. parmerasa–multi-core execution of parallelised hard real-time applications supporting analysability. In *Digital System Design (DSD), 2013 Euromicro Conference on*, pages 363–370. IEEE, 2013.

[29] T. Ungerer and et al. Merasa: Multicore execution of hard real-time applications supporting analyzability. *IEEE Micro*, 30:66–75, September 2010.

[30] R. Wilhelm *et al.* Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. 28(7), 2009.