CISTER
Research Center in
Real-Time & Embedded
Computing Systems

Technical Report

Timing analysis of PCM Main Memory in Multicore Systems

Dakshina Dasari
Vincent Nelis
Daniel Mosse

CISTER-TR-130605
Version:
Date: 6/13/2013

Timing analysis of PCM Main Memory in Multicore Systems

Dakshina Dasari, Vincent Nelis, Daniel Mosse

CISTER Research Unit

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: dandi@isep.ipp.pt, nelis@isep.ipp.pt,

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

Abstract

Given that power is one of the biggest concerns of embedded systems, many devices have replaced DRAM with non-volatile Phase Change Memories (PCM). Some applications need to adhere to strict timing constraints and thus their temporalbehavior must be analyzed before deploying them. Moreover, modern systems typically contain multiple cores, causing an application to incur significant delays due to the contention for the shared bus and shared main memory (PCM in this work). One of the challenges in the timing analysis for PCM main memories is the high discrepancy between read and write latencies and the high contention among cores. Finding an upper bound on these delays is non-trivial mainly because (i) memory requests may be issued by co-executing applications at random times, (ii) it is difficult to determine apriori which applications will be concurrently executing, and (iii) the type of requests applications will issue. This work proposes a method to derive upper bounds on the increase in execution time of applications executing on such PCM-based multicores. It considers the contention on the shared memory and focuses on dealing with the asymmetric read and write latencies of PCM-based memories, while taking into account the specific policy applied to schedule requests by the memory controller.

This work was partially supported by National Funds through FCT (Por-tuguese Foundation for Science and Technology) and by ERDF (European Regional Development Fund) through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within project Ref. FCOMP-01-0124-FEDER-037281 and [REPOMUC] project, ref. FCOMP-01-0124-FEDER-015050, and by ESF (European Social Fund) through POPH (Portuguese Human Potential Operational Program), under PhD grant SFRH/BD/70701/2010.

Timing analysis of PCM Main Memory in Multicore Systems

Dakshina Dasari*, Vincent Nelis*, Daniel Mosse†
*CISTER-ISEP Research Centre, Polytechnic Institute of Porto, Portugal
†Department of Computer Science, University of Pittsburgh, USA
{dndi, nelis}@isep.ipp.pt; mosse@cs.pitt.edu

***Abstract*—Given that power is one of the biggest concerns of embedded systems, many devices have replaced DRAM with non-volatile Phase Change Memories (PCM). Some applications need to adhere to strict timing constraints and thus their temporal behavior must be analyzed before deploying them. Moreover, modern systems typically contain multiple cores, causing an application to incur significant delays due to the contention for the shared bus and shared main memory (PCM in this work).**

One of the challenges in the timing analysis for PCM main memories is the high discrepancy between read and write latencies and the high contention among cores. Finding an upper bound on these delays is non-trivial mainly because (i) memory requests may be issued by co-executing applications at random times, (ii) it is difficult to determine apriori which applications will be concurrently executing, and (iii) the type of requests applications will issue. This work proposes a method to derive upper bounds on the increase in execution time of applications executing on such PCM-based multicores. It considers the contention on the shared memory and focuses on dealing with the asymmetric read and write latencies of PCM-based memories, while taking into account the specific policy applied to schedule requests by the memory controller.

I. Introduction

A key development in the embedded systems arena is the adoption of the multicore technology as their core processing platform. Another interesting development related to memory systems has been the promotion of Phase Change Memory (PCM) as main memory for embedded systems [1], [2], [3], [4], [5]. PCM has been positioned to complement or replace existing volatile memories like Dynamic Random Access Memory (DRAM) as the main memory and as a potential alternative to FLASH memory. PCM is more power efficient than DRAM because it is non-volatile (does not need periodic refreshes).

In spite of the aforementioned benefits, its adoption for real-time embedded systems is not without its own challenges: its read latency is acceptable but the write latency is very high. While DRAM read and write latencies are in the range of 20-50ns, PCM read latency is of the order of 50ns while the write latency is of the order of 0.5–1μs [6]. With such high latencies, from the real-time context, many tasks on systems with PCM-based memory (without any modifications) may miss their deadlines or incur unacceptable delays in their execution times [7]. To address this issue, researchers have proposed PCM memory controller scheduling policies and designs that overcome these challenges, facilitating its adoption in real-time systems [3], [7]. From the architecture side, increasing the cache sizes can also mitigate the penalties associated with the high write latencies. Researchers have also envisioned and architected a multi-tiered vertical memory hierarchy which consists of the on-chip caches, an off-chip DRAM memory and then a PCM main memory as the last memory level.

Our work focuses on developing a mechanism to aid the timing analysis of real-time embedded systems hosted on multicores systems with PCM as the main memory.

A. Problem overview and challenges

To ensure at design time that *real-time* embedded applications deliver the required functionality within pre-set time limits, bounds on key parameters like the worst-case execution time (WCET) must be established. In this paper, we build on state of the art methods that compute these WCET estimates, and address the problem of extending such upper bounds *considering the contention for the PCM memory controller and the asymmetric read and write latencies*. We assume a multicore system with private caches and a PCM-based main memory system.

Another problem of PCM is its limited endurance (up to 10^8 writes), which can be mitigated with a large dedicated on-chip cache (SRAM or embedded DRAM) that can absorb most of the write misses – PCM-only memories then become feasible with the advantage of energy efficiency and density.

Many researchers have dealt with the challenges of resource contention in multicores [8], [9], [10]. With the advent of multicores, architectures are enhanced with various techniques to improve average-case performance (e.g., re-ordering memory requests from the different cores and memory controller with complex scheduling policies). These techniques make it extremely challenging to analyze and do not facilitate time-predictability. Real-time system analysts favor temporal and spatial isolation but the presence of shared resources governed by performance-oriented arbitration policies contradicts this requirement. For example, the shared memory bus in modern commercially available multicores is designed with multiple pipeline stages and support for split transactions, thereby making it extremely complex to analyze [11]. Furthermore, the details of the bus arbitration mechanism are not specified and interfaces to modify them are typically not provided by the vendors.

To enhance performance, the memory bus is work conserving and does not ensure guaranteed time-slots to applications.

This work was partially supported by National Funds through FCT and ERDF (European Regional Development Fund) through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within project Ref. FCOMP-01-0124-FEDER-022701; by FCT and COMPETE (ERDF), within REPOMUC project, ref. FCOMP-01-0124-FEDER-015050; by FCT and ESF (European Social Fund) through POPH (Portuguese Human Potential Operational Program), under PhD grant SFRH/BD/71169/2010.

As a consequence, the timing parameters (e.g., the WCET) of a task that runs in isolation on these subsystems are different from those observed when the task is run in conjunction with other tasks. The WCET of tasks, for instance is known to increase significantly due to contention for these shared resources and thus the designer must be able to determine tight upper bounds on this value, perform schedulability tests and ascertain whether the tasks will meet their deadlines at run time.

B. Main contributions

State-of-the-art analysis techniques to compute the delay due to contention on the shared memory do not consider the request handling mechanisms within the memory controller and treat it as a black box. A fixed latency for servicing read and write memory requests is typically assumed in the analysis, which is appropriate for DRAM. A new memory scheduling policy considering PCM's read/write timing asymmetries [7] reduces the number of deadline misses and makes it practical to deploy real-time applications. This work builds on [7] and provides the timing analysis for PCM main memories in multicore systems. We believe that this is the first work to derive the increased WCET of a task considering asymmetric-latency based systems and the memory request scheduling policy within the realm of real-time systems. Although this particular analysis focuses on PCM, it could be used for other memory technologies with asymmetric read and write latencies like Spin Transfer Torque (STT) memories.

In this work, we propose a method to model the timing of requests in the PCM memory controller considering the memory scheduling policy. We leverage the model to compute the increased WCET of a task considering the contention on the bus and the memory controller. Our method exploits the memory request profile of the analyzed tasks in order to tighten the WCET. The analysis is then validated by running our proposed method on benchmarks from MediaBench [12]. A set of experiments have been performed to highlight the impact of other parameters like the nature of co-scheduled tasks and the task priorities on the WCET.

The rest of the paper is organized as follows: Section II discusses the related work in the area of timing analysis and PCM. The system model is described in Section III. An initial basic approach is proposed in Section IV, followed by Section V which describes our new method. This method is validated and the results are presented in Section VI. The paper finally concludes in Section VII.

II. Related Work

A. Earlier work on PCM

PCM has been proposed as a promising candidate for energy-efficient main memory systems. Lee et al. [13] propose area-neutral buffer organizations and partial write techniques to mitigate the negative impacts of PCM's long latencies, high energy and limited endurance. Qureshi et al. [14] propose a hybrid architecture that uses a DRAM cache to filter accesses to PCM. The hybrid architecture has the latency benefits of DRAM and the capacity and scalability advantages of PCM. Ferreira et al. [15] study page partitioning in the DRAM cache to reduce the amount of data written back to PCM. Zhou et al. [16] propose PCM as a direct replacement for DRAM in main memory without buffer organization. Zhang et al. [17] present a hybrid PCM/DRAM memory architecture that uses a small DRAM as write buffer. OS-level paging scheme is applied to improve PCM write performance and lifetime.

Researchers have also proposed techniques for mitigating the impact of undesirable PCM characteristics. As mentioned above, buffer organizations [15], [13], [14], [17] are effective to hide the impacts of slow PCM writes (compared to DRAM). Techniques like write cancellation and write pausing [18] have also been proposed to improve the performance of PCM reads by delaying the extremely slow write operations.

PCM controller modifications to make it real-time friendly: Tasks executing on a system with the basic PCM memory system can experience high deadline misses. To overcome this issue, the authors of [7] proposed three main features to be integrated into the PCM memory controller, which resulted in substantially reducing the number of tasks that missed their deadlines.

1) Ability to attach external priorities to each memory request, together with the type of the request (read or write) and its arrival time. Priorities are assigned to requests based on the task properties, using algorithms like EDF and RMA.
2) Critical read boosting, which prioritizes critical reads over non-critical prefetch reads.
3) Preference of Reads over Writes. The rationale is that since writes can be buffered and the latency due to a write operation is very high, reads must be prioritized over writes to reduce the waiting time for read responses.

However, their work did not focus on the timing analysis of their proposed model which is the main theme of this paper.

B. Earlier work on Timing Analysis in the area of multicores

While timing analysis for unicore processors has been widely studied (see [19] for a compilation of techniques), the same cannot be said for multicores. The presence of shared hardware resources has brought forth new challenges and has been the focus of ongoing research. In the context of this work, it is noteworthy to cite the works dealing with shared memory and shared bus contention. Time Division Multiple Access (TDMA) based schemes have been proposed in [8], [20], [21], [9] and [10]. The methods approach the problem in different ways: precomputing application specific bus schedules, or analyzing buses with the assumption of separate buses for memories and data, restricting accesses to the bus in specific phases of task execution, division of the tasks into superblocks which execute in specific slots and using FlexRay like approaches to have fixed and reserved slots. These methods therefore require a change in either the application behavior which is not desirable or modification in the hardware which is not easy to achieve. An analysis of a work conserving bus is presented in [22] and [23]. The response time analysis of [22] is pessimistic as it assumes that the analyzed task is blocked by *all* the tasks executing on the other cores. Similarly, the notion of the minimum request distance yields pessimistic bounds for the methods proposed in [23]. The task-packing method employed in [24] to maximize the number of requests generated by the core yielded tighter bounds than that proposed in [23] and its *modified version will be used in this work*. Also of interest are the works of [25] based on timed automata which is restricted to *instruction accesses only* and the work of [26] which again assumes division of tasks into superblocks which run in statically pre-assigned time slots, which limits the flexibility, while the pre-allocation of superblocks for all tasks limits the scalability. The work in this paper is different in that

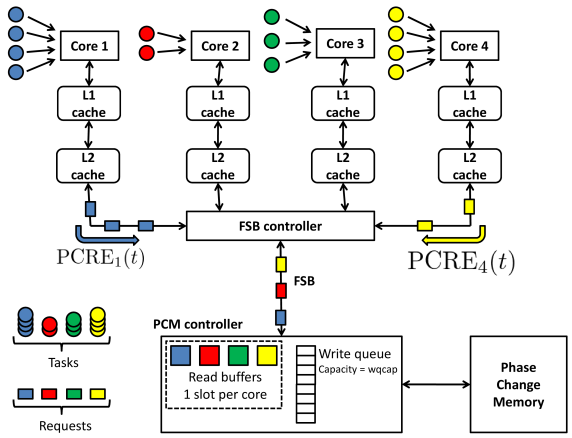


Fig. 1: Platform model

it takes into account the memory scheduling policy, exploits the memory profiling information of the analyzed task and deals with asymmetric read and write times, which was not considered in these previous works.

III. Model of computation

A. Platform model

Figure 1 shows the system we consider, with m processor cores $(\pi_1, \pi_2, \ldots, \pi_m)$, each of which has one or several levels of large private cache, as in the MPC8641D processor from Freescale. The assumption of a private cache is made for the following reasons (i) to focus on the problem of memory contention (ii) data cache analysis on modern shared caches with high set associativity is very difficult [27] or overly pessimistic (iii) spatial partitioning in hardware [28] is preferred by certification experts for safety critical systems and enables composable analysis. All the cores are connected to the memory controller by a single *shared bus*, also called the Front-Side Bus (or FSB)[1]. All the traffic between the cores and the memory controller is transmitted over the FSB and memory requests are scheduled by the PCM controller. Embedded *real-time* application designers favor systems which exhibit predictability and analyzability, have low power consumption, less weight, and small form factor and this may come at the cost of performance – the above model is thus typical of such systems.

B. Workload and memory model

The workload is modeled as a set $\{\tau = \tau_1, \tau_2, \ldots, \tau_n\}$ of n tasks, each of which is characterized by three timing parameters: C_i, T_i and $D_i \leq T_i$. Each task τ_i generates a (potentially infinite) sequence of jobs released at least T_i time units apart (T_i is referred to as the period or minimum inter-arrival time) and each such job has to execute for *at most* C_i time units within D_i time units from its release. The parameter D_i is called the "deadline" of the task and the parameter C_i denotes an *upper bound* on its execution time when it executes uninterrupted and in *isolation*, i.e., with no contention on any of the shared low-level hardware resources. C_i is called the "worst-case execution time"(WCET) of τ_i and can be computed by well-known WCET analysis techniques [19].

Besides the three computation parameters, each task is also characterized by its *worst-case memory request profile* that can be computed by measurements or by static analysis methods [29], [30]. The request profile of a task indicates the maximum number of read and write requests that it can generate in any time interval of a given length t. Given the task memory profiles, the task-to-core assignment, and the timing parameters of the tasks, the per-core memory profile can be computed [24]. This "Per Core Request Evaluator", denoted by $PCRE_j(t)$, computes the maximum number of requests that can be issued from tasks executing on core π_j in any time interval of length t.

C. Scheduler specification

We consider a fully partitioned scheme of task assignment in which every task is assigned a particular core *before* run-time and is not allowed to migrate from one core to another. We denote by $\bar{\pi}_i$, the set of all cores, excluding the one on which τ_i is assigned. The scheduler is assumed to be fixed task-level priority, non preemptive and non-work-conserving. That is, each task is assigned a fixed priority at design time and, when dispatched, it runs to completion without being interrupted or preempted. If a job completes its execution earlier than its WCET, the core remains idle until this WCET, no matter if other jobs are awaiting execution on that core. This is to ensure that the upper-bound on the number of memory requests computed for a given interval at design time is not exceeded at run time by the (early) arrival of the (next) task. Moreover, non-preemptive scheduling avoids cache related preemption delays and task switching overheads.

D. Request scheduling in the FSB controller

Generally in a real-time system, tasks are prioritized and scheduled accordingly so that they all meet their deadlines. While the task scheduler respects these priorities and gives preferential access to the core to tasks with a high priority, in a multicore system with shared main memory, a task may still miss its deadline due to memory contentions if the shared bus and the memory controller do not enforce this prioritization. Therefore, we adopt *globally unique external priorities* to manage memory requests of tasks scheduled on different cores [7]; each memory request inherits the priority of the task issuing it, ensuring that requests from higher-priority tasks arrive earlier at the PCM controller.

The bus is thus priority-driven and is work-conserving: if there is any pending request to be served, the bus cannot be idle. In addition, we assume that there is no hardware prefetching (or it is disabled) as untimely and speculative prefetches can add to the non-determinism, besides adding to the bus traffic and stalling requests of higher importance arriving from the currently executing tasks.

E. Request scheduling in the PCM controller

Constraints on the read requests: We assume that there cannot be multiple outstanding *read* requests from any core, i.e., a core cannot issue a new read request before receiving a response to its previous request. Thus, a core is stalled on issuing a read request until it receives the required response.

Handling the write requests: Since the write latency is much higher than the read latency, non-preemptive writes can considerably increase the task response times. To reduce these

[1]Systems with a large number of (e.g., 64) cores use a network on chip interconnection for communication which is not the focus of this paper.

delays and associated task stalls, the PCM controller queues the write requests in write-buffers so that a task can proceed without waiting for a write operation to be completed. As long as the write buffer is not full, the PCM controller schedules the pending read requests. In the unlikely case that a read request is issued to an memory address pending in a write queue buffer, the controller responds with the data in the write buffer. When the write queue is full, all the pending requests (reads and writes) are sorted in decreasing order of priority (with the highest priority request being positioned at the front of the write queue) and the controller starts serving the reads and writes based on their respective priorities until the write queue is non-full again. The PCM controller then switches back to prioritizing reads over writes. Since the memory controller is work conserving, the write requests are also served when there are no pending read requests issued by any of the tasks.

F. Problem Definition

For each task $\tau_i \in \tau$, given its *WCET C_i and the memory profile in isolation*, compute *the increase* in the WCET C'_i when it runs *in conjunction with other tasks* deployed on a multicore system with a shared PCM. The problem consists of finding a tight upper-bound on the cumulative delay that memory requests may incur in the FSB and PCM controllers. Let N_i^{read} (N_i^{write}) denote the maximum number of read (write, resp.) requests generated by task τ_i during its execution time C_i and $w_{i,k}^{\mathrm{rd}}$ ($w_{i,k}^{\mathrm{wr}}$) denote the waiting time for the k^{th} read (write, resp.) request of τ_i. The objective is to find a tight upper bound on C'_i.

$$C'_i = C_i + \sum_{p=1..N_i^{\mathrm{read}}} w_{i,p}^{\mathrm{rd}} + \sum_{q=1..N_i^{\mathrm{write}}} w_{i,q}^{\mathrm{wr}} \tag{1}$$

IV. An Initial Approach to the Problem

A basic approach to derive C'_i is to compute an upper bound on the delay that a single request can incur and then assign the same delay to each request. That is, if $\bar{w}$ denotes the maximum delay for a single request and N_i denotes the maximum number of requests issued by task τ_i, the resulting WCET can be upper-bounded as follows.

$$C'_i = C_i + N_i \times \bar{w} \tag{2}$$

The above method clearly leads to an overly pessimistic estimation of the increased WCET, C'_i, because it assumes that all the requests of τ_i are subjected to the same (bursty phase of) external task interference from other tasks (which is the worst-case scenario for a single request). It is very unlikely that this assumption is valid since the other tasks will keep progressing in their execution (alternating between computation and memory fetch phases) and will not keep on congesting the memory system at all times. However, this concept of assuming the worst-case scenario for a given parameter and applying it to all other instances is *widely* used in the area of timing analysis. For example, the WCET or the worst-case response time of a task are typically computed by considering the worst-case scenario for a *single* job, and all the jobs are then assigned the same values in the subsequent schedulability analysis. The next section proposes an alternative method which will lead to tighter estimates.

V. Upper Bounds on the External Interference

A. Overview

Let τ_i denote the task under analysis and $\mathrm{hp}(i)$ denote the set of all the tasks of higher priority than τ_i. Also, recall that $\bar{\pi}_i$ represents the set of all the cores, excluding the one on which task τ_i is assigned.

During the execution of τ_i, higher priority tasks running on the other cores (in $\bar{\pi}_i$) may generate requests that interfere with the requests issued by τ_i. The *contiguous intervals* of time during which requests from higher priority tasks are being served by the memory controller will be referred to as "busy periods". Since tasks have alternating phases of computation and memory fetches, there are some "gaps" during which the tasks co-executing with τ_i may *not* be issuing requests (or they issue only requests of lower priority) and the memory controller can thus schedule requests from the analyzed task τ_i or lower priority tasks (if there are no requests from the analyzed task at those instants). These gaps in which the memory controller is not serving requests from higher priority tasks are referred to as "idle periods". Note that these concepts are defined in the context of the *analyzed task τ_i*.

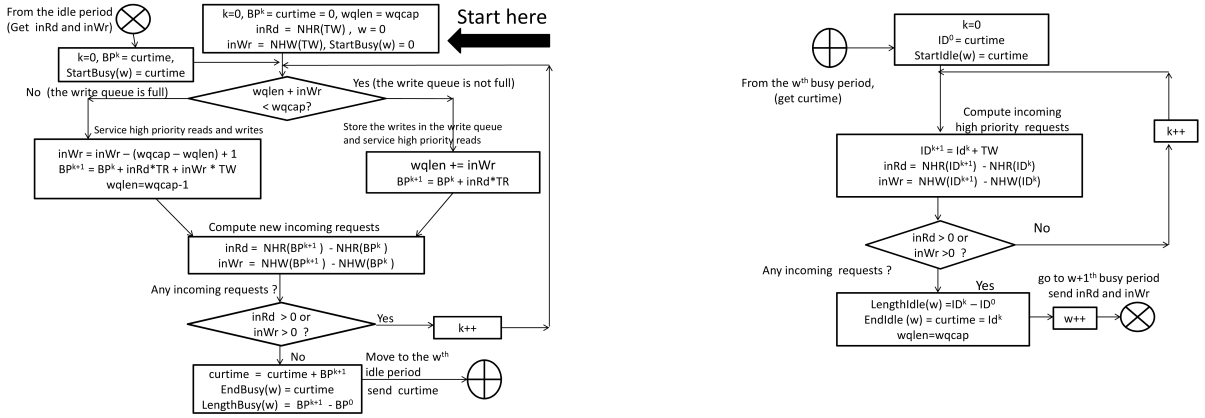
An extended timeline can thus be visualized, which models the schedule of the requests in the controller, consisting of alternating busy and idle periods. The proposed method achieves the objective of computing the increased WCET in two main phases:

1) It determines all the busy and idle periods over an extended duration [0, D_i] where D_i is the deadline of the analyzed task τ_i.
2) It then schedules the requests of the analyzed task τ_i in such a way that its overall execution time is maximized, that is, determine the latest possible time the task will finish, so that it can be checked whether there will be a deadline violation. Towards that goal, we take into account the information on the busy periods to maximize the waiting time of the requests.

B. Phase 1: Determination of the busy and idle periods

1) Overview and notation: The rationale behind the proposed approach is to compute the busy and idle periods by analyzing the working of the PCM controller, considering that the maximum number of requests from the cores in $\bar{\pi}_i$ are generated. The analysis is carried out for a pre-set time interval: from task release to deadline (i.e., until D_i). The computation of the alternating sequence of busy and idle periods is performed by using two automata: the busy and idle automata. In the busy automaton, the algorithm iterates as long as interfering higher-priority requests can be generated, with the aim of maximizing the length of the computed busy period. When no further higher priority requests can be generated by the cores in $\bar{\pi}_i$, the algorithm switches to the idle automaton wherein it keeps increasing the idle period duration until there is a new incoming higher-priority request issued by tasks executing on the other cores, and then switching back to the busy automaton. The algorithm terminates when the deadline is exceeded either in the busy or the idle automaton. While the deadline of the task marks the end of the analysis interval in the proposed approach, other parameters like a specific threshold on the number of busy periods may be used to limit this interval.

Before modeling the working of the PCM controller to capture the worst-case scenario (in terms of sequence of busy and idle periods), a pre-requisite is to capture the *maximum*



(a) Busy period automaton

(b) Idle period automaton

Fig. 2: The busy and the idle period automata

Notation	Meaning
wqcap	the capacity of the write queue
wqlen	the number of slots currently used in the write queue
inRd and inWr	the number of incoming (high priority) read and write requests, respectively
k and $\mathrm{curtime}$	the iteration index and the current time respectively
BP^k	the current time after the k^{th} iteration
$\mathrm{StartBusy}(w)$ and $\mathrm{EndBusy}(w)$	store the time at which the w^{th} busy period starts and ends, respectively
$\mathrm{StartIdle}(w)$ and $\mathrm{EndIdle}(w)$	store the time at which the w^{th} idle period starts and ends, respectively
$\mathrm{LengthBusy}(w)$ and $\mathrm{LengthIdle}(w)$	the length of the w^{th} busy and idle period, respectively
TR and TW	upper bounds on the time to serve a read and a write request by the PCM memory module

TABLE I: Notations used in the automata

number of requests that can be issued from the interfering cores (i.e., the cores in $\bar{\pi}_i$) in any given time interval. We leverage the function $\mathrm{PCRE}_p(t)$ [24] to compute the required interference from tasks of higher priority and compute the lengths of the busy and idle periods. The notation used is shown in Table I.

Augmenting $\mathrm{PCRE}_p(t)$, two new functions, $\mathrm{PCRER}_q(i,t)$ and $\mathrm{PCREW}_q(i,t)$, denote the upper bounds on the number of reads and write requests of higher priority (than the requests of task τ_i) generated by core π_q in a time interval of length t. Then, for the analyzed task τ_i we denote by $\mathrm{NHR}(i,t)$ ($\mathrm{NHW}(i,t)$, resp.) an upper bound on the *cumulative* number of read (write, resp.) requests issued from tasks in $\mathrm{hp}(i)$ executing on the other cores (in $\bar{\pi}_i$) in a time interval of length t.

$$\mathrm{NHR}(i,t) = \sum_{q \in \bar{\pi}_i} \mathrm{PCRER}_q(i,t) \tag{3}$$

$$\mathrm{NHW}(i,t) = \sum_{q \in \bar{\pi}_i} \mathrm{PCREW}_q(i,t) \tag{4}$$

For brevity, we will *drop the task index* i in the automata and denote the functions as $\mathrm{NHR}(t)$ and $\mathrm{NHW}(t)$.

2) The busy period automaton: The flowchart in Figure 2a models the working of the PCM controller when read and/or write requests are generated by the higher priority tasks running on the cores in $\bar{\pi}_i$. To create the scenario leading to the maximum duration of the busy period, the algorithm begins with the initial condition that the write queue is full, reflected by $\mathrm{wqlen} = \mathrm{wqcap}$, and that a write request is currently being served (hence $\mathrm{BP}^0 = \mathrm{TW}$). Before each iteration in the main loop, the algorithm checks if there is any new incoming read or write requests from the higher priority tasks. The incoming write requests may cause the write queue to overflow.

The PCM controller decides which request to schedule based on the current write queue occupancy. Note that the status of the write queue (Full or Non-Full) is decided taking into account the current occupancy of the write queue *and* the number of incoming write requests. Two cases may arise:

Case 1. If the write queue is *not* full, the algorithm takes the right branch of the flowchart. Since the incoming writes can be buffered in the queue (reflected by "wqlen+=inWr"), the controller serves only the *read* requests. The delay $\mathrm{inRd} \times \mathrm{TR}$ is thus added to the total busy period length.

Case 2. If the write queue is or will be full, at least one new incoming write request cannot be buffered and the cores issuing them are stalled (the algorithm takes the left branch). The controller then starts serving *read and write* requests in priority order *until the write queue is non-full again* (in other words, the controller does not have to serve *all* the pending write requests). In the worst-case scenario, it has to serve all the pending read requests plus enough write requests (including new incoming requests) so that the write queue is no longer full. For example, suppose that the capacity of the write queue is 6, 4 slots are currently occupied and there are 2 incoming read requests and 5 incoming write requests. In the worst case, the controller has to serve the 2 incoming reads but only $\mathrm{inWr} - (\mathrm{wqcap} - \mathrm{wqlen}) + 1 = (5 - (6-4) + 1) = 4$ writes, after which 5 slots will be occupied in the write queue

and thus there will be one vacant slot (i.e., the queue is non-full again).

The variables wqlen and inWr are correspondingly updated to reflect the execution of this procedure and the delay $(\mathrm{inRd} \times \mathrm{TR} + \mathrm{inWr} \times \mathrm{TW})$, computed with the reduced value of inWr, is added to the total busy period length. When there are no more read nor write requests issued between BP^k and BP^{k+1} from higher priority tasks in $\mathrm{hp}(i)$ running on cores in $\bar{\pi}_i$, the process terminates and the controller is free to serve requests of other lower priority tasks (including those of τ_i). The length of the current (i.e., the w^{th}) busy period is given by $\mathrm{LengthBusy}(w) = \mathrm{BP}^k - \mathrm{BP}^0$. The variable $\mathrm{curtime}$ is updated by a delay of $\mathrm{LengthBusy}(w)$ and the algorithm moves to the idle automaton.

a) Example: A given busy period is computed by an iterative process. The process initially starts with the notion that the controller is busy serving a write request which needs TW units to be completed. Hence, the initial value, BP^0, is set to TW. In the interval $[0, \mathrm{TW}]$, assume that there are 3 new incoming read requests from the higher priority tasks. The memory serves these 3 requests and the length of the busy period is increased to $\mathrm{BP}^1 = \mathrm{TW} + 3\mathrm{TR}$. While serving these 3 requests, assume that there are 2 incoming high priority requests, a write and a read requests. If the write queue is not full, then the write request is buffered and hence the write does not contribute to the delay; the controller serves the pending read and the busy period is now $\mathrm{BP}^2 = \mathrm{TW} + 4\mathrm{TR}$. If the write queue is full, one of the buffered write requests must be served to prevent the core issuing the incoming write from being stalled. In that case, one of the write requests plus the incoming read request are served and $\mathrm{BP}^2 = \mathrm{TW} + 4\mathrm{TR} + \mathrm{TW}$. The write request that was pending is now buffered in the write queue and the algorithm checks for new incoming requests in the time interval $[\mathrm{BP}^1, \mathrm{BP}^2]$. If no new requests were issued in that time interval, it marks the end of the busy period. Otherwise, the algorithm keeps on iterating through the main loop until no more higher priority request is generated (or until the current time exceeds the deadline).

Note: It can be shown that the length of the first busy period is the maximum waiting time that a single request can incur. This maximum delay can be used to compute the resulting WCET for the initial approach described in Section IV by applying it in Equation (2). By construction, the first busy period is the longest because the analysis starts with an initial configuration to maximize the waiting time of any given request (the write queue is full and a write is being processed). Also, to compute the worst-case, the NHR and the NHW functions consider that co-executing (interfering) tasks from other cores are generating the maximum number of requests.

3) The idle period Automaton: The idle period marks the phase in which there are no new requests from the higher priority tasks in $\mathrm{hp}(i)$ running on the cores in $\bar{\pi}_i$. The requests generated by the analyzed task τ_i, if issued, may be served by the memory controller[2]. The algorithm determines the length of the idle period by starting from the end of the last busy period; this time-instant is recorded in $\mathrm{curtime}$. The iteration index k is initialized to 0 and ID^0 is set to $\mathrm{curtime}$. The central idea of identifying an idle period is to poll at regular time instants if there are new requests being issued by the higher priority tasks. If there are no new requests, then the algorithm increases the idle period duration by the poll interval and continues looping in the idle automaton. If there are new incoming requests, the algorithm switches back to the busy automaton. Note that at the beginning, we assume that a write request was issued by a lower priority task in order to initiate the loop.

An important design issue is determining the ideal poll interval. A very small poll interval will allow us to capture the idle periods in small steps, leading to a longer analysis time if there are no higher priority requests issued during a long time, whereas a large poll interval will capture the arrival of new requests faster, but as a consequence overlooks (precious) idle gaps between two distant polling points. We assume a polling step of TW in the analysis, (assuming a hypothetical write request to be issued) as seen in Figure 2b.

There can be two cases depending on the arrival of requests between two polling instants.

1) No new requests are issued: The algorithm increases the length of the idle period ($\mathrm{ID}^{k+1} = \mathrm{ID}^k + \mathrm{TW}$) and proceeds to the next iteration.
2) New requests are issued: This marks the end of the idle period. The algorithm updates the current time to ID^k, computes the duration of the idle period and switches back to the busy period automaton.

C. Phase 2: Using the pre-computed busy and idle periods of the analyzed task to compute its increased WCET

1) Modeling: This section focuses on computing a tight upper-bound for the cumulative waiting times of all the requests generated by a given task τ_i by considering the busy periods computed in Phase 1. The waiting time for a given request is maximized if it is issued just before the longest (feasible) busy period (the request is issued but the bus has just started serving a contiguous stream of requests of higher priority). The cumulative waiting time is maximized by adding up the maximum waiting times of each requests (delays incurred due to the busy periods) which in turn results in an upper bound of the worst-case execution time of τ_i. To compute the increase in WCET, we start by modeling the memory request profile of the analyzed task τ_i *in isolation*. The memory profiling is done by dividing τ_i into logical *sampling regions* and determining the maximum number of requests issued in each of these regions. The number of memory requests generated in each region can be determined by static cache analysis [19] or by measurements by instrumenting the L2 cache misses [26] (using performance monitoring counters [31]).

For this analysis, we assume that the analyzed task τ_i is sampled in intervals of length $\mathrm{lenregion}$ and has NSR_i such sampling regions. That is, the worst-case execution time C_i of τ_i is split into NSR_i regions, each of length $\mathrm{lenregion}$: $\mathrm{NSR}_i \times \mathrm{lenregion} = C_i$. We can also generalize it to different regions of unequal length $\mathrm{lenregion}_j$ where $\{j \in 1 \ldots \mathrm{NSR}_i\}$, but will keep it simple at this stage. We denote by $\mathrm{Nr}_{i,j}$ and $\mathrm{Nw}_{i,j}$ the maximum number of read and write requests (respectively) that can be generated in the j^{th} sampling region $SR_{i,j}$ of task τ_i, where $1 \leq j \leq \mathrm{NSR}_i$.

2) Description of Algorithm CompConDelay(): During the sampling of τ_i, the WCET of each region is determined by considering a finite service time of TR time units for the read request but a zero waiting time for the write requests. That is, the memory traces obtained at design time assume

[2]Note that requests from low-priority tasks can also be serviced, but the algorithm we are describing is considering only requests from τ_i

Algorithm 1: CompConDelay(τ_i)

input : τ_i: the task under analysis
output: C_i': The increased WCET
/* Compute required time for a region considering its write requests and blocking write requests from lower priority tasks */
1 **for** $j \leftarrow 1$ **to** NSR_i **do**
$\mathrm{wcet}_{i,j} \leftarrow \mathrm{lenregion} + \mathrm{Nw}_{i,j} \times \mathrm{TW} + (\mathrm{Nw}_{i,j} + \mathrm{Nr}_{i,j}) \times \mathrm{TW}$;
2 **for** $j \leftarrow 1$ **to** NSR_i **do**
3 $t_j^{\mathrm{start}} \leftarrow t_{j-1}^{\mathrm{end}}$;
4 $t_j^{\mathrm{end}} \leftarrow t_j^{\mathrm{start}} + \mathrm{wcet}_j$;
5 **for** $k \leftarrow 1$ **to** $(\mathrm{Nr}_{i,j} + \mathrm{Nw}_{i,j})$ **do**
/* Compute candidate set of busy periods that may delay requests in region j */
6 $B \leftarrow \{x$ such that either of the 2 conditions is met$\}$
7 1. $t_j^{\mathrm{start}} \leq \mathrm{StartBusy}(x) \leq t_j^{\mathrm{end}}$ or ;
8 2. $\mathrm{StartBusy}(x) < t_j^{\mathrm{start}} \wedge \mathrm{EndBusy}(x) > t_j^{\mathrm{start}}$;
/* Do not consider the busy periods that delayed previous requests */
9 k $B \leftarrow B \setminus \bigcup_{x=1}^{k-1}\{b_x\}$;
// Find max. delay among the candidates
10 $b_k \leftarrow \underset{w \in B}{\mathrm{argmax}}\{\mathrm{LengthBusy}(w)\}$;
// current region is extended due to extra delay
11 $t_j^{\mathrm{end}} \leftarrow t_j^{\mathrm{end}} + \mathrm{LengthBusy}(b_k)$;
12 **return** $t_{\mathrm{NSR}_i}^{\mathrm{end}}$;

that the write queue is never full and all the write requests are thus buffered in the queue as soon as they are generated. This implies that the time for servicing every read request is accounted for in the original WCET C_i (and thus in the per-region WCET (lenregion) as well), whereas the time for servicing the write requests must be taken into consideration in the analysis. To this end, lines 1 and 2 of Algorithm 1 adds the following to lenregion:

1) $\mathrm{Nw}_{i,j} \times \mathrm{TW}$, for the reason mentioned above, and
2) $(\mathrm{Nw}_{i,j} + \mathrm{Nr}_{i,j}) \times \mathrm{TW}$, because every request of τ_i may be generated just after the PCM controller starts serving a lower-priority write request. Since the PCM serves requests in a non-preemptive manner, every request of τ_i can potentially be subjected to an extra delay of TW time units.

For each sampling region of τ_i, lines 4 and 5 compute the interval of time $[t_j^{\mathrm{start}}, t_j^{\mathrm{end}}]$ during which the j^{th} region executes (in the worst-case scenario): the j^{th} region starts after the $(j-1)^{\mathrm{th}}$ region completes (assuming $t_0^{\mathrm{end}} = 0$) and ends wcet_j time units later. During this time interval, each request of τ_i will be assigned the maximum possible delay. The requests are considered one-by-one (line 6).

For each request k, the algorithm first creates a candidate set of busy periods, denoted by set B which can potentially delay it. Specifically, this set B contains all the busy periods that start within $[t_j^{\mathrm{start}}, t_j^{\mathrm{end}}]$ (condition 1), plus the busy period (if any) that overlaps the time-instant t_j^{start} (condition 2). Then, the algorithm eliminates $(k-1)$ members from set B, that were already used to delay the previous $(k-1)$ requests of τ_i in the current region j (line 9).

To maximize the waiting time for the given request, the algorithm determines (at line 10) which of these busy periods in set B is the longest and assigns the corresponding delay (assumed to be zero if B is empty) to the current request k. As the request is delayed, the length of the region is extended,

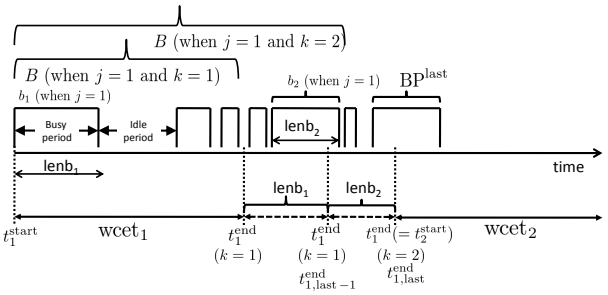


Fig. 3: Visualization of the variables used in Algorithm 1.

which is implicitly reflected by the increase of t_j^{end} at line 12. Finally, the increased WCET C_i' of τ_i corresponds to the end of last region, NSR_i and is captured in the variable $t_{\mathrm{NSR}_i}^{\mathrm{end}}$, which is returned at the end of the entire analysis.

Note that if the increased WCET (C_i') is less than the deadline D_i does not automatically mean that τ_i is schedulable (i.e., will meet its deadline when scheduled with other tasks). All the tasks parameters (including their increased WCET) have to be further provided as an input into a schedulability analysis tool, which will assess the system schedulability by also considering the on-core interference. The focus of this work is to compute the increased WCET; the schedulability analysis should be carried out using existing approaches [32], [22].

D. Proof of safety of Algorithm CompConDelay()

Next we provide a proof that our method indeed computes an upper bound, as desired, on the execution time of a task including the delays due to accesses to PCM.

Lemma 1: The value of C_i' returned by Algorithm 1 is a safe upper-bound on the execution time of τ_i, considering the contention on the shared memory.

Proof. The proof is obtained step-by-step, by examining the properties of all the time-instants t_j^{end} computed by the algorithm. Recall that j indexes the region being examined.

For the first region ($j = 1$), the value of t_1^{end} computed at line 4 is an upper-bound on the completion time of the first region of τ_i, assuming that none of the requests (indexed by k) generated by this region is blocked by higher-priority requests (see Figure 3). When $j = 1$ and $k = 1$, it can be seen that the value of t_1^{end} (re-computed at line 11) is an upper-bound on the completion time of this first request since it considers the maximum blocking for that request. Therefore, during the second iteration in the inner loop (i.e., when $k = 2$), the set B computed at line 6 is guaranteed to contain the maximum number of busy periods that can potentially be used to block a second request. This implies in turn that the value of t_1^{end} computed for the third time at line 11 (during this second iteration when $k = 2$) is an upper-bound on the completion time of the first region, assuming that two requests are blocked by higher-priority requests. The same reasoning can be applied for every subsequent request until the $(\mathrm{Nr}_1 + \mathrm{Nw}_1)^{\mathrm{th}}$ request and thus, t_1^{end} is guaranteed to eventually provide an upper-bound on the completion time of the first region when all its requests can be blocked by higher-priority requests.

Note that during the last iteration (when $j = 1$ and $k = \mathrm{Nr}_1 + \mathrm{Nw}_1$), t_1^{end} is increased for the last time at line 11. Let $t_{1,\mathrm{last}-1}^{\mathrm{end}}$ and $t_{1,\mathrm{last}}^{\mathrm{end}}$ denote the values of t_1^{end} before and after this last increase. To visualize this, let us

assume in Figure 3 that the first region can generate only two requests. By construction of the algorithm, none of the busy periods starting within $[t^{\text{end}}_{1,\text{last}-1}, t^{\text{end}}_{1,\text{last}}]$ can be used to block any request generated in this first region (since there are no more requests from the first region to block). Among those busy periods, some may have their starting *and ending times* within this interval $[t^{\text{end}}_{1,\text{last}-1}, t^{\text{end}}_{1,\text{last}}]$ while *at most* one busy period may start within $[t^{\text{end}}_{1,\text{last}-1}, t^{\text{end}}_{1,\text{last}}]$ and end after $t^{\text{end}}_{1,\text{last}}$. Let us denote by BP^{last} this last busy period that overlaps $t^{\text{end}}_{1,\text{last}}$. Regarding the busy periods that start *and end* within $[t^{\text{end}}_{1,\text{last}-1}, t^{\text{end}}_{1,\text{last}}]$, it is not interesting (in order to maximize the WCET) to assume that the first region finishes earlier than their starting times (i.e., at time $t^{\text{end}}_{1,\text{last}-1}$) so that requests from the second region can "use" these busy periods to increase the overall delay. Assuming so would imply that: (i) after $t^{\text{end}}_{1,\text{last}}$ time units of execution, τ_i is already progressing in its second region (while it could still be executing its first region without this assumption) and, (ii) it uses some requests from the second region to take advantage of these busy periods (and these requests could be used later to further increase the overall delay). However, in order to maximize the cumulative delay, it might be interesting to consider the busy period BP^{last} to block a request of the second region (this will be taken care of during the next iteration of the outer loop).

During the second iteration of the outer loop (i.e., when the algorithm goes back to line 3 with $j = 2$), the algorithm first computes the interval of time $[t^{\text{end}}_1, t^{\text{end}}_1 + \text{wcet}_2]$, where the value of t^{end}_2 is an upper-bound on the completion time of the second region, assuming that all the requests of the first region have incurred the maximum possible delay but none of the requests of the second region have been blocked by higher-priority requests. Then, at line 6, the set B is computed and it can be seen that the busy period BP^{last} is included in that set, thanks to the second condition. By using the same reasoning as above, we can infer that after the $(\text{Nr}_2 + \text{Nw}_2)^{\text{th}}$ iteration in the inner loop (lines 6–11), t^{end}_2 is an upper-bound on the completion time of the second region of τ_i. Following this reasoning, we can see that ultimately $t^{\text{end}}_{\text{NSR}_i}$ is an upper-bound on the execution time of τ_i. ■

VI. Summary of the evaluations

A. Details of the Setup

The tasks used for evaluation are benchmarks chosen from the MediaBench Test Suite [12]. An upper bound on the number of L2 cache misses and WCET is obtained by applying the analysis techniques presented in [22] to the MediaBench tasks. We used Simics, a popular simulator for multi-core architectures [33], to generate the traces *in isolation*. It comprises cores of speed 1GHz, each with a L1 and a L2 cache. The L1 I-cache and D-cache are 4-way, 16KBytes with a cache line size of 64 bytes. The L2 is an 8-way, 512 KBytes unified instruction and data cache with a cache line size of 64 bytes. Unless stated otherwise, unique external priorities are assigned based on the periods of the tasks as in the Rate Monotonic Algorithm. Lower numbers indicate higher priority.

1) Demonstrating the idle and busy period schedule: Figure 4 shows the time at which idle slots are available to tasks. The number of slots is restricted to 50 in this figure for clarity. There are 2 main observations. (i) The first busy period is the longest of all the busy periods in the schedule. To ensure maximum interference, the analysis assumes that the co-executing higher priority tasks generate the highest possible number of memory requests, while the analyzed task begins executing, inorder to stall its progress. As seen in Figure 4, the first idle slot is available at the time 89 to the task with priority 7. (ii) Tasks with lower priority *may* have to wait longer to receive an idle slot, because they are prone to greater interference. Thus, to avail 50 idle slots, task with priority 7 (lowest priority) needs around 150 time units, while it is around 80 time units for a task with priority 6.

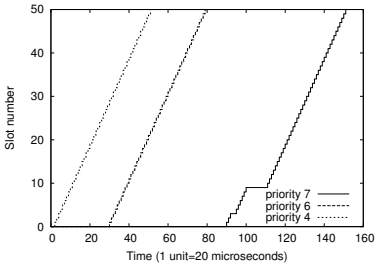


Fig. 4: Slot Availability for tasks with different priorities

2) Comparison with the naive approach: Figure 5 illustrates the tightness of our proposed approach over the naive approach from Section IV. With the naive approach, the WCET of many tasks exceeds the deadlines (in this case the tasks unepic, jpeg-decode, gsm decode, epic and fractal). Epic and unepic are highly memory intensive tasks and thus issue a lot of memory requests and applying the naive approach to these tasks significantly increases their resulting execution times.

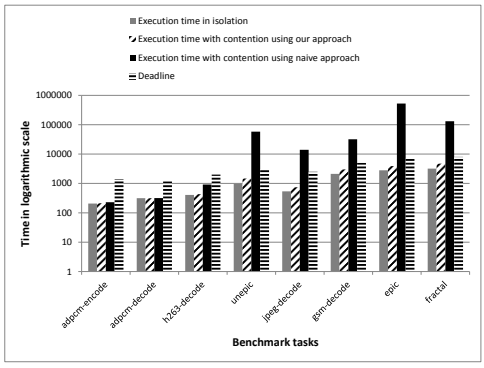


Fig. 5: Comparison with the naive approach (Note: Y-axis is in log scale and 1 unit corresponds to 20 microseconds)

3) Correlation between Task Priorities and the Increase in the WCET: A counterintuitive result is that the impact of external interference from other cores cannot be directly co-related to their priorities, even with priority enforcements. While it generally holds that for the highest priority tasks, the external interference is smaller, this is not the case amongst all lower priority tasks (see Table II). A task of lower priority might incur a lesser interference on its overall execution time than a task with a relatively higher priority.

4) Components of the Increase in WCET: As per the Algorithm CompConDelay, the increase in WCET can be attributed to 3 main components (i) the additional time for each write (ii) external blocking delay by lower priority non-preemptive writes (iii) external interference from higher priority tasks. In Table II we show, for each task, the blocking component and

Benchmark	Priority	%WCET increase	%Blocking	Type
adpcm-decode	1	1.17%	35.00%	L
adpcm-encode	2	1.90 %	21.47%	L
h263-decode	3	15.42%	17.53%	H
unepic	4	92.41%	11.05%	VH
jpeg-decode	5	34.03%	2.38%	M
gsm-decode	6	18.68%	0.43%	L
epic	7	60.12 %	1.30%	H
fractal	8	18.90 %	0.28%	L

TABLE II: Contribution of blocking

the type of task with respect to their memory profiles: **L**ight, **M**oderate, **H**eavy, and **V**ery **H**eavy. It can be seen that the blocking delays contribute to a large percentage of increase in the WCET for the higher priority tasks. The impact on the lower priority tasks is smaller, especially for less memory intensive tasks.

5) Impact of varying the number of cores: It has been observed that while the tasks with higher priorities are impacted marginally by scaling/increasing the number of cores, low priority tasks which are memory intensive are significantly impacted because of the increased external interference. Moreover, the average performance degradation per task increases as the number of cores accessing the same shared memory bus increases (and explains why the single FSB model does not scale and other inter-communication designs are warranted). In this example task set, the increase is 26%, 30% and 33% for 2, 4 and 8 cores, respectively.

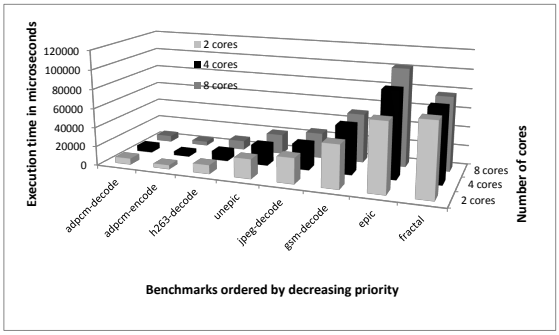


Fig. 6: Tasks spread across 2, 4 and 8 cores

6) Impact of task mix: As a proof of concept to ensure that priorities are respected and to study the effect of core assignments, the analysis with a reference task set was carried out by removing a task with a medium level priority. As expected, the higher priority tasks did not see any changes while the tasks with lower priority suffered less external interference.

7) Impact of task assignments based on request densities: To improve responsiveness, a possible intuitive scheduling algorithm is to prioritize tasks based on their memory request densities (more requests, higher priority, so that they finish earlier). With this set of experiments, we demonstrated that this strategy will lead to the performance degradation for most of the tasks. The effect is worse when highly memory intensive tasks with higher priorities arrive more frequently.

Our experiments show that the increase in execution time for tasks is a complex function of the task profiles (memory or computation intensive), the task assignments to cores, the priority enforcement mechanisms, and the temporal characteristics (the execution time and the period of tasks). For more in-depth evaluation of the results, please see [34].

VII. Conclusions and Future Work

To ensure safe upper bounds, the impact of shared low-level resources on the timing behavior of tasks deployed on multicores must be taken into account while carrying out the timing analysis. In this work, we presented a method to compute the increase in the worst-case execution time of a task considering the contention on the shared Phase Change Memory. Our proposed method takes into consideration the different read and write latencies of the PCM controller, the priorities of the tasks, the request scheduling of the controller, and the interference arising from the co-executing tasks. Our results using embedded benchmarks shows that there is a modest (for most real-time systems) increase in the worst-case computation time of a task, in comparison when the task is run in isolation; surprisingly, we noticed that the lower priority tasks do not always have a higher increase in execution time. Comparisons against a basic approach shows that the proposed method provides tight upper bounds. Lastly, our results also hold for non-embedded benchmarks.

In the future, we will analyze a system consisting of a multi-tiered memory hierarchy in which DRAM is used as an off-chip cache and PCM serves as the main memory.

Acknowledgment: We would like to thank Miao Zhou who provided the traces for the MediaBench and SPEC2006 benchmarks used in the evaluation, and for fruitful discussions of the internals of the PCM controller.

References

[1] "Phase Change Memory: Altered states," http://www.economist.com/node/21560981.

[2] "Market Applications of Phase Change Memory (PCM)," http://ovonyx.com/technology/market-applications-of-phase-change-memory-pcm.html.

[3] A. P. Ferreira, B. R. Childers, R. G. Melhem, D. Mossé, and M. Yousif, "Using pcm in next-generation embedded space applications," in *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2010, pp. 153–162.

[4] J. Handy, "Phase Change Memory will Change Memory System Design," http://www.rtcmagazine.com/articles/view/101556.

[5] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam, "Phase-change random access memory: A scalable technology," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 465 –479, july 2008.

[6] "Phase Change Memory," http://www.pdl.cmu.edu/SDI/2009/slides/Numonyx.pdf.

[7] M. Zhou, S. Bock, A. Ferreira, B. Childers, R. Melhem, and D. Mosse, "Real-time scheduling for phase change main memory systems," in *TrustCom, ICESS11*, nov. 2011, pp. 991 –998.

[8] 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.

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

[10] 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.

[11] "Intel 4 Series Chipset Family Datasheet, For the Intel 82Q45, 82Q43, 82B43, 82G45, 82G43, 82G41 Graphics and Memory Controller Hub (GMCH) and the Intel 82P45, 82P43 Memory Controller Hub (MCH)," 2010, http://www.intel.com/content/www/us/en/processors/xeon/4-chipset-family-datasheet.html.

[12] C. Lee, M. Potkonjak, and W. Mangione-Smith, "Mediabench: a tool for evaluating and synthesizing multimedia and communications systems," in *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on*, dec 1997, pp. 330 –335.
[13] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *ISCA '09*, 2009, pp. 2–13.
[14] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," ser. ISCA '09.
[15] A. P. Ferreira, B. Childers, R. Melhem, D. Mosse, and M. Yousif, "Using PCM in next-generation embedded space applications," in *RTAS '10*, 2010.
[16] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *ISCA '09*, 2009, pp. 14–23.
[17] W. Zhang and T. Li, "Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architectures," in *PACT '09*, 2009, pp. 101–112.
[18] M. K. Qureshi, M. Franceschini, and L. A. Lastras-Montaño, "Improving read performance of phase change memories via write cancellation and write pausing," in *HPCA*, 2010, pp. 1–11.
[19] 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.
[20] 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.
[21] T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury, "Bus-Aware Multicore WCET Analysis through TDMA Offset Bounds," in *2011 Euromicro Conference on Real-Time Systems*, 2011, pp. 3 –12.
[22] D. Dasari, B. Andersson, V. Nelis, S. M. Petters, A. Easwaran, and J. Lee, "Response time analysis of COTS-based multicores considering the contention on the shared memory bus," in *ICESS11,TrustCom*, nov. 2011, pp. 1068 –1075.
[23] 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.
[24] D. Dasari and V. Nelis, "An analysis of the impact of bus contention on the wcet in multicores," in *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS),*, june 2012, pp. 1450 –1457.
[25] W. Yi, "Multicore embedded systems: the timing problem and possible solutions," in *Proceedings of the 12th international conference on Formal engineering methods and software engineering*, 2010, pp. 22–23.
[26] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele, "Worst case delay analysis for memory interference in multicore systems," in *Conference on Design, Automation and Test in Europe*, 2010, pp. 741–746.
[27] J. Reineke, D. Grund, C. Berg, and R. Wilhelm, "Timing predictability of cache replacement policies," *Real-Time Systems*, vol. 37, no. 2, pp. 99–122, Nov. 2007.
[28] IEC 61508, *Functional safety of electrical/electronic/programmable electronic safety-related systems*, 2010.
[29] C. Ferdinand, "Worst case execution time prediction by static program analysis," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, april 2004, p. 125.
[30] D. Grund, "Static cache analysis for real-time systems – LRU, FIFO, PLRU," Ph.D. dissertation, Saarland University, 2012, https://www.epubli.de/shop/buch/Static-Cache-Analysis-for-Real-Time-Systems-Daniel-Grund-9783844216998/13092.
[31] B. Sprunt, "The basics of performance-monitoring hardware," *IEEE Micro*, vol. 22, pp. 64–71, 2002.
[32] 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.
[33] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002.
[34] D. Dasari, V. Nelis, and D. Mosse, ""technical report, timing analysis of pcm-based multicore systems"," CISTER/ISEP, Polytechnic Institute of Porto, Tech. Rep. [Online]. Available: http://webpages.cister.isep.ipp.pt/ dndi/pcmtechrep.pdf