Memory Contention Analysis for 3-Phase Tasks

Jatin Arora
CISTER Research Centre, ISEP, IPP
Porto , Portugal

Syed Aftab Rashid
CISTER and VORTEX CoLab
Porto, Portugal

Geoffrey Nelissen
Eindhoven University of Technology
Eindhoven, the Netherlands

Cláudio Maia
CISTER Research Centre, ISEP, IPP
Porto, Portugal

Eduardo Tovar
CISTER Research Centre, ISEP, IPP
Porto, Portugal

ABSTRACT

In multiprocessor-based real-time systems, the main memory is identified as the main source of shared resource contention. Phased execution models such as the 3-phase task execution model has shown to be a good candidate to tackle the memory contention problem. It divides the execution of tasks into computation and memory phases that enable a fine-grained memory contention analysis. However, the existing work that focuses on the memory contention analysis for 3-phase tasks can overestimate the memory contention that can be suffered by the task under analysis due to the write requests. This overestimation can yield pessimistic bounds on the memory access times and memory contention suffered by tasks which in turn lead to pessimistic worst-case response time (WCRT) bounds. Considering the limitation of the state-of-the-art, this work proposes an improved memory contention analysis for the 3-phase task model. Specifically, we propose a memory contention analysis for the 3-phase task model by tightly bounding the memory contention suffered by the task under analysis due to the write requests. The proposed memory contention analysis integrates memory address mapping of tasks to improve the bounds on the maximum memory contention suffered by tasks.

1 INTRODUCTION

The adoption of multicore platforms in *hard real-time systems*, i.e., systems that run applications with stringent timing requirements, is still under the scrutiny of academia and industry. The main challenge that hinders the use of commercial off-the-shelf (COTS) multicore platforms in hard real-time systems is their unpredictability, which originates from the sharing of different hardware resources, e.g., shared caches, interconnects, and the main memory. Specifically, the *main memory* has been identified as the main source of shared resource contention (see survey [15]). To solve this problem, a plethora of works have focused on analyzing the *memory contention* that can be suffered by tasks [4, 8–11, 19, 20]. Specifically, these works propose *white-box* modeling based solutions for the *Dynamic Random Access Memory* (DRAM), i.e., the solutions take into account the organization of DRAM and the low-level arbitration mechanism employed by the memory controller of DRAM.

It has been shown in the state-of-the-art that phased execution models such as the 3-phase task model [6, 16] enable precise bounds on the memory and bus contention suffered by tasks [1–4, 14, 17]. The 3-phase task model divides the task execution into three phases namely *Acquisition* (A), *Execution* (E), and *Restitution* (R). Specifically, the task first executes its A-phase to prefetch the data/code required by the task from the main memory and store it in the core's local memory (e.g., L1 or L2 cache). It then executes its E-phase by accessing the data/code that is already available in the core's local memory, without the need to access the main memory. Finally, the task writes the modified data back to the main memory during the R-phase. The 3-phase task model divides task execution into distinct *computation* and *memory phases* such that the shared memory is accessed by tasks only during their memory phases and the main memory is not accessed during the computation phase. As a consequence, it is possible to infer the specific time intervals in which memory accesses can happen, i.e., memory phases, and the time intervals in which specific memory operations can happen, i.e., read memory operations during the A-phase and write memory operations during the R-phase. Leveraging upon this, an existing work [4] focuses on analyzing the maximum memory contention that can be suffered by 3-phase tasks considering partitioned fixed-priority non-preemptive scheduling. Their memory contention analysis assumes that the system use *write batching* in which the memory controller prioritizes read requests over write requests. Write requests are then served by the memory controller in *batches* [5] to improve the turnaround time of the data bus [7], i.e., the shared bus which is responsible for the data transfer between the memory controller and memory banks. Even though the analysis presented in [4] provides an important solution, it has some limitations. The analysis in [4] pessimistically computes the memory contention that can be suffered by read requests due to the write requests. Specifically, the analysis in [4] assumes that either one batch of write requests is triggered upon the completion of each read request or the overall delay that can be suffered by the A-phase is given by the length of the write-buffer plus all R-phases of all jobs of all tasks that can be released on all other cores during the A-phase under analysis. This assumption is pessimistic because, in the 3-phase task model, an R-phase can only be issued by a core after the completion of an A-phase. In such a scenario, the actual number of R-phases that can be issued by a core depends on the number of A-phases that can be completed on that core during a given time window and not necessarily on the number of jobs released by tasks running on that core. Consequently, the bound on the total memory contention can be overestimated which can produce pessimistic bounds on the WCRT of tasks.

To address these issues, this work has the following ***contributions***.

1. We propose a memory contention analysis for 3-phase tasks by providing a tighter bound on memory contention that can be caused by write requests.

2. We also discuss the impact that *memory address mapping* and tighter bound on write requests can have on memory access times and memory contention of tasks.

2 SYSTEM MODEL

We assume a multicore platform comprising m identical cores ($\pi_1, \pi_2, \ldots, \pi_m$). The DRAM is shared among all the cores. Similarly to the existing work [4], we assume that the shared DRAM is accessed by cores via a set of crossbar switches that facilitates the point-to-point connection between each core and main memory. We assume that the shared cache is partitioned among cores such that each core has its non-overlapping partition. Furthermore, the local memory of each core is large enough to store all the data/code required by the task with the largest memory footprint that can execute on that core.

Task Model: We consider the 3-phase task model [6], in which the execution of each task is divided into A, E, and R-phases. Each phase as well as the complete task execute non-preemptively. We consider a task set Γ comprising n sporadic tasks ($\tau_1, \tau_2, \ldots \tau_n$) partitioned among cores at design time. T_i denotes the minimum inter-arrival time between two consecutive jobs of task τ_i, and D_i denotes its relative deadline. We assume that tasks have constrained deadlines, i.e., $D_i \leq T_i$. We assume that the maximum number of memory requests that can be issued during the A-phase (resp. R-phase) of task τ_i *in isolation* is denoted by MD_i^A (resp. MD_i^R). Similarly, the WCET of the E-phase of task τ_i is given by C_i^E. Note that the values of MD_i^A, MD_i^R, and C_i^E can be obtained by static analysis, measurement-based analysis, or by using the combination of both [18]. We assume that tasks are scheduled using fixed-priority non-preemptive scheduling with priorities assigned using any fixed-priority algorithm such as Rate Monotonic or Deadline Monotonic [12].

Throughout the paper, we refer to the core on which task τ_i (i.e., the task under analysis) executes as the *local core*, denoted by π_l. Similarly, any core other than the local core is referred to as a *remote core*, usually denoted by π_r. The set of all tasks mapped to a remote core π_r is denoted by Γ_r'.

Main Memory Model: We consider a DRAM as the main memory. We assume a single rank composed of multiple banks. Each bank is organized in rows and columns to store the data of tasks. Each bank has a *row buffer* that stores the data accessed during the most recent access to that bank. We assume that memory requests targeting each bank are enqueued in their respective *per-bank queues*. Each per-bank queue is then exposed to the *inter-bank scheduler* which is responsible to schedule the memory requests from all the per-bank queues. When a memory request targets a different row than the activated row of the bank, it results in a *row miss* and the memory request can be served by issuing the sequence of commands, *PRE*, i.e., to move back the current content of the row buffer to its corresponding row in the DRAM bank, *ACT*, i.e., to activate the requested row in the row buffer, and *CAS*, i.e., to perform the intended read/write operation on the activated row. On the contrary, when a memory request targets the same row as the activated row of the bank, it results in a *row hit* and the memory request can be served using the *CAS* command only. To formalize the properties of the considered memory controller, we will now define a set of *rules*.

R1: Each bank has its *per-bank queue* in which memory requests targeting respective banks are inserted. Each per-bank queue is scheduled using the *First-Ready First-Come-First-Serve* (FR-FCFS) policy which means 1) memory requests that result in a row-hit are prioritized over memory requests that result in a row-miss; 2) in case of a tie, older memory requests are prioritized over newer memory requests.

R2: We consider that banks are partitioned to cores such that each core has its set of banks [13, 20]. Specifically, A-phases of all tasks mapped on each core cannot access any bank assigned to another core. However, for the purpose of data sharing, the R-phases of all tasks in the system can access any bank.

R3: The inter-bank scheduling policy is *Round-Robin* (RR) which serves the memory requests from each per-bank queue with the granularity of one memory request, i.e., one memory request per bank in each turn. Furthermore, to avoid unbounded delay, we assume that the inter-bank scheduler cannot reorder requests [4].

R4: Unlike [4], we relax the assumption that each core issues at most one request per core by assuming that each core can issue multiple memory requests given that the core issue all outstanding memory requests in the correct order, i.e., in the sequence.

R5: We assume that reads have higher priority than writes since writes do not stall the processing pipeline. Write requests are enqueued in a write buffer of size Q_{write} and then served in batches with the watermarking mechanism [5] to improve the turnaround time of data bus [7]. Specifically, if there are pending read requests, the memory controller only starts serving write requests if the number of write requests are greater than or equal to the *watermarking threshold* W_{th} and serves at least one *batch* of write requests where the length of the batch is denoted by N_{Wb}. Similarly to [4], we assume that $W_{th} > Q_{write} - N_{Wb}$.

R6: For each task τ_i, we assume that $MD_i^A \geq MD_i^R$, i.e., each read request (A-phase memory request) can result in at most one write request (R-phase memory request).

3 PROPOSED MEMORY CONTENTION ANALYSIS FOR 3-PHASE TASKS

In this work, we consider two different memory address mappings.

1. Bank Level Mapping: In this mapping, we assume that all the memory blocks that can be requested by an A-phase are mapped to a single bank. We make no assumption about how the requested memory blocks are mapped within the bank.

2. Bank Level Contiguous Mapping: This mapping is similar to the above-mentioned mapping. Additionally, we assume that within the same bank, contiguous address mapping is used which means that subsequent memory requests of the A-phase are mapped to the subsequent columns of the same row. When a memory request is mapped to the last column of a row, the subsequent memory requests are mapped to the columns of another row of the same bank. We do not assume the specific row that will be accessed when switching to a different row of that bank. Contiguous address mapping is commonly used to improve the overall performance since mapping memory requests to the same row provides a better row-buffer locality.

3.1 Memory Contention Analysis for Bank Level Mapping

When analyzing the memory contention, tasks can suffer *intra-bank contention*, i.e., due to interfering memory requests targeting the same bank as the task under analysis, and *inter-bank contention*, i.e., due to interfering memory requests targeting a different bank than the task under analysis. We start by computing the maximum memory contention that can be suffered by *read requests* of the A-phase of task τ_i due to *read requests* of tasks running on all *remote cores*. Since banks are partitioned between cores (see rule R2), the A-phase of task τ_i can only suffer *inter-bank contention*, which is the contention suffered by a task when accessing the shared command and data buses that connects the memory controller to all memory banks.

Lemma 3.1. *The maximum number of read memory requests of all tasks running on all remote cores that can interfere with read memory requests of the A-phase of one job of task τ_i is upper bounded by N_i^{read}, where*

$$N_i^{read} = MD_i^A \times (m-1) \tag{1}$$

Proof Sketch: Due to fixed priority non-preemptive scheduling, there can be at most one A-phase active per core at a time as the E-phase of a task can only start after all read requests of its A-phase completes. Furthermore, due to the bank-level mapping, we know that all the read requests of an A-phase are mapped to a single bank. This implies that despite having multiple private banks per core, there can be at most one active bank per remote core at a time. Due to the RR inter-bank scheduling policy (see Rule R3), the inter-bank scheduler will serve one memory request per bank which means that each read request of task τ_i can be delayed by at most one read request per active bank. As there can be at most one active bank per core, the maximum number of interfering read memory requests of $m-1$ remote cores is upper bounded by $MD_i^A \times (m-1)$. □
Having bounded the number of interfering read requests, we bound the *maximum contention* that can be caused by those interfering requests to read requests of the A-phase of one job of task τ_i.

Lemma 3.2. *The maximum memory contention that can be suffered by read requests of the A-phase of one job of task τ_i due to read requests of tasks running on all remote cores is upper bounded by MC_i^{read}, where*

$$MC_i^{read} = MD_i^A \times \max_{N_{PRE}+N_{ACT}+N_{CAS}=m-1} \big(L^{PRE}(N_{PRE}) + L^{ACT}(N_{ACT}) + L^{CAS}(N_{CAS})\big) \tag{2}$$

Proof Sketch: In bank-level mapping, we do not assume how the A-phase is mapped within the bank. In the worst case, all memory requests may target different rows, thus, each memory request results in row-miss. Furthermore, from Lemma 3.1, we know that each read request of the A-phase of task τ_i can be delayed by $m-1$ read requests. It has been proven in Theorem 1 of [20] that the maximum inter-bank contention that can be suffered by a read request from N read requests is upper bounded by $\max\limits_{N_{PRE}+N_{ACT}+N_{CAS}=N} \big(L^{PRE}(N_{PRE}) + L^{ACT}(N_{ACT}) + L^{CAS}(N_{CAS})\big)$, i.e., the maximum inter-bank contention that can be suffered by a request at any of its commands PRE, ACT, and CAS. Extending this to all read requests of the A-phase of task τ_i, Equation 2 upper bounds the maximum inter-bank contention that can be suffered by the A-phase of one job of task τ_i. □

Having bounded the contention caused by read requests, the next step is to compute the maximum contention that can be caused by write requests to read requests of the A-phase of task τ_i. We start by briefly discussing how such a bound is derived in [4] and identify sources of pessimism. We then propose a new bound in Lemmas 3.3 and 3.4.

From Lemma 3 of [4] *The overall interference suffered by read requests of the A-phase of task τ_i due to write requests in any time interval of length t is bounded by*

$$MC_i^{wr}(t) = L_{WB}(\min(NR(t) \times N_{wb}, NW(t) + Q_{write})) \tag{3}$$

where $L_{WB}(N)$ is the maximum delay that can be caused by N write requests; $NR(t)$ is the sum of the maximum number of read requests that can be issued by the A-phase of task τ_i and all interfering read requests from all remote cores during t; N_{wb} is the number of requests that will be served in one batch; $NW(t)$ is all write requests that can be issued by all jobs of all tasks running on all remote cores during t; and Q_{write} is the length of the write buffer.

In Equation 3, $NR(t) \times N_{wb}$ specify that each read request of the task τ_i and each interfering read request from all remote cores suffer contention from one batch of write requests. This can be pessimistic since it assumes that every read request will suffer from one batch of write requests without analyzing the maximum number of batches that can be triggered during the execution of the A-phase of τ_i. Similarly, in Equation 3, the term $NW(t) + Q_{write}$ specify that all write requests of all jobs of all tasks released on all remote cores during t plus all previously enqueued write requests in the write buffer will cause memory contention. This is extremely pessimistic because in the 3-phase task model, a core can only issue an R-phase after the completion of an A-phase. In such a case, the actual number of write requests issued by a remote core depends on the number of read requests served on that remote core and not necessarily on all jobs released on that core during t. To accurately quantify the memory contention that can be caused by write requests, we need to determine the maximum number of write batches that can be triggered during the execution of the A-phase of τ_i. Building on these insights, we will now bound memory contention that can be caused by write requests as follows.

Lemma 3.3. *The maximum number of batches of write memory requests that can interfere with read requests of one job of the A-phase of τ_i is upper bounded by N_i^{wb}, where*

$$N_i^{wb} = 1 + \left\lceil \frac{\sum_{r=1,r\neq l}^{m} \max\limits_{\tau_u \in \Gamma_r'} \{MD_u^R\} + N_i^{read} - (W_{th} - (Q_{write} - N_{wb}))}{N_{wb}} \right\rceil \tag{4}$$

Proof. When read requests of A-phase arrive at the memory controller, in the worst case, the number of write requests inserted in the write buffer is equal to the length of the write buffer Q_{write}. This will trigger one batch of write requests as integrated into Equation 4. At this point in time, the maximum number of write requests inserted into the write buffer is equal to $Q_{write} - N_{wb}$. Now there can be a scenario in which a remote core just completed an E-phase and starts executing an R-phase. Considering this, we need to account for write requests that can be issued by one R-phase on that remote core. In the worst case, the remote core executes the R-phase that issued the largest number of write requests among the R-phases of all tasks running on that remote core, i.e., $\max\limits_{\tau_u \in \Gamma_r'} \{MD_u^R\}$. Extending this to all remote cores, $\sum_{r=1,r\neq l}^{m} \max\limits_{\tau_u \in \Gamma_r'} \{MD_u^R\}$ bounds the number of write requests that can be issued by R-phases of tasks that already completed their A-phases prior to the arrival of the A-phase of task τ_i. Note that to produce another R-phase on the same remote core, the core first needs to execute an A-phase.

From Lemma 3.1, we know that N_i^{read} bounds the maximum number of interfering read requests. Since the length of the R-phases is assumed to be less than or equal to their A-phases (see Rule R6), in the worst case, there can at most N_i^{read} number of write requests that can be issued by all remote cores. We do not need to account for write requests issued on the local core because 1) task τ_i will only issue R-phase after the completion of its A-phase; and 2) the R-phase of any other previously executed task on the local core must have already inserted all its write requests in the write buffer before the start of τ_i.

Finally, we subtract $(W_{th} - (Q_{write} - N_{wb})$ number of memory requests because after serving the first batch of write requests, the status of the write buffer must be $Q_{write} - N_{wb}$ (remember $W_{th} > Q_{write} - N_{Wb}$). Consequently, another batch of write requests can only be triggered if the watermarking threshold is reached, expressed as $(W_{th} - (Q_{write} - N_{wb})$. As we compute the number of batches, we need to divide the sum of all write requests issued during arrival to completion of the A-phase of task τ_i with the number of write requests per batch N_{wb}. To maximize the number of batches, we take the ceiling operation in Equation 4. □

The maximum *number* of write requests that can interfere with the A-phase of task τ_i is bounded by N_i^{write}, where

$$N_i^{write} = N_i^{wb} \times N_{wb} \tag{5}$$

Lemma 3.4. *The maximum memory contention that can be suffered by the A-phase of task τ_i due to write requests is upper bounded by MC_i^{write}, where*

$$MC_i^{write} = L_{WB}(N_i^{write}) \tag{6}$$

Proof Sketch: From Equation 2 of [8], the term $L_{WB}(N)$ upper bounds the maximum memory contention that can be caused by N write requests (served in batches) assuming that each write request will result in a row-miss and can potentially target any bank in the system. Extending this to N_i^{write} write requests, Equation 6 upper bounds the maximum memory contention that can be suffered by the A-phase of task τ_i due to write requests. □

Lemma 3.5. *The total memory contention that can be suffered by the A-phase of task τ_i is upper bounded by MC_i^{total}, where*

$$MC_i^{total} = MC_i^{read} + MC_i^{write} \tag{7}$$

Proof Sketch: We know that Equation 2 upper bounds the memory contention that can be caused by interfering read requests. Similarly, Equation 6 upper bounds the memory contention that can be caused by write requests. Consequently, Equation 7 upper bounds the *total memory contention* that can be suffered by the A-phase of task τ_i by taking the sum of Equations 2 and 6. □

As proven in [4], we do not need to account for memory contention that can be suffered by the R-phase of task τ_i. This is mainly because the write requests do not stall the processing pipeline, e.g., E-phase execution depends on the A-phase but not on the R-phase. As a consequence, we only need to ensure that all write requests of the R-phase arrive at the memory controller. Similarly to [4], we assume that the length of the write buffer is large enough such that all write requests of all cores can be inserted in it. This ensures that the R-phase of a task does not cause any additional delay to the A-phase of the subsequent task on the same core.

3.2 Memory Contention Analysis for Bank Level Contiguous Mapping

In the bank-level contiguous mapping, subsequent memory requests of an A-phase are mapped to the subsequent columns of the same row of the same bank. Upon a row switch, i.e., accessing the last column of a row, subsequent memory requests are mapped to subsequent columns of another row in the same bank. Due to such mapping, we can also compute the minimum number of read requests that will result in *row-hits*. A memory request resulting in a row hit can 1) reduce memory access times of requests as a row-hit request can only be served using the CAS command; and 2) suffer less inter-bank contention from interfering read requests as the row-hit request can only suffer contention at its CAS command. Despite having contiguous address mapping, it is extremely complex to bound the minimum number of row hits due to the write batching. We explain this using the following example.

Assume that all read requests of the A-phase of task τ_i are mapped to the single row of a bank. In such a scenario, ideally, there should be at most one row miss request and the remaining memory requests should result in a row hit. However, when the system use write batching and the worst-case is derived by assuming that one batch of write requests can be triggered upon serving each read request (as assumed in [4]), we cannot guarantee the minimum number of row hits. This is mainly because each time a batch of write requests is triggered, some or all write requests can target the same bank but a different row than task τ_i. Consequently, despite mapping all read requests to the same row, all memory requests may result in row-miss.

This problem has been highlighted by the state-of-the-art, see Section 3 of [9]. Thanks to Lemma 3.3, we know the maximum number of write batches that can be triggered during the A-phase of task τ_i. Using the bound provided by Lemma 3.3, we can tightly bound the maximum number of row miss requests by integrating the number of row hit requests that became row miss due to a batch of write requests. Due to space constraints, the bound on the maximum number of row miss and the total memory contention suffered by tasks are left as future work.

4 CONCLUSION

In this work, we propose the memory contention analysis for the 3-phase task model by leveraging memory address mapping of tasks. We provide a tighter bound on memory contention that can be caused by write requests. We also highlight how such a bound can be useful in improving the memory access times and memory contention suffered by tasks when using a contiguous address mapping scheme. In the future, we will formulate a detailed analysis for bank-level contiguous mapping. Furthermore, we can also improve the bank-level mapping analysis proposed in this work by directly bounding and integrating total memory contention that can be suffered by the task under analysis during its WCRT. For example, Lemma 3.3 can be further improved by considering the specific set of R-phases that can be released on remote cores during the WCRT of the task under analysis.

ACKNOWLEDGMENT

This work was financed by FCT and EU ECSEL JU within project ADACORSA (ECSEL/0010/2019 - JU grant nr. 876019) - The JU receives support from the EU's Horizon 2020 R&I Programme and Germany, Netherlands, Austria, France, Sweden, Cyprus, Greece, Lithuania, Portugal, Italy, Finland, Turkey (Disclaimer: This document reflects only the author's view and the Commission is not responsible for any use that may be made of the information it contains); it is also a result of the work developed under the CISTER Unit (UIDP/UIDB/04234/2020), financed by FCT/MCTES (Portuguese Foundation for Science and Technology); and under project POCI-01-0247-FEDER-045912 (FLOYD), financed in the scope of the CMU Portugal, by the European Regional Development Fund (ERDF) under COMPETE 2020, also by FCT under PhD grant 2020.09532.BD.

REFERENCES

[1] Jatin Arora, Cláudio Maia, Syed Aftab Rashid, Geoffrey Nelissen, and Eduardo Tovar. 2021. Bus-Contention Aware Schedulability Analysis for the 3-Phase Task Model with Partitioned Scheduling. In *29th International Conference on Real-Time Networks and Systems* (NANTES, France) *(RTNS'2021)*. Association for Computing Machinery, New York, NY, USA, 123–133. https://doi.org/10.1145/3453417.3453433

[2] Jatin Arora, Cláudio Maia, Syed Aftab Rashid, Geoffrey Nelissen, and Eduardo Tovar. 2022. Bus-contention aware WCRT analysis for the 3-phase task model considering a work-conserving bus arbitration scheme. *Journal of Systems Architecture* 122 (2022), 102345. https://doi.org/10.1016/j.sysarc.2021.102345

[3] Jatin Arora, Cláudio Maia, Syed Aftab Rashid, Geoffrey Nelissen, and Eduardo Tovar. 2022. Schedulability analysis for 3-phase tasks with partitioned fixed-priority scheduling. *Journal of Systems Architecture* 131 (2022), 102706. https://doi.org/10.1016/j.sysarc.2022.102706

[4] D. Casini, A. Biondi, G. Nelissen, and G. Buttazzo. 2020. A Holistic Memory Contention Analysis for Parallel Real-Time Tasks under Partitioned Scheduling. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 239–252. https://doi.org/10.1109/RTAS48715.2020.000-3

[5] Niladrish Chatterjee, Naveen Muralimanohar, Rajeev Balasubramonian, Al Davis, and Norman P. Jouppi. 2012. Staged Reads: Mitigating the impact of DRAM writes on DRAM reads. In *IEEE International Symposium on High-Performance Comp Architecture*. 1–12. https://doi.org/10.1109/HPCA.2012.6168943

[6] Guy Durrieu, Madeleine Faugère, Sylvain Girbal, Daniel Gracia Pérez, Claire Pagetti, and W. Puffitsch. 2014. Predictable Flight Management System Implementation on a Multicore Processor. In *Embedded Real Time Software (ERTS'14)*. TOULOUSE, France. https://hal.archives-ouvertes.fr/hal-01121700

[7] Leonardo Ecco and Rolf Ernst. 2017. Tackling the Bus Turnaround Overhead in Real-Time SDRAM Controllers. *IEEE Trans. Comput.* 66, 11 (2017), 1961–1974. https://doi.org/10.1109/TC.2017.2714672

[8] Mohamed Hassan and Rodolfo Pellizzoni. 2018. Bounding DRAM Interference in COTS Heterogeneous MPSoCs for Mixed Criticality Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2323–2336. https://doi.org/10.1109/TCAD.2018.2857379

[9] Mohamed Hassan and Rodolfo Pellizzoni. 2020. Analysis of Memory-Contention in Heterogeneous COTS MPSoCs. In *ECRTS 2020 (LIPIcs, Vol. 165)*. Dagstuhl, Germany, 23:1–23:24. https://doi.org/10.4230/LIPIcs.ECRTS.2020.23

[10] Hyoseung Kim, Dionisio de Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. 2014. Bounding memory interference delay in COTS-based multi-core systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 145–154. https://doi.org/10.1109/RTAS.2014.6925998

[11] Hyoseung Kim, Dionisio de Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. 2016. Bounding and reducing memory interference in COTS-based multi-core systems. *Real-Time Systems* 52 (05 2016). https://doi.org/10.1007/s11241-016-9248-1

[12] C. L. Liu and James W. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM* 20, 1 (jan 1973), 46–61. https://doi.org/10.1145/321738.321743

[13] Lei Liu, Zehan Cui, Mingjie Xing, Yungang Bao, Mingyu Chen, and Chengyong Wu. 2012. A software memory partition approach for eliminating bank-level interference in multicore systems. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 367–375.

[14] Claudio Maia, Geoffrey Nelissen, Luis Nogueira, Luis Miguel Pinho, and Daniel Gracia Perez. 2017. Schedulability analysis for global fixed-priority scheduling of the 3-phase task model. In *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, Hsinchu, Taiwan, 1–10. https://doi.org/10.1109/RTCSA.2017.8046313

[15] Claire Maiza, Hamza Rihani, Juan M. Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I. Davis. 2019. A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems. *Comput. Surveys* 52, 3 (June 2019), 1–38. https://doi.org/10.1145/3323212

[16] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. 2011. A Predictable Execution Model for COTS-Based Embedded Systems. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, Chicago, IL, USA, 269–279. https://doi.org/10.1109/RTAS.2011.33

[17] Thilanka Thilakasiri and Matthias Becker. 2023. An Exact Schedulability Analysis for Global Fixed-Priority Scheduling of the AER Task Model. In *Proceedings of the 28th Asia and South Pacific Design Automation Conference* (Tokyo, Japan) *(ASPDAC '23)*. Association for Computing Machinery, New York, NY, USA, 326–332. https://doi.org/10.1145/3566097.3567850

[18] Reinhard Wilhelm, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, Per Stenström, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, and Reinhold Heckmann. 2008. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems* 7, 3 (April 2008), 1–53. https://doi.org/10.1145/1347375.1347389

[19] Zheng Wu, Rodolfo Pellizzoni, and Danlu Guo. 2016. A composable worst case latency analysis for multi-rank DRAM devices under open row policy. *Real-Time Systems* 52 (11 2016). https://doi.org/10.1007/s11241-016-9253-4

[20] Heechul Yun, Rodolfo Pellizzon, and Prathap Kumar Valsan. 2015. Parallelism-Aware Memory Interference Delay Analysis for COTS Multicore Systems. In *2015 27th Euromicro Conference on Real-Time Systems*. 184–195. https://doi.org/10.1109/ECRTS.2015.24