Worst Case Delay Analysis for Memory Interference in Multicore Systems

Rodolfo Pellizzoni*, Andreas Schranzhofer†, Jian-Jia Chen†, Marco Caccamo* and Lothar Thiele†
*University of Illinois at Urbana-Champaign, Urbana, IL, USA, {rpelliz2, mcaccamo}@illinois.edu
†Swiss Federal Institute of Technology (ETH), Zurich, Switzerland, {schranzhofer, jchen, thiele}@tik.ee.ethz.ch

***Abstract*—Employing COTS components in real-time embedded systems leads to timing challenges. When multiple CPU cores and DMA peripherals run simultaneously, contention for access to main memory can greatly increase a task's WCET. In this paper, we introduce an analysis methodology that computes upper bounds to task delay due to memory contention. First, an arrival curve is derived for each core representing the maximum memory traffic produced by all tasks executed on it. Arrival curves are then combined with a representation of the cache behavior for the task under analysis to generate a delay bound. Based on the computed delay, we show how tasks can be feasibly scheduled according to assigned time slots on each core.**

I. Introduction

Real-time embedded systems are increasingly using Commercial-Off-The-Shelf (COTS) components in an effort to raise performance and lower production costs. In particular, fast multicore CPUs and high-performance DMA peripherals are required to service demanding applications such as video processing that are becoming more and more popular in markets such as avionics. Unfortunately, COTS components are not designed with timing predictability in mind, which makes it challenging to integrate them in real-time systems. In particular, most COTS architectures feature a single-port main memory that is shared among all CPU cores and peripherals. When a task suffers a cache miss, contention for access to main memory can significantly delay cache line fetch and greatly increase the worst case execution time (WCET) of the task. We performed an experiment on a standard Intel dual core platform to understand the severity of this issue (details are provided in [8]). We engineered a task that continuously suffers cache misses and measured its WCET while running it in isolation. We then added to the experiment a second copy of the task running on the other core and a PCI-E [5] peripheral using DMA to saturate main memory with write requests, and measured a WCET increase of 2.96 times for the task.

Solutions to this problem have been presented in the literature. Several works (see [2], [9] for example) have proposed modifications to either memory arbitration or cache behavior to improve predictability, typically enforcing some TDMA scheme; however, such modifications are incompatible with COTS reuse. A different approach has been used in other works, such as [11], [6], [7], [10]: they focus on estimating the maximum delay that a task can suffer in a COTS system due to memory interference. In particular, in [6], [7] an analytical technique was developed to compute an upper delay bound given a representation of peripheral traffic. Unfortunately, such technique is only applicable to monoprocessor systems with a single peripheral bus. The analysis employed in [10] can compute bounds for multicore systems; however, it assumes that cache misses can occur anywhere in the task period. As we will show in Section IV, this can lead to overestimation.

This material is based upon work supported by Lockheed Martin, NSF under Awards No. CNS0720702, CNS0720512, the European Community's Seventh Framework Programme FP7/2007-2013 under Grant No. 216008 project Predator and the European Network of Excellence on Embedded System Design ARTISTDesign. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the sponsors.

In this paper, we introduce a novel WCET analysis framework that can compute memory delay bounds for systems comprising any number of cores and any number of peripheral buses sharing a single main memory. In particular, we provide two main contributions: **(1)** we introduce the key idea of computing a memory traffic arrival curve for each core, given a set of executed tasks. The arrival curve provides an upper bound to the amount of memory traffic generated by the core in any interval of time. **(2)** We describe an innovative algorithm that computes a delay bound for a task given traffic curves for all other cores and peripheral buses in the system. The algorithm is able to distinguish the behavior of DMA peripherals, whose traffic is buffered, from the behavior of CPU cores, which stall on cache misses.

The rest of the paper is organized as follows. In Section II we detail our system model. In Section III we describe how to derive arrival curves for cores, while in Section IV we introduce our delay algorithm. In Section V we show simulation results. Finally, in Section VI we provide concluding remarks.

II. System Model

We consider a COTS system comprising multiple *processing cores* implemented on CPU dies. Each processing core PE_i can employ one or more cache levels, but caches are private and not shared with other cores. Execution is stalled whenever a core suffers a cache miss, e.g. cores do not employ mechanisms such as HyperThreading. Processing cores execute periodic tasks. Tasks are not allowed to migrate and task schedules on different cores are asynchronous. Scheduling follows a *restrictive preemption model*: the control flow graph for each task τ_i is divided into a series of S_i sequential *superblocks* $\{s_{i,1}, \dots, s_{i,S_i}\}$. Each superblock can include branches and loops, but superblocks must be executed in sequence. Multiple tasks executed on the same processing core are scheduled according to fixed time slots, with a given set of superblocks assigned to each slot. For the sake of simplicity, we first assume that the set of superblocks assigned to each time slot is known and that cache is invalidated before the start of the time slot. In Section IV-A we show how to use our delay analysis to compute the static task schedule.

Each task τ_i is characterized by a *cache profile* $c_i^{prof} = \{\mu_{i,j}^{\min}, \mu_{i,j}^{\max}, exec_{i,j}^L, exec_{i,j}^U\}$. $\mu_{i,j}^{\min}, \mu_{i,j}^{\max}$ are the minimum and maximum number of access requests to main memory (cache fetches and write-backs; in [8] we show how to handle

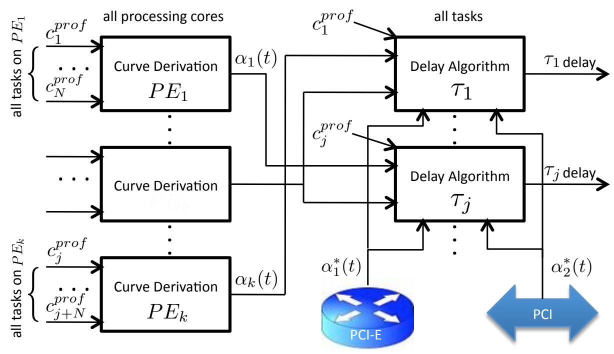

Fig. 1. Analysis Methodology.

write buffers) in superblock $s_{i,j}$. $exec_{i,j}^L$ and $exec_{i,j}^U$ are lower and upper bounds on computation time for superblock $s_{i,j}$ assuming that access requests are completed instantaneously, e.g. they are the times required to execute the instructions in the superblock with no cache misses. Methodologies to compute the cache profile using either experimental measures or static analysis are discussed in [6]. Note that our model implicitly assumes that if the CPU fetch unit is delayed Δ time units in a superblock, its computation time increases by at most Δ. Therefore, we shall either assume a CPU architecture where execution time and communication time are decoupled [13] or that the effect of all timing anomalies can be captured in (pessimistic) bounds provided by static analysis.

Peripherals are connected to the system through dedicated interconnections such as the Peripheral Component Interconnect (PCI) [5]. Before reaching main memory, peripheral requests are typically buffered in the interconnection. Therefore, all peripheral requests coming from the same interconnection are aggregated (see [4] for details) into a single *buffered flow* $\alpha_i^*(t)$: for any interval of length t, $\alpha_i^*(t)$ is the maximum amount of time required by the aggregated peripheral to perform DMA operations in main memory. Each core PE_i is characterized by a parameter C_i, which is the time interval (assumed to be fixed) needed to service a memory request. Furthermore, for each buffered flow and processing core we define an arbitration parameter L_i, which is the maximum duration of an atomic operation in main memory. For core PE_i, C_i is an integer multiple of L_i.

Our analysis methodology uses two successive steps, as shown in Figure 1 for a system with two peripheral interconnections. **(1)** For each core PE_i, we use the technique in Section III to obtain an *unbuffered flow* with arrival curve $\alpha_i(t)$: for any interval of length t, $\alpha_i(t)$ is the maximum amount of time required by tasks running on PE_i to perform operations in main memory. $\alpha_i(t)$ is computed based on the cache profiles of all tasks executed on PE_i. **(2)** For each task τ_i, we use the algorithm in Section IV to compute an upper delay bound. The bound is based on the cache profile c_i^{prof}, all peripheral buffered flows and the unbuffered flows for all cores except the one that executes the task under analysis.

III. Computing Arrival Curves

We specify the interference caused by a task τ_i to other tasks in the system as an arrival curve [12] by considering the superblocks of task τ_i and their respective bounds on execution time and accesses to the shared resource. We initially consider each task in isolation, assuming that no other system component accesses main memory; in Section III-B, we then show how to derive an arrival curve for multiple tasks executed on the same core. Arrival curves are later used in Section IV to compute a delay bound for the task under analysis. Note that when computing the arrival curve for a core, we do not consider the interference caused by other flows on it. As shown in Section IV, this is because each atomic operation of the task under analysis can be delayed for its worst case amount while the interfering cores themselves suffer no delay.

Deriving arrival curves involves **(1)** computing all possible sequences of subsequent superblocks, **(2)** computing the feasible time windows for each sequence, **(3)** computing the minimal and maximal number of cache misses for each time window and **(4)** constructing the arrival curves accordingly.

A. Single Task per Processing Element

In this section we introduce our approach to represent a tasks accesses to a shared resource as arrival curve, assuming a single periodic task per processing element.

1) Computing sequences of super-blocks: Based on the parameters c_i^{prof} of task τ_i, we can derive time windows for which the minimal and maximal number of accesses to a shared resource are known. The time windows can be computed from the set of all possible sequences of subsequent superblocks, i.e., the *sequence set*. As an example, let $\tau_1 = \{s_{1,1}, s_{1,2}\}$ then the sequence set is $*\tau_1 = \{\{s_{1,1}\}, \{s_{1,2}\}, \{s_{1,1}, s_{1,2}\}\}$.

In order to account for the transition phase between succeeding periods of a task, we consider two subsequent instances of a task for the arrival curve derivation. Therefore we specify $\tau_i' = \{\tau_i\ \tau_i\}$, such that $\tau_i' = \{s_{i,1} \dots s_{i,S_i}, s_{i,1} \dots s_{i,S_i}\}$ and $*\tau_i'$ is the corresponding *sequence set*. Element $t_{m,d}' \in *\tau'$ is described by the offset m, representing the index of its first superblock from τ' and d, representing the number of superblocks considered, such that:

$$t_{m,d}' = \{s_{i,m}, \dots, s_{i,m+d}\}\ \forall d \in [0 \dots S_i - 1], \forall m \in [1 \dots S_i]. \quad (1)$$

2) Computing time windows: Each element $t_{m,d}' \in *\tau_i'$ results in four different time windows, considering all combinations of minimal and maximal execution times and accesses to the shared resource respectively. Two of these time windows represent the worst case, i.e., they represent the maximal number of accesses for the shortest time windows. The time windows Δ and their corresponding number of accesses to the shared resource γ are represented as tuples $\hat{t} =< \gamma, \Delta >$ and we show how to compute them in Section III-A3. Accesses to the shared resource can happen at any time during a superblocks execution. In other words, for the first and last super-block in a sequence $t_{m,d}'$, the accesses to the shared resource happen at the end and at the beginning respectively. As a result, the first and last super-blocks' execution times $exec_{i,j}$ are not considered for the representative time windows but their accesses to the shared resource are considered.

Consider Fig. 2 for an example how to compute time windows. In the first example, denoted `1 super-block`, the time window computes as zero, meaning that accesses to the shared resource occur concurrently at one instant of time. For example `2 super-blocks`, the time window Δ computes as the time required to process the first superblocks accesses, while the number of accesses γ computes as the sum of both superblocks' accesses. Execution times are not considered for the time window, since in the worst case the actual computation is performed before and after the first and second superblock respectively. The first superblocks accesses to the shared resource are processed before the second superblock is activated, specifying the time window. In other words, we arrange accesses to the shared resource in subsequent superblocks such that the resulting time window is minimized, thus maximizing the interference onto other tasks.

Computing the time window for elements $t'_{m,d}$, whose superblock sequence spans over the period, needs to consider the gap g between the last superblock of a task and its period. Example `4 super-blocks` in Fig. 2 illustrates such a case. Minimizing the gap, and deductively the time window, is done by assuming the maximal execution time $exec^U_{i,j}$ and number of accesses $\mu^{max}_{i,j}$ for superblocks not included in $t'_{m,d}$.

$$\begin{aligned} g(e,r) &= p_i - \sum_{\forall s_{i,j} \in \tau_i \setminus (\tau_i \cap t'_{m,d})} exec^U_{i,j} + \mu^{max}_{i,j} \cdot C_i \\ &\quad - \sum_{\forall s_{i,j} \in \tau_i \cap t'_{m,d}} exec^e_{i,j} + \mu^r_{i,j} \cdot C_i, \end{aligned} \tag{2}$$

where e and r denote the actual values for execution time and accesses to the shared resource, e.g., $e = U$ and $r = min$.

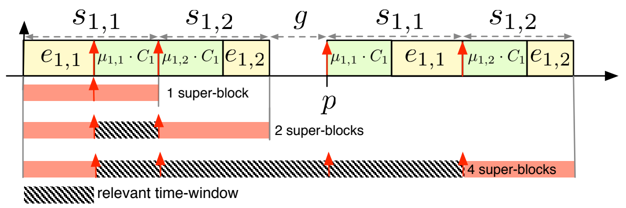

Fig. 2. Computing time windows for sequences of 1, 2 and 4 super-blocks including the gap between periods.

3) Computing the cache misses for each time window: Time windows Δ and the corresponding number of accesses to the shared resource γ for an element $t'_{m,d}$ are computed in Equations 3 and 4. Based on these values, the tuples for element $t'_{m,d} \in *\tau'$ are computed in Equations 5 to 6.

$$\gamma^{min} = \sum_{j=m}^{m+d} \mu^{min}_{i,j} \tag{3}$$

$$\Delta^{min^L} = \sum_{j=m+1}^{m+d-1} exec^L_{i,j} + \sum_{j=m}^{m+d-1} \mu^{min}_{i,j} \cdot C_i \tag{4}$$

$$\hat{t}^{min^L}_{m,d} = < \gamma^{min} \;;\; \Delta^{min^L} + g(L, min) > \tag{5}$$

$$\hat{t}^{max^L}_{m,d} = < \gamma^{max} \;;\; \Delta^{max^L} + g(L, max) > \tag{6}$$

Equation 5 can be transformed into the tuple computed with Equation 6 by simply increasing $\mu^{min}_{i,j}$ to $\mu^{max}_{i,j}$. In other words, they show a linear relation, since the time required to process an access is constant. For any number of accesses within the range of the tuples computed in Equations 5 and 6, we can therefore compute a safe upper bound to the number of accesses performed in the corresponding time window by linear approximation, see Fig. 3.

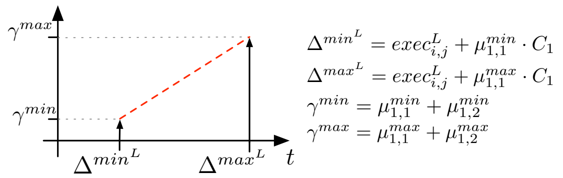

Fig. 3. Linear approximation between minimum and maximum number of accesses to the shared resource for a single super-block

4) Deriving arrival curves: Retrieving the minimal and maximal number of accesses to the shared resource for every time interval $\Delta = \{0 \dots 2p_i\}$ from the computed tuples and linear approximations allows to compute the arrival curve. Consider the function $\delta(\hat{t})$ to return the length of the time window and $\nu(\hat{t})$ to return the number of cache misses for each tuple, then the upper arrival curve $\tilde{\alpha}_i$ can be obtained as:

$$\tilde{\alpha}_i(\Delta) = \underset{\forall \hat{t}_{m,d};\delta(\hat{t}_{m,d})=\Delta}{\operatorname{argmax}} \quad \nu(\hat{t}_{m,d}). \tag{7}$$

We construct the infinite curves $\hat{\alpha}_i$ as an initial aperiodic part, that is represented by $\tilde{\alpha}_i$ and a periodic part which is repeated k-times for $k \in \mathbb{N}$.

$$\hat{\alpha}_i(\Delta) = \begin{cases} \tilde{\alpha}_i(\Delta) & 0 \leq \Delta \leq p \\ \max\left\{ \tilde{\alpha}_i(\Delta), \tilde{\alpha}_i(\Delta - p_i) + \sum\limits_{\forall j} (\mu^{max}_{i,j}) \right\} & p_i \leq \Delta \leq 2p \\ \tilde{\alpha}_i(\Delta - k \cdot p_i) + k \sum\limits_{\forall j} (\mu^{max}_{i,j}) & \text{otherwise} \end{cases} \tag{8}$$

The computational complexity to obtain the overall arrival curves is $O(S_i^2)$. Following the previous computation we derive Lemma 1.

Lemma 1: Deriving alpha curve $\hat{\alpha}_i(\Delta)$ by Equation 8 is the upper bound of accesses to a shared resource by task τ_i for any time window Δ.

Arrival curve $\alpha_i(t)$ represents the maximum amount of time a task τ_i requires to perform its accesses to the shared resource in a time window of length t and is obtained as $\alpha_i(t) = \hat{\alpha}_i(t) \cdot C_i$.

B. Multiple Tasks per Processing Element

In this section we show how to extend the previously shown approach to multiple tasks executing on each processing element. Consider a set of periodic tasks $T = \{\tau_1 \dots \tau_N\}$

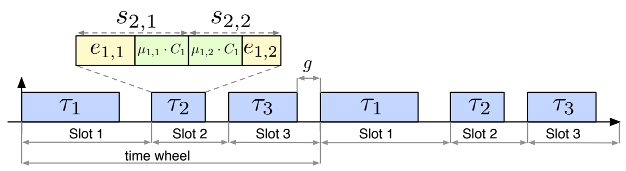

Fig. 4. Example of 3 periodic tasks executing in a static time wheel

scheduled statically, e.g., a static time slot assignment as in Fig. 4. Then we can compute a sequence of all superblocks that constitute tasks in T as:

$$\sigma = \{s_{1,1} \dots s_{1,S_1}, s_{2,1} \dots s_{2,S_2} \dots s_{N,1} \dots s_{N,S_N}\} \tag{9}$$

Based on σ the *sequence set* $*\sigma$ is derived, as shown in Section III-A1. We define $\sigma' = \{\sigma \;\; \sigma\}$ and compute the sequence set $*\sigma'$ such that $t'_{m,d} \in *\sigma'$, resulting in two periods of the static time wheel being considered.

Time windows are computed from the sequence set $*\sigma'$ following the concept presented for a single task per processing element. We compute the maximal and minimal time windows for any possible sequence of subsequent superblocks in $*\sigma'$ and count the corresponding minimal and maximal number of accesses to the shared resource respectively. Elements $t'_{m,d} \in *\sigma'$ contain superblocks from different statically scheduled tasks and therefore the gap between each tasks' last superblock and its period has to be considered. Similarly to the gap g for the single task approach, minimizing the gap results in the worst case.

Equation 2 can be rewritten to compute the gap for a sequence of superblocks, by maximizing all the superblocks that are not considered by element $t'_{m,d}$:

$$g(e,r) = \sum_{\forall \tau_i \in t'_{m,d}} p_i - \sum_{\forall s_{i,j} \in \sigma \setminus (\sigma \cap t'_{m,d})} exec^U_{i,j} + \mu^{max}_{i,j} \cdot C_i \quad (10)$$
$$- \sum_{\forall s_{i,j} \in \sigma \cap t'_{m,d}} exec^e_{i,j} + \mu^r_{i,j} \cdot C_i.$$

The tuples can now be computed as shown in Equations 5 to 6 and based on them the arrival curve can be derived as shown in Sections III-A3 and III-A4.

IV. Delay Analysis

In this section, we initially compute the worst case delay for the task under analysis assuming that it runs in isolation on its assigned processing core. In Section IV-A, we will cover how superblocks can be assigned to fixed timeslices. To simplify notation, we drop the subscript denoting the number of the task under analysis and use c^{prof} as its cache profile, $\{s_1, \ldots, s_S\}$ as its superblocks and C, L as the parameters for its core. We also use $i \in F$ in place of $\alpha_i \in F$ or $\alpha^*_i \in F$. Let $D_{j,k}$ be the maximum delay suffered by the task in superblocks $\{s_j, \ldots, s_k\}$; the overall task delay is equal to $D_{1,S}$. The analysis is based on the following main idea: we first compute an upper bound $Ub_{j,k}$ to the maximum delay $D_{j,k}$ for all $j,k : 1 \leq j \leq k \leq S$, meaning $D_{j,k} \leq Ub_{j,k}$. We then progressively decrease the bound by taking the intersection of multiple such constraints. Due to space limitations, proofs for this section are available in [8].

Since multiple flows contend with the task under analysis for access to main memory, we divide the delay contribution among all interfering flows: we use $D^i_{j,k}$ to denote the maximum total delay caused by flow α_i (or α^*_i) on all operations in superblocks $\{s_j, \ldots, s_k\}$ and $Ub^i_{j,k}$ for its upper bound. We can then obtain $Ub_{j,k}$ as follows:

$$Ub_{j,k} = \sum_{i \in F} Ub^i_{j,k}. \quad (11)$$

Delay bound derivation depends on the memory arbitration scheme. In details, we assume that arbitration among the task under analysis and other unbuffered flows follows either a Round-Robin (RR) or First-Come-First-Served (FCFS) policy, while arbitration among those and buffered flows follows either RR, FCFS or Fixed-Priority (FP) with buffered flows being assigned lowest priority; this is a common optimization in system controllers for embedded systems, see [3]. Arbitration effects are captured by the following lemma.

Lemma 2: Under RR arbitration (or FP for unbuffered flows with lowest priority), each atomic memory operation of the task under analysis can be delayed by flow α_i (or α^*_i) for at most L_i. Under FCFS arbitration, an unbuffered flow can delay each atomic operation for at most C_i, while a buffered flow can delay it for at most b_i, where b_i is the maximum time required to service the backlog (buffered data) of the flow.

Note that in a superblock s_j, the number of atomic memory operations performed by the task under analysis is at most $\mu^{\max}_j \frac{C}{L}$. We can capture the arbitration property expressed by Lemma 2 introducing a *blocking function* B^i_j.

$$B^i_j \equiv \begin{cases} \mu^{\max}_j \frac{C}{L} L_i & \text{for RR and FP} \\ \mu^{\max}_j \frac{C}{L} C_i & \text{for FCFS, unbuffered flow} \\ \mu^{\max}_j \frac{C}{L} b_i & \text{for FCFS, buffered flow} \end{cases} \quad (12)$$

We can then express a first upper delay bound as follows:

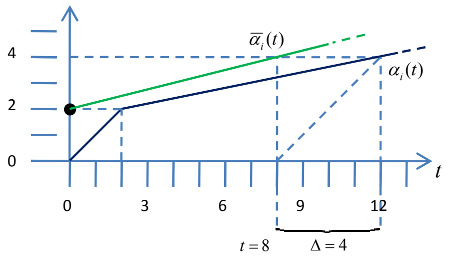

Fig. 5. Derivation of Delay Curve $\bar{\alpha}_i$.

Lemma 3: Blocking Delay Bound: For each flow and superblocks $\{s_j, \ldots, s_k\}$:

$$D^i_{j,k} \leq \sum_{p=j}^{k} B^i_p \quad (13)$$

The bound expressed by Lemma 3 is not tight, because each flow might not produce enough traffic to cause maximum delay to the task under analysis. We can refine the bound by expressing a condition on the amount of service time required by each flow in superblocks $\{s_j, \ldots, s_k\}$. For simplicity, let $\Delta^{\max^U}_{j,k} \equiv \sum_{p=j}^{k} (exec^U_p + \mu^{\max}_p C)$, e.g. $\Delta^{\max^U}_{j,k}$ is the WCET of superblocks $\{s_j, \ldots, s_k\}$ assuming no flow interference.

Lemma 4: Consider a flow α_i, and let $Ub_{j,k}$ be an upper bound to the total delay suffered by the task under analysis in superblocks $\{s_j, \ldots, s_k\}$. Then:

$$D^i_{j,k} \leq \alpha_i \big(\Delta^{\max^U}_{j,k} - C + Ub_{j,k} \big) \quad (14)$$

The C term in Equation 14 captures the constraint that the last memory operation must arrive at least C time units before the end of s_k. Lemma 4 holds for unbuffered flows because $\alpha_i(t'' - t')$ represents the maximum amount of service received in any interval $[t', t'']$. However, the same assumption is not true for a buffered flow, since at time t' there can be additional buffered data, hence the total amount of service time required in $[t', t'']$ can be greater than $\alpha_i(t'' - t')$. In [7], it is shown that the problem can be solved by replacing each buffered flow arrival curve $\alpha^*_i(t)$ with a new arrival curve $\alpha_i(t) = \alpha^*_i(t) + b_i$, where b_i is the maximum time required to service the backlog as previously defined. Lemma 4, as well as the remaining theorems in this section, can then be applied to both unbuffered and buffered flows using arrival curve α_i. The computation of b_i in our model is a simple application of Network Calculus [1] which we detail in [8].

There is a remaining issue. Based on Equation 11, the $Ub_{j,k}$ term in Lemma 4 depends on the delay bound $Ub^i_{j,k}$ for flow α_i, but in turn we would like to compute $Ub^i_{j,k}$ based on the delay bound for $D^i_{j,k}$ provided by Lemma 4. To solve this mutual dependency problem, we introduce a new *traffic delay curve* $\bar{\alpha}_i(t)$:

$$\bar{\alpha}_i(t) \equiv \max\{\Delta | \Delta = \alpha_i(t + \Delta)\}. \quad (15)$$

A graphical representation of the traffic delay curve is shown in Figure 5 for a simple arrival curve α_i. Intuitively, $\bar{\alpha}_i(t)$ is the ordinate of the intersection of $\alpha_i(x)$ with the line $x - t$ (where t is fixed and x varies). We can then obtain $Ub^i_{j,k}$ according to the following lemma.

Lemma 5: Traffic Delay Bound: Consider flow α_i, and let $Ub^{(i)}_{j,k} \equiv \sum_{p \in F, p \neq i} Ub^p_{j,k}$ be an upper bound to the total delay in superblocks $\{s_j, \ldots, s_k\}$ caused to the task under analysis by all flows except flow α_i. Then:

$$Ub^i_{j,k} = \bar{\alpha}_i \big(\Delta^{\max^U}_{j,k} - C + Ub^{(i)}_{j,k} \big) \quad (16)$$

is a valid upper bound to $D^i_{j,k}$.

An improved bound $Ub_{j,k}$ can be obtained by taking the minimum over Lemmas 3, 5; however, such a bound is still fairly pessimistic. Note that since $D_{j,k}$ is the maximum delay for superblocks $\{s_j, \ldots, s_k\}$, it must hold: $D_{j,k} \le D_{j,q} + D_{q+1,k}$ for all $q : j \le q < k$. We can therefore refine the bound on $D_{j,k}$ by considering the minimum between $Ub_{j,k}$ and $Ub_{j,q}+Ub_{q+1,k}$. As an example, consider a task with three superblocks, $\Delta^{\max^U}_{1,1} = 11, \Delta^{\max^U}_{2,2} = 29, \Delta^{\max^U}_{3,3} = 11, L = C = 1$, and a single flow with $\bar{\alpha}_1(t) = \frac{2}{5}t, B^1_1 = 9, B^1_2 = 2, B^1_3 = 9$. Then by computing $Ub_{1,1}+Ub_{2,2}+Ub_{3,3}$ we obtain a delay bound of $\bar{\alpha}_1(10)+B^1_2+\bar{\alpha}_1(10) = 10$, while computing $Ub_{1,3}$ yields $B^1_1 + B^1_2 + B^1_3 = \bar{\alpha}_1(11 + 29 + 11 - 1) = 20$.

The example shows that to obtain a better bound on $D_{j,k}$, multiple subintervals might need to be analyzed, but the number of possible sets of subintervals is exponential in $k-j$. Luckily, as it was shown in [6] for a single flow α_i, there is no need to analyze an exponential number of subinterval: rather, it is possible to use an algorithm that only analyzes a quadratic number of subintervals. The main idea is to compute a delay term $u^{i,j}_k$ for each superblock s_k by iteratively checking all superblocks in $\{s_j, \ldots, s_k\}$. The algorithm iterates over j, k, obtaining at each iteration a bound $Ub^i_{j,k} = \sum_{p=j}^{k} u^{i,j}_p$, which uses the newly computed $u^{i,j}_k$ term. More in details, $u^{i,j}_k$ is computed based on the blocking term B^i_k, the delay terms $u^{i,j}_j, \ldots, u^{i,j}_{k-1}$ and the traffic delay bound of Lemma 5 for each subinterval $\{s_q, \ldots, s_k\}$ with $j \le q \le k$. By computing a delay term $u^{i,j}_k$ for each interfering flow i, the described main idea allows us to produce a tighter bound than Lemmas 3, 5. However, handling multiple flows has an added complexity: when at iteration j, k we compute the traffic delay bound for flow α_i in any subinterval $\{s_q, \ldots, s_k\}$ according to Lemma 5, we must know the maximum delay $Ub^{(i)}_{q,k}$ caused by other flows in that subinterval. The problem can be solved using a dynamic programming approach: instead of iterating over j, k, we first compute delay bounds $Ub_{j,j} = \sum_{i\in F} Ub^i_{j,j}$ for all $j : 1 \le j \le S$, then we compute a delay bound $Ub_{j,j+1}$ for all $j : 1 \le j < S$, then $Ub_{j,j+2}$ and so on and so forth. As shown in Algorithm 1, this is done by iterating over variables d, j, obtaining at each step delay bounds $Ub^i_{j,j+d}$, with $k = j + d$.

Delay term $u^{i,j}_k$ is computed in Equation 20 as the minimum of three delay terms: **(1)** term B^i_k for the blocking delay bound of Lemma 3; **(2)** the minimum over all subinterval $\{s_q, \ldots, s_k\}$, with $j + 1 \le q \le k$, for the traffic delay bound of Lemma 5 (the case of $q = j$ is covered in the third term). Assume that this term becomes minimum among all three terms for a specific choice of q. Then Equation 20 can be rewritten as:

$$\sum_{p=q}^{k-1} u^{i,j}_p + u^{i,j}_k = \sum_{p=q}^{k} u^{i,j}_p = \bar{\alpha}_i(\Delta^{\max^U}_{q,k} - C + Ub^{(i)}_{q,k}), \qquad (17)$$

where following Lemma 5, the rightmost part is an upper bound to $D^i_{q,k}$. Note that since $q \ge j + 1$, it holds $k - q < d$, thus the $Ub^{(i)}_{q,k}$ values have already been computed by the algorithm. Also note that for $d = 0$, $j = k$ and there is no valid value for q, so the term is ignored altogether. **(3)** The traffic delay bound for superblocks $\{s_j, \ldots, s_k\}$. This is a special case of the second term for $q = j$: in the equation, $Ub^{(i)}_{j,k-1} + \sum_{p\in F,p\neq i} u^{p,j}_k = Ub^{(i)}_{j,k}$, but we can not directly use $Ub^{(i)}_{j,k}$ because the $u^{p,j}_k$ values need to be computed together with $u^{i,j}_k$. Hence, we actually need to solve a system of equations computing values $u^{i,j}_k$ for all $i \in F$ simultaneously. Finally, Lines 5-6 are used to update all delay bounds. The algorithm complexity is dominated by the complexity of solving the system of Equation 20, which must be done $O(S^2)$ times. A discussion of how the system can be solved is provided in Section IV-B. The correctness of Algorithm 1 is formally proven in [8] together with an in-depth discussion of its complexity.

A. Multitasking

The analysis of Section IV can be easily extended to a multitasking scenario where tasks are assigned to fixed timeslices. In [7], an algorithm is introduced to compute the minimum number of time slots of fixed length T that must be assigned to the task under analysis. The algorithm works by iterating over superblock $s_k, 1 \le k \le S$: at each step, the algorithm tries to fit s_k in the current time slot based on its WCET and computed upper delay bound. If there is not enough time left, s_k is assigned to a new time slot and the cache profile for s_k and subsequent superblocks is modified assuming that the cache is invalid at the start of s_k. The algorithm can be used in our model substituting the delay analysis of [7] with our new delay analysis for multiple flows.

B. Solving the Delay System

In this section, we detail how to solve the system of Equation 20. Due to the third term in Equation 20, each $u^{i,j}_k$ value depends on all other $u^{p,j}_k$ values, which must thus be computed at the same step. We can obtain a solution for all $u^{i,j}_k$ terms using a recurrence: the idea is to start from a vector of values $(u^{1,j}_k(0), \ldots, u^{i,j}_k(0), \ldots)$, where $\forall i \in F, u^{i,j}_k(0) \ge u^{i,j}_k$. At each step of the recurrence we then compute a new vector $(u^{1,j}_k(c+1), \ldots, u^{i,j}_k(c+1), \ldots)$ based on the previous vector $(u^{1,j}_k(c), \ldots, u^{i,j}_k(c), \ldots)$ and we show that the series converges to a fixed point that is equal to $(u^{1,j}_k, \ldots, u^{i,j}_k, \ldots)$, e.g. it is the only solution to Equation 20.

The series is defined as follows:

$$u^{i,j}_k(0) = \min\Big(B^i_k, \qquad (18)$$
$$\min_{q:j+1\le q\le k} \big\{\bar{\alpha}_i\big(\Delta^{\max^U}_{q,k} - C + Ub^{(i)}_{q,k}\big) - \sum_{p=q}^{k-1} u^{i,j}_p\big\}\Big),$$
$$u^{i,j}_k(c+1) = \min\Big(u^{i,j}_k(c), \qquad (19)$$
$$\bar{\alpha}_i\big(\Delta^{\max^U}_{j,k} - C + Ub^{(i)}_{j,k-1} + \sum_{p\in F,p\neq i} u^{p,j}_k(c)\big) - \sum_{p=j}^{k-1} u^{i,j}_p\Big);$$

intuitively, we compute the initial elements as the minimum of terms (1) and (2) in Equation 20, while each successive element is based on term (3). Note that the $Ub^{(i)}_{j,k-1}$ and $u^{i,j}_p$ values are not part of the recurrence because they have already been computed in Algorithm 1 at a previous step. In [8] we formally prove the correctness of the recurrence.

V. Simulations

We performed extensive simulations to understand how the delay bound varies as a function of task parameters and how fast the delay iteration of Section IV-B converges. In particular, we decided to simulate a quad-core system with RR arbitration, $C = L$ and one task for each core with $S = 10$ superblocks. Tasks are synthetically generated according to

Algorithm 1 Compute $\{Ub_{j,k}\}$

```
1: for $d = 0 \dots S-1$ do
2:     for $j = 1 \dots S-d$ do
3:         $k := j + d$
4:         solve the following system of equations $\forall i \in F$:
```

$$u_k^{i,j} = \min\Big(B_k^i, \tag{20}$$

$$\min_{q: j+1 \le q \le k} \Big\{\bar{\alpha}_i\big(\Delta_{q,k}^{\max^U} - C + Ub_{q,k}^{(i)}\big) - \sum_{p=q}^{k-1} u_p^{i,j}\Big\},$$

$$\bar{\alpha}_i\big(\Delta_{j,k}^{\max^U} - C + Ub_{j,k-1}^{(i)} + \sum_{p \in F, p \neq i} u_k^{p,j}\big) - \sum_{p=j}^{k-1} u_p^{i,j}\Big)$$

```
5:         $\forall i \in F : Ub_{j,k}^i := \sum_{p=j}^{k} u_p^{i,j}$
6:         $\forall i \in F : Ub_{j,k}^{(i)} := \sum_{p \in F, p \neq i} Ub_{j,k}^p$
7: return $\{Ub_{j,k} := \sum_{i \in F} Ub_{j,k}^i\}$
```

three parameters σ, β and α. For each task and superblock, we first generate $exec_{i,j}^U$ according to a uniform distribution with mean $100C$ and coefficient of variation σ. We then generate a cache stall ratio for the superblock, defined as the ratio $\frac{\mu_{i,j}^{\max} C}{exec_{i,j}^U + \mu_{i,j}^{\max} C}$, according to a uniform distribution with mean β and coefficient of variation σ and compute $\mu_{i,j}^{\max}$ accordingly. Finally, we set $exec_{i,j}^L = \alpha\ exec_{i,j}^U, \mu_{i,j}^{\min} = \alpha\ \mu_{i,j}^{\max}$ and we set the period to $p_i = \Delta_{1,S_i}^{\max^U}$. In this section, we show a significant subset of the simulations; complete results are provided in [8].

Figure 6 shows results in terms of the ratio between the computed upper delay bound Ub_{1,S_i} and the WCET $\Delta_{1,S_i}^{\max^U}$ for the task executed on the first core. In the figure, we fix $\alpha = 0.8, \sigma = 0.2$ and vary the cache stall parameter β in $[0, 0.4]$ for the task under analysis and in $[0, 0.2]$ for the other three interfering tasks. Each point in the graph is computed as the average over 100 runs; each run, which involves generating the tasks, computing arrival curves and applying Algorithm 1 took less than a second on a modern PC.

Note that the delay ratio increases almost linearly with the stall ratio for the interfering tasks, until it saturates at roughly three times the stall ratio for the task under analysis (for example, for $\beta = 0.4$ the graph saturates at a value slightly higher than 1.2). This is expected: at a certain point, the memory traffic generated by each interfering core becomes so high that the delay computed by Algorithm 1 is dominated by the blocking factor B_j^i, which does not depend on flow traffic. Also note that a stall ratio of 0.2 for the interfering tasks is enough to cause delay saturation for a task under analysis with stall ratio of 0.4. This is mainly because Algorithm 1 must take into account the mutual effect of multiple flows; the delay caused by one flow can "stretch" a superblock allowing more interfering traffic from other flows. Finally, the series defined by Equations 18, 19 converged in at most 7 steps in all simulations. This is because the delay functions $\bar{\alpha}_i(t)$ obtained from the arrival curves computed in Section III resemble step functions; when step functions are used in the series, at least one element $u_k^{i,j}(c+1)$ decreases by the size of one "step" at each iteration; this in turn causes quick convergence.

VI. Conclusions and Future Work

In a COTS system comprising multiple CPU cores and DMA peripherals, contention for access to main memory can significantly increase a task's WCET. In this paper, we have introduced a new analysis methodology that computes upper bounds to the contention delay suffered by each task. In particular, our analysis is able to abstract each interfering core into an arrival curve which can then be combined with peripheral traffic to yield a delay bound for the task under analysis. The methodology is applicable to a variety of COTS arbitration schemes and cache parameters.

As future work, we plan to extend the analysis to cover more general cache architectures, in particular cache levels shared among a subset of cores. Furthermore, we will investigate how to apply the analysis to dynamic real-time schedulers such as rate-monotonic and earliest-deadline-first.

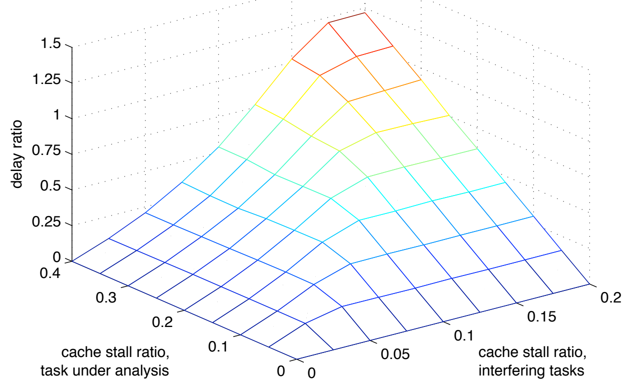

Fig. 6. Delay ratio, $\sigma = 0.2, \alpha = 0.8$.

References

[1] J.-Y. Le Boudec and P. Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer, LNCS, 2001.
[2] S. Edwards and E.A. Lee. The case for the precision timed (pret) machine. Technical Report UCB/EECS-2006-149, EECS Department, University of California, Berkeley, November 17 2006.
[3] Marvell. *Discovery II PowerPC System Controller MV64360 Specifications*. available at http://www.marvell.com/.
[4] M.-Y. Nam, R. Pellizzoni, L. Sha, and R. M. Bradford. ASIIST: Application Specific I/O Integration Support Tool for Real-Time Bus Architecture Designs. In *Proc. of the 14th IEEE ICECCS*, June 2009.
[5] PCI SIG. *Conventional PCI 3.0, PCI-X 2.0 and PCI-E 2.0 Specifications*. http://www.pcisig.com.
[6] R. Pellizzoni, B. D. Bui, M. Caccamo, and L. Sha. Coscheduling of CPU and I/O transactions in COTS-based embedded systems. In *Proc. of the* 29th *IEEE Real-Time System Symposium*, Dec 2008.
[7] R. Pellizzoni and M. Caccamo. Impact of peripheral-processor interference on WCET analysis of real-time embedded systems. *IEEE Transactions on Computers*, 2009. To appear. Available at: http://netfiles.uiuc.edu/rpelliz2/www/.
[8] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele. Memory interference delay estimation for multicore systems. Technical report, 2009. Available at http://netfiles.uiuc.edu/rpelliz2/www/.
[9] J. Rosen, P. Eles A. Andrei, and Z. Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Proc. of the* 28th *IEEE RTSS*, Dec 2007.
[10] S. Schliecker, M. Negrean, G. Nicolescu, P. Paulin, and R. Ernst. Reliable performance analysis of a multicore multithreaded system-on-chip. In *Proceedings of the 6th CODES+ISSS*, Oct 2008.
[11] S. Schönberg. Impact of pci-bus load on applications in a pc architecture. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, Cancun, Mexico, Dec 2003.
[12] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems. In *ISCAS 2000*, volume 4, pages 101–104, Geneva, Switzerland, March 2000.
[13] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):966–978, Jul 2009.