Arbitration-Induced Preemption Delays

Farouk Hebbache, Florian Brandner, Mathieu Jan, Laurent Pautet

▶ To cite this version:

Farouk Hebbache, Florian Brandner, Mathieu Jan, Laurent Pautet. Arbitration-Induced Preemption Delays. 31st Euromicro Conference on Real-Time Systems (ECRTS 2019), Jul 2019, Stuttgart, Germany. 10.4230/LIPIcs.ECRTS.2019.19 . hal-02447339

HAL Id: hal-02447339
https://hal.science/hal-02447339

Submitted on 21 Jan 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Arbitration-Induced Preemption Delays

Farouk Hebbache
CEA, List, 91191 Gif-sur-Yvette, France
farouk.hebbache@cea.fr

Florian Brandner
LTCI, Télécom ParisTech, Université Paris-Saclay, France
florian.brandner@telecom-paristech.fr

Mathieu Jan
CEA, List, 91191 Gif-sur-Yvette, France
mathieu.jan@cea.fr

Laurent Pautet
LTCI, Télécom ParisTech, Université Paris-Saclay, France
laurent.pautet@telecom-paristech.fr

Abstract

The interactions among concurrent tasks pose a challenge in the design of real-time multi-core systems, where blocking delays that tasks may experience while accessing shared memory have to be taken into consideration. Various memory arbitration schemes have been devised that address these issues, by providing trade-offs between predictability, average-case performance, and analyzability. Time-Division Multiplexing (TDM) is a well-known arbitration scheme due to its simplicity and analyzability. However, it suffers from low resource utilization due to its non-work-conserving nature. We proposed in our recent work dynamic schemes based on TDM, showing work-conserving behavior in practice, while retaining the guarantees of TDM. These approaches have only been evaluated in a restricted setting. Their applicability in a preemptive setting appears problematic, since they may induce long memory blocking times depending on execution history. These blocking delays may induce significant jitter and consequently increase the tasks' response times.

This work explores means to manage and, finally, bound these blocking delays. Three different schemes are explored and compared with regard to their analyzability, impact on response-time analysis, implementation complexity, and runtime behavior. Experiments show that the various approaches behave virtually identically at runtime. This allows to retain the approach combining low implementation complexity with analyzability.

2012 ACM Subject Classification Computer systems organization → Real-time system architecture

Keywords and phrases Dynamic Time-Division Multiplexing, Predictable Computing, Multi-Criticality, Preemption

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.19

1 Introduction

Multi-core architectures pose many challenges in real-time systems, which arise from the manifold interactions between concurrent tasks during their execution – most notably accesses to shared main memory. These interactions make it difficult to tightly bound the Worst-Case Execution Time (WCET) of real-time tasks. Systematically considering the worst-case behavior of an arbitration policy with regard to memory accesses in the presence of concurrent requests is too pessimistic, as it leads to low resource utilization at runtime. Considering Mixed-Criticality (MC) systems is one approach to increase resource utilization. In an MC system, tasks with different levels of criticality, and even non-critical tasks, execute on the same multi-core architecture. The non-critical tasks in such a system may then exploit resource budgets that have not been fully consumed by critical tasks – which rarely fully

© Farouk Hebbache, Florian Brandner, Mathieu Jan, and Laurent Pautet;
licensed under Creative Commons License CC-BY
31st Euromicro Conference on Real-Time Systems (ECRTS 2019).
Editor: Sophie Quinton; Article No. 19; pp. 19:1–19:22

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

exploit the worst-case budgets reserved for them. However, most MC systems take decisions only at the level of task scheduling. Our aim here is, hence, to apply similar ideas to the arbitration of memory requests in order to improve memory utilization.

Amongst the memory arbitration policies, Time-Division Multiplexing (`TDM`) divides time into slots and allocates them to cores to exclusively access memory. Using `TDM`, the accesses within a slot no longer depend on whether concurrent requests exist or not. `TDM` provides predictable behavior and improves composability by bounding access latencies and guaranteeing bandwidth independently from other cores. However, the access latency of a memory request when using `TDM` now depends on the scheduling of these time slots, even if they are unused. Such unused slots appear when the owner of a `TDM` slot does not (yet) have a memory request ready to be served. Under a strict `TDM` scheme, these unused slots cannot be reclaimed by another task. This *non-work-conserving* behavior of `TDM` often leads to low resource utilization. This problem is further amplified as the number of cores increases, leading to longer `TDM` schedules. Another source of `TDM` pessimism stems from the length of `TDM` slots, expressed in clock cycles, which have to be longer than the worst-case latency for handling memory requests. The arbiter consequently has to wait for the beginning of the next `TDM` slot, even when memory requests complete earlier than this worst-case latency. Resulting in low average-case performance and poor memory utilization.

To overcome these limitations, we recently explored novel dynamic `TDM`-based arbitration schemes [13]. In that work, we envisioned that the task's criticality level should not only be used by the task scheduler, but also by the memory arbiter. The arbiter associates a deadline with each memory request of a critical task, which corresponds to the end of its corresponding slot under a strict `TDM` scheme. The deadline then allows to compute the *slack time* of a critical task, i.e., the amount of time that the task's last request completed before the request's deadline. This slack time then can be exploited by the arbiter, under certain conditions, to favor requests of non-critical tasks over requests from critical tasks, to freely reorder memory requests, and to schedule memory requests at the granularity of clock cycles – instead of being confined to `TDM` slots. The resulting arbiter significantly reduces the memory idle time, compared to a regular `TDM` arbiter. The improvements go up to a factor of 350 and even remain above a factor of 50 under high memory load [13].

However, the proposed dynamic `TDM`-based arbitration techniques face issues under a preemptive execution model. In this paper, we define two memory delays induced by preemptions, the *memory blocking delay* and the *misalignment delay*, which may lead to significant jitter and increase task response times. Even worse, due to non-critical tasks, the memory blocking delay may be unbounded in some circumstances. We explore three different approaches to analyze the impact of these arbitration-induced preemption delays considering preemptive [18] (`SHDp`) and non-preemptive [1] (`SHDw`) memory requests. Finally, we propose a new technique (`SHDi`) to resolve these issues by adapting (priority or rather) *criticality inheritance* known from scheduling theory. This allows us to manage and easily bound these preemption delays. Our evaluation shows that our new approach successfully limits the worst-case preemption delays experienced at runtime under our dynamic `TDM`-based arbitration schemes. At the same time we see virtually no impact on average-case performance and success rate. Note, in addition, that the proposed technique is not limited to the dynamic `TDM`-based arbitration schemes and is also applicable to other arbitration techniques, e.g., arbitration based on fixed priorities [1].

The remainder of this paper is organized as follows. Section 2 describes the considered system model. In Section 3, we briefly review the dynamic `TDM`-based arbitration schemes. Section 4 identifies the different delays caused by preemptions in dynamic `TDM`-based arbitra-

tion strategies, while Section 5 proposes three preemption models to handle these delays. In Section 6, we evaluate our contributions and demonstrate the improvements in terms of blocking delay reductions and success rates on randomly generated task sets. Section 7 presents related work, before concluding in Section 8.

2 System Architecture

2.1 Task Model

In our previous work [12, 13] on `TDM`-based dynamic memory arbitration, we assumed a restricted task model, where each core executes a single independent and periodic task without preemption. Here, we relax these assumptions to allow multiple tasks to be scheduled on each core under a fixed-priority preemptive scheduler. We assume a finite task set Γ consisting of independent and periodic tasks. Each task $\tau_i \in \Gamma$ is assigned a fixed priority[1] i and is characterized by the tuple $\tau_i = (C_i, T_i, D_i, M_i)$, representing the task's WCET, its period, its implicit deadline, and finally its worst-case number of memory requests respectively. Each task generates an infinite sequence of jobs at runtime, which, in turn, generate sequences of memory requests $\tau_{i,j}$ where $j \leq M_i$. Requests are separated by a dynamic number of processor clock cycles (see Section 3). This *distance* represents the amount of computation performed between the completion of request $\tau_{i,j}$ and the issuing of the consecutive request $\tau_{i,j+1}$ or the job's termination. This allows us to model any *dynamic* job execution (even input-dependent) considering a deterministic hardware platform (see below).

Ultimately we aim at mixed-criticality systems, where tasks can be associated with multiple WCETs depending on the system's execution mode (criticality). This work focuses on the impact of the memory arbitration scheme on such MC systems, more precisely the interplay between dynamic TDM-based arbitration and task preemptions. We consider two classes of tasks: *critical* tasks τ_i^c and *non-critical* tasks τ_i^{nc}. However, we currently only consider a single C_i value per task in our task model, i.e, we assume a multi-criticality system. Our experiments are based on simulations to evaluate memory arbitration under realistic circumstances. We here assume that the WCETs (C_i) and deadlines (D_i) of critical tasks are firm and have to be respected under all circumstances. An execution thus fails if a critical task misses its deadline. Non-critical tasks, on the other hand, are executed in a best-effort manner by the underlying computer platform and memory arbitration scheme. During execution, they may exceed their WCET budget, potentially causing deadline misses – for themselves or other (critical) tasks. Handling timing failure events, in particular switching critical tasks to a different behavior as in MC scheduling [5], is out of the scope of this work. It will be addressed in future work. For the formal analysis, based on response-time analysis (RTA) [2], we simply require (for now) *firm* WCETs and deadlines for non-critical tasks in order to bound the response times of critical tasks.

2.2 Hardware Architecture

We assume a hardware platform consisting of m cores connected via a central arbiter to a shared main memory. The memory requests dynamically generated by the jobs at runtime thus compete for this shared resource. We assume that each core is equipped with internal caches. Memory requests thus represent transfers of cache blocks resulting from cache misses.

[1] A larger task index indicates higher priority.

In order to ensure that the aforementioned *distances* between requests are independent from the execution of other tasks, we assume composable compute cores [9] and the absence of any external events that may interfere with the execution of a core. The interference between the independent tasks consequently stems from accesses to the shared memory only and, in particular, depends on the employed memory arbiter.

For simplicity, we assume that all cores, the memory bus/arbiter, and the memory itself operate at the same clock speed. We thus generally refer to time in *clock cycles*.

2.3 Scheduling Policy

On multi-core platforms, task scheduling can be performed globally among all/a subset of the available m cores [16] or in a partitioned manner [3, 4]. Partitioned scheduling statically assigns each task onto a fixed core, while global scheduling allows tasks to migrate among cores dynamically. In principle both of these scheduling policies could be combined with the approaches proposed here. For brevity, we limit our discussion on *partitioned scheduling* using *fixed priorities*.

This enables the scheduler on each core to determine at any moment in time and *in advance*, which task will need to be activated next on its core. Instead of triggering preemptions periodically, we assume that the scheduler programs a hardware component that signals the need to preempt the currently running task to both the core and the memory arbiter. This ensures that preemption handling does not interfere with the execution on a given core – up until to the moment when a preemption is triggered. We assume that each core is equipped with such a component, separately tracking the next upcoming preemptions stemming from a non-critical or a critical task. This also ensures that the preemption mechanism does not interfere with the computation (*distance*) between memory accesses.

3 Background

This section provides a brief introduction to our recent work on TDM-based dynamic memory arbitration as well as well-established iterative techniques for response-time analysis.

3.1 TDM-Based Dynamic Memory Arbitration

TDM-based arbitration is popular due to its simplicity and predictability, despite the fact that strictly applying TDM often results in under-utilization of resources. Our recent work on dynamic TDM-based memory arbitration schemes [12, 13] addresses resource under-utilization by (a) introducing two classes of memory requests (critical and non-critical) and (b) by exploiting slack to allow for more dynamic scheduling decisions as long as the deadlines of critical requests can be met safely.

The key idea of the *dynamic TDM with slack counters* (TDMds) arbitration scheme [12] is to interpret TDM scheduling as driven by deadlines. Under strict TDM each request completes precisely at the end of the request owner's next TDM slot, which can be seen as a deadline. The deadlines of TDMds similarly correspond to the end of TDM slots. However, instead of systematically delaying requests until their respective deadlines, requests are processed dynamically in any order – as long as deadlines are met. This allows to compute the *slack time* of a critical task, i.e., by how much the task's last request completed earlier w.r.t. the request's deadline. This slack is stored in dedicated counters and allows to prioritize non-critical requests, i.e., spend the slack of a critical task in favor of a non-critical task. Note, however, that slack accumulated within one job of a task naturally is not preserved for

subsequent jobs. Slack counters are consequently reset at job start. Also note that deadlines are ensured for critical requests only, while non-critical requests are processed in a best-effort manner. The dynamic processing allows non-critical tasks, for instance, to reclaim otherwise unused `TDM` slots.

A critical request may arrive earlier than expected under a strict `TDM` scheme if the request's owner accumulated some slack before. The request's deadline would then also appear earlier than under strict `TDM` if one would simply consider the request's issue date to compute the deadline, i.e., the end of the next `TDM` slot after the issue date. Under `TDMds` the deadline of a new critical request is instead computed from a *delayed issue date*, which is derived by adding the previously accumulated slack to the request's original issue date. Given enough slack this may push the request's deadline into the future, i.e., past the next `TDM` slot, and provide additional freedom to the arbiter. This also ensures that the deadlines under `TDMds` **exactly** correspond to the deadlines/completion dates that would be observed under strict `TDM`. This holds for **any** dynamic execution of **any** program [13].

The `TDMds` arbiter keeps critical requests in a priority queue (ordered by increasing deadlines) and non-critical requests in a simple FIFO. At the beginning of each `TDM` slot the arbiter schedules the top-most request from one of these two queues in order to be processed by the shared memory. The arbiter prioritizes non-critical requests over critical requests, as long as the deadline of the top-most critical request is *not* at the end of the current `TDM` slot. This ensures that non-critical requests can quickly access memory as long as critical requests have enough slack, while also guaranteeing that deadlines of critical requests are met.

An extension of `TDMds`, called *dynamic TDM with early release* (`TDMer`) even goes a step further by performing arbitration at the level of (bus) clock cycles instead of `TDM` slots [13]. The key insight is that memory can begin the processing of a request at any moment (even in the middle of a slot), if the arbiter can ensure that subsequent critical requests do not miss their deadlines. Two cases can be distinguished. Firstly, consider that memory is idle during `TDM` slot i, while a request by the owner of the upcoming `TDM` slot $i+1$ is pending. This request can immediately be processed, as the `TDM` slot i is unused and the request is guaranteed to finish before its deadline, which may not lie before the end of `TDM` slot $i+1$. The processing cannot interfere with requests of other slots (e.g., $i+2$). A similar argument can be built for arbitrary pending requests, if the owner of the upcoming `TDM` slot has enough slack (e.g., more than a slot length). In this case, the processing of another request can start in the middle of the unused `TDM` slot i and complete in slot $i+1$. This may, in the worst-case, delay a request of the owner of slot $i+1$. However, the deadline of the delayed request, under all circumstances, falls into the next `TDM` period, due the use of the *delayed issue date* to derive its deadline. Exploiting these two cases, `TDMer` was shown to be work-conserving even under high memory load, when every job of critical tasks is granted an initial slack counter value of a single slot length [13] (not shown in the following example).

Figure 1 shows an execution of a task set under `TDMer`, considering two critical tasks (τ_0^c, τ_1^c) and a non-critical task (τ_2^{nc}). All tasks execute on separate cores, which issue critical and non-critical requests on behalf of the respective tasks. Only critical tasks have dedicated `TDM` slots (`C0`, `C1`) that take 8 clock cycles each and 16 clock cycles for both, corresponding to the `TDM` slot length Sl and the `TDM` period P respectively. The non-critical task thus may only access memory during unused slots or when the slack of critical tasks permits. The visualization of a request includes the request's *issue date* (↰) and *deadline* (⋯]). The processing of a request by the main memory is indicated using a solid green hatched bar (▨), whose right edge indicates the request's *completion date*. The memory is not always busy, memory idling is indicated by a red hatched bar (▨). The deadline may lie after the

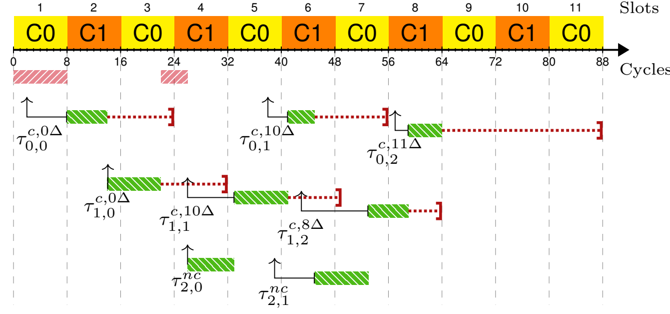


Figure 1 Dynamic arbitration using TDMer of critical tasks τ_0^c and τ_1^c and non-critical task τ_2^{nc}.

request's completion date, which then generates slack for the task issuing the request. The value of the slack counter when issuing a request is displayed as a superscript. For instance, task τ_0^c accumulated 11 cycles of slack due to the early completion of requests $\tau_{0,0}^c$ and $\tau_{0,1}^c$. When issuing request $\tau_{0,2}^c$ this slack (superscript 11Δ) pushes the request's deadline beyond the next TDM slot owned by τ_0^c (beyond slot 9 to 11). The arbiter is no longer tied to TDM slots and can choose one of the issued requests, independently from the actual owner of the slot and the alignment with the TDM schedule – given enough slack. This is illustrated by the non-critical request $\tau_{2,1}^{nc}$, which is issued in TDM slot 5. The request first has to wait for the ongoing request $\tau_{1,1}^c$ to complete. Then, $\tau_{0,1}^c$ is prioritized, since τ_0^c is the owner of the upcoming TDM slot 7. At the moment when the processing of $\tau_{2,1}^{nc}$ starts, τ_0^c is the owner of the upcoming slot 7, but has no pending request. However, any request issued by τ_0^c is known to have a deadline that lies beyond slot 7, due to its slack counter value of 11. It is thus safe to begin the processing of $\tau_{2,1}^{nc}$. We can also see that the non-critical request is prioritized over the pending critical request $\tau_{1,2}^c$, as τ_1^c has accumulated 8 cycles of slack. Note, that the slack counters are all reset for subsequent job instances of the critical tasks τ_0^c and τ_1^c.

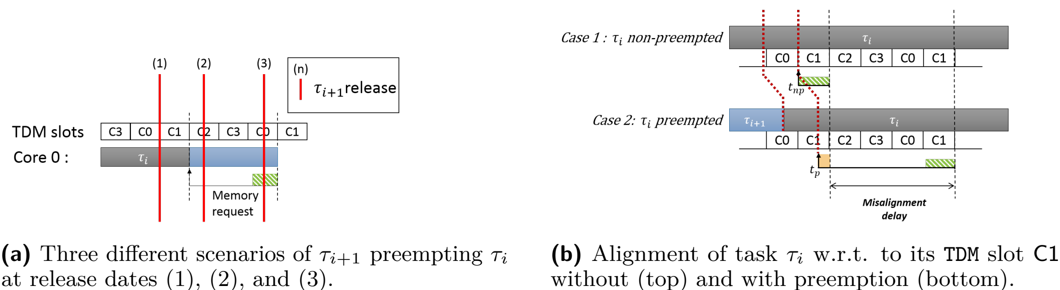
3.2 Worst-Case Response Time Analysis

We assume that schedulability is verified using a variant of *Response-Time Analysis* (RTA) [2] based on the usual recurrence equations:

$$R_i^{n+1} = C_i + B_i + \sum_{\forall j \in \text{hp}(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j \tag{1}$$

The recurrence equations are initialized to $R_i^0 = 0$ and then iteratively reevaluated until a fixed point is reached. R_i then indicates the response time of task i, having a WCET bound C_i (see Section 2). In addition, the impact of preemptions by tasks with higher priority than i ($j \in \text{hp}(i)$) is considered via their WCET bounds C_j and periods T_j. B_i indicates an upper bound on the time task i may be blocked (e.g., by semaphores). However, we will ignore this part of the equation in the remainder of this paper and instead consider additional delays potentially caused by the memory arbitration policy.

The dynamic memory arbitration approaches presented above may induce additional preemption delays and *jitter* that need to be considered during schedulability analysis in order to be certain that the tasks actually meet their deadlines safely. Both issues are caused by the way memory requests are processed as explained in the following section.



(a) Three different scenarios of τ_{i+1} preempting τ_i at release dates (1), (2), and (3).

(b) Alignment of task τ_i w.r.t. to its `TDM` slot `C1` without (top) and with preemption (bottom).

Figure 2 Preemption effects w.r.t. `TDM` memory arbitration.

4 Arbitration-Induced Preemption Delays

We now investigate the issues raised by TDM-based arbitration in general and our dynamic schemes [12, 13] in particular, considering the preemptive system model from Section 2. For brevity, we will focus on preemption costs caused by the arbitration policy and ignore other costs due to the scheduler invocation, context switching, pipeline flushes, or Cache-Related Preemption Delays (CRPD). From here on, we use the term *preemption cost* to refer to the costs related to the *arbitration policy only*.

4.1 Preemption Costs for strict TDM Arbitration

Subfigure 2a depicts three preemption scenarios of a task τ_{i+1} that preempts a lower-priority task τ_i at different release dates (red vertical lines). The first case (1) refers to a preemption occurring while the CPU performs computations (gray area). The task then can be preempted right away (ignoring potential pipeline stalls). The second case (2) refers to a situation where the preemption occurs while the CPU stalls, waiting to access the shared memory. While it would be possible to abort the pending memory request and immediately preempt τ_i [18], this requires modifications to the processor pipeline. In the last case (3), the preemption occurs while the memory actually processes a memory request. It would theoretically be possible to also abort the request at this stage – requiring modifications to the processor pipeline and throughout the entire memory hierarchy. A simpler alternative for cases (2) and (3), both in terms of hardware and timing analysis, would be to avoid aborting the request and simply wait for its completion [1].

Clearly, all three cases may induce preemption-related delays that need to be bounded and taken into consideration by the schedulability test (in addition to classical CRPD). The worst case delay experienced for case (1) is *trivial* and only depends on the characteristics of the processor pipeline. Case (3) similarly is analyzable and can be bounded by the worst-case memory latency, e.g., a `TDM` slot length [1]. The analysis of case (2) is more complex, since the behavior potentially depends on the other tasks in the system and the memory arbitration policy. We first analyze the timing behavior for strict `TDM` and later extend this analysis to the dynamic TDM-based approaches.

▶ **Definition 1.** *Under a system with fixed-priority preemptive scheduling, the* ***memory blocking delay*** *(MB) denotes the number of clock cycles that a higher-priority task τ_h is blocked from executing on its core after its release, due to a pending memory request by a lower-priority task τ_l ($l < h$).*

However, the blocking time does not cover all preemption-related delays of strict TDM. Recent work [14, 19] proposed sophisticated WCET analyses, which exploit the relative alignment of the program execution with regard to the TDM schedule, without preemption. A preemption may impact the program's relative alignment – unless task scheduling itself is aligned with the TDM period.

▶ **Definition 2.** *The* ***misalignment delay*** *(MA) denotes the number of additional clock cycles that the first memory access of a task takes, w.r.t. the worst case considered by the WCET analysis, when resuming after a preemption.*

Subfigure 2b depicts such a misalignment delay under strict TDM for a request of a critical task τ_i, owning TDM slot C1. Case 1 (top) illustrates an execution without preemption, where a request is issued right at the beginning of the task's TDM slot at time instant t_{np}. The program's alignment w.r.t. the TDM schedule is ideal and the request is processed immediately. The second case (bottom) shows the *same* execution of τ_i after a preemption by τ_{i+1}. In the absence of other side-effects, such as CRPDs, the same computations are performed by τ_i up to its first memory request (as indicted by the red dotted lines), which now is issued at time instant t_p. However, the task's alignment w.r.t. the TDM schedule was slightly shifted due to the preemption. The request thus has to wait longer than expected by the WCET analysis, which assumed an execution without preemption.

4.2 Preemption Costs for Dynamic TDM-based Arbitration

The dynamic TDM-based arbitration schemes [12, 13] inherit the memory blocking and misalignment delays from strict TDM. Figure 3 shows 3 tasks executing on 2 cores that share slots C0 and C1 within a TDM period under the TDMds arbitration scheme. The mapping between tasks and cores is indicated by matching colors (yellow ▬ and orange ▬). Critical tasks τ_0^c and τ_1^c are executed on the same core, which results in a preemption of task τ_0^c by τ_1^c (▬). Non-critical task τ_2^{nc} executes alone on the other core. A core has a dedicated TDM slot when it executes a critical task (τ_1^c, τ_0^c), while the slots of cores executing non-critical tasks (here only τ_2^{nc}) are *shared* by the running non-critical tasks (see the next section).

Lets assume that tasks cannot be preempted while performing a memory access [1]. Task τ_0^c then blocks the higher-priority task τ_1^c after its release in TDM slot 4. The preemption's blocking time, highlighted by the blue cross-hatched bar (▬), appears to be larger than the worst-case memory access latency under strict TDM. The latency of request $\tau_{0,2}^c$ amounts to two entire TDM periods and τ_1^c starts executing only after its completion. The reason for this long delay is the high slack counter value ($\Delta 16$) for request $\tau_{0,2}^c$, combined with the interference of the non-critical task τ_2^{nc} running on the other core. The non-critical task

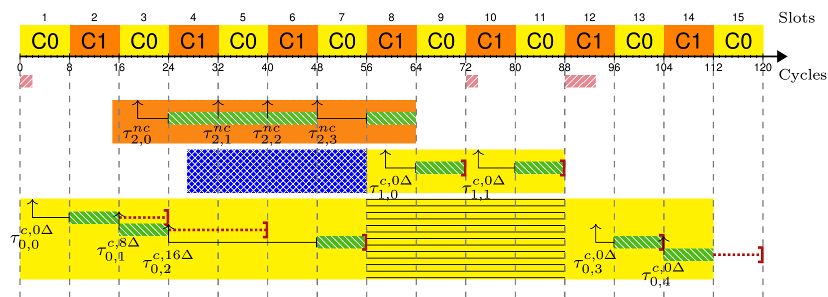


Figure 3 Impact of slack accumulation on the memory blocking time for TDMds.

consumes all the slack of the critical request $\tau^c_{0,2}$, delaying its completion, and thus also delaying the preemption. The situation is potentially even worse when a non-critical task is preempted during a memory request. Recall that non-critical requests are executed in a best-effort manner. The blocking time caused by such a request may even be unbounded.

The example illustrates that preemption-induced delays need to be taken into consideration during system analysis. Note that the same reasoning (see Section 4.1) used for strict `TDM` to establish the misalignment delay bound applies for all dynamic `TDM`-based approaches considered here. Next, we investigate three different approaches to integrate the memory blocking delay due to preemptions for our dynamic `TDM`-based arbitration schemes by extending response-time analysis. In addition, we compare the approaches w.r.t. their implementation complexity and actual runtime behavior.

5 Arbitration-Aware Preemption Techniques

Let us assume that memory requests are never aborted – cf. cases (2) and (3) from Subfigure 2a. In this case, preemptions have to be delayed until a potentially ongoing request completes. Under strict `TDM` arbitration, this corresponds to the usual worst-case memory access latency w.r.t. the `TDM` period P and `TDM` slot length Sl, which can be bounded by [1]:

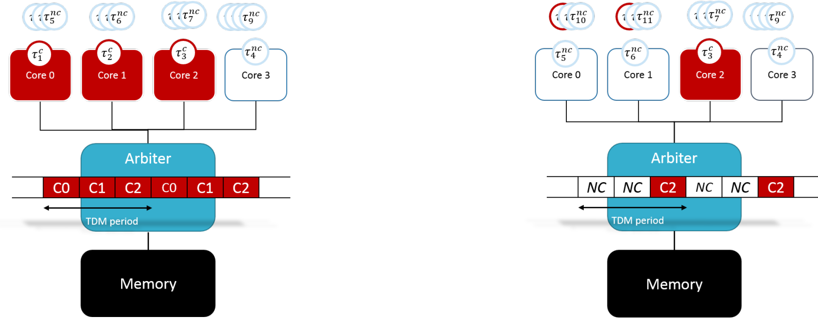
$$\mathrm{MB}^{\texttt{TDM}} = P + Sl - 1 \tag{2}$$

As demonstrated in Section 4, this bound is not valid for `TDMds` and `TDMer`, due to additional delays caused by the slack counters. While it certainly appears feasible to determine bounds on the slack counter values of tasks, it can be expected that these bounds would be rather pessimistic. In this section, we first explore this option to simply *wait* [1] due to its uncomplicated hardware implementation and later refer to it as `SHDw` (for *ScHeDuling wait*). Next, we explore an alternative option (`SHDp`, for *ScHeDuling preempt*), which assumes that ongoing requests can be *preempted* as long as they are not yet processed by the main memory [18] – cf. case (2) from Subfigure 2a. Finally, we propose a new solution (`SHDi`, for *ScHeDuling inheritance*) that tries to limit the impact of the slack counters by imposing a (new) deadline on an ongoing request when the core's timer signals a preemption by a critical task, i.e., the request of the preempted task *inherits* the criticality of the preempting task. However, before discussing these preemption models, we first need to refine the architecture model considering the fact that the cores can be shared by the tasks. Finally, we show how these models can be integrated into a response time analysis.

5.1 `TDM` Schedule and Preemption

We assume a multi-core platform with m cores and partitioned fixed-priority scheduling on each core. Critical and non-critical tasks may reside on the same core – without any restrictions on priority assignment. Notably, non-critical tasks may have a higher priority than critical tasks.

This raises the question on how `TDM` slots are assigned among cores/tasks. We propose a pragmatic solution for this work, but other alternatives are obviously possible. Since each core that executes at least one critical task may require a `TDM` slot at some moment, we assign one `TDM` slot to each such core. However, the slot is only reserved exclusively for that core when it actually executes a critical task, i.e., the core is considered critical. Cores that do not execute a critical task at a specific moment are themselves non-critical. The `TDM` slots that are not reserved by critical cores are marked as *NC* and shared among all running tasks on all cores in the system. This strategy is illustrated by Figure 4, showing a system with



(a) All TDM slots are reserved by critical cores. **(b)** One reserved TDM slot, two are shared (*NC*).

Figure 4 Hardware architecture.

4 cores. The TDM schedule consists of three slots (C0, C1, and C2), since only three of the cores may execute critical tasks (red). In Subfigure 4a, these three cores execute critical tasks and are thus themselves critical. In Subfigure 4b, on the other hand, only a single critical task executes. Consequently two of the TDM slots are marked *NC* and shared by all running tasks. Note, that non-critical tasks on core 3 may suffer from starvation on the depicted platform. For the formal analysis we require that at least one TDM slot is marked *NC* whenever a non-critical task executes in order to rule starvation out. This can be seen as a form of hierarchical arbitration similar to the work by Paolieri et al. [18] on Round-Robin. We thus define a larger TDM period for non-critical tasks using an application-specific constant k (e.g., k represents the number of non-critical cores divided by the number of *NC* slots when applying FIFO arbitration among non-critical requests): $P^{nc} = k \cdot P$.

Also, recall that we assume that preemptions are pre-programmed through a timer-like hardware component (see section 2.3). Using these components, we can control under which conditions preemptions are actually triggered, e.g., to block the current preemption until an ongoing memory request has completed or to preempt a pending request, et cetera.

5.2 Scheduling with Request Waiting (SHDw)

This strategy simply waits that an on-going memory request finishes [1]. Compared to strict TDM, the memory blocking time can however be considerably larger due to the slack counters. Consequently, a timing analysis technique is required that allows to bound the maximum slack counter value of a critical task τ_i^c. A trivial bound Δ_i^{max} can be computed by multiplying a task's worst-case number of memory requests (M_i) with the maximum slack possibly accumulated per request: $M_i \cdot (P - Sl)$ (TDMds) or $M_i \cdot (P + Sl - 1 - l)$ (TDMer), where l indicates the minimum memory latency. The resulting bound appears highly pessimistic, but more precise bounds are out of the scope of this work.

To bound the memory blocking time of a critical task τ_i^c two cases need to be considered. Firstly, the blocking delay of preempting some lower-priority non-critical tasks ($\text{lpnc}(i)$) has to be considered via $\text{MB}_i^{nc,\texttt{SHDw}}$ – which is similar to Equation 2 for strict TDM. Secondly, the blocking delay of preempting another lower-priority critical task ($\text{lpc}(i)$) has to be considered via $\text{MB}_i^{c,\texttt{SHDw}}$. Here, the maximum slack counter values (Δ_l^{max}) over all lower-priority critical

tasks τ_l^c, has to be considered in addition to the cost of a single memory access as follows:

$$\mathrm{MB}_i^{c,\mathtt{SHDw}} = \begin{cases} 0 & \text{if } \mathrm{lpc}(i) = \emptyset, \\ P + Sl - 1 + \max\limits_{l \in \mathrm{lpc}(i)} \Delta_l^{max} & \text{else} \end{cases}$$

$$\mathrm{MB}_i^{nc,\mathtt{SHDw}} = \begin{cases} 0 & \text{if } \mathrm{lpnc}(i) = \emptyset, \\ P^{nc} + Sl - 1 & \text{else} \end{cases}$$

$$\mathrm{MB}_i^{\mathtt{SHDw}} = \max(\mathrm{MB}_i^{c,\mathtt{SHDw}},\ \mathrm{MB}_i^{nc,\mathtt{SHDw}}) \tag{3}$$

The main advantage of this approach is its simplicity in terms of hardware complexity. The timer-like component triggering the preemption on behalf of the task scheduler simply has to detect whether a memory request is pending and, if so, delay the preemption until the request completes. The downside is that it requires precise information on slack counters, so a complex analysis that appears to go against a simple analysis, main advantage of `TDM`.

5.3 Scheduling with Request Preemption (`SHDp`)

An alternative approach is to preempt ongoing memory requests. We consider that requests can only be preempted while pending at the arbiter, but not while being processed by the memory [18]. Consequently, preemptions are still delayed when a request is currently processed by the memory (cf. case 3 of Subfigure 2a). The memory blocking delay for critical and non-critical tasks then can trivially be bounded by the worst-case memory latency, which in turn is bounded by Sl:

$$\mathrm{MB}_i^{\mathtt{SHDp}} = Sl - 1 \tag{4}$$

However, the preempted task later has to reissue the memory request that was preempted, which causes additional delays that need to be accounted for in its response time. A trivial bound for this reissuing delay is the `TDM` period P, i.e., the maximum latency of a pending request after the preemption. Note that there is no need to consider the slack counter value, since it is already covered by the WCET. Furthermore, the misalignment delay always applies to the preempted/reissued request. The misalignment delay thus already covers this overhead (see Subsection 5.5).

Figure 5 shows an execution of the same task set introduced in Subsection 4.2 using `TDMds` and the `SHDp` preemption scheme. This time request $\tau_{0,2}^c$, waiting to access the memory, is immediately preempted by task τ_1^c and is reissued once τ_0^c resumes execution. The request $\tau_{0,2}^c$ thus appears twice in the figure, once before the preemption (aborted) and once thereafter (reissued). Note that the slack counter, due to the request's preemption, diminished from 16Δ to 13Δ.

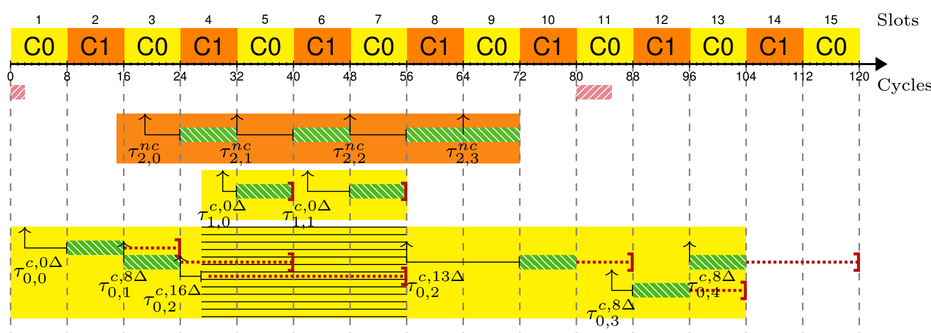


Figure 5 Memory blocking delays considering request preemption (`SHDp`).

This solution appears to be ideal in terms of preemption costs. However, preempting ongoing memory requests can be complex to implement. The processor pipeline has to be extended such that the memory access instruction and all instructions that started execution after it can be aborted. All these instructions have to be reexecuted and thus are not allowed to cause side-effects on the processor state. Such side-effects are, unfortunately, very common. Examples include branch prediction and instruction cache accesses, which occur early in processor pipelines and whose side-effects cannot (easily) be reverted. These effects consequently need to be taken into consideration through dedicated timing analyses. Similarly extensions are required on the memory hierarchy, including caches (aborting cache updates), the memory bus (cache coherence), and the memory arbiter itself. It thus appears preferable to explore an alternative approach that strikes a compromise in terms of implementation and analysis complexity.

5.4 Scheduling with Criticality Inheritance (`SHDi`)

The aim of this approach is to provide a means to control the impact of the slack counters and the interference from non-critical tasks on the memory blocking delay of preempting tasks. The dynamic `TDM`-based arbiters considered here are all based on the notion of *deadlines*. The idea of criticality inheritance is to *impose* a new deadline on a pending request at the moment when a critical task is released, regardless of the criticality of the preempted task. The preemption is still delayed – as before under the `SHDw` scheme. However, the blocking time is bounded by the newly imposed deadline.

This new deadline is computed in the same way as ordinary request deadlines. The only difference is that the issue date is replaced by the release date of the preempting task and that the current value of the slack counter, belonging to the preempted task, is always considered to be 0 (`TDMds`) or *Sl* (`TDMer`). This yields a deadline that certainly falls within the current or the next `TDM` period, and thus effectively bounds the memory blocking delay. An important aspect is that the slack counter for `TDMer` needs to be *Sl* for this computation, and not 0. This is required for the simple reason that, without slack, the deadline could fall on the immediate next `TDM` slot. Under `TDMer` this could cause a clash with an ongoing request from another core that was granted access by the arbiter based on the – then valid – slack counter value of the preempted task. Setting the slack counter to *Sl* thus ensures that any ongoing request can finish before the request from the preempted task is handled.

At runtime two scenarios can be distinguished, depending on the criticality of the preempted task. Firstly, if another critical task is preempted it is clear that its pending critical request already carries a deadline. Replacing this deadline is easy, it suffices to signal

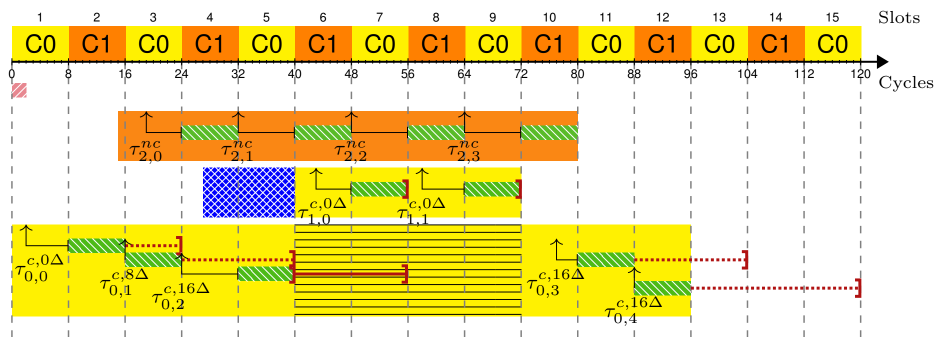


Figure 6 Memory blocking delays considering criticality inheritance (`SHDi`).

to the memory arbiter that the recomputation of the deadline is needed using the current cycle, i.e., the release of the preempting task. Non-critical requests do not carry a deadline and may be held in a structure separated from critical requests (e.g., a FIFO for TDMer). The request thus needs to be taken out of this structure and reissued as a critical request with the appropriate deadline. Finally, the core has to reclaim its TDM slot, which up to now has been marked non-critical, i.e. *NC*. These operations only concern the arbiter and do not impact the processor pipeline or other parts of the memory hierarchy.

Assuming that both cases require a constant amount of clock cycles t_{id} to associate the ongoing request with the newly imposed deadline, we obtain the following bound for the memory blocking delay under SHDi for TDMds and TDMer respectively:

$$\mathrm{MB}_i^{\mathtt{SHDids}} = P + Sl - 1 + t_{id} \quad (5)$$

$$\mathrm{MB}_i^{\mathtt{SHDier}} = P + 2 \cdot Sl - 1 + t_{id} \quad (6)$$

Note that the second Sl in Equation 6 stems from the non-zero slack counter, which ensures that all requests can access memory without clashes. We currently do not apply criticality inheritance when the preempting task is non-critical.[2] This falls in the domain of defining a (sensible) task model, which is not the subject of this work. We plan to address this in future work, along with the handling of timing failure events, in order to fully implement mixed-criticality systems. Non-critical tasks currently implicitly follow the SHDw strategy.

Figure 6 again shows an execution of the same task set as in the previous examples, this time under TDMds combined with SHDi. Critical task τ_1^c is released while critical request $\tau_{0,2}^c$ is pending. The pending request has its original deadline at the end of TDM slot 7 (⊣). As before, it is subject to interference from task τ_2^{nc} on the other core, which otherwise would be prioritized due to the far deadline of $\tau_{0,2}^c$. However, the preemption imposes a new deadline at the end of the immediate next TDM slot (slot 5). The arbiter thus has to prevent the interference from the other core and has to assign the next slot to the preempted task – effectively bounding the memory blocking delay.

This approach combines a reasonable memory blocking delay bound with moderate implementation complexity and simple analysis. The hardware modifications only concern the memory arbiter that reacts to the core's timer, providing the release date and the criticality of the preempting task.

5.5 Misalignment Delays

Strict TDM as well as the dynamic TDM-based schemes all suffer from misalignment delays, highlighted in Section 4, for the same reasons. The delay appears when the task's misalignment *at the first* memory access after a preemption is larger w.r.t. the task's own TDM slot as determined by the WCET analysis [14, 19]. In the worst case the associated memory request misses the task's TDM slot by a single cycle, i.e., the issue date (TDM) or delayed issue date (dynamic TDM) miss the slot by a cycle. The request consequently completes in the worst case in the next TDM period, resulting in the following bound for all the TDM-based approaches:

$$\mathrm{MA}^{\mathtt{TDM}} = \mathrm{MA}^{\mathtt{TDMds}} = \mathrm{MA}^{\mathtt{TDMer}} = P \quad (7)$$

The delay can be larger for non-critical tasks ($P^{nc} = k \cdot P$). Also note that the bound is smaller than the worst-case memory access latency of strict TDM ($P + Sl - 1$), since the memory wait time of to the first TDM slot is accounted for in the WCET (highlighted in beige in Subfigure 2b).

[2] Deadlines could easily be imposed for preemptions by non-critical tasks.

5.6 Response-Time Analysis

The memory blocking (MB) and misalignment delay (MA) bounds, as described above for the preemption mechanisms `SHDw`, `SHDp`, and `SHDi` as well as the arbitration policies `TDMds` and `TDMer`, can now be integrated into the recurrence equations of the response-time analysis.

With regard to a task τ_i, the misalignment delay may appear every time any task τ_j ($i < j$) resumes after a preemption. This is independent of whether τ_j directly preempts τ_i or some other task. The misalignment delay bound MA of the respective arbitration scheme thus needs to be added for every potential preemption that might occur.

Every preempting critical or non-critical task τ_i ($i > 0$) may experience the memory blocking delay *before* starting to execute. The bound thus can be seen as part of the task's WCET, i.e., the ongoing memory request of the preempted task is essentially considered to be executed by the preempting task. We thus add MB to those parts of the equation that represent a WCET (C_i, C_j):

$$R_i^{n+1} = (C_i + \mathrm{MB}_i) + \sum_{\forall j \in \mathrm{hp}(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil ((C_j + \mathrm{MB}_j) + \mathrm{MA}) \tag{8}$$

For the `SHDw` and `SHDp` schemes, independently of the `TDM` policy, but also for `SHDi` when combined with `TDMds`, the term MB_j can thus be safely removed from this equation. The latency of the ongoing memory request, owned by the preempted task, is indeed counted twice: once for the preempting task (MB_j) and the preempted task (C_i). This is illustrated by Figure 6, were the blue area (▩), representing the MB_j term for task τ_1^c, ends at the deadline of request $\tau_{0,2}^c$ and thus the C_i of τ_0^c. For `TDMer` under `SHDi`, the imposed deadline may be greater than the original deadline of the ongoing request, since the imposed deadline is recomputed with a slack counter of Sl. The term MB_j is thus pessimistic and a possible refinement is to subtract the minimal memory latency l from this term.

6 Experiments

We use simulation to evaluate the runtime behavior of the various preemption mechanisms. Before discussing the results, we provide details on the experimental setup.

6.1 Experimental Setup

We developed a simulation engine that is able to simulate a dynamic execution trace of an entire multi-core platform. Traces are specified according to the system/hardware model from Sections 2 and 5. The engine includes a fixed-priority preemptive task scheduler, and various memory arbitration schemes (`TDMfs`, `TDMds`, `TDMer`), which can be combined with the aforementioned preemption mechanisms (`SHDw`, `SHDp`, `SHDi`). The `TDMfs` arbiter represents a variant of strict `TDM`, where non-critical tasks may only reclaim unused slots, and serves as a baseline for our measurements.

The engine can be configured in terms of the number of (non-)critical cores, `TDM` slots in a period (at least 1 per critical core), and (non-)critical tasks. The task scheduler respects user-defined core affinities that can be assigned freely to tasks. However, for our experiments each task is assigned to a single fixed core only (partitioned scheduling). Tasks are represented by a sequence of jobs, which, in turn, represent dynamic execution traces consisting of memory accesses that are separated by a fixed amount of computation time (cf. the various examples). This allows to simulate the same execution trace of a task set using different platform configurations and compare the results. Note that the framework does not model the actual computations, only the *time* needed for computations.

6.1.1 Task Set Generation

The *UUniFast* algorithm [6] allows to randomly generate a task set based on two parameters n and U, where n specifies the total number of tasks and U the total utilization desired. As we assume partitioned scheduling, we apply the algorithm for each core of the multi-core system separately and combine the resulting tasks, with the corresponding affinities, into a final task set Γ.

For each core, UUniFast first generates n different utilization values $\{u_0, u_1, \ldots, u_{n-1}\}$, one for each task τ_i. The sum of these utilization numbers yields the core utilization $U \leq 1$. From the utilization parameters, the task periods T_i are generated. Note that we constrain our system to harmonic periods, which ensure that hyper-periods and simulation times remain reasonable. The period of the first task T_0 is 20ms, while all other periods are random multiples of T_0, obtained from a uniform random distribution in the range $[1, 5]$. The individual task periods are hence in the range from 20ms to 100ms. From the task periods and the utilization numbers as well as the task set's hyper-period h, we then derive the worst-case execution time of each task $C_i = T_i \cdot u_i$, the implicit deadline $D_i = T_i$, and the number of jobs for each task $J_i = h/T_i$. Finally, task priority and criticality (on critical cores) are randomly assigned when the final task set Γ is constructed.

6.1.2 Traffic Generator

The simulation framework then requires a memory trace for each job of the generated task set, i.e., the (time) distances between consecutive memory accesses (see Section 2). We use the same traffic generator as in our previous work [13]. We thus only provide a brief overview here and invite interested readers to consult the original paper for additional details.

The goal of the traffic generator is to provide synthetic memory access patterns representing *cache misses* of some random dynamic program execution. The generator is thus calibrated against *actual dynamic execution traces* collected using the MiBench benchmark suite [8] on the Patmos processor architecture [20]. Note, however, that the generated memory accesses have to be consistent with the task's WCET (C_i). The generator thus tracks the evolution of a WCET bound as it proceeds. For each newly generated memory access this WCET bound is incremented by the worst-case memory access latency, which is bounded by $P - 1 + Sl$ cycles. Note that the same bound is applied for critical and non-critical tasks, even if no `TDM` slot is allocated to a core executing only non-critical tasks. The generator simply stops once the bound reaches the task's C_i. The resulting execution times thus rather closely approach the tasks' WCETs. The memory load is higher than can be expected in the average case. Note that we capture this effect in our experiments by varying the total system load.

6.1.3 `TDM` Schedule

The duration of a `TDM` slot length Sl, corresponds to an upper bound of the memory access latency previously determined on a Terasic DE-10 Nano evaluation board that is equipped with an Intel Cyclone V SoC-FPGA and 1 GB of DDR3 memory. A single Patmos processor was implemented in the FPGA and performed memory accesses in isolation via the SoC's multi-port memory controller running at 100 Mhz (the remaining components of the SoC were deactivated). At any moment a single memory access was in-flight during these measurements. Depending on the internal state of the memory controller and DDR memory (refresh, open page, etc.), we measured a memory latency between 21 and 40 cycles. In the simulation runs we thus consider a `TDM` slot length of 40 cycles. For `TDMer`, which is able to exploit memory requests completing faster than the `TDM` slot length, we simulate a varying latencies using a uniform random distribution in the range $[21, 40]$ clock cycles.

6.1.4 Simulation Setup

Using the task set and memory traffic generators, we perform simulations with varying numbers of cores (2, 4, 8, 16), critical cores (power of 2 in the range from 1 to the number of cores), and the normalized system utilization (steps of 10 from 10% to 100%, normalized to the number of cores). The number of tasks is randomly chosen between the number of cores and 32 tasks, while the number of critical tasks is randomly chosen between 1 and the number of tasks assigned to a critical core.

The `TDM` schedule assigns a single slot to each critical core, where each slot takes $Sl = 40$ cycles. The period P hence ranges from 40 to 640 cycles. The slack counters are reset at every job start to 0 or 40 cycles for `TDMds` and `TDMer` respectively. For each of these system configurations, 10 simulation runs were performed using 10 different task sets, which results in 12 600 runs.

6.2 Results for Preemption Schemes

We start by analyzing the memory blocking delays on the simulated task sets with the three preemption mechanisms considered. Subfigure 7a shows the cumulative average memory blocking delay over an entire simulation run for the `SHDw` preemption mechanism considering the `TDMfs`, `TDMds`, and `TDMer` arbitration policies.[3] As expected, the cumulative overhead can become very large going up to $1.6 \cdot 10^6$ cycles for both `TDMfs` and `TDMds` (showing virtually identical results), while only reaching $3 \cdot 10^3$ cycles for `TDMer`– despite the fact that preemptions are rare events.[4] We observed a task set executing on 16 cores experiencing **502**ms (an average of **31**ms per core) of blocking delays within **590**ms of total execution time (assuming a clock speed of 100 Mhz). The other approaches, `SHDp` and `SHDi`, have significantly lower overhead compared to `SHDw` (not shown due to space considerations).

Since non-critical tasks execute in a best-effort manner, their memory blocking delays are expected to potentially become quite large, in particular, when preempting another non-critical task. Subfigure 7b thus highlights the average maximum memory blocking delay experienced by non-critical tasks across utilization levels for `SHDw` in combination with either `TDMfs`, `TDMds`, or `TDMer`. As can be seen, individual preemptions consistently take thousands of cycles, which corresponds to about **18**ms (maximum observed memory blocking delay). As these events are still rare, some volatility in the simulations is visible through the large drop at 90% utilization. Note that typically non-critical tasks represent the majority of the computation and memory load of the generated task sets. Consequently, non-critical tasks experience most of the preemptions as well as the associated memory blocking delays.

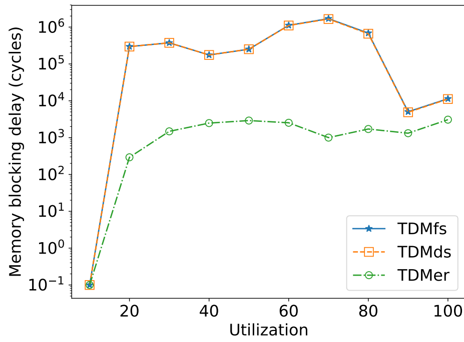
Finally, Subfigure 8a shows the maximum memory blocking delays experienced by critical tasks considering all three preemption mechanisms in combination with `TDMfs`, `TDMds`, and `TDMer`. The delays are normalized w.r.t. the `TDM` period in order to avoid penalizing simulation configurations with shorter periods. These delays are representative of the upper bounds defined in Subsection 5. Note that the results for `TDMfs` are virtually identical to those for `TDMds` and are thus not shown.

As expected, the `SHDp` scheme presents very low memory blocking delays, as can be seen in more detail in Subfigure 8b. Under `TDMds` this preemption mechanism leads to no noticeable memory blocking. This is explained due to the non-work-conserving nature of this arbitration technique, which leads to long memory wait times and consequently increases

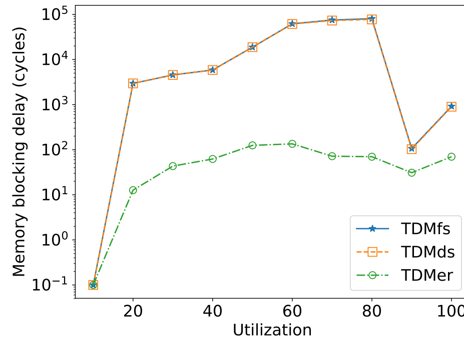
---

[3] Each point represents an average value over all schedulable task sets at the corresponding normalized utilization.

[4] On average, there are 7 preemptions per run, with a maximum number of 288 preemptions in rare cases.



(a) Average cumulative memory blocking delay over all tasks.



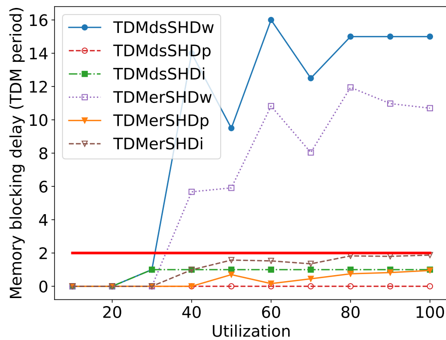
(b) Average maximum memory blocking delay for non-critical tasks.

Figure 7 Memory blocking delays across normalized system utilization for SHDw.

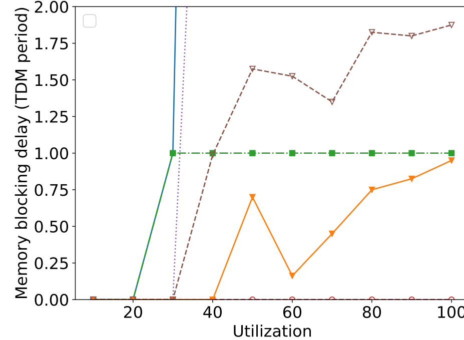
the probability of preempting a pending request. The situation is different for TDMer. Due to its high efficiency, the probability of preempting a pending request is much lower. It is instead more likely to preempt a request that is currently processed by the memory, resulting in moderate memory blocking delays for SHDp. Note that these delays may never exceed a single TDM period, notably for configurations with a single critical core. The highest memory blocking delays are observed for SHDw under TDMds. The memory delay amounts to up to 16 TDM periods for a configuration with 2 cores, where both cores are critical (i.e., $P = 80$ cycles). The TDMer arbiter fares slightly better, with a maximum of about 12 TDM periods. This demonstrates the high overhead experienced even by critical tasks when using SHDw.

The SHDi scheme, as expected, falls in-between the two other schemes. In combination with TDMds the memory blocking delay is at most one TDM period. However, starting from 50% utilization, we can notice that TDMer exhibits slightly worse results compared to TDMds. Nevertheless, the preemption cost for TDMer always stays below 2 TDM periods (highlighted by Subfigure 8b) – notably for configurations with a single critical core (cf. Equation 6).

These result confirm the intuitive expectation that the overhead induced by the three preemption schemes is strictly increasing from SHDp over SHDi to SHDw. However, this is not always the case due to the extra costs induced by the slack counters under TDMer (a counter-example was encountered for a larger slot length of $Sl = 100$, while evaluating the



(a) Maximum memory blocking delay w.r.t. the arbitration and preemption schemes.



(b) Zoom on the plot from the left, focusing on blocking delay up to 2 TDM periods.

Figure 8 Maximum memory blocking delay for critical tasks across normalized system utilization.

impact of varying the memory access latency[5]). Recall that the newly imposed deadline during a preemption is computed considering a minimum slack of a single TDM slot length (Sl) in order to avoid clashes on the immediate next TDM slot (see Subsection 5.4). Accounting for this additional runtime overhead $\varepsilon \leq Sl$, we can derive a relationship between the various preemption schemes: $\mathrm{MB}_i^{\mathtt{SHDp}} \leq \mathrm{MB}_i^{\mathtt{SHDi}} \leq \mathrm{MB}_i^{\mathtt{SHDw}} + \varepsilon$.

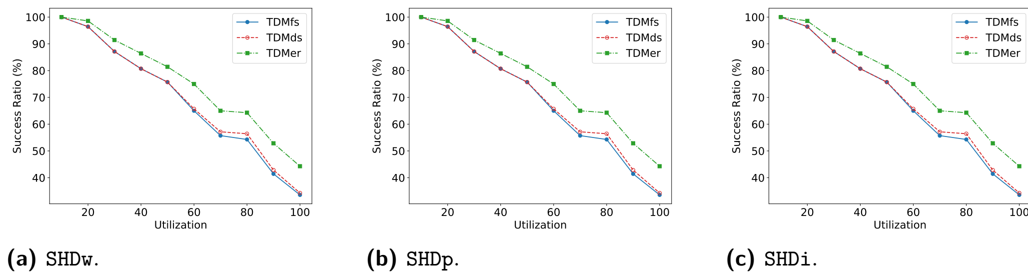
To conclude, the arbitration-induced preemption costs for critical tasks can be very high at runtime when the SHDw scheme is used, while it is similar for the SHDi and the SHDp schemes. This means that other criteria, such as predictability and implementation complexity, can be used to choose among the schemes. SHDi appears to strike a reasonable balance between these two criteria. Besides, the memory blocking delays are lower using TDMer than when using TDMfs or TDMds.

6.3 Results for (Preemptive) Arbitration Schemes

We now evaluate the success rate of our preemptive arbitration policies. The success ratio refers to the number of task sets[6] that were *schedulable* for each level of utilization, i.e., simulation ended its execution without any deadline miss for critical tasks. Subfigures 9a, 9b, and 9c depict this success rate for our 3 arbitration policies under SHDw, SHDp, and SHDi respectively. Results are shown considering the same task set for all combinations. Comparing the impact of the preemption schemes across all utilization levels, little difference among them is visible. This can be explained by the number of preemptions, which is small compared to the total execution time of each simulation. As seen in Section 6.2, individual preemptions may however cause considerable overhead.

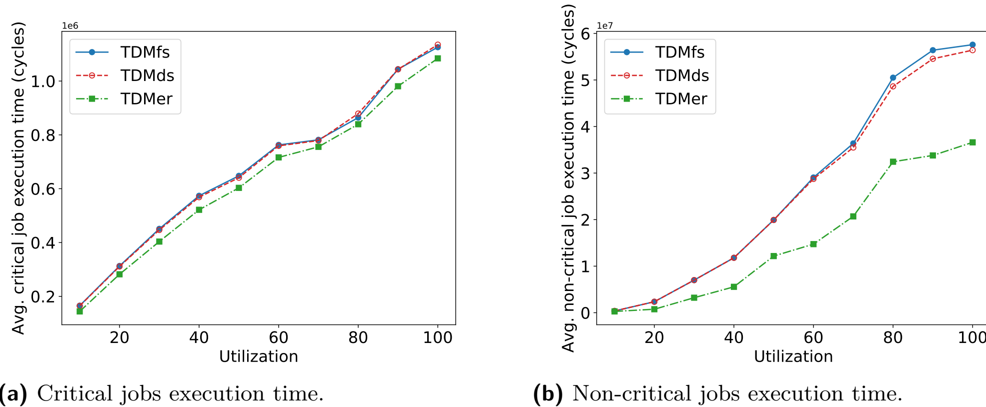
Overall, there was no significant difference in the success ratio of TDMfs and TDMds, except for a normalized utilization above 70%. In that case, we can notice better results for TDMds, regardless of the preemption technique. This is different from the results obtained in our previous work [12, 13], which shows an improved memory utilization for TDMds compared to TDMfs. In this previous work, the observed improvement reached up to a factor of 3.3 [13] in terms of memory utilization at low total system utilization, which then leveled off considerably at higher load. It appears that, TDMds improves the memory utilization mostly for situations where system load is not jeopardizing *schedulability*, explaining the small impact on success rates. However, regardless of the preemption techniques, TDMer shows better results at almost all utilization levels. This can be explained by the decoupling from TDM slots and

[5] Results not shown due to space constraints.
[6] Each point represents an average value over 124 runs.



(a) SHDw. **(b)** SHDp. **(c)** SHDi.

Figure 9 Average *schedulability* success ratio through normalized system utilization.



(a) Critical jobs execution time. **(b)** Non-critical jobs execution time.

Figure 10 Average jobs execution times with varying normalized system utilization using `SHDi`.

the fine-grained dynamic memory arbitration. This effectively renders the `TDMer` scheme work-conserving. These results confirm those previously obtained considering a restricted task model (one task per core), where the observed memory utilization improvements for `TDMer` were considerable, even at high utilization levels [13]. `TDMer` thus has a relevant impact on the success rates.

As the most reasonable choice for a preemption scheme is `SHDi`, Subfigures 10a and 10b depict the average execution time of critical and non-critical tasks respectively using this scheme. The relative gain by `TDMer` is particularly visible for the average execution time of non-critical tasks at high utilization levels. For instance, at 100% system utilization `TDMer` achieves an improvement of 36%. Clearly concluding that `TDMer` improves the execution times of critical and non-critical tasks, and thus can reduce the probability of missing the deadline of critical tasks. Additionally, as preemptions are rare events, the runtime impact of the different preemption schemes is small compared to the impact of the arbitration policy.

7 Related work

`TDM` strategies have been used at different design levels of real-time systems. Tabish et al. [21] split tasks into execution phases with and without main memory activities, which are scheduled in order to avoid contention by applying `TDM`. `TDM` is applied at the task scheduling level, with slot sizes accommodating the longest phase. Phases without main memory activity may only access data stored in a scratchpad. Preemption-related delays due to memory accesses thus cannot appear.

The *slot* shifting approach by Fohler [7], applies a similar idea as our dynamic arbitration schemes, but to task scheduling. The granularity of slots allocated to tasks is consequently much larger than individual memory requests considered here. Memory blocking delays may be present, depending on the underlying platform, *even* when only a single core is considered, but not taken into consideration.

Altmeyer et al. [1] worked on a compositional framework for multi-core response time analysis. Among other things, they explored the impact of different arbitration schemes (Fixed-Priority, FIFO, Round-Robin and TDM) and defined upper-bounds for the arbitration induced preemptions delays. They assume that all requests sent to the bus are non-preemptive, which is similar to our SHDw approach. Their results only apply to strict TDM and not to our dynamic TDM-based approaches. However, their bounds for Fixed-Priority arbitration seem to be incorrect. Suppose a higher-priority task τ_i preempts another tasks τ_j $(j < i)$. The bus interference experienced by an ongoing request of τ_j may then block τ_i, i.e., the memory blocking delay. The proposed equation [1, Eq. 12], however, misses interference from requests issued by a higher priority task τ_k running on another core, where $j < k < i$. This cannot be resolved easily using the framework of Altmeyer et al. – except if a scheme similar to SHDi is applied. In that case it suffices to consider the ongoing request of the preempted task in the context of the preempting task – via the term $S_i^x(t)$.

Yonghui et al. *skip* unused slots in a TDM period, supporting variable-sized TDM slots [17]. Hassan et al. [11, 10] similarly propose a work-conserving variant of TDM along with a technique to generate harmonic TDM schedules accommodating critical and non-critical tasks. The approaches do not preserve the relative alignment of the program execution with the TDM schedule, offset analyses [14, 19] thus cannot be applied. Preemption mechanism are not explicitly discussed, it appears that a strategy similar to SHDp is envisioned by the authors. Memory blocking delays exist and have not been analyzed previously. The bounds for strict TDM appear to be applicable – except for preemptions of non-critical tasks in a harmonic TDM schedule. Finally, it is unclear whether the TDM schedule can be changed in response to a task preemption.

Paolieri et al. [18] propose a multi-core platform for mixed-criticality systems that is equipped with a hierarchical Round-Robin arbiter, which always to prioritizes critical tasks. In contrast to the approaches presented here, their arbiter is not able to exploit task criticalities to improve memory utilization or to reduce the average execution time of non-critical tasks. The authors envision a preemption scheme similar to SHDp, but do not analyze the preemption costs. The ideas underlying Altmeyer's analysis for Round-Robin and Fixed-Priority [1] could be used to model the system.

Kostrzewa et al. [15] propose a mechanism, which provides latency guarantees for hard real-time transmissions in a network-on-chip. For each critical task the amount of slack time is computed off-line and programmed into a counter at the level of NoC interface at every job start. Best-effort transmissions, similar to the dynamic arbiters considered here, may then delay critical transmissions as long as slack is available. Task preemptions are then considered for hard real-time and best-effort transmissions. Critical tasks may run out of slack in the middle of a best-effort transmission which are thus preempted. A delay – similar to the memory blocking delay – is taken into consideration to avoid exhausting the slack budget before all best-effort transmissions are successfully suspended.

8 Conclusion

The work presented in this paper extends previous work on dynamic TDM-based memory arbitration schemes by adding support for a preemptive execution model and by identifying the limitations of such a system. We identify two sources of arbitration-induced preemption delays, the *memory blocking delay* and the *misalignment delay*, and propose means to manage, and finally bound these delays. While bounding the misalignment delay is straightforward, limiting the memory blocking delay is more involved. We thus propose formal bounds for two

obvious preemption schemes based on waiting and preemptable memory requests (`SHDw` and `SHDp`). Additionally, we propose an alternative scheme (`SHDi`), which imposes new deadlines for critical requests and leverages *criticality inheritance* when a critical task is blocked by a non-critical request. The experimental results showed that the preemption mechanisms show little difference at runtime, which allows us to select the best approach according to other criteria, such as low implementation complexity and analyzability. The new `SHDi` approach appears to offer the best trade-off in terms these criteria.

References

1. Sebastian Altmeyer, Robert I. Davis, Leandro Indrusiak, Claire Maiza, Vincent Nelis, and Jan Reineke. A Generic and Compositional Framework for Multicore Response Time Analysis. In *Proceedings of the International Conference on Real Time and Networks Systems*, RTNS '15, pages 129–138. ACM, 2015. doi:10.1145/2834848.2834862.
2. N. Audsley, A. Burns, M. Richardson, K. Tindell, and A.J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8:284–292, 1993.
3. S. Baruah and N. Fisher. The partitioned multiprocessor scheduling of sporadic task systems. In *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, pages 9 pp.–329, December 2005. doi:10.1109/RTSS.2005.40.
4. Sanjoy Baruah and N. Fisher. The partitioned multiprocessor scheduling of deadline-constrained sporadic task systems. *IEEE Transactions on Computers*, 55(7):918–923, July 2006. doi:10.1109/TC.2006.113.
5. Alan Burns and Robert I. Davis. A Survey of Research into Mixed Criticality Systems. *ACM Comput. Surv.*, 50(6):82:1–82:37, November 2017. doi:10.1145/3131347.
6. P. Emberson, R. Stafford, and R.I. Davis. Techniques For The Synthesis Of Multiprocessor Tasksets. In *Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, pages 6–11, 2010.
7. G. Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *Proc. Real-Time Systems Symposium (RTSS)*, pages 152–161, Pisa, Italy, December 1995.
8. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. of the Int. Workshop on Workload Characterization*, pages 3–14, 2001.
9. S. Hahn, J. Reineke, and R. Wilhelm. Towards Compositionality in Execution Time Analysis: Definition and Challenges. *SIGBED Rev.*, 12(1):28–36, 2015.
10. M. Hassan and H. Patel. Criticality- and Requirement-Aware Bus Arbitration for Multi-Core Mixed Criticality Systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11, April 2016. doi:10.1109/RTAS.2016.7461327.
11. M. Hassan, H. Patel, and R. Pellizzoni. A framework for scheduling DRAM memory accesses for multi-core mixed-time critical systems. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 307–316, April 2015. doi:10.1109/RTAS.2015.7108454.
12. F. Hebbache, M. Jan, F. Brandner, and L. Pautet. Dynamic Arbitration of Memory Requests with TDM-like Guarantees. In *International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS'17)*, December 2017.
13. F. Hebbache, M. Jan, F. Brandner, and L. Pautet. Shedding the Shackles of Time-Division Multiplexing. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 456–468, December 2018. doi:10.1109/RTSS.2018.00059.
14. T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury. Static Analysis of Multi-core TDMA Resource Arbitration Delays. *Real-Time Syst.*, 50(2):185–229, March 2014.

15 A. Kostrzewa, S. Saidi, and R. Ernst. Slack-based Resource Arbitration for Real-time Networks-on-chip. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '16, pages 1012–1017. EDA, 2016.

16 H. Leontyev and J. H. Anderson. Generalized Tardiness Bounds for Global Multiprocessor Scheduling. In *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pages 413–422, December 2007. doi:10.1109/RTSS.2007.33.

17 Y. Li, B. Akesson, and K. Goossens. Architecture and Analysis of a Dynamically-scheduled Real-time Memory Controller. *Real-Time Syst.*, 52(5):675–729, September 2016.

18 Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. Hardware Support for WCET Analysis of Hard Real-time Multicore Systems. In *Proceedings of the International Symposium on Computer Architecture*, ISCA '09, pages 57–68. ACM, 2009. doi:10.1145/1555754.1555764.

19 Hamza Rihani, Matthieu Moy, Claire Maiza, and Sebastian Altmeyer. WCET analysis in shared resources real-time systems with TDMA buses. In *Proceedings of the International Conference on Real Time and Networks Systems*, RTNS '15, pages 183–192. ACM, 2015. doi:10.1145/2834848.2834871.

20 M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, C. W. Probst, S. Karlsson, and T. Thorn. Towards a Time-predictable Dual-Issue Microprocessor: The Patmos Approach. In *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, OASICS, pages 11–21, 2011.

21 R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S. S. Phatak, R. Pellizzoni, and M. Caccamo. A Real-Time Scratchpad-Centric OS for Multi-Core Embedded Systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11, April 2016.