Technical Report

Response Time Analysis of COTS-Based Multicores Considering The Contention On The Shared Memory Bus

Dakshina Dasari

Björn Andersson

Vincent Nelis

Stefan M. Petters

Arvind Easwaran

Jinkyu Lee

HURRAY-TR-110705

Version:

Date: 09-27-2011

Response Time Analysis of COTS-Based Multicores Considering The Contention On The Shared Memory Bus

Dakshina Dasari, Björn Andersson, Vincent Nelis, Stefan M. Petters, Arvind Easwaran, Jinkyu Lee

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, baa@isep.ipp.pt, nelis@isep.ipp.pt, smp@isep.ipp.pt, aen@isep.ipp.pt, jule@isep.ipp.pt

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

Abstract

Abstract—The current industry trend is towards using Commerciallyavailable Off-The-Shelf (COTS) based multicores for developing realtimeembedded systems, as opposed to the usage of custom-madehardware. In typical implementation of such COTS-based multicores,multiple cores access the main memory via a shared bus. This oftenleads to contention on this shared channel, which results in an increaseof the response time of the tasks. Analyzing this increased responsetime, considering the contention on the shared bus, is challengingon COTS-based systems mainly because bus arbitration protocolsare often undocumented and the exact instants at which the sharedbus is accessed by tasks is not explicitly controlled by the operatingsystem scheduler; they are instead a result of cache misses. This paperproposes three contributions towards analyzing tasks scheduled onCOTS-based multicores. Firstly, we describe a method to model thememory access patterns of a task. Secondly, we apply this model toanalyze the worst-case response time for a set of tasks. Finally, thispaper describes a method to experimentally obtain the parametersrequired for such an analysis, by using performance monitoringcounters. We compare our work against an existing approach andshow that our approach outperforms it by providing tighter upperboundson the number of bus requests generated by the tasks.

Response Time Analysis of COTS-Based Multicores Considering the Contention on the Shared Memory Bus

Dakshina Dasari*, Björn Andersson†*, Vincent Nelis*, Stefan M. Petters*, Arvind Easwaran* and Jinkyu Lee‡
*CISTER-ISEP Research Centre, Polytechnic Institute of Porto, Portugal
†Software Engineering Institute, Carnegie Mellon University, USA
‡Dept. of Computer Science, KAIST, South Korea
{dndi, baa, nelis, smp, aen}@isep.ipp.pt; jinkyu@cps.kaist.ac.kr; baandersson@sei.cmu.edu

***Abstract*—The current industry trend is towards using Commercially available Off-The-Shelf (COTS) based multicores for developing real-time embedded systems, as opposed to the usage of custom-made hardware. In typical implementation of such COTS-based multicores, multiple cores access the main memory via a shared bus. This often leads to contention on this shared channel, which results in an increase of the response time of the tasks. Analyzing this increased response time, considering the contention on the shared bus, is challenging on COTS-based systems mainly because bus arbitration protocols are often undocumented and the exact instants at which the shared bus is accessed by tasks are not explicitly controlled by the operating system scheduler; they are instead a result of cache misses. This paper makes three contributions towards analyzing tasks scheduled on COTS-based multicores. Firstly, we describe a method to model the memory access patterns of a task. Secondly, we apply this model to analyze the worst-case response time for a set of tasks. Although the required parameters to obtain the request profile can be obtained by static analysis, we provide an alternative method to experimentally obtain them by using performance monitoring counters (PMCs). We also compare our work against an existing approach and show that our approach outperforms it by providing tighter upper-bound on the number of bus requests generated by a task.**

I. Introduction

Currently, multicore processors are generic building blocks in the design of embedded real-time computing systems. In a typical multicore system, each core has its own resources (architectural state, registers, execution units, some or all levels of caches). Data is transferred from the memory to the processor over an interconnection network and the focus of this paper is on multicore implementations that use the Front Side Bus (FSB) as the interconnection network. Since all the cores access memory via the FSB, cores may stall while waiting for requests to be served, thereby increasing the execution time of the tasks running on those cores. For the deployment of real-time applications on multicores it is often of utmost importance to determine whether tasks meet their deadlines by analyzing their worst-case response time (WCRT) at design time, considering the contention on shared low-level hardware resources. Unfortunately, established methods to compute WCRT of tasks for a uniprocessor cannot be used unmodified for multicores, since they do not take into account the additional impact of shared low-level hardware resources. For the multicore domain, methods to profile the bus request patterns and to compute the increased execution time due to contention on the shared low-level hardware resources are still in the initial stages of research. Hence there is a need to develop such a method which will help in leveraging the computing power of multicores for real-time applications. To do so, we first discuss the solutions which exist in the state of the art and then proceed towards defining our objectives and proposing our method for analyzing tasks deployed on multicores, considering the contention on the memory bus.

This work was supported by the CISTER Research Unit (608FCT), the REPOMUC project (ref. FCOMP-01-0124-FEDER-015050), funded by FEDER funds through COMPETE (POFC - Operational Programme 'Thematic Factors of Competitiveness), and by National Funds (PT) through FCT - Portuguese Foundation for Science and Technology; by the RECOMP project, funded by National Funds through FCT under grant ref. ARTEMIS/0202/2009, and by the ARTEMIS JU under grant agreement no 100202.

A. Related Work

The research community has made important initial contributions to advance the state of the art. Some TDMA-based schemes have been analyzed since they are predictable and hence real-time friendly. However, it is difficult to point out currently existing architectures that actually support TDMA, for lack of documentation from the vendors. A TDMA-based arbitration algorithm has been proposed by Rosén et al. [1], in which different time slots to access the bus are allocated to different processors by static scheduling, stored in a memory directly connected to the bus arbiter (not available in COTS systems). Schranzhofer et al. [2] developed a framework for analyzing the worst-case response time of real-time tasks when TDMA is applied to arbitrate access to a shared resource. This was followed by their work on resource adaptive arbiters [3]. Unfortunately, their rigid assumption of tasks being split into superblocks which execute in some statically pre-defined order and divided into mutually exclusive acquisition, execution and replication phases limits the applicability of their solution. In the TDMA-based schemes proposed in [4] and [5], the authors have considered the effect of shared instruction caches with a shared bus as well. Since they do not model data accesses and assume separate buses and memories for code and data (uncommon in commodity hardware), their solution though interesting, has limited applicability. Another method to model request patterns and the memory bus using timed automata has been proposed in [6], but they model only instruction accesses. Pellizoni et al. [7] have developed a method to compute an upper bound to the contention delay incurred by a task, for systems comprising any number of cores and any number of peripheral buses sharing a single main memory. They assume time triggered (periodic) tasks and a restrictive preemption model and hence their solution does not cater to event-triggered tasks. Schliecker et al. [8] have proposed a method to address the issue of bounding the shared resource load for multiprocessor systems using a general event

based model, but their method is based on the computation of the minimum time between two consecutive accesses to a shared resource, leading to an over-estimation of the number of requests since it inherently implies a uniform distribution of requests. We will show in Section V that the approach proposed in this paper provides a tighter upper-bound on the number of requests when compared to the approach by Schliecker et al.

B. Objectives and Assumptions

Given the WCET of a task in isolation, we aim to develop a method to compute the WCRT, considering the increased execution time due to contention on the bus for COTS-based systems. As chip makers often do not publish the underlying bus arbitration protocols, the method must not be tightly bound to a particular bus arbitration mechanism and therefore should be generic. In addition, the method should also deal efficiently with event-triggered tasks to reflect real world applications. In order to realize such a method, we believe that it is warranted to make the following assumptions:
A1. The interconnection network to memory is a bus: The rationale for this assumption is that although the general trend among chip makers is towards switched interconnection networks, the shared front side bus is still the dominant technology in multicore processors and is expected to be used for some considerable time.
A2. Non preemptive tasks: This assumption is made as a first step to avoid dealing with cache-related preemption delays and the effect of context switching overhead associated with preemptive scheduling.
A3. A constrained deadline sporadic task model: sporadic tasks have proven remarkably useful for the modeling of event-triggered real-time systems.
A4. Partitioned scheduling (tasks have been assigned to processors before run-time and they do not migrate at run-time): Again, this is to focus on the problem of bus contention.
A5. Arbitration for the memory bus is work conserving: This is the current standard of COTS-based multicore implementations.
A6. Only one memory request can be handled at a time. Today, most of the commercial memory controllers implement complex and optimized features to improve the memory performance, such as multiple data rates or multiple channels. In such memory controllers, memory requests can be overlapped and multiple requests can then be served simultaneously. However, this assumption is made to simplify the analysis while still providing safe results.

We present a method which attempts to fulfill the objectives stated above, based on assumptions A1–A6. Our main contributions are towards developing a method to (i) Characterize the bus request pattern of the task (ii) Find an upper bound on the number of bus requests generated in a time window of length t (iii) Compute the WCRT of tasks for multicores under a partitioned scheme, considering the contention on the bus (iv) Experimentally obtain the requisite parameters on a COTS-based multicore. These are presented in Sections III, IV and VI in the paper. We also compare our method with the method proposed in [8] in Section V.

II. System and Task Model

A. Hardware Model

The hardware is composed of a set of m processor cores denoted by $\pi_1, \pi_2, \ldots, \pi_m$, and as stated, the cores do not share caches. This model applies to systems in which each core has a private cache, or the shared cache if present, is disabled. All the cores communicate over a shared bus (the Front-Side-Bus) in order to access the shared main memory.

B. Task Model

The application is composed of a set of tasks $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$. We also assume a constrained-deadline sporadic task model in which each task τ_i is characterized by $\langle C_i, D_i, T_i \rangle$; a worst-case execution time, C_i, a minimum inter-arrival time T_i and a deadline $D_i \leq T_i$, with the interpretation that, during the execution of the system, task τ_i releases a sequence of jobs such that two subsequent jobs from τ_i are released at least T_i time units apart and the exact times of the releases of these jobs cannot be controlled by the scheduling algorithm. In order to meet its deadline, each job released by τ_i needs to be executed for C_i time units within D_i time units from its release. We denote by R_i an upper-bound on the worst-case response time (WCRT) of task τ_i. The response time of a job denotes the time between its arrival and its completion and the WCRT of a task is the maximum amongst the response time of all the jobs released by the task.

In this paper, we are interested in finding the WCRT when τ_i executes *with* contention on the memory bus, i.e., assuming that other tasks are running on other cores. Clearly, this value is not an inherent property of τ_i, but is co-runner dependent and it depends on the memory request pattern of the other tasks scheduled to run during its execution. To compute this contention-aware WCRT, we extend the general WCRT time equation for non-preemptive tasks and incorporate the extra delay introduced due to cores competing for the same shared FSB to access the shared main memory. To compute the contention delay, we next introduce the notations $\mathrm{BR}_i(t)$ and TR.

Given a task τ_i, we define a function $\mathrm{BR}_i(t)$, that returns an upper bound on the number of bus requests that task τ_i can generate in a time interval of length t. Since the tasks do not share any cache in our hardware model, the initial value of $\mathrm{BR}_i(t)$, is clearly dependent on task τ_i only, and independent of the behavior of tasks running on the other processors/cores. But, as will be seen, since the computation of $\mathrm{BR}_i(t)$, takes as a parameter the response time of the task, which is in turn affected by the other tasks run in the system, the value of $\mathrm{BR}_i(t)$ will change accordingly.

We denote by TR an upper bound on the time needed to perform a *bus transaction*. In general, a bus transaction is a complete sequence of bus actions required to perform a read (or write) operation.

C. Scheduler Specification

As noted, tasks are assigned to processors before run-time; i.e., we consider a partitioned scheme of task assignment in which tasks are not allowed to migrate from one core to another. Also, remember that tasks run to completion and are not preempted. For analysis, we will assume that each task assigned to a core is assigned a unique priority at design time. It has to be noted that the assumption of fixed priority scheduling has only been made for clarity of representation, but in principle our approach can be used with any fixed job priority algorithm which allows the computation of the WCRT R_i. To summarize, our current approach assumes a non-preemptive, fixed priority, partitioned model for the task set under analysis.

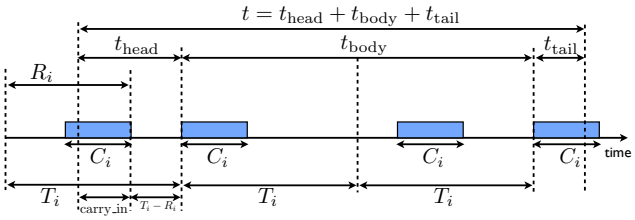


Figure 1. Calculation of $\text{BR}_i(t)$ for $t \geq C_i$

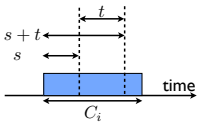


Figure 2. Calculation of $\text{BR}_i(t)$ for $t < C_i$

We denote by $\pi(i)$, the set of tasks, excluding τ_i, that are assigned to the same core as τ_i. The notation $\bar{\pi}(i)$ will be used to denote the set of tasks not assigned to the same core as τ_i. Also, we denote by $\text{lp}(i)$ and $\text{hp}(i)$ the subset of tasks executed *on the same core* as τ_i and which have a lower and higher priority than τ_i, respectively.

III. A METHOD TO COMPUTE $\text{BR}_i(t)$

As stated previously, $\text{BR}_i(t)$ denotes an upper bound on the number of bus requests that task τ_i can generate during any time interval of duration t. The following notations are used in context of the computation of $\text{BR}_i(t)$:

1) A lower and upper bound, $\text{ARL}_i^j(t)$ and $\text{ARH}_i^j(t)$ (respectively) on the number of bus requests in an interval $[0, t]$, where 0 denotes the beginning of execution of the j^{th} execution path of task τ_i up to time t.
2) The execution time C_i^j of the j^{th} execution path of task τ_i.

We note that different executions of the same path may result in different number of bus requests as a result of the underlying cache replacement policy; this is the reason why we distinguish between $\text{ARH}_i^j(t)$ and $\text{ARL}_i^j(t)$. We let $\text{paths}(\tau_i)$ denote the set of all the execution paths of task τ_i. By definition, $\text{ARH}_i^j(t)$ and $\text{ARL}_i^j(t)$ are non-decreasing functions for all i, j.

Consider a time window of a given length t, for which we need to compute an upper bound $\text{BR}_i(t)$ on the number of bus requests generated by a task τ_i. We can have three types of jobs in this time interval (i) A job that is released before the start of the time window but with its deadline in the time window, which means it executes partially or completely within the window(ii) Jobs that are released within this time window and complete their entire execution within it, and (iii) A job released within the given time window but having its deadline outside the window and hence executes partially or completely within the time window. To calculate $\text{BR}_i(t)$, we divide the time window t into three subintervals correspondingly: the *head* portion of length t_{head}, the *body* portion of length t_{body}, and the *tail* portion of length t_L, such that $t_{\text{head}} + t_{\text{body}} + t_{\text{tail}} = t$ and $t_{\text{head}}, t_{\text{tail}} < T_i$.

As shown in Figure 1, the head has a length of less than T_i, implying either one partial or one complete execution. The head is in turn divided into two parts, namely, the *carry_in* and the *arrival gap* (a_gap). The carry-in portion represents the execution segment of the task which lies within the time window (under consideration) and it ranges from 0 to C_i. Since the task executes in the carry-in portion, we can also view the carry-in part as the request generating portion of the head. On the other hand, the a_gap part specifies the time between the termination of τ_i in the head and its next release time-instant, i.e., it is the time between the end of the carry-in and the next release of τ_i. In this portion, no bus requests can be generated since the CPU is waiting for the next release of τ_i. In order to maximize the number of requests in the whole time window under consideration, it can be easily shown that the a_gap interval should be as short as possible, and its shortest length is given by $(T_i - R_i)$, which assumes the scenario in which the job of τ_i that executes in the head has completed execution exactly R_i time units after its release. We represent this by

$$t_{\text{head}} \stackrel{\text{def}}{=} \begin{cases} \text{carry\_in} + \text{a\_gap} & \text{if carry\_in} > 0 \\ 0 & \text{otherwise} \end{cases}$$
$$\text{with } 0 \leq \text{carry\_in} \leq C_i \text{and a\_gap} = T_i - R_i$$

Since the head part starts from any arbitrary point of an execution of τ_i, but includes the end point of that execution, an upper bound on the number of bus requests generated in the head portion is given by:

$$f_i^{\text{head}}(\text{carry\_in}) \stackrel{\text{def}}{=} \max_{j \in \text{paths}(\tau_i)} \left\{ \text{ARH}_i^j(C_i^j) - \text{ARL}_i^j([C_i^j - \text{carry\_in}]) \right\} \quad (1)$$

In the body portion, there are *exactly* t_{body}/T_i complete executions of τ_i and the maximum number of request generated in the body portion is given by:

$$f_i^{\text{body}}(t_{\text{body}}) \stackrel{\text{def}}{=} \frac{t_{\text{body}}}{T_i} \times \max_{j \in \text{paths}(\tau_i)} \left\{ \text{ARH}_i^j(C_i^j) \right\} \quad (2)$$

Finally, the length of the tail part is less than T_i, implying either one partial or one complete execution. The number of bus requests generated in the tail part can be bounded from above by:

$$f_i^{\text{tail}}(t_{\text{tail}}) \stackrel{\text{def}}{=} \max_{j \in \text{paths}(\tau_i)} \left\{ \text{ARH}_i^j(\min\{t_{\text{tail}}, C_i^j\}) \right\} \quad (3)$$

Algorithm 1 describes a method to compute the function $\text{BR}_i(t)$. The input to the algorithm is t, the duration for which the number of requests needs to be upper bounded, R_i, the response time of the task, C_i and T_i. When the task is run in isolation, we assign $R_i = C_i$. The algorithm computes the maximum number of requests by considering every feasible combination of t_{head}, t_{body} and t_{tail}. To do so, it initially fixes the carry_in which ranges from 0 to C_i, computes the arrival gap given by $T_i - R_i$ and then correspondingly calculates t_{body} and t_{tail} in lines 3 to 14 of the algorithm. Next the body portion is computed and the rest of the time interval is assigned to the tail portion. This is represented as:

$$t_{\text{body}} \stackrel{\text{def}}{=} \max\left\{0, \left\lfloor \frac{(t - t_{\text{head}})}{T_i} \right\rfloor \right\} \times T_i \quad (4)$$

$$t_{\text{tail}} \stackrel{\text{def}}{=} \max\left\{0, t - t_{\text{head}} - t_{\text{body}}\right\} \quad (5)$$

For every combination of t_{head}, t_{body} and t_{tail}, the algorithm computes the number of requests in line 15. The maximum

Algorithm 1: ComputeBR()

input : R_i, C_i, T_i and time interval t
output: $\mathrm{BR}_i(t)$

```
begin
    total ← maxreq ← 0 ;
    for carry_in ← 0 to min(C_i, t) do
        if (carry_in == 0) then
            t_head ← 0 ;
            t_body ← ⌊t / T_i⌋ × T_i;
            t_tail ← t − t_body;
        else
            a_gap ← T_i − R_i;
            t_head ← carry_in + a_gap;
            if t_head > t then t_body ← t_tail ← 0 ;
            else
                t_body ← ⌊(t − t_head) / T_i⌋ × T_i;
                t_tail ← t − t_head − t_body;
        total ← f_i^H(carry_in) + f_i^M(t_body) + f_i^T(t_tail);
        if total > maxreq then maxreq ← total
    if t < C_i then
        Compute maxreq1 as per Equation (6) ;
        if maxreq1 > maxreq then maxreq ← maxreq1
    return maxreq ;
```

recorded value of the number of requests generated is updated as the algorithm proceeds and the final value is returned as $\mathrm{BR}_i(t)$.

For the special case in which $t < C_i$, the maximum number of requests may be generated across two jobs (with only a carry_in and tail portion, and no body portion), or *in any arbitrary segment of the task*. In the latter case, we compute $\mathrm{BR}_i(t)$ as follows (see Figure 2):

$$\mathrm{BR}_i(t) = \max_{\substack{j \in \mathrm{paths}(\tau_i) \\ 0 \le s < (C_i - t)}} \left\{ \mathrm{ARH}_i^j(\min\{s+t, C_i^j\}) - \mathrm{ARL}_i^j(s) \right\} \quad (6)$$

For this scenario $t < C_i$, $\mathrm{BR}_i(t)$ is thus computed by taking the maximum between the value returned by the algorithm described above and the value returned by Equation (6), which handles the case in which the maximum number of requests is generated within a task segment.

Although it appears that the algorithm loops over all the values from 0 to C_i, in practice it is not feasible to compute the value of $\mathrm{ARH}_i(t)$ or $\mathrm{ARL}_i(t)$ for all t from 0 to C_i as it is computationally expensive and hence the values must be computed at a coarser granularity. In reality and as described in the experiment section, a limited number (say k) of sampling points are chosen from 0 to C_i and readings are recorded only at these k points. In such a method, whenever t is not equal to one of these k sampling point while reading $\mathrm{ARH}_i(t)$ or $\mathrm{ARL}_i(t)$, it is always important for these two functions to round the returned value to the next higher sampling point. This may result in a over-approximated number of request for a given t, but the returned value will be safe. The algorithm is presented as such, to separate the theoretical method which is generic, from the implementation which may depend on the hardware (e.g. the resolution of timers, which will decide the frequency of sampling).

The current method of exploring all paths is inevitable in static analysis, measurement-based or hybrid methods to ensure safe upper bounds. It can be optimized on an application-to-application basis, considering the input sets and eliminating paths which will not contribute to the maximum number of requests (for e.g. simple error reporting/recovery paths which return immediately or paths with certain conditional clauses). The proposed method can thus be applied after a path truncation phase and application of other optimization techniques which is not in the focus of the paper. The proposed solution as such, is meant to serve as a generic method, irrespective of the application or the input set.

IV. Response Time Analysis

In this section, we describe the function to compute the general response time and then propose an extension to handle interference from the other cores.

A. General Response-Time Analysis

The research literature provides methods for computing the exact response-time of tasks scheduled by non-preemptive fixed-priority scheduling on uniprocessor system [9], [10]. For the task model considered in this paper, a simplified version which does not explore the entire busy period and is hence faster can be derived from [11]. Instead of computing the response time R_i of task τ_i *exactly*, it computes an *upper-bound* $\widehat{R}_i$ on it by using the following recursive equation:

$$\widehat{R}_i^{(k+1)} = C_i + B_i + \sum_{j \in \mathrm{hp}(i)} \left\lceil \frac{\widehat{R}_i^{(k)}}{T_j} \right\rceil \times C_j \quad (7)$$

where B_i is the maximum blocking time imposed on task τ_i due to lower-priority tasks, i.e., $B_i \stackrel{\text{def}}{=} \max_{j \in \mathrm{lp}(i)}\{C_j\}$. The WCRT R_i of the task τ_i is computed in an iterative manner, starting from $\widehat{R}_i^{(0)} = C_i + B_i$, and is given by the smallest value of $\widehat{R}_i^{(k)}$ that satisfies Equation (7). The process terminates when either it reaches the first fixed-point value of the equation at which $\widehat{R}_i^{(k+1)} = \widehat{R}_i^{(k)}$, in which case the WCRT $R_i = \widehat{R}_i^{(k+1)}$is obtained, or it reaches $\widehat{R}_i^{(k)} > D_i$ which implies that the deadline of task τ_i is missed.

B. Extended Response-Time analysis

The computation of the WCRT in the *multicore* scenario must consider the increased delay due to tasks executing on the same core and the additional delays due to contention on the FSB from tasks running on the other cores. We now introduce the extended response-time equation which, in addition to the original WCRT equation, also factors-in the contention delay due to requests generated by the co-scheduled tasks on the other cores competing for the shared FSB.

$$\widehat{R}_i^{(k+1)} = C_i + B_i + \sum_{j \in \mathrm{hp}(i)} \left\lceil \frac{\widehat{R}_i^{(k)}}{T_j} \right\rceil \times C_j + \sum_{j \in \bar{\pi}(i)} \mathrm{BR}_j(\widehat{R}_i^{(k)}) \times \mathrm{TR} \quad (8)$$

Equation (8) encapsulates the effects of the delay due to interference by higher priority tasks on the same core, the blocking by the lower priority tasks on the same core and the delay caused by interference from tasks running on the other cores. If τ_i is the task under analysis, all the tasks with a higher priority than

Algorithm 2: ComputeRespTime()

input : τ_i
output: R_i^{new} or failure code

1 $R_i^{\text{new}} \leftarrow C_i$;
2 **repeat**
3 $\quad R_i^{\text{old}} \leftarrow R_i^{\text{new}}$;
4 $\quad R_i^{\text{new}} \leftarrow C_i + B_i + \sum_{j \in \text{hp}(i)} \left\lceil \frac{R_i^{\text{old}}}{T_j} \right\rceil \times C_j + \sum_{j \in \bar{\pi}(i)} \text{BR}_j(R_i^{\text{old}}) \times \text{TR}$;
5 $\quad$ **if** ($R_i^{\text{new}} > D_i$) **then return** failure ;
6 **until** ($R_i^{\text{new}} == R_i^{\text{old}}$) ***or*** ($R_i^{\text{new}} > D_i$);
7 **return** R_i^{new} ;

τ_i and assigned on the same core as τ_i, will also be impacted by the requests generated by tasks scheduled on the other cores, thereby increasing their execution time. This in turn will impact the WCRT of task τ_i. In the increased response time, more requests may be generated by the tasks running on the other cores. This will continue till the value of $\widehat{R}_i$ stabilizes (like the regular response time equation). Hence, to incorporate the extra delay due to the bus requests, which are generated during the increased response time, $\text{BR}_j()$ is parameterized with $\widehat{R}_i^{(k)}$.

Although the process is largely standardized, we present the method to compute the above equation as an algorithm (Algorithm: ComputeRespTime()) for easier readability. In order to prove that this algorithm terminates, we have to prove that the value of $R_i^{new} \geq R_i^{old}$ in each iteration of the algorithm. The following conditions will cause the algorithm to terminate: (i) $R_i^{new} > D_i$ implying that the task will miss its deadline, hence making the task set non schedulable (ii) $R_i^{new} = R_i^{old}$ implying that the fixed point value of the equation is reached and the recurrence relation has converged. The term representing the interference from the higher priority tasks is known to be a monotonically increasing function. To guarantee increasing monotonicity of the entire right-hand side (RHS) of the equation, we then need to prove the monotonic increasing property of the term, $\sum_{j \in \bar{\pi}(i)} \text{BR}_j(R_i^{old}) \times \text{TR}$, representing the interference from the other cores.

We know from Algorithm ComputeBR() for task τ_i, which is being delayed by requests from τ_j ($j \in \bar{\pi}(i)$), we have to compute $\text{BR}_j(t)$. In the case presented here, the value of t is R_i^{old}. Hence we can express BR as a function of two parameters and represent it here as $\text{BR}_j(R_j, R_i^{\text{old}})$. The value of R_j stays constant during the entire iterative process ComputeRespTime() but the value of the input R_i^{old} increases at each iteration (see line 3). By the very definition of the function BR(), we know that for all tasks τ_j having the property $C_j \leq R_j \leq D_j$ and for all $t, t' > t$: $\text{BR}_j(R_j, t') \geq \text{BR}_j(R_j, t)$. Hence we obtain the monotonic increasing property of the RHS of the equation.

C. System Analysis

To analyze the entire system, we need to find the response times of all the tasks in the system. To facilitate this, we have formulated the process as two algorithms: PerCoreAnalysis() and SystemAnalysis() (Algorithm 3 and 4). The value of any variable X in the iterative step k is denoted by X^k in both algorithms.

Algorithm PerCoreAnalysis(): This algorithm captures the computation of R_i^k for a set of tasks assigned to a core, during

Algorithm 3: PerCoreAnalysis()

/* π_j denotes the core index */
input : stepnum, π_j
output: statuscode (TRUE(1) or FALSE(0) or failure(-1))

1 **begin**
2 $\quad k \leftarrow$ stepnum, RiModified $\leftarrow$ FALSE, $R_i^0 \leftarrow C_i^0 \leftarrow C_i$;
$\quad$ /* Compute the WCRT for the set of n tasks */
3 $\quad$ **foreach** (τ_i *assigned to* π_j) **do**
4 $\quad\quad R_i^{k+1} \leftarrow ComputeRespTime(\tau_i)$;
5 $\quad\quad$ **if** ($R_i^{k+1} > D_i$) *OR* $R_i^{k+1} =$ failure **then** **return** failure;
6 $\quad\quad$ **if** ($R_i^{k+1} \neq R_i^k$) **then** RiModified $\leftarrow$ TRUE;
7 $\quad$ **return** RiModified ;

Algorithm 4: SystemAnalysis()

begin
$\quad$ stepnum $\leftarrow 0$;
$\quad$ **repeat**
$\quad\quad$ **foreach** $\pi_i \in \{\pi_1, \pi_2, \ldots, \pi_m\}$ **do**
$\quad\quad\quad$ status $\leftarrow$ PerCoreAnalysis(stepnum, π_i);
$\quad\quad\quad$ **if** (status == failure) **then**
$\quad\quad\quad\quad$ print "Task Set Not Schedulable", Exit
$\quad\quad\quad$ /* status returns TRUE if any R_i was modified. */
$\quad\quad$ stepnum ++ ;
$\quad$ **until** (status $\neq$ TRUE);

the iteration step k. For every iteration k, R_i^{k+1} is computed using Equation (8), considering the interference (modeled by $BR_j(R_i^k)$) from the tasks running on the other cores. If the response time of any task exceeds its deadline, the algorithm returns a failure status. If the response time of any task in the current iteration differs from the previous iteration, the algorithm, sets the value of the variable RiModified to TRUE, implying that another round of iteration is required.

Algorithm SystemAnalysis(): This algorithm applies the PerCoreAnalysis() algorithm to the set of tasks running on each core in a sequence of iterative steps, passing the step index as a parameter. The algorithm PerCoreAnalysis() for step k is first applied to the tasks set on core π_1, then on core π_2 and subsequently to core π_m. The algorithm terminates when the response time, R_i for every task τ_i in the entire system (all task sets on all cores) converges to a stable value across iterations ($R_i^{k+1} = R_i^k$), or the WCRT of any task of the entire system exceeds its deadline, implying that the task set is not schedulable, thereby terminating the entire offline analysis process. It is important to note that if the call to PerCoreAnalysis() results in a modified response time of any task, it affects the resulting value, generated by the ComputeBR() algorithm (which uses the value of R_i as an input). Hence, the next call to PerCoreAnalysis, must be made, with the modified R_i as input to the ComputeBR() algorithm, to reflect the modified external core interference represented by the term $\sum_{j \in \bar{\pi}(i)} \text{BR}_j(t) \times \text{TR}$ in Equation (8).

In order to prove that the algorithm terminates, we have to prove that the $(k+1)^{\text{th}}$ call to the function $\text{PerCoreAnalysis}()$ generates a value $R_i^{k+1} \geq R_i^k$. Note that the value of k now denotes

the iteration index in the context of Algorithm $\mathrm{SystemAnalysis}()$. On exiting the $\mathrm{PerCoreAnalysis}()$ function, the response time of every task has reached a fixed-point equation or a status indicating that the deadline of the task is exceeded is returned. In the former case, the attainment of the fixed point value implies that:

$$R_i = C_i + B_i + \sum_{j \in \mathrm{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \times C_j + \sum_{j \in \bar{\pi}(i)} \mathrm{BR}_j(R_i) \times \mathrm{TR} \quad (9)$$

The computeBR() algorithm for task τ_j needs R_j (amongst other parameters) as an input, to compute the maximum number of requests in time t. As mentioned earlier, $\mathrm{BR}()$ can be expressed as a function of two parameters $\mathrm{BR}_j(R_j, t)$. In the above equation t has the value R_i.

Lemma 1: For all tasks τ_j having the property $C_j \leq R_j \leq R'_j$, it holds for all $t > 0$ that $\mathrm{BR}_j(R_j, t) \leq \mathrm{BR}_j(R'_j, t)$

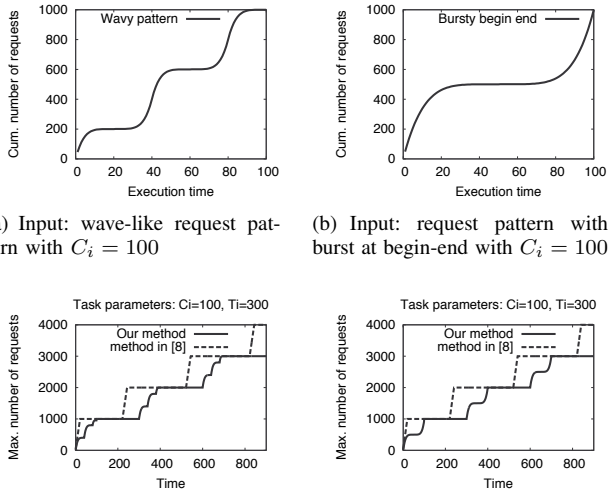
Proof: The proof follows from Algorithm computeBR(). Increasing the response time R_i implies that the maximum arrival gap $(T_i - R_i)$ decreases, thereby decreasing the idle segments of the time window in which no requests are generated between task releases: It potentially allows more requests to be generated in the carry-in portion, the body portion or the tail portion during the time window under consideration[1]. ■

V. Comparison with Related Work

An arbitration algorithm agnostic method is proposed by [8] (and an extended version in [12]) and hence warrants a comparison with our method, since the method may look similar in principle to the reader. The approach presented in [8] uses an event activation model to compute the upper bound to access shared resources in a given time t. To compute the maximum number of requests for a single task instance, they assume that there is a known minimum time dsr between two requests to a shared resource. They propose a simple lower bound to compute the minimum time that a task must execute, to generate n requests, given by $\delta^-(n) = (n-1) \times \mathrm{dsr}$. This is then extended, to compute the minimum time to make n requests by multiple instances of the task. An inverse function $\eta^+(t)$, is used to derive the maximum number of requests in time t. The assumption of a minimum request distance is request pattern agnostic and inherently implies a uniform distribution of requests and hence leads to an over-estimation of the maximum number of requests that a task can generate in a given time t.

It is important to note that the method proposed here, uses a different technique (compared to the method proposed in [8]) to compute the maximum number of requests for a task in a time interval t and hence the experiments here are used to highlight only that phase of the overall analysis. Since our approach to compute $\mathrm{BR}_i(t)$ takes into consideration, the request profile of a task and is request pattern sensitive, the bounds computed are tighter, as seen in Figure 3(c) and 3(d). These results are drawn from artificial request patterns depicted in Figure 3(a) and 3(b). These patterns are representative of applications having a (i) burst of requests at the beginning and end and (ii) wave like request distribution. In these graphs, C_i = 100, T_i = 300 and the maximum number of requests, $\mathrm{BR}(C_i)$ in one task instance (referred to $N_j^{\max}$ in their approach)

[1]Although intuitive, it is to be noted that when $R_i = T_i$, no more requests can be generated, as the entire carry_in portion is within the time window.



(a) Input: wave-like request pattern with $C_i = 100$

(b) Input: request pattern with burst at begin-end with $C_i = 100$

(c) Output: wave-like request pattern (wave-like) over $t = 900$ time units

(d) Output: request pattern (burst at begin-end) over $t = 900$ time units

Figure 3. Comparison of the approaches

is 1000. The experiments are run with $R_i = C_i$ as inputs to both algorithms. The maximum number of requests are computed for all values from 0 to 900 (i.e. 3*T_i time units). The curve denoted by "method in [8]" reflects the number of requests as per the method proposed in [8], while the curve denoted by "Our method" reflects the number of requests reported by our method. As seen in the graphs, our method for determining the maximum number of requests first characterizes the task behavior and then derives the bounds. In contrast, in the method proposed in [8], the authors do not consider the request distribution and base their analysis on the basis of request distances. As a result, their approach yields more pessimistic upper bounds on the number of requests that a task can generate. We compared the two approaches for other types of request patterns (like bursts at the beginning of the application, bursts at the end of the application, etc.) as well and found that our method outperforms their method. As expected, for tasks with uniform distribution of requests, both methods yield the same upper bounds. The graphs with other patterns are not presented here due to space limitations. Summarizing the discussion above, we believe that our approach dominates their approach in yielding tighter upper bounds on the number of requests in a given time interval.

VI. A method to obtain parameters experimentally

In *principle*, it is possible to use the substantial amount of work developed in the WCET analysis community [13] to provide suitable bounds on ARH and ARL. However, these approaches generally need an important amount of information about the hardware in order to provide accurate results. Since it is difficult to obtain suitably accurate documentation for COTS hardware, those techniques might provide highly pessimistic results and we focus on an alternative technique based on measurements, as this is still the de-facto standard in the analysis of safety critical systems. This alternative is also preferable when the underlying cache replacement policy is pseudo-LRU, because static/offline

analysis methods generally lead to highly pessimistic results for such policies (and pseudo-LRU is usually employed in COTS-based hardware).

The approach proposed in the paper requires as an input, the parameters TR, an upper bound on the time to complete one bus transaction. It also needs the cache profile of a task modeled by the ARH and ARL values. This section details how these values can be obtained by measurement on the actual hardware.

The experiments were carried out on an Intel^{TM}Core2 Quad Q8300 processor consisting of four cores placed on two dies on a single chip. Each die has two cores and each core has its own instruction and data cache (denoted as I$ and D$). However,the two cores on the same die share the L2 cache i.e., (i) Core-1 and Core-2 share a L2 cache on one die and (ii) Core-3 and Core-4 share a L2 cache on another die. All the 4 cores access the main memory via a single shared bus. On one die, tasks were run only on Core-1, keeping Core-2 idle, thereby giving Core-1 access to the entire L2 cache available on that die. Analogously, on the other die tasks were run only on Core-3, keeping Core-4 idle, thereby giving Core-3 access to the entire L2 cache available on that die. Experiments were performed on the VxWorks 6.8 [14] real-time operating system. Other relevant details of the experimental setup are presented in Table I.

System characteristics	
Processor model	Intel® Core2^{TM}Quad Processor
CPU	Q8300 @ 2.50GHz
L1 cache	32 KB D-cache, 32KB I-cache, 8-way associative
L2 cache	2048 KB, unified, 8-way associative
FSB Specs	333 MHz, 1333 MTps, 10656 MBps
OS kernel	VxWorks 6.8

Table I
TEST SYSTEM DESCRIPTION

A. Measurement Setup

Before each run of the experiments, the cache was invalidated, ensuring that the state of the cache was consistent across runs. The experiments were run with the same input, thereby forcing single execution paths. To reduce the non determinism, the hardware prefetching and adjacent cache line prefetching features were disabled in the processor. To avoid migration of the tasks across cores, tasks were pinned to the cores using the taskAffinity feature in VxWorks. Another feature namely, "CPU Reservation" (in VxWorks terminology) that dedicates a core to a task was also used to ensure that the task to which the core is dedicated runs non-preemptively. Events were monitored at the micro-architectural level by writing to model-specific registers and reading PMCs directly. PMCs are a set of special-purpose registers built into modern microprocessors to store the counts of hardware related activities, such as cache misses ([15], [16]). It is necessary to disable the prefetching feature (i) to isolate the bus contention problem and (ii) to have more determinism while taking the measurements, as prefetchers speculatively fetch data and add to the traffic on the bus and run in the background at arbitrary times, thus making the timing measurements inaccurate. Since the memory is shared between several peripherals, the interference from these must be kept to a minimum. Hence the experiments were run with a basic console device and a diskless system to avoid any DMA activity to influence measurement results.

B. Measurement of TR

TR is defined as an upper bound on the time to complete one bus transaction. To obtain this value experimentally, a task was generated that constantly accessed the memory and generated an L2 cache miss on each access. We programmed this task by declaring an array twice the size of the cache and accessing each line of the cache sequentially, thereby causing an L2 miss for every access. Since the array size is twice the cache size, the task scans the entire cache twice in each run, hence evicting all the cache lines that were already fetched, prior to the next run. The number of bus requests, denoted by NBR, is obtained by monitoring the Bus_Requests_Mem event, for each run and the time taken for each run, denoted by TBR is recorded. The number of bus requests generated was verified against the expected number of bus requests (which is twice the number of cache lines) to validate the approach and was found to be consistent. The value of TR is thus computed for thousands of runs and the maximum is recorded over all the runs. Then the final TR is given by Equation (10).

$$\mathrm{TR} = \max_{k=1..nr} (\mathrm{TBR}_k \,/\, \mathrm{NBR}_k) \tag{10}$$

where k denotes the run index, nr denotes the number of runs and TBR_k and NBR_k denote the corresponding values in that run. The value of TR from the experiments was 46.6 nano seconds.

C. Measurement of ARH and ARL

The ARH and the ARL values described in Section III, represent the upper and lower bound on the number of bus requests generated by a task from the beginning of its execution up to time t. To measure these values for a given task, we chose some sampling points by dividing the execution time of the tasks into subintervals. We obtained the cumulative number of bus requests upto that point by interrupting the task and reading the performance monitoring counters at the required sampled point. We then re-ran the task and interrupted the task at the next sampling point. At each sampling point, the highest measured value was recorded as ARH and the lowest value was recorded as ARL, over multiple iterations. It is to be noted that unlike simulations, where it is assumed that a task will have fixed number of memory accesses at a given time instance, this presents a more realistic approach, as it takes into account the variations in the number of requests issued due to the underlying cache replacement policy employed and makes this method very generic. For the given system, the Bus_Requests_Mem_This_Core_This_Agent event was monitored to precisely measure the number of requests issued by the task. An example of the AR curve, showing the ARH and ARL values at each sampling point for the Search Benchmark from the MiBench Suite [17] is presented in Figure 4. The AR curve for the search program shows a variability in the number of cache misses across runs during one complete execution. It can be seen that after a certain time, the number of bus requests remains almost constant and then increases. The constant number of requests seen in the graph corresponds to the time when the task is not issuing any

requests and this was achieved by the introduction of a task delay in the program (to demonstrate the variability in the request pattern which can be captured by PMCs).

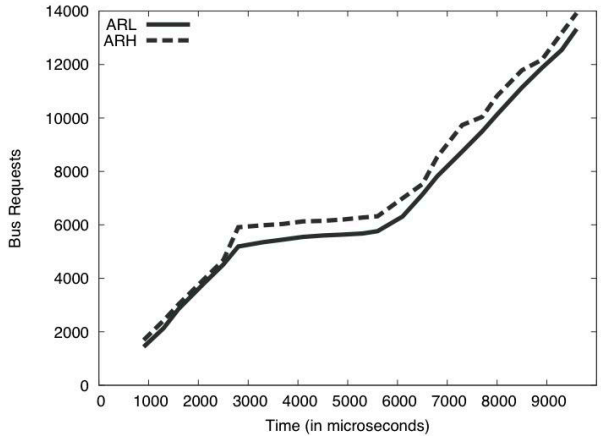


Figure 4. ARH, ARL Curve for the Search Benchmark

VII. Conclusions and Future work

In this paper, we have proposed a method to analyze the response times of tasks in a multicore system, considering the contention on the shared front-side-bus. We have presented a method to model the memory access patterns of a task, and used it to derive an upper bound on the number of requests it can generate within a given time window. By comparing our approach with an existing approach, we have shown that we can derive tighter bounds on the number of requests. We also outline the steps to obtain the required parameters on an actual hardware set-up. In the current work we considered a non-preemptive task model and multicores with non-shared caches. In the future, we plan to extend the current work to analyze shared caches.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury, "Bus-Aware Multicore WCET Analysis through TDMA Offset Bounds," in *ECRTS '11: Proceedings of the 2011 Euromicro Conference on Real-Time Systems*, 2011.

[6] M. Lv, W. Yi, N. Guan, and G. Yu, "Combining abstract interpretation with model checking for timing analysis of multicore software," in *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*, ser. RTSS '10, 2010, pp. 339–349.

[7] 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.

[8] 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.

[9] R. J. Bril, J. J. Lukkien, and W. F. Verhaegh, "Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption," *Real-Time Systems*, vol. 42, pp. 63–119, August 2009.

[10] K. Tindell, A. Burns, and A. Wellings, "Calculating controller area network (can) message response times," *Control Engineering Practice*, vol. 3, no. 8, pp. 1163–1169, 1995.

[11] K. W. Tindell, A. Burns, and A. J. Wellings, "An extendible approach for analyzing fixed priority hard real-time tasks," *Real-Time Systems*, vol. 6, pp. 133–151, March 1994.

[12] S. Schliecker and R. Ernst, "Real-time performance analysis of multiprocessor systems with shared memory," *ACM Transactions in Embedded Computing Systems*, vol. 10, pp. 22:1–22:27, 2011.

[13] 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.

[14] VxWorks, *Applications Programmers Guide, 6.8*.

[15] B. Sprunt, "The basics of performance-monitoring hardware," *IEEE Micro*, vol. 22, pp. 64–71, 2002.

[16] *Intel 64 and IA-32 Architecture Software Developers Manual Volume 3B: System Programming Guide, Part 2*, Intel Corp.

[17] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Workload Characterization*, 2001, pp. 3–14.