Technical Report

An Analysis of the Impact of Bus Contention on the WCET in Multicores

Dakshina Dasari

Vincent Nelis

HURRAY-TR-120504

Version:

Date: 05-13-2012

An Analysis of the Impact of Bus Contention on the WCET in Multicores

Dakshina Dasari, Vincent Nelis

IPP-HURRAY!

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail: dndi@isep.ipp.pt, nelis@isep.ipp.pt

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

Abstract

The use of multicores is becoming widespread inthe field of embedded systems, many of which have real-time requirements. Hence, ensuring that real-time applications meet their timing constraints is a pre-requisite before deploying them on these systems. This necessitates the consideration of the impact of the contention due to shared low-level hardware resources like the front-side bus (FSB) on the Worst-CaseExecution Time (WCET) of the tasks. Towards this aim, this paper proposes a method to determine an upper bound on the number of bus requests that tasks executing on a core can generate in a given time interval. We show that our method yields tighter upper bounds in comparison with the state of-the-art. We then apply our method to compute the extra contention delay incurred by tasks, when they are co-scheduled on different cores and access the shared main memory, using a shared bus, access to which is granted using a round-robin arbitration (RR) protocol.

An Analysis of the Impact of Bus Contention on the WCET in Multicores

Dakshina Dasari, Vincent Nelis
CISTER-ISEP Research Centre, Polytechnic Institute of Porto, Portugal
{dndi, nelis}@isep.ipp.pt

***Abstract*—The use of multicores is becoming widespread in the field of embedded systems, many of which have real-time requirements. Hence, ensuring that real-time applications meet their timing constraints is a pre-requisite before deploying them on these systems. This necessitates the consideration of the impact of the contention due to shared low-level hardware resources like the front-side bus (FSB) on the Worst-Case Execution Time (WCET) of the tasks. Towards this aim, this paper proposes a method to determine an upper bound on the number of bus requests that tasks executing on a core can generate in a given time interval. We show that our method yields tighter upper bounds in comparison with the state-of-the-art. We then apply our method to compute the extra contention delay incurred by tasks, when they are co-scheduled on different cores and access the shared main memory, using a shared bus, access to which is granted using a round-robin arbitration (RR) protocol.**

I. Introduction and Motivation

Embedded systems are increasing at a rapid rate and so is the percentage of processors designed for such systems. Almost 99% of the processors that are manufactured today are deployed in the embedded market. Apart from having specific functional requirements, many of the applications in embedded systems have stringent timing requirements. For systems exposing such timing requirements (called "real-time" embedded systems), researchers have proposed many scheduling algorithms over the last decades, together with associated "schedulability analysis", that enable certification agencies to verify at design time whether the system will always fulfill its timing requirements at run-time. The rigorousness of the certification process varies according to the "Safety Integrity Level" (SIL) of the task under scrutiny, where every task of the system is assigned a SIL reflecting its level of "criticality".

The desired "wish-list" of embedded applications across the market segments is a computing platform that can provide high performance with reduced SWaP (size, weight and power) properties and multicore systems have naturally emerged as a promising solution. When deployed on the same multicore system, tasks of different SILs can co-exist and share some low-level hardware resources such as cores, cache subsystems, communication buses and main memory. It is of *chief importance* to understand that, *unless these tasks of different SILs are shown to be sufficiently independent*, the standards require that the hardware and software are developed at the highest SIL among the SILs of all these tasks, which is very expensive. This requirement is clearly stated in the automotive domain (req. 7.4.2.3 of ISO 26262-4 [1]), as well as in the international standard (req. 7.6.2.10 of IEC 61508 [2]). This is why substantial efforts are put to (i) render the tasks of a same SIL as independent and isolated as possible from the tasks with different SILs and (ii) upper-bound the impact that the execution of the tasks of a same SIL may have on the execution behavior of the tasks of different SILs, with the objective of certifying each subset of tasks at its own SIL level.

In order to limit the risk of failure of tasks with high SILs, systems must be designed to isolate the execution of the tasks, both in the spatial and temporal domains, and if total isolation cannot be achieved then designers must be able to upper-bound the impact that tasks executions have on each other. To cater to these requirements, international standards typically favor simple and safe designs such as partitioned scheduling, partitioned caches[1], time-triggered architectures and cyclic scheduling algorithms (CSA) as recommended in [2] (Annex F, page 103).

Unfortunately by design constraints, complete isolation of tasks by partitioning at the hardware level has its limit. A simple manifestation of this is the range of multicore designs (from different vendors) in which the cores are typically connected to a shared off-chip main memory by a single shared communication channel (which does not conform to the "total-isolation" paradigm). In the Intel L7400 and E8n00 series processors for example, this communication channel is referred to as Front-Side-Bus or FSB. In the Infineon Tricore architecture, the bus subsystem is duplicated enabling the cores to communicate to the memories in parallel, without conflicts, unless the cores access the same memory subsystem (in which case the same problem persists). Hereafter, we assume that the cores communicate with the main memory over a single shared FSB.

As the traffic on the shared FSB increases, the bus gets saturated and thus becomes a bottleneck causing tasks to stall during their execution. The difference between the processor speed and the time to access the main memory is high, leading to a non-negligible increase in the WCET of the tasks assigned to these cores. Since the contention for the FSB cannot be totally avoided, the impact of the generated traffic on the task executions has to be analyzed and upper-

[1] *[1] page 14: "To achieve independence and to avoid propagation of failure, the system design can implement the partitioning of functions and components". [3] page 12: "Partitioning can be used for fault containment to avoid cascading failures.*

bounded so that its influence on the temporal isolation can be controlled and taken into account in the certification process[2]. Key properties like the WCET and the response-time of the tasks are dependent on the traffic generated by the co-scheduled tasks on the other cores since they share the same FSB. It is therefore challenging to ensure at design time that tasks meet their timing requirements. Although there has been considerable research in the uniprocessor domain towards analyzing the WCET of tasks [4], these methods cannot be applied as is and need to be augmented by further analyses to factor-in the extra contention delay due to the shared low-level hardware resources.

Contribution of this paper. The contribution of this paper is twofold. First, we propose a method to compute an upper-bound $\mathrm{PCRE}_p(t)$ on the number of requests that can be generated on a given core p in any time window of length t. Then, for a given task τ_i which executes for at most C_i time units on a given core p', we use the computed functions $\mathrm{PCRE}_p(C_i)$ of the other cores p to upper-bound the increased execution time of task τ_i due to the contention between the cores for the shared FSB. In our method we assume that the access to the FSB is granted using a Round-Robin protocol (as in Intel processors [5]).

Organization of the paper. Section II describes the computational model; Section III describes our method to obtain an upper-bound $\mathrm{PCRE}_p(t)$ on the number of bus requests generated by a given core p in a given interval of time t; Based on this technique, Section IV proposes an improved method to compute a tighter WCET considering RR arbitration for a given task; This is extended to a system-wide analysis in Section V; We present the related work in Section VI and the paper concludes in Section VII.

II. System model

Platform model. The system is composed of a set of m processor cores denoted by $\pi_1, \pi_2, \ldots, \pi_m$ and we assume that no cache memory is shared between them (as in the MPC8641D processor from Freescale) or all levels of shared cache, if present, are disabled or partitioned. All the cores share the FSB in order to access the shared main memory. To focus on the requests which are generated by cache misses, we consider that the hardware prefetching mechanism is disabled in the processor: By doing so, we reduce the non-determinism caused by arbitrary speculative prefetches which consume bus bandwidth and can block important requests from being served. We consider that there is no pipelining of requests in the memory controller or any support for split transactions. To re-iterate, having temporal and spatial isolation between subsystems is a key requirement for embedded systems to ensure composability and timing predictability and hence the above assumptions are motivated by the problem domain itself and thereby not restrictive.

Task model. We consider a sporadic and constrained-deadline task model in which a task τ_i is characterized by four parameters: $\mathrm{C}_\mathrm{i}^{\mathrm{iso}}$, T_i, $\mathrm{BR}_\mathrm{i}^{\mathrm{iso}}$ and $D_i \leq T_i$. The parameter $\mathrm{C}_\mathrm{i}^{\mathrm{iso}}$ denotes an upper-bound on the execution time of task τ_i when it executes in *isolation*, i.e., with no contention on the FSB. T_i denotes the minimum inter-arrival time between two consecutive jobs from τ_i. D_i denotes the deadline of the task: every job released by the task τ_i has to execute for at most $\mathrm{C}_\mathrm{i}^{\mathrm{iso}}$ time units within D_i time units from its release in order to meet its deadline. The parameter $\mathrm{C}_\mathrm{i}^{\mathrm{iso}}$ can be computed by well-known techniques in WCET analysis [4]. The current study focuses on determining $\mathrm{C}_\mathrm{i}^{\mathrm{mix}}$, which denotes an upper-bound on the execution time when τ_i executes *with* contention on the FSB, i.e., when the co-scheduled tasks are running on the other cores. Clearly, the value of $\mathrm{C}_\mathrm{i}^{\mathrm{mix}}$ is not an inherent property of τ_i and depends on the traffic on the FSB and hence on the memory request pattern of the co-scheduled tasks on the other cores during its execution. $\mathrm{BR}_\mathrm{i}^{\mathrm{iso}}$ (short for Bus Requests) denotes an upper-bound on the number of bus requests that a job of τ_i can generate when it executes non-preemptively in isolation[3]. We assume that this parameter can be derived by static analysis or measurements [6] or hybrid techniques.

Scheduler specification. We consider a partitioned scheme of task assignment in which tasks are assigned to cores at design time and are not allowed to migrate from one core to another at run-time (non-migrative). We denote by $\bar{\pi}(i)$ the set of $m-1$ cores to which task τ_i is *not* assigned (called the "interfering cores" of task τ_i). We consider a non-preemptive scheduler and hence do not deal with cache related and task switching overheads. We assume that the number of tasks in the system is known at system design-time and make the non work-conserving assumption as follows: whenever a task completes earlier than its WCET (say on CPU π_p), the scheduler idles the core π_p up to the theoretical WCET of the task. This assumption is made to ensure that the number of bus-requests within a time window computed at design time, is not higher at run-time due to early completion of a task and the subsequent early execution of the next tasks. The effect of jitter which is inherent to any timing based design is not the focus of this paper and thus will not be handled explicitly in the theory that follows.

III. Determination and Analysis of $\mathrm{PCRE}_p(t)$

This section describes a method to calculate the Per-Core-Request-Evaluator function $\mathrm{PCRE}_p(t)$. Recall that this function gives an *upper-bound* on the number of requests generated by the set of tasks assigned to core π_p in any time interval of length t.

[2] *[2] (Annex F): "Where a resource (such as a peripheral device) is shared between elements, the design shall ensure that the elements will not function incorrectly because the shared resource is locked by another element. The time required to access a shared resource shall be taken into account in determining temporal non-interference."*

[3] Note that $\mathrm{BR}_\mathrm{i}^{\mathrm{iso}}$ may not correspond to execution path that results in $\mathrm{C}_\mathrm{i}^{\mathrm{iso}}$, but assuming both estimates together ensures that the worst-case behavior of the task τ_i is captured.

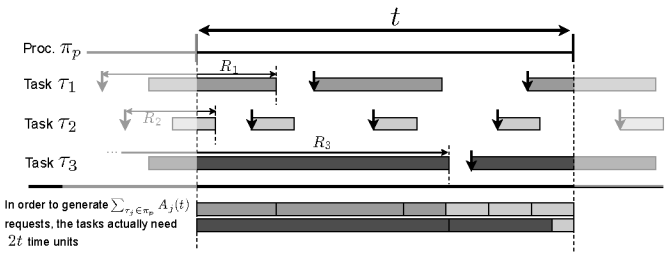

Figure 1. Illustration of the pessimism of the approaches in [6], [7]. For each of the three tasks τ_i, the job-release pattern drawn is the one leading to the maximum number of requests within t time units, i.e., it is the pattern considered in the computation of $A_i(t)$. In these computations, the response time R_i of each task τ_i is taken into account.

State-of-the-art methods: A method to compute the cumulative interference from a given core is also proposed in [6] and [7] (Equation 6, Corollary 1). The method is basically divided into two steps. First, it computes an upper-bound $A_j(t)$ on the number of requests generated by multiple instances of a given task τ_j in a time interval of length t. Second, the total interference $\text{PCRE}_p(t)$ from all the tasks running on a given core π_p is obtained simply by adding up the interference $A_j(t)$ of all the tasks running on that core π_p. That is, for core π_p, the interference generated is roughly given by $\text{PCRE}_p(t) = \sum_{\forall \tau_j \in \pi_p} A_j(t)$. The method clearly leads to an over-estimation[4], since all tasks executing on π_p may not co-execute when the analyzed task executes. Figure 1 illustrates the over-estimation.

A. Description of the method

To circumvent the overestimation made by the state-of-the-art methods, we propose a technique that considers all the possible combinations of tasks, to ensure that the maximum number of requests are generated by the tasks. Instead of the (additive) two-step procedure as stated above, we carry out the analysis considering all the tasks in the task-set in one phase. Within any given time window of length t, the schedule that leads to the maximum number of requests can be split into 3 portions: (i) the "carry_in", which contains the job (if any) that starts its execution outside the window and completes within the window (i.e., this job partially executes at the beginning of the window) (ii) the "body", which contains all the jobs of different tasks that execute entirely within the window and (iii) the "carry_out", which contains the job (if any) that starts its execution within the window but completes outside the window. The length of the carry_in and the carry_out portions of task τ_i can range from 0 to $\text{C}_\text{i}^{\text{iso}}$. The three portions are depicted in Figure 2(a). All the blocks of the same color represent different jobs of the same tasks.

The description of the pseudo-code of our method given by Algo. 1 follows. For a given core π_p and duration t, it

[4]In [7], the authors do mention that the approach is conservative and that the problem is an instance of a bounded non-linear knapsack problem which is NP-hard, but do not provide a solution.

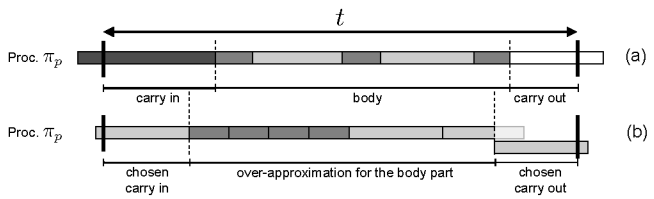

Figure 2. (a) The worst-case scheduling scenario (in terms of number of requests) that can occur at run-time in a time frame of length t. (b) Illustration of how Algo. 1 packs the tasks within the body portion, and the final schedule it assumes.

computes the function $\text{PCRE}_p(t)$ in 3 main steps explained below.

Step 1 (lines 1–5). Since a job of any task $\tau_i \in \tau^p$, can occupy the carry_in or the carry_out portion, the maximum maxcarrylen is first computed in line 2. Then, for every possible length k in 0 to maxcarrylen, the algorithm computes the maximum number of requests that can be generated within these portions and record the maximum values in the inreq and outreq arrays. The methods $\text{head}_i(t)$ and $\text{tail}_i(t)$

Algorithm 1: $\text{PCRE}_p(t)$

```
   input : t, p is the core index
   output: maxrequests
   // τ^p is the task set assigned to core π_p
1  C_max = max_{τ_i ∈ τ^p} {C_i^iso} ;
2  maxcarrylen = min(t, C_max) ;
   // STEP 1
3  foreach k ∈ [0, maxcarrylen] do
4      inreq[k] = max_{τ_i ∈ τ^p} {head_i(k)} ;
5      outreq[k] = max_{τ_i ∈ τ^p} {tail_i(k)} ;
   // STEP 2
6  foreach b_len ∈ [0, t] do
7      foreach τ_i ∈ τ^p do njobs_i = ⌈b_len/T_i⌉;
8      capacity = b_len ; filled_cap = 0;
9      list = list containing njobs_i instances of each task τ_i ∈ τ ;
10     Sort list by descending order of BR_i^iso/C_i^iso;
11     while list ≠ φ and filled_cap + list.first().C_i^iso ≤ capacity
       do
12         filled_cap += list.first().C_i^iso ;
13         breq[b_len] += list.first().BR_i^iso ;
14         list.deleteFirst() ;
15     rem_cap = capacity − filled_cap ;
16     if list ≠ φ and rem_cap > 0 then
17         (filled_cap += rem_cap);
18         breq[b_len] += (rem_cap / list.first().C_i^iso) × list.first().BR_i^iso;
19         list.delete first();
   // STEP 3
20 maxrequests = 0 ;
21 foreach in_len ∈ [0, maxcarrylen] do
22     foreach out_len ∈ [0, maxcarrylen − in_len] do
           // Assign the rest to the body portion
23         b_len = t − in_len − out_len ;
24         total = inreq[in_len] + breq[b_len] + outreq[out_len] ;
25         if (maxrequests < total) then maxrequests = total ;
   // Case :t < Execution time of the task
26 maxrq1 = 0 ;
27 foreach (τ_i ∈ τ^p) do
28     if (t < C_i^iso) then maxrq1 = max(maxrq1, inscan(i, t)) ;
29 return max(maxrq1, maxrequests) ;
```

compute an upper bound on number of requests generated by the task τ_i in the time window $[\mathrm{C}_\mathrm{i}^\mathrm{iso} - t, \mathrm{C}_\mathrm{i}^\mathrm{iso}]$ (the end part of its execution) and $[0, t]$ (the beginning part of its execution), respectively. The detailed computation of these upper-bounds is presented in [6].

Step 2 (lines 6–20). Depending on the scheduling algorithm employed, the body portion of the task contains different combinations of jobs. However, irrespective of the scheduling algorithm, we know that each task τ_i can admit at most $\mathrm{njobs}_i = \lceil \mathrm{b\_len}/T_i \rceil$ complete executions of jobs within a body portion of length $\mathrm{b\_len}$. The aim of this step is to select the tasks that will generate the maximum number of requests within the body portion. The problem is therefore analogous to the bounded knapsack problem in which the capacity of the knapsack can be equated to the size of the body portion (i.e., $\mathrm{b\_len}$); Each object to be packed is a job (of task τ_i) for which the weight is given by its WCET $\mathrm{C}_\mathrm{i}^\mathrm{iso}$ and the value is given by the maximum number of requests, $\mathrm{BR}_\mathrm{i}^\mathrm{iso}$, that it can generate. We first transform the bounded knapsack problem to an instance of a 0/1 knapsack problem by creating njobs_i jobs for each task τ_i and store them in a list (lines 9–10). The algorithm then sorts the job list in the descending order of the $\mathrm{BR}_\mathrm{i}^\mathrm{iso}$ / $\mathrm{C}_\mathrm{i}^\mathrm{iso}$ values (line 11). Then, it employs a greedy approach: it successively pops the job at the front of the list and packs it within the body window until the capacity of the knapsack (i.e., the size of the body window) is not exceeded (lines 12-14)[5]. As the jobs are packed, the algorithm accordingly updates the $\mathrm{breq}[\mathrm{b\_len}]$ array which holds the maximum number of requests and $\mathrm{filled\_cap}$, the currently occupied capacity. If the last-popped job cannot fit entirely in the remaining capacity, then the algorithm allows a fraction of this job to fit in the remaining capacity and adds up to $\mathrm{breq}[\mathrm{b\_len}]$ the corresponding number of requests, assuming that the requests are uniformly generated over the job's execution (reflected in line 18). This fractional assignment transforms the problem to a fractional knapsack problem and is a source of pessimism of our approach but ensures safe upper bounds on the number of requests. It also violates our description of the body portion in which only complete executions are permitted and can lead to a final schedule containing two partial executions of tasks (one at the end of the body portion and one in the carry_out), but given that the objective is not to draw up the optimal schedule (which requires the knowledge of the scheduling algorithm), we believe that this is an acceptable solution.

Step 3 (lines 21–25). The algorithm considers all possible combinations of length for the carry_in and carry_out portions ($\mathrm{in\_len}$ and $\mathrm{out\_len}$) and assigns the remaining length to $\mathrm{b\_len}$ (line 23), such that $\mathrm{in\_len} + \mathrm{out\_len} + \mathrm{b\_len} = t$. We use the pre-computed number of requests stored in arrays inreq, outreq and breq to compute the final upper bound on the maximum number of requests generated in the time window of length t.

Special case (lines 27–28). If the duration t of the time window is less than the execution time of a task, then this task could have started its execution before the beginning of the window and finish its execution outside the window. In such a case, the maximum number of requests may be found within the task or across 2 successive partial executions of the task. The method $\mathrm{inscan}(i, t)$ internally scans the task for the maximum number of requests and records the maximum in the variable $\mathrm{maxrq1}$.

B. Sources of over-estimation in Algorithm $\mathrm{PCRE}_p(t)$

As seen in Figure 2(b), Algorithm 1 may schedule different tasks in the carry_in and carry_out portions compared to the actual "worst-case schedule" (WCS) (Figure 2(a)) and the generated schedule may allow for a longer body portion. A task which is part of the carry in portion in the WCS may be disallowed (by scheduling constraints) in the body portion. But since Algo. 1 does not record which tasks have been scheduled in the carry in and out portions, the jobs of these tasks can be re-considered while analyzing the body portion to maximize the number of generated requests, hence leading to a first over-approximation. The second over-approximation is attributed to the number of requests computed for the fractional assignment of the last packed task in the body portion of the window and the assumption that its requests are assumed to be uniformly distributed over its execution. In short, the pessimism arises from a) the over-approximation made while computing the number of requests in the body and b) the fact that the maximum number of requests generated in each of the three portions is computed regardless of which tasks are scheduled in the other portions, hence allowing potentially more jobs of the same task to execute within the t time units.

C. Complexity of Algorithm $\mathrm{PCRE}_p(t)$

As explained in Section III, the size t of the window is the execution time of the analyzed task; That is, the maximum value of t is given by $C_\mathrm{max} = \max_{\tau_i \in \tau} \mathrm{C}_\mathrm{i}^\mathrm{iso}$. Let n be the number of tasks deployed on processor π_p. Therefore, assuming that the value of $\mathrm{head}_i(k)$ and $\mathrm{tail}_i(k)$ is given for all tasks τ_i and durations k, it can be seen that the time-complexity of Step 1, 2 and 3 are $n \times C_\mathrm{max}$, C_max^2 and $C_\mathrm{max} \times n \times \log n$, respectively, leading to a total complexity of $\mathrm{O}(\max(n \times C_\mathrm{max}, C_\mathrm{max}^2, C_\mathrm{max} \times n \times \log n))$ for Algo. 1. Since C_max is typically greater than the number n of tasks in practice, the algorithm runs with a pseudo-polynomial time-complexity of $\mathrm{O}(C_\mathrm{max}^2)$.

D. Algorithm $\mathrm{PCRE}_p(t)$ provides a safe upper-bound

Theorem 1: Algorithm 1 computes a safe upper-bound on the number of requests.

Proof: Let $S^\mathrm{M}(t)$ denote the schedule of the tasks leading to the maximum number of requests in a time window of length t. For example, suppose that $S^\mathrm{M}(t)$ is the

[5]The method primarily aims to compute the maximum requests, and not to generate the exact schedule leading to it. Hence placing tasks in order τ_1,τ_2,τ_1 will lead to the same number of requests as the schedule τ_1,τ_1,τ_2

schedule depicted in Figure 2(a). Since Algo. 1 considers all combinations of feasible carry_in and carry_out portions, it eventually covers the case leading to $S^{\mathrm{M}}(t)$. Therefore, Algo. 1 finds the same number of requests generated in these portions as in $S^{\mathrm{M}}(t)$. In the body portion, we know that $S^{\mathrm{M}}(t)$ contains only complete executions. We relax this requirement to include a fractional execution at the end and this enables the problem to be expressed as an instance of a fractional bounded knapsack problem for which the greedy approach is known to be optimal [8] and is guaranteed to provide an upper-bound to the corresponding integer knapsack problem. The algorithm finds the maximum over all feasible carry_in and carry_out and body portions and thereby is guaranteed to find an upper-bound which is safe and is greater or equal to the solution resulting from the schedule $S^{\mathrm{M}}(t)$. ■

E. Monotonically Increasing Property

We now introduce a basic property of the function $\mathrm{PCRE}_p(t)$ which can be formally expressed as:

Property 1 (Monotonically Increasing Property): Given core π_p and durations t_1 and t_2, we have $t_1 \leq t_2 \Leftrightarrow \mathrm{PCRE}_p(t_1) \leq \mathrm{PCRE}_p(t_2)$.

Proof: Let $\mathrm{maxrequests}_1$ and $\mathrm{maxrequests}_2$ be the maximum number of requests returned by $\mathrm{PCRE}_p(t_1)$ and $\mathrm{PCRE}_p(t_2)$ (computed as per Algo. 1), respectively. Let t_1^{in} and t_1^{out} be the lengths of the carry_in and carry_out portions in the solution returned by $\mathrm{PCRE}_p(t_1)$. Since all possible lengths for the carry_in and carry_out portions are considered by Algo. 1, the computation of $\mathrm{PCRE}_p(t_2)$ considers these lengths t_1^{in} and t_1^{out} as well. Therefore, two cases may arise: (i) $\mathrm{maxrequests}_2$ is obtained for these lengths of carry_in and carry_out, which results in a larger body portion (i.e., $t_2 - t_1^{\mathrm{in}} - t_1^{\mathrm{out}} \geq t_1 - t_1^{\mathrm{in}} - t_1^{\mathrm{out}}$) and thus a greater number of requests within the body since the set of jobs that can execute within $t_2 - t_1^{\mathrm{in}} - t_1^{\mathrm{out}} \supseteq$ the set of jobs that can execute within $t_1 - t_1^{\mathrm{in}} - t_1^{\mathrm{out}}$, hence $\mathrm{maxrequests}_2 \geq \mathrm{maxrequests}_1$, or (ii) $\mathrm{maxrequests}_2$ is obtained for different lengths of the carry_in and carry_out, which means that Algo. 1 found other values of carry_in and carry_out for which $\mathrm{maxrequests}_2$ is even greater (hence $\mathrm{maxrequests}_2 \geq \mathrm{maxrequests}_1$ holds as well). ■

F. Comparison against the approach in [6]

In order to compare the $\mathrm{PCRE}(\cdot)$ function, with the approach in [6] we generated a set of 25 tasks varying the WCET (from 10 to 500 ms), periods (from 4 to 10 times the WCET) and randomly generated memory traces and assigned the tasks randomly to cores. To have a fair comparison with [6], we ensured that the cumulative utilizations on tasks assigned to the core is less than one. Figure 3 shows the comparison of the upper-bound on the number of requests generated over time for core 1 to which 7 tasks were assigned. As seen in Figure 3, our method (legend: "New Approach") provides tighter upper-bounds on the number of bus requests in comparison with the approach in [6]

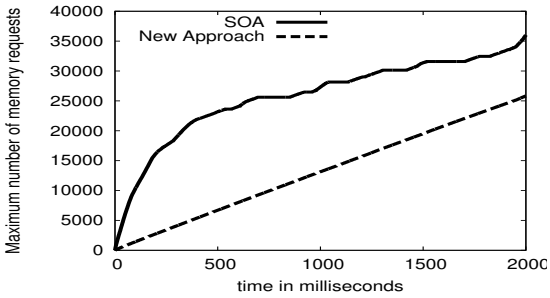

Figure 3. Comparison with the approach in [6]

(legend "SOA"). The graph also illustrates the monotonically increasing property of the PCRE() function.

G. PCRE() for periodic tasks with static schedules.

The problem of computing the function $\mathrm{PCRE}_p(t)$ is inherently simpler for a system in which tasks are periodic and the scheduler enforces a fixed ordering of task execution (static scheduling) that repeats itself (cyclic scheduling or CSA) every H time units, where H is the period of the schedule. In this approach also, we make the non work-conserving assumption, i.e., if tasks do not execute up to their WCET, the core must be idled up to the WCET. In contrast to the previously proposed approach of finding the combination of tasks that leads to the maximum number of bus requests within a time window of length t, we first draw up the entire schedule of the tasks from time 0 to time $H + t$. Since the task set and the order of task arrivals is known, this schedule can be constructed at design time. Then, this schedule is scanned by sliding a time window of length t from time 0 to time $H + t$. The extra t time units beyond the period of the schedule must be considered, as the maximum number of bus requests may be generated across two schedule periods. As the interval is scanned, the maximum number of requests observed so far within t time units is recorded and is finally returned by the $\mathrm{PCRE}_p(t)$ function.

IV. WCET Analysis: Basic RR and Improved RR

As mentioned in Sec. II, we assume that the contention over the FSB is resolved based on a RR arbitration mechanism, in which all the cores are treated equally. The order in which the cores acquire the ownership of the bus is fixed apriori. When more than one core tries to access the bus, ties are resolved based on the fixed-ownership ordering. Given the WCET $\mathrm{C_i^{iso}}$ of a task τ_i in isolation, here we compute the extra execution time that should be added to $\mathrm{C_i^{iso}}$ in order to take into account the delay due to contention on the FSB. We introduce the notation $\mathrm{TR^{iso}}$ which denotes an upper-bound on the time to serve a memory request when a task runs in isolation. $\mathrm{TR^{iso}}$ includes the time for arbitration over the FSB, the time spent in the memory controller and finally the time for the memory subsystem to serve the request and the bi-directional communication delay (for the request and response).

A. Basic Round Robin equation

Given a RR bus arbitration mechanism, an upper-bound $\mathrm{C_i^{mix}}$ on the WCET of each task τ_i considering bus contention is given by

$$\mathrm{C_i^{mix}} \stackrel{\text{def}}{=} \mathrm{C_i^{iso}} + \mathrm{BR_i^{iso}} \times (m-1) \times \mathrm{TR^{iso}} \tag{1}$$

Since the access to the shared FSB is granted using the RR protocol, each request generated by τ_i can be blocked by *at most* 1 request issued by the tasks running on each of the other $(m-1)$ cores, hence creating an extra delay of *at most* $\mathrm{BR_i^{iso}} \times (m-1) \times \mathrm{TR^{iso}}$ time units. Equation (1) implicitly assumes that the tasks running on each of the $(m-1)$ interfering cores **will** generate $\mathrm{BR_i^{iso}}$ requests during the execution of task τ_i, which might not be true as tasks running on these cores may generate lesser number of requests. Hence the above equation may lead to pessimistic bounds. We tackle this over-pessimism by providing a tighter upper-bound in the following subsection.

B. Improved Round Robin equation

Lemma 1: Considering that a task τ_i is executing with contention on the FSB, an upper-bound $\mathrm{C_i^{mix}}$ on its execution time is given by the first solution (i.e., $C_i^k = C_i^{k-1}$) at which the following fixed-point iteration converges:

$$C_i^k \stackrel{\text{def}}{=} \mathrm{C_i^{iso}} + \sum_{\pi_p \in \bar{\pi}(i)} \min(\mathrm{BR_i^{iso}}, \mathrm{PCRE}_p(C_i^{k-1})) \times \mathrm{TR^{iso}} \tag{2}$$

with $C_i^0 \stackrel{\text{def}}{=} \mathrm{C_i^{iso}}$. The fixed point iteration terminates when $C_i^k = C_i^{k-1}$, at which, the value of $\mathrm{C_i^{mix}}$ is given by the corresponding C_i^k.

Proof: Equation (2) recursively computes a new value of C_i at each iteration $k \geq 1$, and incorporates the extra delay incurred by task τ_i due to the requests generated by the tasks running on the interfering cores, i.e., the cores $\pi_p \in \bar{\pi}(i)$. By definition, we know that τ_i generates at most $\mathrm{BR_i^{iso}}$ requests during its execution. For a RR bus arbitration algorithm, each of the interfering cores can delay every request issued by τ_i by at most 1 request and hence can delay the execution of τ_i by at most $\mathrm{BR_i^{iso}} \times \mathrm{TR^{iso}}$ time units. Hence, at each iteration k, Equation (2) considers for each core $\pi_p \in \bar{\pi}(i)$ the minimum between (i) the number of requests that π_p might actually generate in the (currently computed) execution time of τ_i (i.e. $\mathrm{PCRE}_p(C_i^{k-1})$) and (ii) the maximum number of requests that can be used by π_p to block τ_i's execution (i.e., $\mathrm{BR_i^{iso}}$). ■

In the next subsection, we discuss some properties of this improved RR equation.

C. Properties of the Improved RR equation

Lemma 2: In Equation (2), for any iteration $k \geq 1$, the value of C_i^k monotonically increases, i.e $C_i^k \geq C_i^{k-1}$

Proof: The proof is by induction. Initially, $C_i^0 = \mathrm{C_i^{iso}}$ and it can be inferred from Equation (2) that $C_i^1 \geq C_i^0$. The induction step consists in showing that, if $C_i^k \geq C_i^{k-1}$ then $C_i^{k+1} \geq C_i^k$. According to Equation (2), the expression $C_i^{k+1} \geq C_i^k$ can be rewritten as

$$\begin{aligned} &\mathrm{C_i^{iso}} + \textstyle\sum_{\pi_p \in \bar{\pi}(i)} \min(\mathrm{BR_i^{iso}}, \mathrm{PCRE}_p(C_i^k)) \times \mathrm{TR^{iso}} \\ \geq\ &\mathrm{C_i^{iso}} + \textstyle\sum_{\pi_p \in \bar{\pi}(i)} \min(\mathrm{BR_i^{iso}}, \mathrm{PCRE}_p(C_i^{k-1})) \times \mathrm{TR^{iso}} \end{aligned}$$

By subtracting/dividing the common terms from both sides we have:

$$\begin{aligned} &\textstyle\sum_{\pi_p \in \bar{\pi}(i)} \min(\mathrm{BR_i^{iso}}, \mathrm{PCRE}_p(C_i^k)) \geq \\ &\textstyle\sum_{\pi_p \in \bar{\pi}(i)} \min(\mathrm{BR_i^{iso}}, \mathrm{PCRE}_p(C_i^{k-1})) \end{aligned} \tag{3}$$

From Property 1, for any π_p we have $\mathrm{PCRE}_p(C_i^k) \geq \mathrm{PCRE}_p(C_i^{k-1})$. Therefore, only three cases may arise:

1) $\mathrm{BR_i^{iso}} \geq \mathrm{PCRE}_p(C_i^k) \geq \mathrm{PCRE}_p(C_i^{k-1})$
2) $\mathrm{PCRE}_p(C_i^k) \geq \mathrm{BR_i^{iso}} \geq \mathrm{PCRE}_p(C_i^{k-1})$
3) $\mathrm{PCRE}_p(C_i^k) \geq \mathrm{PCRE}_p(C_i^{k-1}) \geq \mathrm{BR_i^{iso}}$

and it can be easily shown for all of them that

$$\min(\mathrm{BR_i^{iso}}, \mathrm{PCRE}_p(C_i^k)) \geq \min(\mathrm{BR_i^{iso}}, \mathrm{PCRE}_p(C_i^{k-1}))$$

which provides Inequality (3) and thereby, establishes the proof. ■

Lemma 3: Equation (2) always terminates in at most $(m-1) \times \mathrm{BR_i^{iso}}$ iterations and may provide a tighter upper-bound than Equation (1).

Proof: The proof is a direct consequence of Lemma 2. The highest value of C_i^k is reached when $\forall \pi_p \in \bar{\pi}(i)$ it holds that $\mathrm{PCRE}_p(C_i^k) \geq \mathrm{BR_i^{iso}}$. In this case, Equation (2) becomes $C_i^k \stackrel{\text{def}}{=} \mathrm{C_i^{iso}} + \sum_{\pi_p \in \bar{\pi}(i)} \mathrm{BR_i^{iso}} \times \mathrm{TR^{iso}}$ which corresponds to Equation (1). In order to maximize the number of iterations to reach this highest value of C_i^k, we have to consider that at each iteration k, there exists only one core $\pi_\ell \in \bar{\pi}(i)$ such that $\min(\mathrm{BR_i^{iso}}, \mathrm{PCRE}_\ell(C_i^{k-1})) = \min(\mathrm{BR_i^{iso}}, \mathrm{PCRE}_\ell(C_i^{k-2})) + 1$ and for all cores $\pi_p \in \bar{\pi}(i)$ with $p \neq \ell$, it is the case that $\min(\mathrm{BR_i^{iso}}, \mathrm{PCRE}_p(C_i^{k-1})) = \min(\mathrm{BR_i^{iso}}, \mathrm{PCRE}_p(C_i^{k-2}))$. In this scenario, we get $C_i^k = C_i^{k-1} + 1$ at each iteration k and it takes $(m-1) \times \mathrm{BR_i^{iso}}$ iterations to reach the highest value of C_i^k given above. Finally, if Equation (2) converges to a solution $C_i^k = C_i^{k-1}$ before this extreme value, then the resulting $\mathrm{C_i^{mix}}$ (where $\mathrm{C_i^{mix}} = C_i^k$) provides a tighter upper-bound than the $\mathrm{C_i^{mix}}$ computed by Equation (1). ■

D. Comparison: Basic RR vs Improved RR

The resulting WCET of a task (with contention) is sensitive to co-scheduled task's $\mathrm{C_i^{iso}}$, T_i), memory profile besides core assignments and the scheduling algorithm and cannot be summarized without exhaustive tests. However, the aim here is to validate and compare our approach and hence we provide a "proof of concept" with a small set of tasks. We generated 16 random tasks with different memory profiles (generated randomly) and assigned them to 4 different cores randomly. We used Algo. 1 to compute the PCRE() function in Eq. (2) to compute the WCET. As seen in Fig. 4, for all the tasks, our method performs equally or better (in most

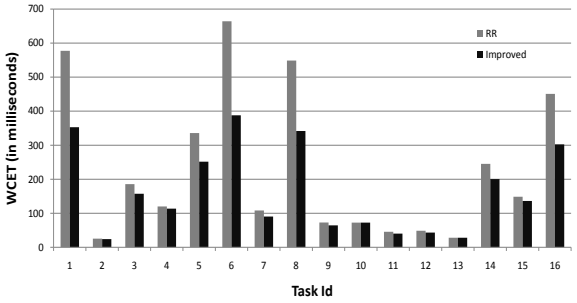

Figure 4. WCET comparison: improved approach vs. basic RR

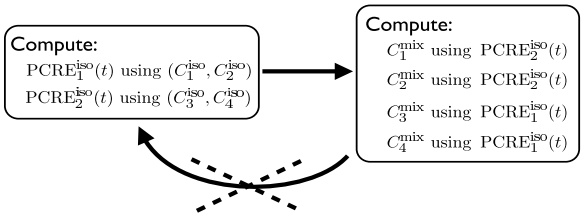

Figure 5. Our system-wide analysis is a non-cyclic process.

cases) against the basic RR arbitration. The tightness of the improved WCET varies from (0 to 41%). There is no improvement using our approach for task 2 and 13, which can be explained as follows: These tasks have a very low request density ($\mathrm{BR}_{\mathrm{i}}^{\mathrm{iso}}$ / $\mathrm{C}_{\mathrm{x}}^{\mathrm{iso}}$) ratio) of around 2 and a low execution time. Hence their WCET increases marginally due to the contention on the bus and there is little scope for improvement. For tasks 1, 5, 6, 8, 16 the request densities varied from 25 to 30 (high for this example). For such tasks, the impact of contention is high and the tightness varied from (25-41%). The rest of the tasks showed moderate improvements (5 to 18%) with the new approach.

V. System wide Analysis

In this section, we describe the process of applying the method described earlier to *all* the tasks in the system. Consider an example system with 2 cores $\{\pi_1,\ \pi_2\}$ and 4 tasks $\{\tau_1,\ \tau_2,\ \tau_3,\ \tau_4\}$. Let tasks τ_1 and τ_2 be assigned to π_1 and tasks τ_3 and τ_4 be assigned to π_2. We calculate the functions $\mathrm{PCRE}_1(t)$ and $\mathrm{PCRE}_2(t)$ using the method described in Section III (left box in Fig. 5). Tasks assigned to core π_1 are subject to interferences from tasks co-scheduled on π_2 and vice versa. Therefore, we compute $\mathrm{C}_1^{\mathrm{mix}}$ and $\mathrm{C}_2^{\mathrm{mix}}$ against $\mathrm{PCRE}_2(t)$ and $\mathrm{C}_3^{\mathrm{mix}}$ and $\mathrm{C}_4^{\mathrm{mix}}$ against $\mathrm{PCRE}_1(t)$, using the method introduced in Section IV (right box in Fig. 5). At this point, it may seem that the functions $\mathrm{PCRE}_1(t)$ and $\mathrm{PCRE}_2(t)$ can be refined using as input, the newly computed $\mathrm{C}_1^{\mathrm{mix}}, \mathrm{C}_2^{\mathrm{mix}}, \mathrm{C}_3^{\mathrm{mix}}$ and $\mathrm{C}_4^{\mathrm{mix}}$. If so, the process may re-iterate and a natural question is "when should the process stop iterating?". We answer this question based on the following Lemma 4.

Lemma 4: For every core p, the function $\mathrm{PCRE}_p(t)$ monotonically decreases as the WCET of the tasks running on core π_p is inflated due to the contention for the FSB.

Proof: The proof is a consequence of the fact that incorporating the interference from other tasks into the WCET estimate of a given task τ_i does not increase the number of requests that τ_i can potentially generate in any time window of any length t. That is, the number of requests that are inherently generated by each task τ_j assigned to π_p does not increase as its WCET inflates. In the computation of $\mathrm{PCRE}_p(t)$, the only side-effect of inflating the WCET of each task τ_i (without modifying its maximum number of request $\mathrm{BR}_{\mathrm{i}}^{\mathrm{iso}}$) is that potentially lesser jobs of τ_i can execute within the body portion and lesser requests can potentially be generated within a given length of carry_in and carry_out portions. Hence the lemma. ■

As proven in Lemma 4, re-iterating the process of computing $\mathrm{PCRE}_p(t)$ (for all cores π_p) and $\mathrm{C}_{\mathrm{i}}^{\mathrm{mix}}$ (for all tasks τ_i) alternately will decrease the value returned by the function $\mathrm{PCRE}_p(t)$ at each iteration, hence ultimately providing an unsafe upper-bound for the WCETs. As a result, our analysis is a one-step process, i.e., it is not cyclic. This also means that the given model facilitates the determination of the value of $\mathrm{C}_{\mathrm{i}}^{\mathrm{mix}}$ for every task τ_i at design-time itself.

VI. Related work

The problem of analyzing the impact of shared hardware resource contention on the WCET of the tasks is of significant importance and the research community has taken initial steps to address this problem. Time Division Multiple Access (TDMA) based schemes have been proposed in [9], [10], [11] and [12] and [13]. The methods approach the problem in different ways: precomputing application specific bus schedules, or analyzing buses with the assumption of separate buses for memories and data, restricting accesses to the bus in specific phases of task execution, division of the tasks into superblocks which execute in specific slots and using FlexRay like approaches to have fixed and reserved slots. These methods therefore require a change in either the application behavior or modification in the hardware. An analysis of a work conserving bus is presented in [6] and [7] and we have shown that they can result in overestimated WCETs. Also of interest are the works of [14] based on timed automata which is restricted to instruction accesses and the work of [15] which again assumes division of tasks into superblocks which run in pre-assigned time slots.

It is important to cite the works related to the contention due to shared caches which is another major problem and has been addressed in [16],[17],[18],[19]. Alternative ideas have been proposed that circumvent the shared cache contention by spatial isolation. Cache Partitioning and cache locking techniques have been proposed [20],[21] and [22] to ensure that dedicated cache space is available to critical real-time tasks. As seen above, many of the approaches favor a non work-conserving TDMA bus. We present some views regarding this in the following subsection.

A. TDMA and the Non work-conserving assumption

The TDMA bus arbitration is predictable and composable, allowing tasks to be analyzed in isolation, making it a real-time friendly protocol. But it is non work-conserving and hence the bus is idle when the core owning a time-slot does not have any requests to be served. Although it is favored in the research community, existing COTS-based systems (which are designed for high performance) do not employ it. Therefore in this paper, we assume that the FSB uses a work conserving and predictable (known upper bounds) RR bus arbitration algorithm.

The approach in our paper shifts the TDMA-like approach from the bus-level to the task-level through the non work-conserving assumption. With the TDMA approach at the bus-level, reclaiming the unused time-slots is difficult as most of the COTS-based systems do not allow for reprogramming the bus arbitration rule. On the other hand, a non work-conserving scheduler enables an easier reclamation of the slack – scheduling rules can be modified to achieve this: for example, if a task τ_i completes δ units before its WCET then the scheduler could start executing any pending task τ_j such that $C_j \leq \delta$ and $\mathrm{BR}_{\mathrm{j}}^{\mathrm{iso}} \leq \mathrm{tail}_i(\delta)$. In this manner, the maximum number of requests generated at run-time in any time window of length t is guaranteed not to exceed the one computed at design time using $\mathrm{PCRE}_p(t)$ (for all cores p). A significant amount of research work on slack time reclamation techniques exists and we will further investigate how to relax our non work-conserving assumption as part of the future work.

VII. Conclusions

In this paper, we highlighted the need for isolation among components and the need to determine an upper-bound on the interference. An analysis was presented to compute an upper-bound on the number of bus requests generated by a set of tasks running on a given cores considering (i) a static schedule (ii) no assumptions on the scheduling algorithm. We showed that the proposed method provides tighter bounds than the existing methods in literature. This method was used to further compute a tighter bound on the WCET against the WCET computed by the basic RR mechanism.

Acknowledgements

This work was supported by the REPOMUC project, ref. FCOMP-01-0124-FEDER-015050, co-funded by FCT-MCTES (Portuguese Foundation for Science and Technology) and ERDF (European Regional Development Fund) through COMPETE (Operational Programme Thematic Factors of Competitiveness); the RECOMP project, ref. ARTEMIS/0202/2009, co-funded by FCT-MCTES, and by EU funds through the ARTEMIS Joint Undertaking, under grant nr. 100202.

References

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

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

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

[4] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem – overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, pp. 36:1–36:53, 2008.

[5] "Dual-Core Intel Itanium 2 Processor 9000 Series," Intel Corporation,Santa Clara (US), pp. 95–96, 2006. [Online]. Available: ftp://download.intel.com/design/Itanium2/datashts/31405401.pdf

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

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

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 2nd Ed.* McGraw-Hill, 2001.

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

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

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

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

[13] A. Schranzhofer, R. Pellizzoni, J.-J. Chen, L. Thiele, and M. Caccamo, "Timing analysis for resource access interference on adaptive resource arbiters," in *Real-Time and Embedded Technology and Applications Symposium*, 2011.

[14] W. Yi, "Multicore embedded systems: the timing problem and possible solutions," in *Proceedings of the 12th international conference on Formal engineering methods and software engineering*, 2010, pp. 22–23.

[15] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele, "Worst case delay analysis for memory interference in multicore systems," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2010, pp. 741–746.

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

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

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

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

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

[21] N. Guan, M. Stigge, W. Yi, and G. Yu, "Cache-aware scheduling and analysis for multicores," in *EMSOFT '09: Proceedings of the 7th ACM international conference on Embedded software*, 2009, pp. 245–254.

[22] B. Lesage, I. Puaut, and A. Seznec, "PRETI: Partitioned REal-TIme shared cache for mixed-criticality real-time systems." [Online]. Available: http://hal.inria.fr/hal-00661687