Noname manuscript No.
(will be inserted by the editor)

Work-Conserving Dynamic Time-Division Multiplexing for Multi-Criticality Systems

Farouk Hebbache · Florian Brandner · Mathieu Jan · Laurent Pautet

Received: date / Accepted: date

Abstract Multi-core architectures pose many challenges in real-time systems, which arise from contention between concurrent accesses to shared memory. Among the available memory arbitration policies, Time-Division Multiplexing (TDM) ensures a predictable behavior by bounding access latencies and guaranteeing bandwidth to tasks independently from the other tasks. To do so, TDM guarantees exclusive access to the shared memory in a fixed time window. TDM, however, provides a low resource utilization as it is *non-work-conserving*. Besides, it is very inefficient for resources having highly variable latencies, such as sharing the access to a DRAM memory. The constant length of a TDM slot is, hence, highly pessimistic and causes an underutilization of the memory. To address these limitations, we present dynamic arbitration schemes that are based on TDM. However, instead of arbitrating at the level of TDM slots, our approach operates at the granularity of clock cycles by exploiting *slack* time accumulated from preceding requests. This allows the arbiter to reorder memory requests, exploit the actual access latencies of requests, and thus improve memory utilization. We demonstrate that our policies are analyzable as they preserve the guarantees of TDM in the worst case, while our experiments show an improved memory utilization. We furthermore present and evaluate an efficient hardware implementation for a variant of our arbitration strategy.

Keywords Time-Division Multiplexing · Dynamic Arbitration · Predictable Computing · Multi-Criticality Systems

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

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

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

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

Extended Version

In recent work (Hebbache et al., 2018), we addressed several challenges in multi-core architectures for real-time systems, which mainly arise from contention between concurrent accesses to shared memory. Among the available memory arbitration policies, our approach operates at the granularity of clock cycles by exploiting *slack* time accumulated from preceding requests. This allows the arbiter to reorder memory requests, exploit the actual access latencies of requests, and thus improve memory utilization. We demonstrated that our policies are analyzable as they preserve the guarantees of TDM in the worst case, while our experiments show an improved memory utilization on average.

We extend this previous work by discussing additional details on the simulation setup (Subsection 6.2) and by proposing an efficient hardware implementation of our approach along with its evaluation. In Section 7, we give an overview of the architecture design and describe how the different features of our arbitration strategy (notably, deadline and slack counters) can be implemented. We also show how this work led us to propose a variant of our initial scheme that takes implementation trade-offs and costs into consideration, a formal proof of the worst case behaviour for the new approach is also discussed in this section. In Section 8, we demonstrate that these trade-offs do not impact the overall performance of our approach, while enabling a simple and efficient implementation. Another contribution addresses the issues raised by highly variable access latencies and excessively pessimistic latency bounds. In Subsection 6.6, we evaluate the impact of our arbitration schemes w.r.t. the variability of the memory access latency, i.e., differences between the best and worst memory access latency. We show that even in this context, our approach allows to achieve the maximum memory utilization with the guarantee of respecting the timing constraints of critical tasks for real-time systems.

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 run-time. This problem is further amplified as real-time systems today allow tasks with different levels of criticality (Vestal, 2007; Burns and Davis, 2017), and even non-critical tasks, to execute on the same multi-core architecture. Another approach is to divide time into slots and allocate them to cores to exclusively access memory. Using a Time-Division Multiplexing (`TDM`) policy, 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.

The access latency of a memory request when using `TDM`, however, now depends on the scheduling of these time slots, even if they are unused. Such unused slots appear when an 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 (as for instance under Round-Robin). 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 pessimism of `TDM` stems from the length of `TDM`

slots, expressed in clock cycles, which have to be longer than the worst-case latency of handling memory requests. Memory requests targeting a DRAM memory, however, have highly variable latencies (Wu et al., 2013). The temporal behavior of the DRAM depends, for instance, on memory refresh operations or whether the accessed memory page changed. Besides, the access latencies of memory read requests is higher than that of memory writes, since data must be sent back to the requesting core.

To overcome these aforementioned limitations, we explore the definition of dynamic arbitration schemes based on TDM. We claim that the level of criticality should not only be used by task schedulers, but also by memory arbiters. We thus recently started exploring TDM-based arbitration schemes that allow the arbiter, under certain conditions, to favor requests of non-critical tasks over request from critical tasks (Hebbache et al., 2017). This is achieved by associating deadlines to the memory accesses of critical tasks, which correspond to the end of their corresponding slots under a strict TDM scheme. These deadlines allow the arbiter to compute the *slack time* of each pending request from critical tasks in the system, i.e., the amount of time a request completed before its deadline. Later on, and if slack times permit, the arbiter can change the order in which requests are handled by reallocating unused slots by critical tasks to non-critical tasks. This arbitration policy is called TDMds, for *dynamic TDM with slack counters*, and addresses one source of pessimism of strict TDM.

A first contribution of this paper consists in proposing two dynamic TDM arbitration schemes that extend TDMds in order to address the sources of pessimism related to TDM slots. This is achieved by decoupling the arbitration from the TDM slots, i.e. arbitration decisions are taken at the granularity of clock cycles. Our experiments show that this allows to improve delays suffered by traditional TDM by a factor of at least 1.5, and up to a factor of 4.2. A second contribution is a formal correctness proof of the newly proposed approaches, which also applies to TDMds. Most notably, we prove that TDM's temporal behavior is preserved for critical tasks. Consequently, analysis results valid under TDM, such as offset analyses (Kelter et al., 2014; Rihani et al., 2015), are equally valid under our schemes. Finally, we present a hardware implementation of a variant of our scheme that takes implementation trade-offs and costs into consideration. We show that these trade-offs do not impact the overall performance of our approach, while enabling a simple and efficient implementation.

The remainder of this paper is organized as follows. Section 2 describes the considered system model. In Section 3, we then motivate our contributions by identifying the sources of pessimism within TDM slots. Section 4 presents two TDM-based schemes that no longer perform arbitration at the granularity of slots, but at the level of clock cycles. We demonstrate that their worst-case behavior is equivalent to a strict TDM arbitration policy in Section 5. The proposed approaches are next evaluated in terms of memory utilization and efficiency in Section 6. In Section 7, we present a simple and efficient hardware implementation of a variant of our proposed arbitration schemes. Finally, Section 8 contains an evaluation of the hardware implementation and the associated trade-offs. Section 9 presents related work, before concluding in Section 10.

2 System model

In this paper, we explore memory request arbitration on a multi-core architecture consisting of m cores with private caches and a single shared memory, i.e., cache misses result in memory requests to transfer cache blocks. For now we assume a restricted task model, where each core executes a single independent and periodic task τ_i, $1 \leq i \leq m$. Tasks are modeled

as a sequence of memory requests separated by a given number of processor clock cycles, representing the amount of computation that is performed between two memory accesses. For a task τ_i, the distance between memory requests $k-1$ and k is given by $dist_k$. We assume a composable computer architecture (Hahn et al., 2015), which ensures that the distance between requests is independent from the execution of other tasks. The only interference between the independent tasks thus stems from accesses to the shared memory and, most importantly, the employed memory arbitration scheme.

Figure 1 illustrates an execution under such a system model using traditional TDM arbitration, considering 3 tasks $(\mathsf{A},\mathsf{B},\mathsf{C})$ that execute on separate cores and perform concurrent memory requests to the shared memory. Each task is assigned a dedicated TDM slot (vertical columns, labeled A trough C) that alternate over time. The slot length Sl in this example is 8 processor clock cycles, which results in a global TDM period P of 24 clock cycles (i.e., $P = 3 \cdot Sl$). The tasks in our system model are represented by sequences of memory requests as follows: $(\mathsf{A}\colon 2,24,12)$, $(\mathsf{B}\colon 14,4,2)$ and $(\mathsf{C}\colon 26,6)$. Task A, for instance, performs its first memory access after 2 clock cycles, the second access 24 clock cycles after completing the first one, and the third access another 12 cycles later. In the remainder of the paper, all figures only show a single instance, or job, of each task.

Each memory access of a task blocks the task's execution, depending on the memory's speed, i.e., the TDM slot length or DRAM latency (Wu et al., 2013), and the arbitration policy (TDM). This blocking time is visualized in the figure by considering the following dates for each request: (a) the *issue date* (↑⌙) indicates the moment when a task issues a request to the memory arbiter (e.g., through a bus or on-chip network), (b) the *start date* (⊣) indicates the moment when the memory starts processing a request, and (c) the *completion date*. The time span between these dates correspond to the *Request Inter-Task Delay* $(b-a)$ and the *Request Execution Time* $(c-b)$ of Paolieri et al. (2013). Request B_1, for instance, is issued 4 cycles after the completion of request B_0 in slot 6. The request is granted access to the memory in slot 8, which starts processing immediately, and completes at the end of the TDM slot indicated by the green hatched bar (▨). Its request inter-task delay is thus 12 cycles, while its request execution time is equal to Sl. In this work, we assume that requests are granted access to the memory only at the beginning of a TDM slot, as this minimizes TDM periods. The results presented hereafter can easily be generalized to other TDM schemes. The memory is not always busy, unused slots are thus indicated by a red hatched bar (▨). The schedule length of our example is 13 TDM slots, where A_2 is the last request to complete at the end of the 13^{th} slot. The blocking time of requests A_0, A_1, and A_2 are respectively 3.75, 3, and 1.5 slots respectively. The total blocking induced by TDM on the requests from task

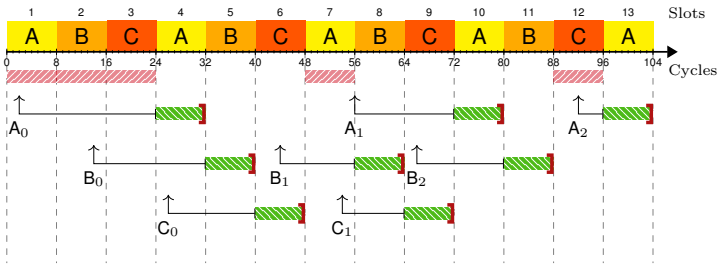


Fig. 1 Regular TDM arbitration of three tasks A, B, and C, including periods of memory activity (▨) and idling (▨).

A is thus 66 cycles. The last memory access of task A therefore completes after 104 cycles, since the amount of processing of task A is 38 cycles ($2 + 24 + 12$).

Similar to mixed-criticality systems (Burns and Davis, 2017), we define two classes of tasks in our model: *critical* and *non-critical* tasks. However, our model is somewhat simplified, as we do *not* introduce different operation modes, e.g., the modes LO/HI from Vestal (2007). We simply assume that critical tasks are associated with a strict deadline that has to be met under all circumstances. The underlying computer platform and memory arbitration scheme thus have to provide a means to bound the worst-case execution times of these tasks. Non-critical tasks, on the other hand, may miss their deadlines. In contrast to typical mixed-criticality systems, we do not demand strict worst-case execution time bounds for them in this work. The underlying hardware can thus execute these tasks in a best-effort manner. Non-critical tasks continue execution even if they miss their deadline, therefore, requests of non-critical tasks are never canceled and remain pending until processed by the main memory.

Our system/task model is kept simple on purpose for this work. We refer interested readers to a discussion of recent extensions (Hebbache et al., 2019), which we consider out of the scope of this work.

3 Motivation

The system model from the previous section can be implemented relatively easily using a TDM arbitration scheme, where each critical task is assigned a dedicated TDM slot. Non-critical tasks may share TDM slots or, in the worst-case, simply recycle unused TDM slots leftover by the critical tasks (i.e., the slots shown in red in Figure 1). This appears to be an attractive solution for critical real-time systems, where TDM is widely popular due to its predictability. The strict separation of critical tasks would allow to easily establish worst-case execution time bounds (Kelter et al., 2014; Rihani et al., 2015), while the non-critical tasks would improve the memory utilization (Hebbache et al., 2017).

We recently started investigating improved arbitration schemes based on TDM, that allow for a more dynamic scheduling of memory requests (Hebbache et al., 2017). We defined an arbitration scheme dubbed TDMds (*dynamic TDM with slack counters*), where the arbitration decisions are driven by deadlines. For critical tasks, we derive a deadline for each request, which simply corresponds to the end of the task's next TDM slot after the request's issue date. The TDMds arbiter is then free to schedule memory requests dynamically, as long as the request deadlines of critical tasks are respected. This dynamic scheduling thus blurs the separation between the TDM slots of tasks, i.e., any task may perform a memory request in any given TDM slot – as long as no deadlines are missed. We associate a *slack counter* with each critical task (implementation details are discussed in Section 7). This counter indicates how many cycles the last request of a job completed before its deadline. When a new request is issued by a job that previously accumulated some slack, the request's issue date occurs earlier than expected under a strict TDM scheme. Consequently, also the corresponding deadline appears earlier than under strict TDM. This may potentially limit the available scheduling choices for the arbiter. We address this issue by computing a so-called *delayed issue date*, which simply consists of adding the previously accumulated slack back to the issue date of a new request. The delayed issue date may then potentially push the deadline farther into the future. This, furthermore, provides a strong guarantee linking executions under strict TDM to executions under our scheme. During the execution of a job, the deadlines computed under TDMds considering the accumulated slack, exactly correspond to the dead-

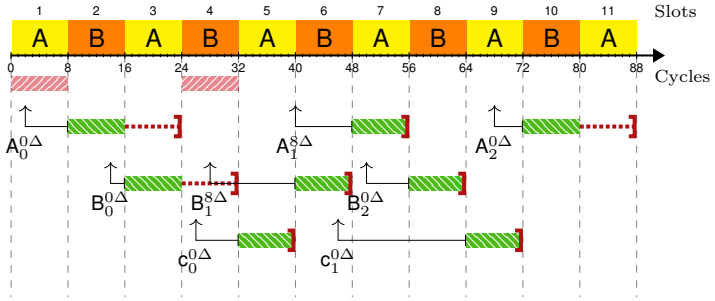


Fig. 2 Improved arbitration using `TDMds` of two critical (A and B) and a non-critical task (c).

lines/completion dates under regular `TDM`. Note, however, that the slack accumulated by a job is not preserved for subsequent jobs of a task, i.e., slack counters have to be reset to zero at job start.

Deadlines only apply to critical tasks, which have to respect strict timing constraints. The `TDMds` approach nevertheless assigns a *soft* deadline to non-critical requests, which corresponds to the end of the immediate next `TDM` slot. On a deadline miss for a non-critical request the deadline is simply pushed back by a `TDM` slot length. Issued requests can so be kept in a priority queue considering the requests' deadlines where critical tasks have higher priority on a tie with non-critical requests. The arbiter then schedules requests according to an earliest-deadline-first (EDF) policy. Our deadline-driven arbitration policy renders `TDM` slots exchangeable between cores and allows to reorder requests to improve memory utilization.

Figure 2 shows an execution of the task set from Section 2 under `TDMds`, considering tasks A and B as critical tasks, task c as non-critical (its label now thus is lowercase). The visualization of a request now also includes the request's deadline (·]). The deadline may well lie far after the request's actual completion date, and thus generate slack for the job issuing the request (e.g, requests A_0, A_2, and B_0). The value of the slack counter is displayed as a superscript for each request. For instance, request A_1 has accumulated 8 cycles of slack (superscript 8Δ in Figure 2). At the beginning of each `TDM` slot, the arbiter chooses one of the issued requests, independently from the actual owner of the slot. This is, for instance, the case for non-critical request c_0, which is granted access to the memory despite the fact that critical request B_1 has been issued. The slack accumulated by task B is here *spent* in favor of the non-critical task. In comparison to regular `TDM` (Figure 1), `TDMds` is more efficient in this example. The last request A_2 completes 3 `TDM` slots earlier. Note, that the slack counters are all reset to zero for subsequent jobs of the critical tasks A and B.

As illustrated by the example above, the use of deadlines and slack counters allows the `TDMds` arbiter to improve the utilization of the memory. However, issues stemming from the very nature of `TDM` remain, both are related to the use of fixed `TDM` slots as the request arbitration only occurs at the beginning of slots.

Like standard `TDM`, `TDMds` remains non-work conserving, i.e., issued requests may not be able to access memory, even when it is idle. Figure 2 shows three such requests, namely A_0, B_1, and c_0. The arbitration of `TDMds` is limited to `TDM` slots and thus cannot immediately grant request c_0 access to the memory in slot 4. Instead, it has to wait until the beginning of the next `TDM` slot 5, where it is indeed scheduled.

Definition 1 During the execution of a task set under TDMds, the **issue delay** denotes the number of clock cycles during which at least one request was pending at the memory arbiter within an unused TDM slot.

For instance, TDMds generates an issue delay of 6 cycles for request c_0. This delay could have been avoided if requests were handled independently from TDM slots. For instance, task A, the owner of TDM slot 5, has accumulated some slack (8Δ), which ensures that the deadline of any request issued by task A after request c_0 will be at the end of slot 7 or later. It thus would be safe for request c_0 to immediately access the memory during the unused TDM slot 4. This also holds if the request stretches partially into the next slot, since any potential request from task A could still meet its deadline at A's next TDM slot.

Another issue is related to the length of TDM slots, which has to be chosen such that the longest possible memory access can safely complete. Most memory accesses will, in practice, complete way earlier than this worst-case memory latency. Requests B_0 and c_1 in Figure 2 are issued close to the end of a used TDM slot. It is thus likely that the requests serviced by the memory at the respective issue dates have already completed. The TDMds arbiter, however, is limited to TDM slots and cannot immediately grant access to a subsequent memory request even when the current memory request completes early.

Definition 2 During the execution of a task set under TDMds the **release delay** denotes the number of clock cycles during which at least one request is issued to the memory arbiter after the completion of the memory request of a used TDM slot.

Similar to the issue delay before, it is possible to avoid this delay by decoupling the arbiter from the TDM slots. A complete approach is presented in the next section.

4 TDM Arbitration without Slots

We now develop a solution to the issues highlighted above. First, we show how the arbitration can be decoupled from TDM slots in order to reduce issue delays by considering slack counters. We then propose a slight variation of the approach to address *release delays*. Both approaches improve the memory utilization, while converging to regular TDM in the worst case (see Section 5). This allows to preserve properties that make TDM popular – including results obtained from advanced TDM-based program analyses (Kelter et al., 2014; Rihani et al., 2015).

4.1 Early Request Processing

Under TDMds, arbitration decisions are taken at the beginning of TDM slots and are based solely on the set of actually issued requests. It is then possible to delay an issued request of a critical task, depending on the request's actual deadline. The task's slack counter itself is not considered during this arbitration decision, it merely has an indirect effect during the calculation of the request's deadline. This can be seen as a forecast, based on the actual requests visible to the arbiter.

However, the slack counter values are also valid when a critical job did not (yet) issue a request to the arbiter. This, in fact, allows an arbiter to take a peek into the near future and take arbitration decisions based on this information. In particular, it is possible to determine a lower bound of the deadline associated with any request coming from the owner of the

Algorithm 1 Condition to apply the *early-start* optimization.

```
1: function EARLY-START(Now, NextSlot, Request)
2:     NextOwner = OWNER(NextSlot)
3:     NextDist = START(NextSlot) − Now
4:     if NextOwner = OWNER(Request) then
5:         return true
6:     else if NextDist < SLACK(NextOwner) then
7:         return true
8:     return false
```

immediate *next* TDM slot (even when the job did not yet issue a request). The memory can then start the processing of any of the issued requests at any moment, if that deadline bound lies past the end of the next TDM slot. This ensures that the memory can process the request partially in the current TDM slot, while completing it in the next slot, without violating the worst-case behavior of TDM. We call the resulting approach TDMes for *early start*.

Two cases have to be considered by the arbiter before applying the early-start optimization to a request, as shown by Algorithm 1. Helper functions are used to retrieve the owner of the next TDM slot or request (OWNER), the start cycle of the TDM slot (START), and the slack counter of a task (SLACK).

The first condition (Line 4) checks whether the task issuing the request owns the upcoming slot (only for *critical* tasks). In this case it is always safe to immediately start processing the request, as the memory will always respect the request's deadline – an overflow into the next TDM slot is not an issue.

The second condition (Line 6), verifies that any potential overflow into the next TDM slot is safe, using the slack counter of the task owning the upcoming slot. Recall that the deadline under TDMds is computed from a delayed issue date, i.e., the issue date plus the value of the slack counter. We can do the same to obtain a lower bound of the deadline, by simply assuming that the owner of the TDM slot may issue a request in the next clock cycle. If the deadline bound corresponds to the end of the immediate next TDM slot, it is not safe to overflow and the memory cannot start processing any request (yet). If the deadline bound lies farther in the future, an overflow is safe and the memory can proceed.

Instead of actually computing the deadline bound, it suffices to compare the distance to the beginning of the next TDM slot (NextDist) with the slack counter of the slot's owner

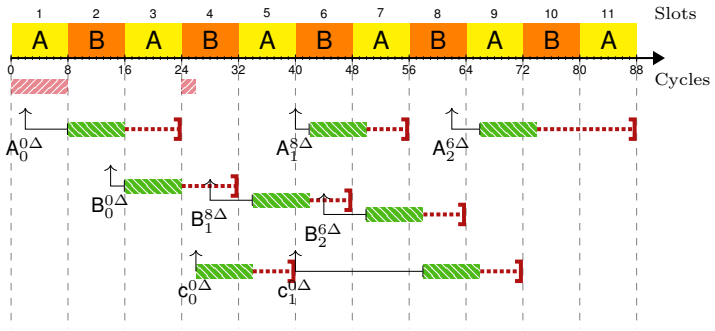


Fig. 3 Reduced issue delays due to the TDMes arbiter, which operates independently from the actual TDM slot length.

(SLACK). If the distance is smaller than the slack counter value, the delayed issue date lies after the beginning of the TDM slot and the deadline correspond to the TDM slot thereafter – the early-start optimization can be applied. If the distance is larger or equal to the slack counter an overflow might be problematic – the optimization cannot be applied.

Figure 3 illustrates the resulting arbitration under TDMes for the task set from before $(\mathsf{A},\mathsf{B},\mathsf{c})$. Requests can now start early, if the conditions described before are met. This is the case for request c_0, a *non-critical* request that is issued during TDM slot 4 at cycle 26. The owner of the next TDM slot starting at cycle 32, is task A, whose slack counter is 8 (stemming from access A_0). At the moment when c_0 is issued, the arbiter thus needs to verify the second condition of Algorithm 1. Task A could potentially issue a request in the next cycle (27), which would yield a delayed issue date of 35 $(27+8)$ and consequently a deadline at the end of TDM slot 7. It is evidently safe to immediately grant c_0 access to the memory, as shown in the figure. The same result can be obtained by comparing the distance $(32-26=6)$ to the next TDM slot with A's slack counter (8), since $6<8$. The same situation arises for request c_1. The arbiter handles request c_1 in the next clock cycle after 4 requests where served, i.e. at cycle 59 $(26+4*8+1)$, leading to a distance to the next slot smaller than A's slack counter. The remaining requests in the example, except for A_0, fall into the first condition of Algorithm 1, i.e., the owner of the request is also the owner of the subsequent TDM slot. For instance, request B_0 is processed within slot 3, as B owns slot 4. The early-start optimization cannot be applied to A_0, since task B, the owner of the next TDM slot (2), has a slack counter value of 0. The request thus suffers from an issue delay of 6 cycles, which indicates that our approach may still exhibit non-work-conserving behavior.

Compared to TDMds (Figure 2), the TDMes policy is again more efficient. The memory is almost always busy and the last request completes at cycle 74 (as opposed to 80 before).

4.2 Early Release after Request Completion

Up to now, the actual behavior of the memory to handle requests (load or store) was irrelevant to the memory arbiter, which simply relied on the fact that all memory requests are guaranteed to complete within the duration of a TDM slot. This, in essence, means that all memory accesses take the worst-case latency, which may introduce considerable pessimism in the form of *release delays*, as the actual memory latency typically varies from access to

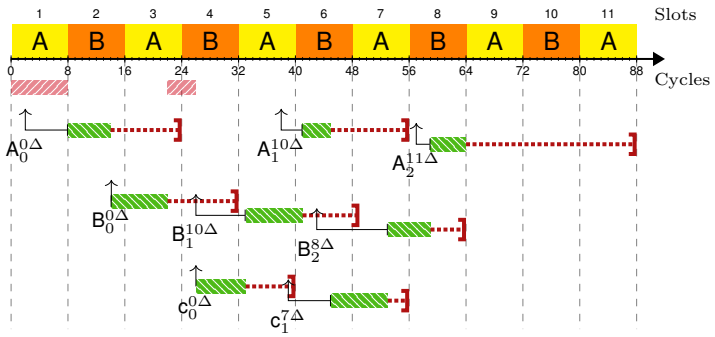


Fig. 4 Elimination of release delays under TDMer arbitration, which considers the actual latency of memory access. Some of the *eliminated* release delays may simply be transformed into issue delays.

access, depending on the internal state of the underlying memory technology, e.g., DRAM (Wu et al., 2013).

The memory processing under the previously presented approach is no longer required to be aligned with the TDM schedule and can perform memory accesses at any moment. It is thus only natural to drop the (artificial) constraint of waiting the entire duration of a TDM slot before *releasing* the memory and allowing the next request to be processed. We refer to this arbitration scheme as TDMer (for *early release*), which entirely eliminates any release delays present under TDMds or TDMes. However, it is not guaranteed that any of the issued requests is granted access to the memory, e.g., due to the lack of slack (see Section 4.1). In this case, the early-release optimization does not actually improve the memory utilization and release delays are simply transformed into issue delays.

Figure 4 again shows an execution trace for the task set $(\mathsf{A},\mathsf{B},\mathsf{c})$ from Section 2 under the TDMer scheme. The main difference concerns the duration of the memory processing time, which is now illustrated by green hatched bars of variable length ( ). Request A_0, for instance, now completes 2 cycles earlier than before. This entails several changes. Firstly, the slack counter of task A increases by an additional 2 cycles, which now amounts to 10 cycles (cf. $\mathsf{A}_1^{10\Delta}$). Secondly, request B_0 can be processed right after being issued, which eliminates the release delay that would otherwise be observed. In this example, the reduced release delay itself is not beneficial, due to the absence of issued requests before cycle 24. Finally, due to the earlier processing of requests from the two other tasks and a reduced memory latency, request A_1 completes in TDM slot 6 instead of TDM slot 7. This allows the arbiter to change the order of request c_1 and B_2, since B_2's deadline is far enough in the future and task A has sufficient slack (11).

In comparison to TDMes (Figure 3), again an improvement is achieved, despite a slight increase in the memory's total idle time (cf. the red hatched bars ( ). The last request (A_2) completes at cycle 64, as opposed to cycle 74 before. In particular the non-critical task c gained from the altered schedule and completes its last memory request 13 cycles earlier. Only task B does not profit and terminates at the same instant. Note, however, that critical tasks never terminate later than under a regular execution under TDM. This can be seen by the fact that the deadlines for critical requests in Figures 2 through 4 match.

The improvements are even greater when comparing with the original TDM-based execution (Figure 1), which completed after 104 cycles. TDMer yields an improvement of 62.5% – while, in the worst case, preserving a strict separation between critical and non-critical tasks. Note, that the deadlines shown in Figure 1 are not comparable to the deadlines shown in the other figures, due to the presence of a third TDM slot for C.

5 Worst-Case Behavior

In the previous sections we claimed that for critical tasks our approach *converges* towards TDM in the worst case. We will now provide a more precise definition of this *worst-case behavior* and provide formal proofs of correctness. Since non-critical tasks are served on a best-effort basis, we do not consider them here.

By converging towards TDM we simply mean that TDMer (as well as TDMes) provides the following guarantee:

Theorem 1 *(Worst-Case Behavior) Considering a given execution (i.e., execution path, runtime conditions, input values, ...) a memory access of a critical task under any possible execution considering TDMer completes no later than the same execution under strict TDM.*

This is a very strong guarantee, which preserves many of the properties that make TDM popular in critical systems. This includes results of worst-case execution time analyses, even sophisticated analyses that exploit information on the relative alignment of program execution with regard to the TDM schedule (Kelter et al., 2014; Rihani et al., 2015).

In order to show the correctness of Theorem 1 we will first refine some essential definitions and show that deadlines of critical tasks under TDM are always aligned with TDM slots and unique between critical tasks. Based on this, we will finally show that TDMer preserves the same deadlines using slack counters.

TDM arbitration guarantees a fixed time window to a task to exclusively access memory, this time window is defined by the TDM slot length Sl. Each of the n critical tasks has its own slot, resulting in a repetitive TDM schedule with a period $P = n \cdot Sl$. For each memory request issued by a critical task τ_i the arbiter is assumed to store or compute the following information: the request's arrival date (aka. issue date) a_k, completion date c_k, and deadline d_k as well as the start date of the current TDM period Sp, and the offset of the task's TDM slot $O(\tau_i)$ (with regard to Sp).

The deadline of a memory request under TDM is then defined as follows:

Definition 3 (TDM Request Deadline) Considering TDM arbitration, the deadline d_k of the kth request issued by a critical task τ_i is given by:

$$d_k = \begin{cases} Sp + O(\tau_i) + Sl & \text{if } a_k \leq Sp + O(\tau_i) \\ Sp + O(\tau_i) + P + Sl & \text{else.} \end{cases}$$

Lemma 1 *Given Definition 3, the deadline of a critical request corresponds to the end date of its dedicated TDM slot. The deadline d_k is consequently always equal to the completion date c_k under regular TDM arbitration.*

Proof The deadline d_k always corresponds to the end of τ_i's TDM slot, as each of the arguments used in the computation is, by definition, a multiple of the TDM slot length Sl (Sp, $O(\tau_i)$, and P). The formula simply distinguishes two cases (1) when the request is issued at or before the start of τ_i's TDM slot, the deadline then corresponds to the end of the TDM slot in the *current* period, or (2) when the request is issued after the start of its TDM slot, the deadline then corresponds to the end of the TDM slot in the *next* period. □

Lemma 2 *The deadlines of critical requests issued by different critical tasks can never be identical.*

Proof This follows trivially, since, by definition, the offsets of two critical tasks $O(\tau_i)$ and $O(\tau_j)$ have to be different when $i \neq j$. □

The two properties from above show that TDM arbitration can simply be interpreted as being driven by deadlines. Any dynamic arbiter respecting these deadlines (i.e., $c_k \leq d_k$) can be used to implement a TDM-based arbitration scheme that preserves Theorem 1, e.g., an implementation based on the earliest-deadline-first strategy like TDMer (see Theorem 2). However, since our approach here is decoupled from the notion of TDM slots once sufficient slack has been accumulated, we also have to show that the deadlines under our approach match those of TDM.

Earlier completion of tasks gives rise to the accumulation of slack w.r.t an execution under regular TDM, which is stored in a dedicated slack counter for each critical task. These slack counters are updated after every completion of a critical memory request.

Definition 4 (Slack Accumulation) Under TDMer, the slack counter of a critical task τ_i after the completion of its kth memory request is given by: $\Delta_k = d_k - c_k$.

The slack is then used to compute a delayed issue date for the next request of τ_i. Depending on the amount of slack accumulated at this moment, this delayed issue date may push the deadline farther into the future and thus provide more flexibility to a dynamic arbiter.

Lemma 3 *The deadlines for critical requests under* `TDMer` *correspond to the same deadlines as under regular* `TDM`*.*

Proof Considering Definition 4, we can show by induction that a request's issue date $a_k^{\texttt{TDM}}$ under regular `TDM` always corresponds to the delayed issue date under `TDMer`. This last date can be computed from the original issue date $a_k^{\texttt{TDMer}}$ and the slack counter value Δ_k, i.e., $a_k^{\texttt{TDM}} = a_k^{\texttt{TDMer}} + \Delta_{k-1}$.
Induction base $k = 0$: $\Delta_0 = 0$ the issue date under `TDM` and `TDMer` are naturally the same, i.e. $a_0^{\texttt{TDM}} = a_0^{\texttt{TDMer}}$.
Induction step: Assuming a composable architecture that ensures that the delay $dist_k$ between the k-th and $(k-1)$th memory request is constant we obtain:

$$dist_k = a_k^{\texttt{TDM}} - c_{k-1}^{\texttt{TDM}} = a_k^{\texttt{TDMer}} - c_{k-1}^{\texttt{TDMer}}$$

Based on the hypothesis that the deadline of the previous request is equal under `TDM` and `TDMer`, i.e., $d_{k-1}^{\texttt{TDM}} = d_{k-1}^{\texttt{TDMer}}$, and the fact that deadline and completion date are identical under `TDM`, we obtain by substitution that:

$$\begin{aligned}
\Delta_{k-1} &= d_{k-1}^{\texttt{TDMer}} - c_{k-1}^{\texttt{TDMer}} = d_{k-1}^{\texttt{TDM}} - c_{k-1}^{\texttt{TDMer}} = c_{k-1}^{\texttt{TDM}} - c_{k-1}^{\texttt{TDMer}} \\
&= c_{k-1}^{\texttt{TDM}} + dist_k - c_{k-1}^{\texttt{TDMer}} - dist_k \\
&= c_{k-1}^{\texttt{TDM}} + (a_k^{\texttt{TDM}} - c_{k-1}^{\texttt{TDM}}) - c_{k-1}^{\texttt{TDMer}} - (a_k^{\texttt{TDMer}} - c_{k-1}^{\texttt{TDMer}}) \\
&= a_k^{\texttt{TDM}} - a_k^{\texttt{TDMer}} \\
&\Longrightarrow a_k^{\texttt{TDM}} = a_k^{\texttt{TDMer}} + \Delta_{k-1}
\end{aligned}$$

Since the issue date $a_k^{\texttt{TDM}}$ and the delayed issue date $a_k^{\texttt{TDMer}} + \Delta_{k-1}$ are identical, and we use the same method to compute the deadlines, it follows that the deadlines are identical, i.e., $d_k^{\texttt{TDM}} = d_k^{\texttt{TDMer}}$. □

The result from above shows that the deadlines of memory requests under `TDMer` correspond to those under `TDM`. It remains to show that the arbiter is actually able to respect those deadlines. For this we also need to consider non-critical tasks and their respective memory requests.

Definition 5 (Non-Critical Requests) The deadline of non-critical requests under `TDMer` corresponds to the end of the immediate next `TDM` slot after the issue date of the request, independent from the actual owner of that slot, i.e.:

$$d_k = \left\lceil \frac{a_k}{Sl} + 1 \right\rceil \cdot Sl$$

The deadlines of non-critical requests may obviously collide with deadlines of critical tasks. The arbitration policy thus has to take these collisions into account.

Definition 6 (Request Arbitration) Under `TDMer` request arbitration is based on a priority queue depending on the requests' deadlines. In case of a tie between a critical task and (possibly many) non-critical tasks, the critical request is assigned higher priority. Deadlines of non-critical tasks are reevaluated after each `TDM` slot.

Definition 7 (Request Admission) The request with the highest priority is granted access to the memory at the granularity of individual clock cycles according the EARLY-START test from Algorithm 1.

Theorem 2 *TDMer ensures that any critical request completes before its deadline, i.e., $c_k \leq d_k$, in addition to Theorem 1.*

Proof Assume that request k of a critical task τ_i is the first critical request in the system that missed its deadline, i.e., its $c_k > d_k$. This means that the memory was busy processing another request at the beginning of τ_i's TDM slot.

First assume the case that the request l of another task τ_j being processed by the memory was granted access to the memory at instant t after the issue date of request k, i.e., $a_k \leq t$. This implies that the deadline d_l is either smaller or equal to d_k, otherwise k would have higher priority and l would not have been admitted first. The deadlines cannot be equal, as this would imply that l is a non-critical task (Lemma 2), which again would exclude its admission due to its weaker priority. It follows that $d_l < d_k$ and, due to Lemma 1, $d_l \leq d_k - Sl$. This, however, would imply that the request l missed its deadline as it was still being processed at the beginning of τ_i's slot. This contradicts the assumption that k was the first to miss its deadline.

Now assume the case that request l was granted access to the memory before the issue date of k, i.e., $t < a_k$. This implies that the early-start optimization was applied. More precisely the second condition (Line 6), as l cannot originate from τ_i. However, this is impossible since $t < a_k + \Delta_k < d_k - Sl$ has to hold, while the early-start admission test at the same time requires $a_k + \Delta_k > d_k - Sl$. □

It is thus impossible that any critical request misses its deadline, which, in addition, is equal to that of an execution under TDM (Lemma 3). Non-critical requests are, as expected, potentially subject to deadline misses.

6 Simulation Experiments

In this section, we evaluate the previously presented dynamic TDM arbitration mechanisms by simulating the concurrent execution of synthetic tasks. We first present the experimental setup and then compare the various approaches using the issue and release delays as well as the memory utilization.

6.1 Experimental Setup

Our simulation framework is based on the system model presented in Section 2. This framework allows us to collect execution traces and statistics from the concurrent execution of n periodic tasks, each executing on a separate core, and competing for a central shared memory. The framework does not model the actual computation performed by the tasks. However, it simulates the memory accesses and their arbitration considering four different arbitration schemes: (1) TDMfs, a variant of regular TDM where non-critical tasks may reclaim unused slots, (2) TDMds, a dynamic TDM-based arbitration policy respecting TDM slots, (3) TDMes, a dynamic approach decoupled from TDM slots (see Section 4.1), and (4) TDMer, a refinement of TDMes addressing release delays (see Section 4.2).

All experiments are based on randomly generated synthetic task sets obtained via *UUniFast* (Emberson et al., 2010) and a memory traffic generator. The goal is to obtain a large number of simulations that reflect a realistic behavior of real task sets considering different parameters, such as task periods, worst-case execution times, memory load, and total system utilization.

Task Set Generation: The *UUniFast* algorithm allows to randomly generate tasks for a task set Γ based on two input parameters n and U, where n specifies the total number of tasks and U the total system utilization desired. The algorithm then generates n different utilization values $\{u_1, u_2, \ldots, u_n\}$, one for each task τ_i, while the sum of these tasks utilizations equals the system utilization U. From the task utilization parameters, the task periods T_i are generated. Note that we constrain our system to harmonic periods, which ensure that hyper-periods and therefore simulation times remain reasonable. The period of the first task T_1 is assumed to be 20ms,[1] while all other periods are random multiples of T_1, i.e., $T_i = k * T_1, 1 < i \leq n$, where k is 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, $hp = \text{LCM}_{1 \leq i \leq n}(T_i)$, we then derive the worst-case execution time of each task $C_i = T_i \cdot u_i$ and the number of jobs for each task, i.e., $J_i = hp/T_i$. The tasks $\tau_i \in \Gamma$ of the final task set are thus represented by a triple $\tau_i = (C_i, T_i, J_i)$. We assume implicit deadlines, i.e., task deadlines are equal to the task periods T_i.

Traffic Generator: The simulation framework then requires a specification of each task in terms of memory accesses (cf. $dist_k$ in Section 2). The memory access sequences are thus obtained from a traffic generator for each job in the task set generated by *UUniFast*.

The aim is to obtain synthetic tasks whose memory patterns are similar to real applications. We thus exploit the applications from the MiBench benchmark suite (Guthaus et al., 2001) in order to calibrate the traffic generator. The MiBench benchmarks were first executed individually on the Patmos architecture (Schoeberl et al., 2011) using a cycle-accurate simulator. The simulator was extended to collect traces from actual program executions matching our system model. The traces were collected for a Patmos hardware configuration based on a 5-stage in-order single-issue pipeline, a 32 KB *method cache* using the *LRU* replacement policy on 32 entries with a cache block size of 32 bytes, a 256 byte *stack cache* with block size 4, a 32 KB write-through *data cache* with *LRU* replacement on 4 sets and 32 byte blocks. Memory latencies are ignored during this trace collection.

The memory access patterns captured by the collected traces are then analyzed with regard to their statistical properties. Most notably, we were interested in the distribution of the distance (in clock cycles) between consecutive memory accesses. Our experiments show that the *Generalized Extreme value Distribution* (GEV) (Hansen et al., 2009) fits well to the data from the collected trace. Based on the empirical trace data and the parameters of distribution functions fitted to this data, we defined a parameters space in order to generate the memory access sequences for the jobs obtained through *UUniFast*. Note that the distribution incidentally describes whether a benchmark is memory intensive or rather compute bound.

In order to obtain a memory access sequence for a job, the traffic generator first randomly chooses the GEV distribution parameters and then generates memory accesses and the respective distance (cf. $dist_k$ in Section 2) between them. Note, however, that the generated memory accesses have to be consistent with the task's worst-case execution time C_i.

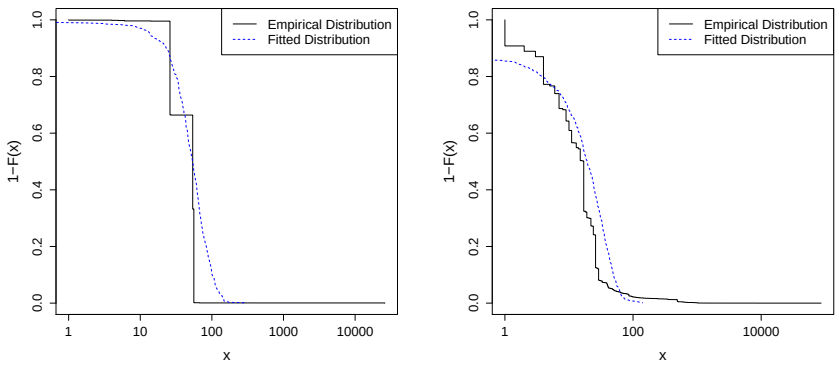
[1] Note that it does not matter whether the first task period T_1 is chosen randomly or is fixed, as in our case.

The generator thus tracks the evolution of a worst-case execution time bound as it proceeds. For each memory access, we add to this bound the worst-case latency for a newly generated request, which is bounded by $P + Sl - 1$ cycles. The generator simply stops once the bound reaches the task's C_i. The execution times of the synthetic tasks thus rather closely approach the tasks' worst-case execution times. This is a rather pessimistic view, inducing a higher memory load than can be expected from average-case executions, as the actual execution times of critical tasks are known to be significantly lower than their worst-case execution times. Note that we capture this effect in our experiments by varying the system load.

6.2 Generated Memory Profiles

The GEV distribution unifies three standard extreme value distributions, namely the Fréchet, Weibull, and Gumbel distributions. GEV is characterized by three parameters the location (μ), scale (σ), and shape (ξ). We used the function *GumbelFit*, from the *fExtreme* (Wuertz et al., 2017) package of the R statistical computing environment, in order to fit the parameters μ, σ, and ξ to the trace data in Subsection 6.1. The shape parameter determines which of the three standard distributions is chosen. The traces of most MiBench benchmarks are best described by a Fréchet distribution ($\xi > 0$). Figure 5 shows the inverted empirical distribution functions of the request distances (in cycles) from two application traces (`rawdaudio` and `cjpeg-small`) and compares them to the cumulative distribution function of the fitted GEV distribution. As can be seen the fitted distributions nicely describe the behavior of benchmarks, while providing a convenient abstraction for the use in our traffic generator.

The traffic generator adds memory requests to the jobs of a task τ_i until the worst-case execution time C_i, provided by UUniFast is reached. Assume that the number of memory request for a job j is given by $NbrAcc_i^j$. This allows us to characterize the job's behavior by comparing the processor demand PD_i^j and memory demand MD_i^j, which together must not exceed the task's WCET, i.e., $PD_i^j + MD_i^j \leq C_i$. These parameters emerge from the generated



(a) `rawdaudio`. (b) `cjpeg-small`.

Fig. 5 Empirical distributions of two MiBench applications compared with the GEV distributions after fitting.

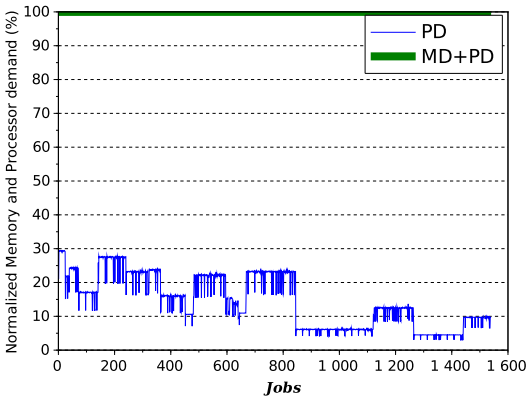


Fig. 6 Comparison of the memory (MD) and processor (PD) demand for the approx. 1500 jobs of the simulation runs with 24 tasks.

memory sequence of a job as follows:

$$PD_i^j = \sum_{k=0}^{NbrAcc_i^j} dist_k$$

$$MD_i^j = (P + Sl - 1) \cdot NbrAcc_i^j.$$

Here, $dist_k$ refers to the distance between the k-th and $(k-1)$-th memory access of a job of task τ_i. Figure 6 illustrates stacked plots of memory and processor demand normalized to the WCET (C_i) for a particular configuration of our simulation runs considering 24 tasks. As can be seen the combined memory and processor demand almost always reaches the specified WCET (C_i), which represents a system with relatively high load compared to its worst-case behavior.

6.3 Simulation Parameters

Based on the generated task sets and the corresponding memory access sequences, we performed a considerable number of simulations, by varying the number of tasks/cores (4, 8, 12, 16, 20, 24) and the global system utilization (between 10% and 100% in steps of 10). Note that the system utilization is normalized to the number of cores in the system. Following previous work in the context of mixed-criticality systems (Baruah et al., 2011; Jan et al., 2013), we chose two scenarios concerning the repartitioning between critical and non-critical tasks: (1) 25% critical and 75% non-critical tasks and (2) an equal repartitioning of 50%/50% between critical and non-critical tasks. For each configuration 10 simulation runs were performed, resulting in 1080 runs overall and several thousand simulated job instances. In order to have comparable results between different task sets and arbitration approaches, the duration of each simulation is limited to the task set hyper-period – potentially terminating the simulation before all non-critical tasks have completed (e.g., in cases when these tasks missed their deadlines).

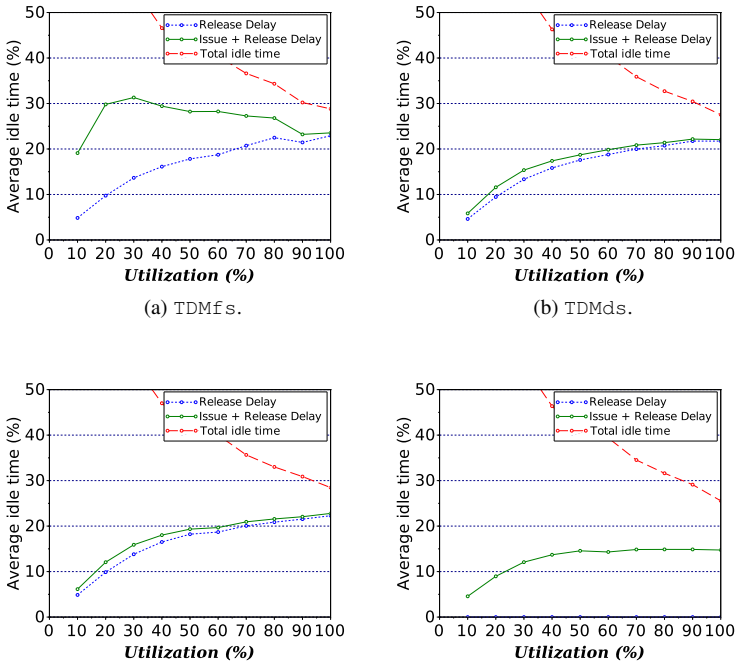
The duration of a `TDM` slot length Sl, corresponds to an upper bound of the memory access latency previously determined on a Terasic DE-10 Nano evaluation board that is equipped with an Intel Cyclone V SoC-FPGA and 1 GB of DDR3 memory. A single Patmos

processor running at 100 Mhz was implemented in the FPGA and performed memory accesses in isolation via the multi-port memory controller provided by the SoC (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 all simulation runs we thus consider a `TDM` slot length of 40 cycles. For `TDMer`, which supports an early release of the memory when completing a memory request faster than the `TDM` slot length, we simulate a varying memory latency obtained from a uniform random distribution in the range $[21,40]$ clock cycles. The slack counters for `TDMds`, `TDMes`, and `TDMer` are reset at the beginning of each job.

6.4 Results for Dynamic TDM-based Arbitration

The first set of figures represents an overall comparison of the various `TDM`-based approaches over all simulation runs.[2] Figure 7 shows a breakdown of the average memory idle time from all runs due to (1) the total release delays, (2) the total issue delays, and (3) the total number of cycles without any memory requests issued to the arbiter ("No request").

[2] We choose to trim Figure 7 in order to achieve a better visualization of the impact of our arbitration policies on the issue and release delays.



(a) `TDMfs`. (b) `TDMds`.

(c) `TDMes`. (d) `TDMer`.

Fig. 7 Evolution of memory idle time, issue, and release delays over all simulations under varying utilization (lower is better).

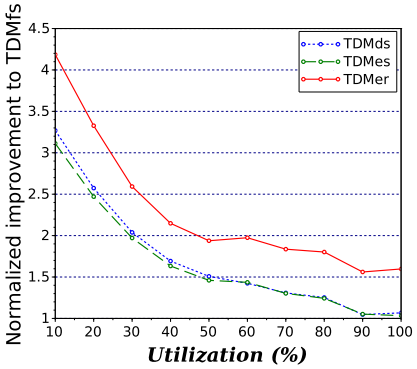


Fig. 8 Normalized sum of issue and release delays for dynamic arbitration schemes compared to `TDMfs` (higher is better).

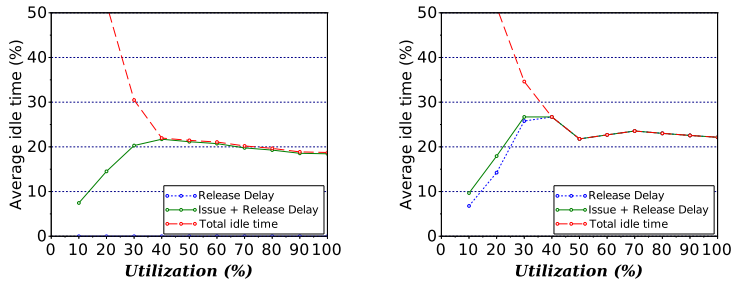
The plotted lines are stacked, i.e., the red line represents the sum of all three forms of memory idling, while the green line represents the sum of the issue and release delays. The idle times are normalized to the total trace length of the simulation.

Looking at the green lines (issue + release delays) reveals that the dynamic arbitration schemes are quite successful in eliminating issue delays. In comparison to `TDMfs` (i.e., regular non-dynamic `TDM`), the distance between the green and blue lines are relatively small. The distance, and thus the issue delays, diminish as system and memory load increases. Release delays thus represent a considerable source of inefficiency for the `TDMds` and `TDMes` approaches. This does not apply to `TDMer`, which completely eliminates release delays. However, as can be seen a non-negligible portion of these release delays are merely transformed into issue delays. Overall, however, `TDMer` achieves considerable improvements in terms of delays due to the non-work-conserving nature of `TDM`.

Comparing the combined impact of release and issue delays (green line), one can see that these typically represent more than 25% of the simulated total execution time for regular `TDMfs`, while it hardly exceeds 15% for `TDMer`. Note that the remaining issue delays stem from situation where requests cannot be scheduled immediately, due to insufficient slack of the critical task owning the immediate next `TDM` slot. It appears that this can be explained due to the absence of slack at the beginning of jobs (see Section 6.5).

Figure 8 shows the relative improvement with regard to the combined issue and release delays of the dynamic arbitration schemes compared to `TDMfs`. We can observe considerable improvements of up to a factor of 3.3 for `TDMds` and `TDMes`, which both follow a very similar trend. However, under very high system utilization ($\geq 90\%$) these approaches do not perform better than regular `TDMfs`. This can be explained by the high memory utilization from critical tasks, which can lead to starvation for non-critical tasks. Recall, though, that the traffic generator results in traces that represent comparatively high load for all tasks. We do not expect that realistic real-time systems actually exhibit such a high load. `TDMer` even outperforms the other approaches and exhibits improvements of up to a factor of 4.2 and remains above 1.5 even at very high utilization.

A general trend common to all approaches is that the total idle time decreases as the system utilization increases. This can be explained by the increasing number of memory requests that are issued by the tasks in the system. The average memory idle time over all simulation configurations hardly drops below 30%. The various approaches have little impact on the total amount of idling, except for `TDMer` which for higher load ($> 60\%$)



(a) TDMer. (b) TDMfs.

Fig. 9 Memory idle time considering 24 tasks with 6 critical and 18 non-critical tasks (lower is better).

shows improvements of a few percent. This is not surprising, as more efficient memory arbitration tends to reduce the execution time of jobs and thus tends to increase the gaps of inactivity between task activations.

A closer look at Subfigure 7d shows that memory is idle due to the absence of requests (cf. the distance between the red and green lines). For TDMer this amounts to more than 10% for a system utilization of 100%. The configuration with 24 tasks in total (6 critical, 18 non-critical) showed the highest level of memory contention in our experiments. The high load is caused by the high number of tasks, which is exacerbated by the low number of TDM slots. Recall that the number of TDM slots has an impact on the period P, which incidentally increases the number of memory accesses that fit into the worst-case execution time of a task – here most notably of the non-critical tasks.

Figure 9 shows detailed results for this configuration in isolation for both TDMer and TDMfs. As can be seen memory idling drops rapidly as system load increases and levels off at about 20% and 25% of the simulated trace lengths respectively. For a system load above 40% the memory idling of TDMfs is exclusively due to release delays. This suggests that all TDM slots are used and memory utilization is limited by the arbiter. For TDMer these release delays are mostly transformed into issue delays – which implies that at least one memory request is constantly pending at the arbiter. This indicates that the slack counter values are too low, which prohibits the early-start optimization.

6.5 Results for Dynamic TDM with Initial Slack

We have seen in the experimental results that the TDMer approach completely eliminates release delays. However, as can be seen in Figure 7, a non-negligible portion of these release delays are merely transformed into issue delays. These remaining issue delays stem from situations where requests cannot be scheduled immediately, due to insufficient slack of the critical task owning the immediate next TDM slot. This is particularly visible for the configuration with 24 tasks (Figure 9). For runs of this configuration using TDMer, memory idling is exclusively caused by issue delays for utilization levels above 40%. This implies that at least one memory request is constantly pending at the arbiter and that the slack counter values of critical tasks are often too low to apply the early-start optimization. Note that this behavior has to appear systematically throughout the entire simulation run or otherwise a noticeable difference between memory idling and issue delays would appear. It appears that memory load is too high for critical tasks to accumulate slack under these circumstances.

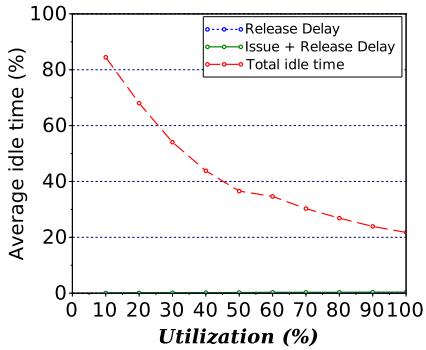


Fig. 10 Evolution of memory idle time, issue, and release delays over all simulations with varying utilization under `TDMer` with slack counters initialized to *Sl* at job start (lower is better).

Our hypothesis is that the remaining issue delays can be eliminated by supplying an initial slack when critical jobs start.

We thus slightly modified the experimental setup from Subsection 6.1. Instead of resetting the slack counter to zero at the start of critical jobs, we reset it to the `TDM` slot length *Sl*, i.e., 40 cycles. This initial slack promises to resolve the aforementioned issues, since it represents the minimum amount of slack required to enable the early-start optimization right from the beginning of the simulation. This is, however, associated with a potential increase of the task's execution time by at most one `TDM` period, which needs to be accounted for in the task's WCET (C_i), i.e., this is equivalent to the overhead of a single additional memory accesses under strict `TDM`. This additional *virtual* memory accesses can then be taken into consideration for the correctness proof from Section 5, e.g., by adapting the base case accordingly.

Figure 10 shows a breakdown of the average memory idle time from all runs using the `TDMer` arbiter. The results appear to confirm our hypothesis, the remaining issue delays are almost eliminated and typically represent less than 0.5% of the simulation trace length. Providing a small amount of initial slack thus effectively rendered our approach work-conserving – while still retaining all the advantages mentioned previously.

This also applies to the configurations with 24 tasks, which are shown separately in Figure 11. In particular, we observe that the arbitration policy no longer limits memory

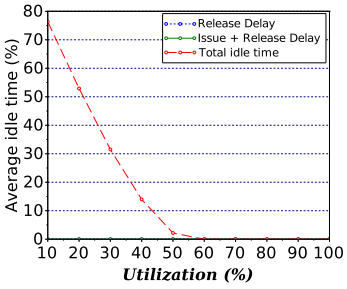


Fig. 11 Memory idle time considering 24 tasks with 6 critical and 18 non-critical tasks, under `TDMer` with initial slack set to *Sl* (lower is better).

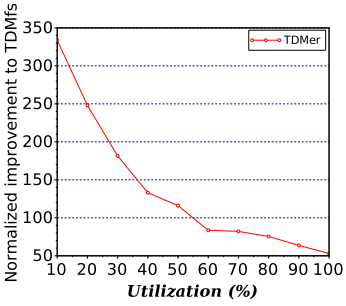


Fig. 12 Normalized sum of issue and release delays for dynamic arbitration schemes compared to `TDMfs`, with initial slack set to *Sl* (higher is better).

utilization. For high system utilization ($\geq 60\%$) memory now truly becomes saturated. This is an interesting observation, as it indicates that the critical tasks *do not* loose slack over time. The slack of individual critical tasks may drop, even become zero, for short periods of time, but is at least preserved on the long run. This has to be true or otherwise a noticeable level of issue delays would eventually manifest.

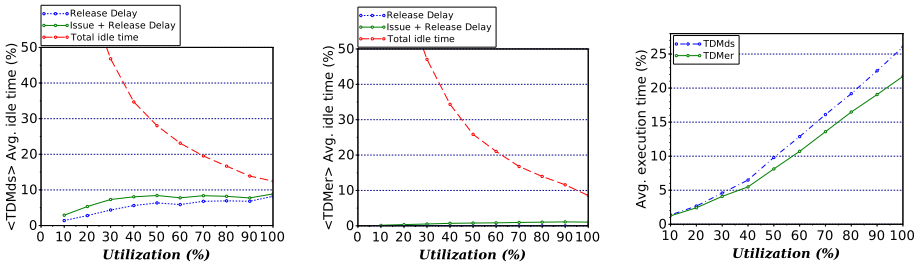
Figure 12 shows the relative improvement with regard to the combined issue and release delays of `TDMer` when an initial slack counter value of a single `TDM` slot length is provided at the start of jobs of critical tasks. The measurements are normalized against `TDMfs`. We observe considerable improvements for `TDMer` of up to a factor of 350 and even at high levels of memory utilization the improvements remain above a factor of 50.

From this evaluation, we conclude that `TDMer` is successful in eliminating release delays and significantly reducing issue delays, even when the memory bandwidth is close to saturation, as shown in Figures 9 and 11. The dynamic arbitration policy combined with slack counters hence allows to decouple the arbiter from constraints imposed by the slots of regular `TDM`, while offering a very fine granularity of memory arbitration and preserving `TDM`'s guarantees for *critical* tasks.

6.6 Results for Varying Memory Access Latencies

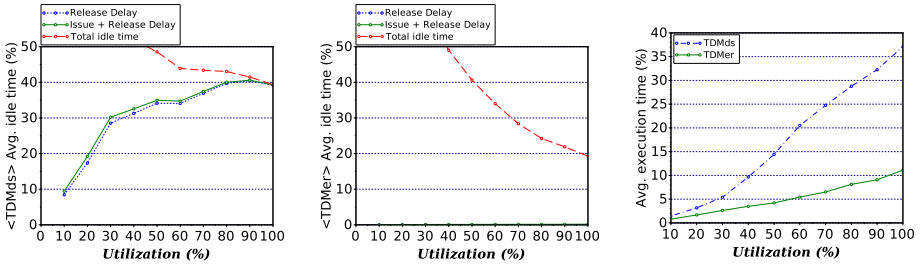
In the previous experiments we assumed a fixed `TDM` slot length of 40 cycles, corresponding to the memory access latency of a DDR3 memory. Hassan (2018) showed that DDR DRAMs, when trying to provide predictable memory behavior, suffer from highly variable access latencies and overly pessimistic latency bounds. Therefore, when targeting real-time systems with strict timing constraints, Hassan (2018) proposes an alternative off-chip memory solution, based on Reduced Latency DRAMs (RLDRAMs). Access latencies for RLDRAMs are generally lower and, in addition, exhibit less variability. In the following experiments we are thus interested in evaluating the impact that varying the memory access latency might have on our arbitration schemes.

We slightly modified our experimental setup by varying the `TDM` slot length Sl using two additional configurations with slot lengths of 25 and 100 cycles respectively. This impacts the generated memory profiles, as described in Subsection 6.2, as the number of memory accesses that fit into the task's WCETs (C_i) derived by UUniFast depends on the slot length. Recall that our traffic generator takes the worst-case memory access latency for each newly



(a) `TDMds` memory idling. (b) `TDMer` memory idling with an initial slack of Sl. (c) Normalized average non-critical tasks execution times.

Fig. 13 Results considering a `TDM` slot length of $Sl = 25$ cycles (lower is better).



(a) TDMds memory idling. (b) TDMer memory idling with an initial slack of Sl. (c) Normalized average non-critical tasks execution times.

Fig. 14 Results considering a TDM slot length of $Sl = 100$ cycles (lower is better).

generated request into account, which is bounded by $P - 1 + Sl$ cycles. Note that both, the TDM period P and the TDM slot length Sl, are impacted in our modified setup. Varying the TDM slot length thus impacts the memory traffic generator and consequently the generated task sets. The results presented in this section are therefore not directly comparable. This applies, in particular, for the total memory idling.

We performed the same number of simulations as in the previous experiments for the two new configurations – considering independently generated task sets according to a varying number of tasks/cores and a varying global system utilization (see Section 6.1). The simulated memory latencies are again randomly chosen in the range $[21,25]$ and $[21,100]$ respectively. Critical tasks are provisioned with an initial slack value of a single slot length at each job start.

Figures 13 and 14 summarize the obtained results from these simulation runs. Subfigures 13a and 14a show a breakdown of the average memory idle time from all runs for TDMds. Looking at the green lines (issue and release delays) reveals that for a TDM slot length of $Sl = 25$, representing a memory with small access latency variability, TDMds still suffers from considerable inefficiency, which is mostly caused by release delays. Even for moderate levels of system utilization (from 40% on) these delays steadily represent almost 10% of the memory idling. This is much better than before, considering a slot length of 40 cycles, but still represents a non-negligible overhead. The TDMer approach, with initial slack, also in this configuration successfully eliminates these delays, as depicted in Subfigure 13b – albeit with a lower gain compared to the previous results. The results are, as expected, different with a TDM slot length of $Sl = 100$, which represents a memory with high access latency variability. The delays induced here are very high for TDMds, going from 10% up to 40% at high load (Subfigure 14a). Subfigure 14b, shows that TDMer is still very effective in eliminating the issue and release delays. This improves the total memory utilization considerably, in particular at high load where the total memory idling drops from 40% for TDMds to less than 20% for TDMer. Due to space considerations, we do not show the results for regular TDMfs here. The results are considerably worse than those for TDMds– the combined issue and release delays here reach a peak at about 47% and 20% of the memory idle time for medium levels of system utilization for the two memory latency configurations respectively.

Subfigure 13c shows a breakdown of the normalized execution times of non-critical tasks w.r.t. the total trace length for TDMds and TDMer, considering $Sl = 25$ cycles. We can see that TDMer reduces execution times, especially at high loads. However, due to the

smaller memory overhead, the gain for non-critical tasks is moderate. The configuration with $Sl = 100$ cycles (Subfigure 14c), on the other hand, shows considerable improvements in the execution times of non-critical tasks – despite the large `TDM` slots. Starting at utilization levels of 30% the normalized execution time is improved by a factor of at least 2 reaching a maximum of a factor of roughly 3.8. This is due to the fact that non-critical tasks can potentially exploit the considerable memory idle time (up to 40%) caused by the release delays of the other approaches that have to respect `TDM` slots.

As a conclusion, our dynamic `TDM`-based arbitration policy `TDMer` is performing well with low latency memories. But the gain is even more significant when using memories with highly variable latencies. So, regardless of the memory type, using our approach allows to achieve the maximum memory utilization with the guarantee of respecting the timing constraints of critical tasks for real-time systems.

7 Hardware Architecture

In this section, we discuss means to implement a variant of the dynamic `TDM`-based arbitration scheme in hardware. Such an implementation faces four main challenges. Firstly, the use of a priority queue to implement the EDF policy for critical requests will, in all likelihood, be costly and slow in hardware. Secondly, the use of modulo operations in the deadline and slack computations should be avoided – likewise due to performance and complexity reasons. Thirdly, the values of deadlines and slack counters need to be bounded in order to limit the number of data bits required in registers and the associated logic circuits. Finally, the implementation needs to make arbitration decisions at the full speed of the memory bus, as opposed to the `TDMds` or `TDMfs` schemes that only take decisions at the beginning of each `TDM` slot.

A nice feature of our proposed schemes is that the deadline and slack computation of one task is independent from other tasks in the system. This allows us to decompose the hardware design as follows: (1) components to forward requests, compute Deadlines, and manage Slack Counters (thus called *DSC*) for each core executing a critical task, (2) components to forward requests from Non-Critical cores (*NC*), (3) global ARbitration routing logic (*AR*), and (3) a component to perform the Data Multiplexing (*DM*) between cores and the main memory.

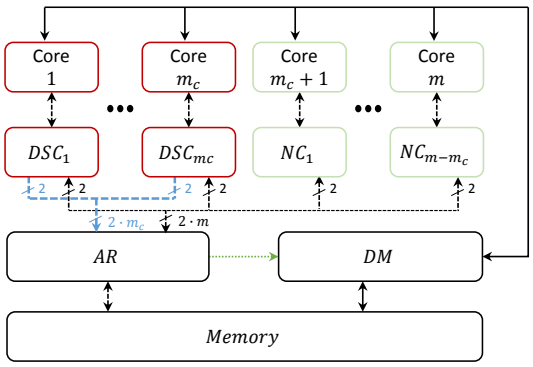


Fig. 15 Overview of the hardware design of our dynamic TDM-based arbiter.

We first provide an overview of the interactions among these components and subsequently discuss relevant components in more detail – along with simple and efficient solutions addressing the aforementioned challenges.

7.1 Architecture Overview

Figure 15 provides an overview of the proposed hardware architecture of our dynamic TDM-based arbitration policy. Each of the m processing cores is connected to a *DSC* or *NC* component. We assume for simplicity that the first m_c cores are critical (highlighted in red), while the remaining cores are non-critical (highlighted in green). These *DSC* and *NC* components receive memory requests from the cores, forward them to the arbiter, and notify the cores of the completion of their requests (◄---►). The *NC* components do not have any internal state or logic themselves. The *DSC* components, on the other hand, have internal state that contains, among others, the deadline and slack counter of the respective core as well as logic to perform the necessary bookkeeping operations on its internal state.

The *NC* and *DSC* components, in turn, forward the requests to the main arbiter *AR* (◄---►). In addition, the arbiter receives two control signals (------►) from each *DSC* component, which communicate deadline (PM_i) and slack (ES_i) information of the respective critical cores. This information allows *AR* to take the arbitration decisions, while ensuring that the `TDM` guarantees are respected at all times. The arbiter selects one of the pending requests (at a cycle-level granularity) and then indicates the selected core identifier (*core_id*, ·····►) to the data multiplexing component. The *DM* then routes the data signals (◄——►) to/from the various cores from/to the main memory. The arbiter at the same time forwards the request to the main memory and subsequently waits for the request to complete (◄---►). The completion signal is then propagated back to the respective *NC* or *DSC* component, which itself notifies the respective core and performs the necessary bookkeeping operations as needed. These handshake signals, for pending requests and request completion, are thus connected to almost all components, i.e., Core i, DSC_i, NC_i, *AR*, and Memory, as indicated by the dashed black lines.

7.2 Arbitration Logic

`TDMer` relies on EDF strategy among critical requests and thus requires a hardware implementation of a priority queue or a similar structure. Preliminary tests on our FPGA board revealed that, with a rising number of cores, the clock frequency would quickly become an issue with such an approach. However, applying EDF is not strictly necessary as hinted to by the correctness proof in Section 5, which only refers to EDF in Definition 7 and Theorem 2.

The main motivation of applying EDF was, on the one hand, to prioritized non-critical requests over critical requests whenever possible. A closer look at the proof reveals that, in terms of correctness, EDF can be replaced by any other scheme as long as it ensures that any critical request can access memory at the beginning of the `TDM` slot that is associated with its deadline.

Consequently, it suffices to prevent other requests from accessing memory during that slot. For this we need to consider two cases: (1) prevent the early-start optimization in the `TDM` slot before the critical request's deadline and (2) prevent any request from accessing memory during the `TDM` slot of the critical request. The former case is independent from

the EDF policy used by the arbiter (see Line 4 of Algorithm 1), while the later case can be detected by checking the critical-request's deadline as described below.

The arbitration logic *AR* receives two 1-bit control signals from each *DSC* component representing a critical task. The *early-start* signal (ES_i) indicates whether the respective task has accumulated sufficient slack to allow the early-start optimization (see Subsection 4.1), whereas the *prioritize me* signal (PM_i) indicates whether a critical core needs to be prioritized now in order to meet its deadline, i.e., it claims the immediate next `TDM` slot.

The arbiter *AR* thus performs the following check at each clock cycle whenever none of the currently pending requests is processed by the memory. Instead of determining the request with the smallest deadline (as under `TDMer`), the hardware implementation first checks the ES_i signal of the owner of the next upcoming `TDM` slot. If the signal is asserted, no deadline miss is imminent and any arbitration policy can be applied in order to select the next request to be processed. In our implementation we simply apply Round-Robin arbitration among all pending requests, we thus call the resulting arbitration policy `TDMrr`.

If the ES_i signal is not asserted, the owner of the upcoming `TDM` slot does not have enough slack and the early-start optimization cannot be applied safely. The arbiter thus cannot select any of the pending requests and has to wait until the beginning of the next `TDM` slot – or until the ES_i signal is asserted.

Finally, the arbiter also checks if any of the PM_i signals is asserted (this may only be the case at the beginning of a `TDM` slot). If one of these signals is asserted, the corresponding request of the slot owner needs to be processed right now in order to prevent a deadline miss.

If the above checks allow to select a new request to be processed, the arbiter signals the respective core identifier of that request to the data multiplexing component (*DM*), which subsequently takes care of transferring the data between the core and the memory.

Before concluding the discussion of the *AR* component, we would like to highlight the consequences of replacing EDF by round-robin. The EDF policy, as described in Section 4, systematically prioritized non-critical requests over critical requests – as long as sufficient slack is available. Consequently, critical requests tend to be delayed until no non-critical request is pending or a deadline miss become imminent. One would expect this to be favorable in terms of execution times for non-critical tasks, and less favorable for critical tasks. This is indeed noticeable, as confirmed by our experimental evaluation in Section 8. However, the impact is marginal.

Replacing EDF by round-robin, on the other hand, does not have an impact on correctness. A proof w.r.t. the worst-case behavior under the `TDMrr` scheme is provided in Section 7.5. In the next section, we first explain how these control signals are derived by the *DSC* components.

7.3 Deadline and Slack Computation

Due to the periodic repetition of the `TDM` schedule during each `TDM` period (P), a naive implementation of the deadline and slack values within the *DSC* components would require several modulo operations, which are expensive in terms of hardware resources. In addition, the bit-width required to perform the related computations may become an issue. For instance, it appears preferable to avoid representing deadlines as absolute dates, which may lead to large values in long running systems and consequently require a large number of bits. Fortunately both issues can be solved elegantly, leading to an efficient and simple solution.

As explained in Section 5, the deadline computation is related to the start date of the current `TDM` period Sp and the offset of the task's `TDM` slot $O(\tau_i)$ (w.r.t Sp). Based on this

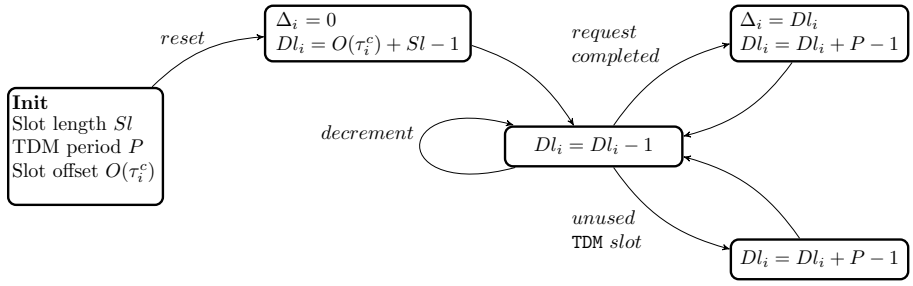


Fig. 16 Summary of the update rules of the Deadline and Slack Computation (*DSC*) components.

observation, we use a dedicated counter register Dl_i to model the current deadline of a request from a critical task τ_i. The value of this counter represents the number of clock cycles left before reaching the deadline of the request. This counter is thus decremented on each cycle, for instance while waiting for the completion of a pending request or the issuing of a new request from its critical task. A deadline miss would theoretically occur when this counter reaches 0 – although this will not occur in practice. We also track the deadlines when no request is pending, i.e., our approach keeps the value of the Dl_i counter current at all times. In this case, Dl_i indicates the deadline that would be computed if a request were issued in the current cycle.

We then define 3 update rules that reset the Dl_i counter when specific events occur, as depicted by Figure 16. The first rule is applied on a system reset (left). The second rule updates Dl_i when a task's slot is *unused* under strict `TDM` (bottom right). Finally, the third rule deals with the case of memory request completion (top right). These 3 cases are detailed in the remainder of this section. Note that, as before, the slack counter (Δ_i) indicates the number of clock cycles that the last request completed earlier than its deadline.

7.3.1 Update Rules

System Reset: During a system reset, each DSC_i module initializes its deadline counter for its respective critical task τ_i depending on its offset $O(\tau_i)$ within the `TDM` schedule, as illustrated in Figure 16 (left). At this moment, obviously no request from τ_i can be pending at the arbiter. The deadline thus rather indicates the number of cycles until the end of the *first* `TDM` slot owned by τ_i is reached. The value of Dl_i can also be seen as the deadline that would be computed if a request were issued right at system reset. The slack counter (Δ_i) is reset to 0, as usual.

Unused `TDM` Slot: Note that the current Dl_i value always corresponds to a `TDM` slot under strict `TDM`. When the beginning of this `TDM` slot under strict `TDM` has passed and no request from τ_i is pending, the slot would have been *unused*. In such a situation, the Dl_i counter must be incremented to the next possible deadline for any potential request issued by τ_i in the future. Such a deadline is one `TDM` period ahead, i.e., equals $Dl_i + P - 1$, as illustrated by Figure 16 (bottom right). The slack counter does not change during this update.

Under strict `TDM` an *unused* slot can be detected by comparing the beginning of τ_i's next slot with the *delayed issue date*, i.e., the current instant plus the slack counter Δ_i (see Section 3). In our hardware implementation these absolute dates are represented as relative dates w.r.t. the current instant. The beginning of τ_i's next slot is thus given by $Dl_i - Sl + 1$, whereas the *delayed issue date* simply corresponds to the current value of the slack counter

Δ_i. Task τ_i thus did not use its TDM slot, when it has no request pending and these relative dates match, i.e., $Dl_i - Sl + 1 = \Delta_i$. This triggers the necessary update of the deadline counter.

Request Completion: When the core executing task τ_i issues a request to its DSC_i component, the pending request is immediately forwarded to the arbiter using a dedicated control signal. The Dl_i counter continues to be decremented, while the DSC_i component waits for the request to complete. Note that the rule for *unused* slots cannot trigger meanwhile.

Once the request completes, which is signaled by the memory and forwarded by *AR* to the respective DSC_i component, the slack counter needs to be updated. Recall that the value of Dl_i represents the number of clock cycles to the next deadline. It thus suffices to copy the value of Dl_i into the slack counter Δ_i. The Dl_i counter also needs to be updated, since, under strict TDM, the slot associated with the current deadline would have been used. The next possible deadline for any future request issued by τ_i falls into the next TDM period and is simply given by $Dl_i + P - 1$. Both updates are illustrated by Figure 16 (top right).

7.3.2 Control Signal Generation

The *DSC* components are connected to the arbiter *AR* using several control signals. The signals that indicate pending requests and their completion are merely forwarded from/to the core and arbiter respectively, but are not generated by the *DSC* components themselves. This is different for the ES_i and PM_i signals, which are derived from the current values of the deadline and slack counters.

The *early-start* signal (ES_i) indicates to the arbiter that the corresponding task τ_i has sufficient slack, i.e., other requests may overflow into its TDM slot without the risk for a deadline miss. The early-start optimization is *not* safe when the value of the Dl_i counter is smaller than $2 \cdot Sl$. This indicates that the deadline of τ_i's pending request, or the potential deadline of a request to be issued in the future, is at the end of the next upcoming TDM slot. A deadline miss cannot be excluded. Hence, $ES_i = 1$ iff $Dl_i \geq 2 \cdot Sl$, $ES_i = 0$ otherwise. Note that, if τ_i left its TDM slot unused, its deadline may advance due to the second update rule from above. This might then enable the early-start optimization in one of the subsequent cycles – this is equivalent to the early-start test shown by Algorithm 1.

A similar argument can be made for the PM_i signal at the beginning of a TDM slot owned by a task τ_i. The beginning of the slot corresponds to the relative date $Dl_i - Sl + 1$ as explained above. The PM_i consequently needs to be asserted when τ_i, the owner of the current TDM slot, reached this date and has to claim its slot in order to prevent a deadline miss. Hence, $PM_i = 1$ iff $Dl_i = Sl - 1$, $PM_i = 0$ otherwise.

7.4 Bit-Width Considerations

As can be seen the solution proposed above only requires two internal counter registers, a simple state machine, and two adders. It does not require any expensive modulo operations and can thus be implemented efficiently. In addition, deadlines are not encoded as absolute values – which allows to minimize the number of bits required for the deadline and slack computations.

However, a naive implementation that simply cuts the Dl_i values off using saturation arithmetic might break correctness. The problem is that the Dl_i values need to be aligned with the TDM schedule, i.e., the deadline represented by Dl_i always has to correspond to the end of one of τ_i's future TDM slots. Let us illustrate this by an example considering a

system that uses saturation arithmetic over 10 bits and a TDM period of $P = 64$ cycles. Dl_i counters here may take any value in the range $[0, 1023]$. Now assume that the request of a critical task in that system completed 1000 cycles before its deadline, i.e., $Dl_i = 1000$ at that instant. Following the update rules from above, this value is copied into the slack counter ($\Delta_i = 1000$). In addition, Dl_i needs to be incremented by $P - 1$ (63). Due to the saturation arithmetic the deadline becomes $Dl_i = 1023$, whereas the actual deadline should have been 1063. The DSC_i component now generates its control signals based on a deadline that is 40 cycles early! Consequently, the PM_i and ES_i signals are no longer correct, which, in the worst-case, may cause a clash with another slot of another critical task.

The problem of clashes can be resolved by subtracting P from the new Dl_i value, i.e., using $Dl_i = 1063 - 64 = 999$ in the above example. This suffices to guarantee that the deadlines remain aligned with the original TDM schedule. It turns out that this solution is, in addition, virtually free in terms of hardware resources. Note that the update rules for Dl_i only indicate two possible options: either Dl_i is decremented by 1 or incremented by $P - 1$. The latter case may cause an overflow, which can be detected with little overhead from the carry out bit of the corresponding adder. Furthermore, one can see that subtracting P again from the new Dl_i value is equivalent to a decrement by 1 cycle ($Dl_i + P - 1 - P = Dl_i - 1$). This value is already available and does not cause any additional hardware overhead.

Note that this allows a safe operation of the arbiter with a reduced bit-width for deadline and slack counters and their associated logic circuits. Due to the artificial limit in the range of the slack counters, the deadlines might be earlier compared to the deadlines under strict TDM. This is not problematic, assuming a timing-composable architecture. However, one might expect that this could impact the arbiters overall efficiency in terms of memory utilization. This is evaluated, among others, in detail in the following experiments.

7.5 Worst-Case behavior w.r.t. the Hardware Design

TDMrr brings some modifications to our dynamic schemes presented in Section 3 and 4, namely the use of Round-Robin arbitration and modifications to the way deadlines and slack counters are computed. The formal proof for TDMer from Section 5 thus does not apply to TDMrr and needs to be adapted accordingly. The goal is to show that the *relative* deadlines of TDMrr match the *absolute* deadlines of strict TDM.

Since slack counters and deadlines under TDMrr are relative, we first introduce a way, and a notation, to obtain the absolute deadline corresponding to a given value of the Dl_i register at a given instant t by introducing a cycle counter cc. Note that there is no actual need to implement this counter in hardware. Also note that, for now, we assume unbounded bit-width for the computation of the various registers involved:

Definition 8 Under TDMrr the **absolute deadline** $d_i^{@t}$ of a critical task τ_i at time instant t, is given by the sum of the relative deadline ($Dl_i^{@t}$) and a (*virtual*) cycle counter $cc^{@t}$, which is reset to 0 on system reset and incremented on every clock cycle. We thus have $d_i^{@t} = Dl_i^{@t} + cc^{@t}$, or, for brevity, $d_i^{@t} = Dl_i^{@t} + t$.

We then start with some auxiliary lemmas that we will use later in the final correctness proof. The first lemma indicates that the relative deadlines under TDMrr are unique – similar to the absolute deadlines under TDMer.

Lemma 4 *Under TDMrr the relative deadlines Dl_i of a critical tasks τ_i always correspond to the end of τ_i's TDM slot and are thus unique.*

Proof The proof is based on induction over successive time instants t:
Induction base $t = 0$: This time instant corresponds to the system initialization, where the Dl_i register is set by the *reset* update rule from Subsection 7.3.1. The relative deadline of τ_i is then determined according to the task's slot offset $O(\tau_i)$: $Dl_i^{@0} = O(\tau_i) + Sl - 1$.

The lemma thus trivially holds, since critical tasks have dedicated `TDM` slots, with a common length Sl, and unique slot offsets, which results in a unique absolute deadline d_i^0 at the end of the task's `TDM` slot.

Induction step $t = n$: Assuming that $d_i^{@t-1}$ represented a unique absolute deadline at the end of a `TDM` slot, we need to ensure that the absolute deadline at t remains unique.

The Dl_i register is modified according to the remaining update rules *decrement*, *unused* `TDM` *slot*, and *request completion*, which gives us two cases to consider:

Case (1): The Dl_i register decrements (*decrement*):

$$\begin{aligned} Dl_i^{@t} &= Dl_i^{@t-1} - 1 \\ Dl_i^{@t} + t &= Dl_i^{@t-1} + t - 1 \\ d_i^{@t} &= d_i^{@t-1} \end{aligned}$$

The absolute deadline $d_i^{@t}$ in this case does not change and the lemma trivially holds.

Case (2): The Dl_i register increments (*unused* `TDM` *slot* and *request completion*):

$$\begin{aligned} Dl_i^{@t} &= Dl_i^{@t-1} + (P - 1) \\ Dl_i^{@t} + t &= Dl_i^{@t-1} + t - 1 + P \\ d_i^{@t} &= d_i^{@t-1} + P \end{aligned}$$

The absolute deadline $d_i^{@t}$ is simply incremented by a `TDM` period P. Since all tasks share the same period the absolute deadlines thus remain unique and still represent the end of a `TDM` slot. □

Corollary 1 *At any time instant t only one of the $PM_i^{@t}$ signals of all critical task $\tau_i \in \Gamma$ can be asserted, i.e., $PM_i^{@t} = 1 \wedge PM_j^{@t} = 1 \Longrightarrow i = j$.*

Proof This follows immediately from Lemma 4 and the fact that the PM_i is only asserted when $Dl_i^{@t} = Sl - 1$. □

The next step is to show that `TDMrr` always respects the absolute deadline of a pending request. For now, this does not take into consideration whether the deadline is *correct* with regard to regular `TDM`. However, as shown before the deadline is known to represent the end of a `TDM` slot and is unique.

Corollary 2 *The absolute deadline under `TDMrr` d_k^{TDMrr} does not change while a memory request r_k is pending, i.e., $\forall t \in [a_k, c_k[, \quad d_i^{@t} = d_i^{@a_k}$.*

Proof The value $d_i^{@t}$ does not change while a request is pending, since the cycle counter (i.e., t) is steadily incremented by 1, while Dl_i is decremented by 1 (cf. the *decrement* update rule in Section 7.3.1). Other update rules are not possible up to the completion of the request. □

Lemma 5 *Given an absolute deadline d_k^{TDMrr} of a critical request r_k with an arrival date a_k^{TDMrr}, `TDMrr` guarantees that the request completes at or before this deadline: $a_k^{TDMrr} < c_k^{TDMrr} \leq d_k^{TDMrr}$.*

Proof Proof by contradiction, assuming that $c_k^{\texttt{TDMrr}} > d_k^{\texttt{TDMrr}}$ holds.

The absolute deadline $d_k^{\texttt{TDMrr}}$ of the request has to be equal to the absolute deadline of task τ_i at the moment of the request's arrival: $d_k^{\texttt{TDMrr}} = d_i^{@a_k} = O(\tau_i) + Sl - 1 + x_k \cdot P$, where $x_k \in \mathbb{N}^0$, which follows from Corollary 2.

Therefore, $c_k^{\texttt{TDMrr}} > O(\tau_i) + Sl - 1 + x_k \cdot P$, which means that, at the beginning of τ_i's `TDM` slot at instant $t = O(\tau_i) + x_k \cdot P$, request r_k was pending. However, at that instant, the relative deadline $Dl_i^{@t}$ under `TDMrr` had to be (cf. Definition 8):

$$Dl_i^{@t} = d_i^{@t} - t = O(\tau_i) + Sl - 1 + x_k \cdot P - O(\tau_i) - x_k \cdot P = Sl - 1$$

Consequently PM_i had to be asserted at this instant (cf. Subsection 7.3.2). As the request missed its deadline the arbiter did not granted r_k access to the memory. This can only happen if another request r_l from another task τ_j blocks the memory:

$$\exists r_l \text{ of } \tau_j \text{ such that:} \quad a_l^{\texttt{TDMrr}} \leq O(\tau_i) + x_k \cdot P < c_l^{\texttt{TDMrr}} < O(\tau_i) + x_k \cdot P + Sl - 1$$

The arbiter had to grant r_l access to the memory no earlier than $c_l^{\texttt{TDMrr}} - (Sl-1)$, due to the worst-case memory access latency/slot length Sl, this gives us:

$$\begin{aligned}(O(\tau_i) + x_k \cdot P) - (Sl-1) &< c_l^{\texttt{TDMrr}} - (Sl-1) < (O(\tau_i) + x_k \cdot P) + Sl - 1 - (Sl-1) \\ t - (Sl-1) &< c_l^{\texttt{TDMrr}} - (Sl-1) < t + Sl - 1 - (Sl-1) \\ t - (Sl-1) &< c_l^{\texttt{TDMrr}} - (Sl-1) < t\end{aligned}$$

This means that memory started processing request r_l before instant t. The request r_l can only be processed iff ES_i is asserted by the owner of the immediate next `TDM` slot with regard to t. In this case τ_i has to be the owner (cf. the in-equations above). However, since r_k was pending at instant t, it is impossible that ES_i was asserted. The relative deadline of task $Dl_i^{@t} = Sl - 1$ (cf. above), which is contradictory to the necessary condition $Dl_i^{@t} \geq 2 \cdot Sl$ needed to assert ES_i. This is also true for all instants up to t in the range $[t - (Sl-1), t]$, which can only yield relative deadlines in $[Sl - 1, 2 \cdot Sl - 2]$.

Note that the range $[t - (Sl-1), t]$ is safe even when $a_k \geq t - (Sl-1)$. As the absolute deadline $d_i^{@t'}$ for $t - (Sl-1) \leq t' \leq a_k$ cannot be larger then $d_i^{@t}$. Either c_{k-1} is processed or $d_i^{@t'} = d_i^{@t}$. The former would contradict the assumption that r_l of task τ_j was processed, the latter would prevent the ES_i signal from being asserted.

The pending request r_k would prevent other requests from starting right *before* τ_i's `TDM` slot. In addition, Lemma 4 and Corollary 1 ensure that r_k is the only request that can claim τ_i's slot. We can thus conclude that the completion date of request r_k will always be smaller than its absolute deadline under `TDMrr`, i.e., $c_k^{\texttt{TDMrr}} \leq d_k^{\texttt{TDMrr}}$. □

Based on the previous intermediate results, we are finally able to proof that the deadlines under `TDMrr` in fact match those under regular `TDM`.

Lemma 6 *Assuming an unbounded bit-width, the absolute deadline d_k^{TDM} of the k-th request of a critical task under regular `TDM` always matches the* absolute *deadline d_k^{TDMrr} of the k-th request under `TDMrr`: $d_k^{TDM} = d_k^{TDMrr}$.*

Proof The proof is based on induction over the set of requests r_k of a critical task τ_i:

Induction base $k = 0$: Depending on the arrival date a_0 of the first request r_0, we can distinguishes two cases:

Case (1): The request arrives before the first TDM slot ($a_0 \leq O(\tau_i)$):
We then know that the deadline under regular TDM is $d_0^{\texttt{TDM}} = O(\tau_i) + Sl - 1$. Under TDMrr the situation is more complex, due to the constant updates of the Dl_i register. However, the absolute deadline (cf. Definition 8) has to be valid, for all time instants t while the request is pending at the arbiter: $\forall t \in \mathbb{N}^0, a_0^{\texttt{TDM}} \leq t < c_0^{\texttt{TDMrr}} \leq d_0^{\texttt{TDMrr}}$: $d_0^{\texttt{TDMrr}} = d_i^{@t}$ (see Lemma 5).

We then need to show that $d_i^{@t} = O(\tau_i) + Sl - 1$. This is trivially true at $t = a_0$, due to the initialization of Dl_i at system reset (cf. the *reset* update rule in Section 7.3.1). The value $d_i^{@t}$ does not change while a request is pending (see Corollary 2), we thus have to show that the request completes in time, i.e., $c_0^{\texttt{TDMrr}} \leq d_0^{\texttt{TDMrr}}$, which follows from Lemma 5.

Case (2): The request misses the first TDM slot ($a_0 > O(\tau_i)$):
The deadline under regular TDM is $d_0^{\texttt{TDM}} = O(\tau_i) + x_0 \cdot P + (Sl - 1)$, where $x_0 \in \mathbb{N}^+$. The variable x_0 refers to the number of TDM slots missed before the request arrival. Therefore, knowing that $a_0^{\texttt{TDM}} > d_0^{\texttt{TDM}} - P + Sl - 1$, we can derive the number of missed TDM slots as follows:

$$x_0 = \left\lfloor \frac{d_0^{\texttt{TDM}}}{P} \right\rfloor = \left\lfloor \frac{d_0^{\texttt{TDM}} - P}{P} \right\rfloor + 1 = \left\lfloor \frac{a_0 + Sl - 2}{P} \right\rfloor + 1$$

Under regular TDM and TDMrr, the first request arrival date is the same for both, i.e., $a_0^{\texttt{TDM}} = a_0^{\texttt{TDMrr}}$. Hence, the number of missed TDM slots x_0 is the same for both arbitration schemes. Under TDMrr, the relative deadline Dl_i steadily decrements until the update rule *unused TDM Slot* (cf. Section 7.3.1) triggers – potentially repeatedly (x_0 times). This update rule relies on the Dl_i register, every time it becomes $Sl - 1$, it is incremented by a TDM period P, actually $P - 1$ for the subsequent cycle. The *unused TDM Slot* rule triggers for the first time at time instant $O(\tau_i)$ and repeatedly triggers up to the TDM slot starting at $d_0^{\texttt{TDM}} - P - (Sl - 1)$.

Therefore, at $t = d_0^{\texttt{TDM}} - P - (Sl - 2)$ the value of the relative deadline has to be $Dl_i^{@t} = O(\tau_i) + x_0 \cdot P + Sl - 1 - t$. We thus can derive the following absolute deadline under TDMrr:

$$d_0^{\texttt{TDMrr}} = d_i^{@t} = Dl_i^{@t} + t = O(\tau_i) + x_0 \cdot P + (Sl - 1)$$

Note that the absolute deadline under TDMrr does not change up until the request's arrival, i.e, in the time range $[t, a_0^{\texttt{TDMrr}}]$, since $a_0^{\texttt{TDMrr}} - t < P$. The deadline also remains unchanged thereafter until the request's completion (see Corollary 2).

We have shown, through Cases (1) and (2), that the deadlines matches for the first request r_0 under regular TDM and TDMrr. In addition, the update rule *request completion* updates the slack counter value using the remaining cycles until the absolute deadline, which is obtained from $Dl_i^{@c_0^{\texttt{TDMrr}}}$, i.e., $\Delta_i = Dl_i^{@c_0^{\texttt{TDMrr}}}$.

Induction step $k = n$: Assuming that the deadlines $d_{n-1}^{\texttt{TDM}}$ and $d_{n-1}^{\texttt{TDMrr}}$ for $(n-1)$-th request matches, we need to ensure that this deadlines also matches for the n-th request, i.e., $d_n^{\texttt{TDM}} = d_n^{\texttt{TDMrr}}$. For this we again distinguish two cases:

Case (1): Under TDM, request r_k arrives before the next TDM slot ($a_k^{\texttt{TDM}} < P - (Sl - 1)$):
This means that r_k's deadline simply falls into the next period and that the distance between the k-th and $(k-1)$-th request is shorter than P, i.e., $d_k^{\texttt{TDM}} = d_{k-1}^{\texttt{TDM}} + P$ and $dist_k = a_k^{\texttt{TDM}} - d_{k-1}^{\texttt{TDM}} < P$.

Under TDMrr we know that the absolute deadline was correct right before the completion of r_{k-1} at instant $c_{k-1}^{\texttt{TDMrr}} - 1$, i.e., $d_k^{\texttt{TDMrr}} = d_i^{@c_{k-1}^{\texttt{TDMrr}} - 1}$. Due to the *request completion* rule the absolute deadline of τ_i in the next cycle becomes $d_i^{@c_{k-1}^{\texttt{TDMrr}} - 1} + P - 1 =$

$d_i^{@c_{k-1}^{\texttt{TDMrr}}} + P = d_k^{\texttt{TDMrr}} + P$, matching the expected deadline of the k-th request. This also indicates that $Dl_i^{@c_{k-1}^{\texttt{TDMrr}}} \geq P$.

It remains to show that this deadline does not change up to the request's arrival at $a_k^{\texttt{TDMrr}}$. Since no request is pending, we have to show that the *unused* `TDM` *Slot* rule does not trigger. This follows trivially from the fact that $dist_k < P$ and the fact that $Dl_i^{@c_{k-1}^{\texttt{TDMrr}}} \geq P$.

Case (2): Under `TDM`, request r_k arrives after the next `TDM` slot ($a_k^{\texttt{TDM}} \geq P - (Sl-1)$):
We then know that the r_k's deadline has to be a number x_k periods later than the deadline of the previous request, i.e., $d_k^{\texttt{TDM}} = d_{k-1}^{\texttt{TDM}} + x_k \cdot P$, where $x_k = \left\lfloor \frac{dist_k + Sl - 2}{P} \right\rfloor + 1$. Based on the arguments put forward for Case (1) of the induction step ($dist_k$) above as well as Case (2) of the induction base (using $Dl_i - \Delta_i$), we can show that the *unused* `TDM` *slot* rule triggers x_k times up to the arrival data of r_k. □

Theorem 3 *(Worst-Case Behavior) Considering a given execution (i.e., execution path, runtime conditions, input values, ...) a memory access of a critical task under any possible execution under* `TDMrr` *completes no later than the same execution under strict* `TDM`*.*

Proof The correctness of Theorem 3 is ensured by by Lemma 5 by guaranteeing that request completes at or before its deadline, i.e., $c_k^{\texttt{TDMrr}} \leq d_k^{\texttt{TDMrr}}$. Lemma 6 shows that the deadlines under `TDMrr` will always corresponds to the deadline under regular `TDM`. □

The proofs above assumed an unbounded bit-width, which is not the case in our hardware implementation. Limiting the bit-width only has an impact on Lemma 6, all preceding lemmas remain valid. The lemma could be adapted in order to show that the absolute deadlines under `TDMrr` are always smaller than those under strict `TDM`, due to the fact that the value of the Dl_i register is smaller.

8 Hardware Experiments

In this section we evaluate the hardware design of the arbitration logic described in Section 7. We first present the evaluation platform and the hardware cost results. Afterwards, we compare the `TDMer` and the `TDMrr` approaches using various metrics, such as, memory utilization, average number of deadline misses for non-critical tasks, and the maximum slack accumulated by critical tasks. Finally, we evaluate the impact of limiting the bit-width of the slack counters on the arbiter's performance.

8.1 Evaluation Platform

The hardware design described in Section 7 has been realized on a Terasic DE-10 Nano evaluation board. The board is, among others, equipped with an Intel Cyclone V SE SoC-FPGA, which we use to evaluate the hardware implementation cost. Logic circuits on various families of FPGA device families from Intel are built from Adaptive Logic Modules (ALMs) that contain registers, programmable logic, but also predefined logic blocks (e.g., adders). The Cyclone V SE (5CSEBA6U23I7) on our evaluation board contains 41 910 ALMs, 83 820 primary logic registers, and 5 530 kbit of distributed memory, which, for instance, is enough to instantiate several Patmos cores (Schoeberl et al., 2011). Hardware is synthesized from the arbiter's Verilog implementation using Intel's Quartus Prime Lite tool (version 18.1) using default parameters for the considered FPGA.

To evaluate the hardware cost of implementing the `TDMrr` approach, we determined the number of ALMs and primary logic registers occupied by the design. The synthesis tool did not make use of any memory resources (neither block- nor distributed RAM). We also determined the attainable clock speed of the design, provided by the synthesis tool in the form of a maximum operating frequency (considering regular operating conditions).

The correctness of the Verilog implementation was thoroughly validated using the Icarus Verilog hardware simulation tool (version 10.1). The hardware simulation accepts the same memory patterns, described in Section 6, as input, which allows an automatic verification against the cycle-accurate software simulator that was already used in the other experiments.

8.2 Results for Hardware Synthesis

After careful validation, we instantiated and synthesized several different versions of our hardware design. The design can be parameterized by the number of cores (m), the number of critical cores (m_c), and the bit-width of the deadlines/slack counters. Table 1 summarizes the results in terms of ALMs and primary logic registers occupied by the various design instances considering 24-bit deadline and slack counters (see Subsection 8.4 for a justification), The table also indicates the maximum operating frequency. Note that the reported numbers only comprise the *DSC*, *NC*, and *AR* components. The numbers do not cover the data multiplexing (*DM*) and processing cores, as these components are independent from the actual arbiter design. The design is relatively small as can be seen by the low relative usage numbers (%). This becomes even more apparent when comparing against the hardware cost of instantiating a Patmos core on the same FPGA device. Considering the processing core and its caches in isolation, i.e., ignoring I/O and memory interfaces, Patmos occupies approximately 10 500 ALMs. The processor design, including I/O interfaces and an Avalon-based bus interface to a silicon DDR memory controller, achieves a maximum operating frequency of 65 MHz, which can likely be improved by fine-tuning the implementation to the board.

Overall we can observe that in terms of hardware costs, the resource utilization is highly dependent on the number of critical cores (m_c) rather then the total number of cores (m). When the number of critical cores increases, the number of occupied ALMs and primary logic registers proportionally increase. The main difference between critical and non-critical cores stems from the deadline and slack computation (*DSC*) components. This module, along with the number of critical cores, therefore plays a major role in the overall hardware cost. When the total number of cores increases, while keeping the number of critical cores constant, the difference is rather small. For instance, increasing the number of cores

Table 1 Implementation details of our hardware design on an Intel Cyclone V SE SoC-FPGA considering 24-bit deadline and slack counters.

Cores		ALMs		Registers		
m	m_c	Number (%)	per m_c	Number (%)	per m_c	Frequency (MHz)
2	2	179 (0.43%)	90	116 (0.14%)	58	147.86
4	2	221 (0.53%)	111	122 (0.15%)	61	157.88
4	4	407 (0.97%)	102	218 (0.26%)	55	142.37
8	4	389 (0.93%)	97	225 (0.27%)	56	129.62
8	8	723 (1.73%)	90	418 (0.50%)	52	117.19
16	8	773 (1.84%)	97	429 (0.51%)	54	97.61
16	16	1424 (3.40%)	89	814 (0.97%)	51	98.18

Table 2 Implementation details of our hardware design on an Intel Cyclone V SE SoC-FPGA considering various bit-widths for deadline and slack counters.

Cores		10 bits			20 bits		
m	m_c	ALMs	Registers	Frequency (MHz)	ALMs	Registers	Frequency (MHz)
2	2	95	60	205.72	157	100	164.02
4	2	111	66	200.72	175	106	157.95
4	4	187	106	165.04	315	186	151.15
8	4	214	113	149.57	344	193	130.55
8	8	378	194	127.67	627	354	124.36
16	8	431	205	112.27	679	365	102.19
16	16	755	366	100.45	1228	686	96.08

form 4 to 8, while keeping 4 critical cores, even results in a decrease in terms of the occupied ALMs (407 vs. 389 ALMs). This is due to heuristic optimizations applied by the synthesis tool. This can also be seen when normalizing the number of ALMs to the number of critical cores (column ALMS per m_c). The resource usage per critical core peaks at 110 ALMs and then appears to converge towards 89 ALMs with an increasing number of critical cores. A similar trend can also be observed for the usage of primary logic registers in the *DSC* components, which peaks at 61 registers per critical core and then levels off. Note that non-critical cores still consume resources in the *AR* component, albeit very little.

The maximum operating frequency follows the same trend as resource consumption, with a peak performance of 157.88 MHz that levels off to 97.61 Mhz. However, this time, the total number of cores is the main factor, which leads to a reduction of the clock frequency. This decrease can be attributed to the *AR* component, more precisely, the critical path is related to the logic circuit that selects the next request to be processed by the memory (round-robin) in combination with the masking induced by the *PM* signals. The depth of this logic circuit, and thus the critical path, in our currently unoptimized implementation depends on the total number of cores. It is very likely that an improved implementation (using partitioning and balanced tree structures) and traditional optimization techniques (such as pipelining or retiming) would allow to improve the clock frequency. However, on the considered FPGA this does not appear to be beneficial, since the memory controller operates at only 100 Mhz. Only the configurations with 16 cores are not able to match the controller's speed. However, due to resource constraints (ALMs, registers, and memory) configurations with 16 Patmos cores are not feasible anyways.

Table 2 shows the consequences of reducing the bit-width for deadline and slack counters in terms of ALMs and primary logic registers along with the operating frequency, considering bit-widths of 10 bits and 20 bits. Overall we can see that the bit-width of deadline and slack counters highly impacts the overall results, while following the same trends observed in Table 1 in terms of core numbers. The resource consumption in relation to the 10 bit version increases by a factor of roughly 1.6 and 2 for widths of 20 bits and 24 bits respectively. The operating frequency when using a low number of cores appears to be impacted more by the bit-width, but appears to converge to roughly 100 MHz for configurations with 16 cores. This decrease can be attributed to the *AR* component, in combination with the masking induced by the PM_i signals. For a low number of cores the critical path is more impacted by the computation of the PM_i and ES_i signals within the DSC_i components, which highly depends on the deadline and slack counter width.

From these results, we can conclude that the proposed design appears feasible for a realistic hardware implementation, both in terms of hardware complexity (ALMs/area) and

clock frequency. It remains to verify that the attainable memory utilization of this design matches the original `TDMer` arbiter.

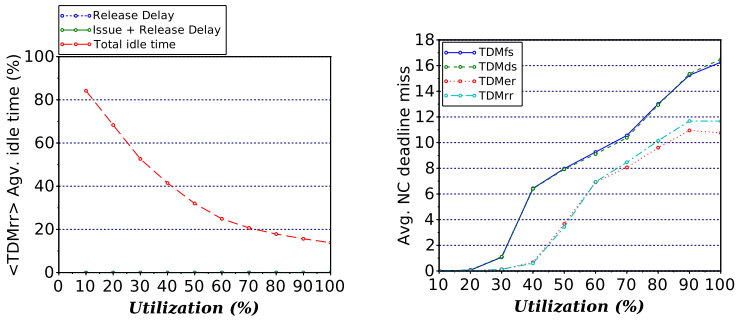
8.3 Results for Dynamic TDM with Round-Robin Arbitration

The designed variant of the dynamic `TDM`-based arbitration policy `TDMrr`, applies a different deadline and slack computation strategy. More importantly, instead of using an EDF arbitration policy with priority queues, `TDMrr` uses a Round-Robin arbitration policy over *all* pending memory requests, while prioritizing requests of critical tasks only when they are about to miss their deadline. The following experiments aim at evaluating whether this choice has an impact on the performance of the dynamic `TDM`-based arbitration scheme at the system level. We, therefore, undertake another series of simulation runs based on the same experimental setup previously used to evaluate `TDMer` in Section 6.

Figure 17a shows a breakdown of the average memory idle time for `TDMrr` over all simulation runs. Recall that, as for Figure 10 from Section 6.5, the plot shows stacked lines representing the release delays, issue delays, and the total memory idling. Overall, the evolution of the memory idling for `TDMrr` shows the same trends as under `TDMer`: memory idling dominates under low system utilization and drops considerably under high load. As before release and issue delays are virtually eliminated. The choice to replace EDF by round-robin apparently did not impact the overall performance of the arbiter negatively.

On the contrary, and somewhat surprising, the performance of `TDMrr` appears to be slightly better than that of `TDMer` starting with a system utilization of about 50%. `TDMer` systematically prioritizes non-critical requests, as long as the critical requests have sufficient slack left. Non-critical tasks systematically *drain* the slack of critical tasks. `TDMrr`, on the other hand, applies Round-Robin among critical and non-critical pending requests alike. Memory bandwidth is, as a result, shared more evenly among critical and non-critical requests, which, most importantly, allows critical tasks to *preserve* more slack on average. This is, apparently, beneficial in some situations where the entire slack of critical tasks was drained under `TDMer`. Our experiments indeed indicate that the average slack for `TDMrr` is higher than that of `TDMer` (not shown by a figure).

This, however, comes at a price. Figure 17b depicts the average number of deadline misses of non-critical tasks. Due to the aforementioned difference between `TDMer` and



(a) Memory idling.

(b) Deadline misses for non-critical tasks.

Fig. 17 Evolution of the average memory idling and average number of deadline misses for non-critical tasks over all simulation runs under `TDMrr` with initial slack (lower is better).

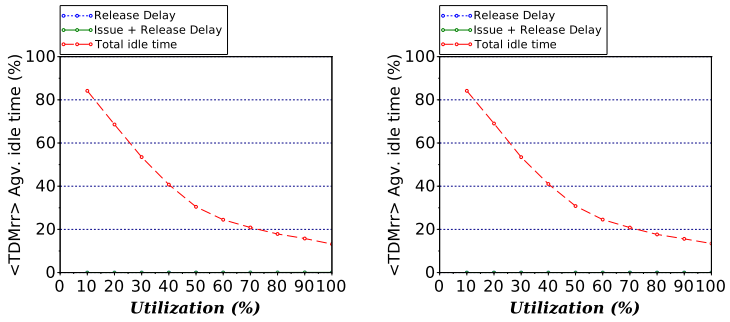
TDMrr, we can notice a difference in terms of deadline misses between TDMer and TDMrr. This trend appears to coincide with the trend regarding the memory idling from before. Starting from 60% system utilization, a difference is visible that increases along with the load. Note, however, that two other factors have a dominating impact on the number of deadline misses: the core number and the ratio between critical and non-critical tasks (50%/50% vs. 25%/75%, see Subsection 6.3). Configurations with at most 4 cores do not cause sufficient memory load, while configurations with more than 16 cores quickly cause overload. The arbitration policy in these situations makes little difference. The situation changes for configurations with 8, 12 or 16 cores. Here, TDMer shows improvements over TDMrr in terms of deadline misses when the repartitioning between critical and non-critical tasks is even (50%/50%). This can be explained by the right level of memory load of these configurations that allows a moderate number of non-critical tasks to exploit a *reasonable* margin of the memory bandwidth.

From these experimental results we conclude that the implementation of our dynamic TDM-based arbitration policy (TDMrr) is efficient, both in terms of hardware complexity and arbitration performance. Notably the good results concerning the memory ideling are preserved over all simulation results, while other metrics are only marginally impacted.

8.4 Results for Bit-Width Constrained Slack Counters

Section 7.4 discusses the consequences of reducing the bit-width of the deadline and slack counters, focusing on implementation and correctness issues. In this section, we turn our attention to the impact on the system-level memory utilization that might result from artificially limiting the bit-width, and thus range, of these counters.

For simplicity, we will considering a hardware implementation of a dynamic TDM-based arbiter running at 100 MHz in the following example. We can then compute the amount of time that can be represented by the slack counters with a given bit-width. Hence, 32 bits corresponds to more than 43 seconds of *slack time* that could theoretically be accumulated by a critical task during execution. This, by far, exceeds the periods (T_i) considered in our simulations, where tasks may exhibit periods in the range $[20, 100]$ ms. For our setup a slack counter width above 24 bits thus cannot impact the arbiter's performance negatively.



(a) 10 bit. (b) 20 bit.

Fig. 18 Evolution of memory idle time considering reduced bit-widths for the slack and deadline counters under TDMrr with initial slack (lower is better).

Reducing the width to 20 bits limits the slack time to roughly 10.5 ms, about 10% of the longest task periods considered in our simulations. One would expect an impact on the arbitration performance. The same applies to a width of 10 bits, which further constrains the slack accumulation to roughly 10 μs, but also reduces hardware costs. For the subsequent experiments, we have chosen to limit the deadline and slack counters to 20 and 10 bits respectively. The experimental setup remains unchanged otherwise (see Section 6.1).

Figure 18 shows the breakdown of the average memory idle times over all simulation runs for `TDMrr`. In terms of memory idling, the results are virtually identical regardless of the used bit-width. However, the average slack counter values are impacted considerably – in particular for configurations with low to medium system utilization. This loss appears to have little impact on the other metrics (deadline misses, execution times, etc.). This is contrary to our observations for the comparison with `TDMer` in the previous subsection, where non-critical tasks drained the slack of critical tasks. The situation is different now considering `TDMrr`, despite the fact that slack is lost, slack counters generally stay above the critical threshold that enables the early-start optimization. As long as critical task have *enough* slack (more than *Sl*, for instance) the arbiter's performance is not degraded. This also applies for all other dynamic `TDM`-based arbitration schemes that are decoupled from slots, i.e., varying the deadline and slack counter width has little to no effect.

9 Related work

Recently two software-based approaches have been proposed to improve `TDM`-based arbitration depending on contention (Tabish et al., 2016; Kritikakou et al., 2014). The first work (Tabish et al., 2016) defines a task model in which tasks are split into sub-tasks consisting of either memory accesses or computation only. The goal is then to find a feasible schedule that ensures that the sub-tasks accessing memory never execute concurrently. The approach thus completely avoids contention by construction by applying `TDM` at a rather high-level of abstraction. However, this approach requires to regroup memory accesses within a single sub-task, e.g., by using a scratch-pad memory, which entails a considerable change in the underlying programming/execution model. Kritikakou et al. (2014) track the slack time of critical tasks in software. Non-critical tasks can access memory as long as all critical tasks still have slack left, otherwise *all* non-critical tasks are stopped. In our case, non-critical tasks may always continue execution and arbitration is performed at the granularity of clock cycles.

MemGuard (Yun et al., 2013) ensures isolation between cores by implementing a credit-based approach for tracking memory requests in software. Tasks executed by a core are suspended when the budget of memory requests, periodically assigned to the core, is depleted. A reclaim manager can donate *predicated* non-used budget of memory requests to other cores, making the approach suitable for soft real-time systems only. Agrawal et al. (2017) extend the MemGuard memory bandwidth throttling approach (Yun et al., 2013) to upper bound the WCET using slot-based time-triggered systems. It constructs schedule tables, assigning partitions and dynamic memory bandwidth to each slot on each core. At runtime, two servers jointly control the contention between the cores, and the amount of memory accesses per slot.

A common approach is to improve the resource utilization of `TDM` by increasing the number of `TDM` slots according to task weights (Yoon et al., 2011). Others apply strict `TDM` arbitration to critical tasks (Gomony et al., 2017), while allowing other schemes for non-

critical tasks. In both cases the TDM strategy itself is not modified, which remains heavily non-work-conserving.

Another approach (Fohler, 1995) uses a similar idea as our dynamic arbitration schemes: shifting *slots*. However, it deals with a different problem: minimizing the response time of aperiodic tasks in statically-scheduled periodic hard real-time systems. Shifting slots is thus applied for scheduling tasks and not memory requests. The granularity of slots allocated to tasks is consequently much larger than individual memory requests considered in our work.

Kostrzewa et al. (2015) propose a technique to dynamically adapt the arbiter to a varying number of *active* tasks, which execute under regular TDM. The approach thus does not address the non-work-conserving nature of TDM. Li et al. (2016) truly *skip* unused entries in a TDM schedule in order to allow for variable-sized TDM slots. Similarly, Hassan et al. (2017) combine a slot skipping technique with a harmonic TDM schedule. Like our approach, slot skipping improves upon the non-work-conserving behavior of TDM. However, both approaches are tied to the notion of TDM slots that are processed by a fixed schedule. Our deadline-driven approach, on the other hand, allows to dynamically reorder memory requests, while keeping the same guarantees as TDM for critical tasks. Most notably our approach is analyzable since it preserves TDM slot offsets (Kelter et al., 2014; Rihani et al., 2015). In our case, the behavior of a task only depends on *its own history*, as opposed to the aforementioned approaches where memory latency/interference is highly dependent on the behavior of other tasks.

Jun et al. (2007) propose a slack-aware arbiter at the granularity of individual requests. However, slack is statically defined by a fixed parameter for each core (master) that is independent from the actual load on the main memory. Timing errors may thus occur, since it cannot be guaranteed that requests complete before their slack is entirely consumed. Also, slack cannot be accumulated across successive requests.

Finally, Kostrzewa et al. (2016) propose a mechanism which provides latency guarantees for hard real-time transmissions in a network-on-chip with a minimum impact on performance sensitive best-effort transmissions. They use a slack-based global and dynamic prioritization of data streams. However, slack for each *critical* task is computed off-line and preset to a fixed value at the beginning of jobs. In our work slack is accumulated on-line across successive memory requests, starting with an initial slack of 0. Note, however, that the slack counters could also be initialized to a non-zero value, e.g., derived by a schedulability analysis. In addition, other forms of slack could be considered in our approach at runtime, e.g., slack with regard to a task's worst-case execution time when diverging from the worst-case execution path.

10 Conclusion and future work

This work presents TDMer, a dynamic TDM-based arbitration policy. Instead of arbitrating at the level of TDM slots, our approach operates at the granularity of clock cycles by exploiting slack time accumulated from preceding requests. In addition to the successful elimination of the release delays by TDMer, a relatively small initial slack counter value at the start of each critical job enables to also eliminate the residual issue delays, one of the two sources of inefficiency in TDM arbitration schemes. Our evaluation reveals considerable gains, in particular, when approaching high system utilization. We furthermore proposed a hardware implementation of a slightly simplified variant of the approach (TDMrr). Using additional simulations we showed that the proposed solution combines the advantages of dynamic TDM-based scheduling with an efficient and simple hardware realization.

Our dynamic TDM-based arbitration policies come with a set of restrictions, that we plan to remove as future work. First, we limited the discussions to uniform TDM schedules, where all TDM slots have the same length and each critical task is assigned a single slot per period. However, the proposed approach can be adapted to other kinds of TDM schedules proposed in the literature, such as weighted and harmonic TDM (Hassan et al., 2017; Yoon et al., 2011). The only requirement is that the deadline of a request is (easily) computable. In some cases, it might also be useful to vary the slot size depending on the bandwidth or throughput requirements of a (critical) task, e.g., to accommodate burst transfers. Scheduling a burst transfer independently from TDM slots may then cause overruns touching several immediately following TDM slots. The simple admission test (Algorithm 6) then needs to be extended in order to consider a bound of the burst's transfer time, as well as the slack counter values of all potentially concerned slot owners.

Another restriction is that our dynamic TDM-based arbitration policies currently assume independent periodic tasks where each task executes on a separate core. Interactions between dependent tasks may however impact the timing behavior. For instance, a task may wait for another task (e.g., release of a lock) or to wait for a precise instant (e.g., 11:30 AM). In particular in the latter case, it is easy to see that the accumulated slack before the wait operation is meaningless afterwards. It then suffices to reset the slack counter of the waiting task. Other forms of interactions might allow to preserve the slack counter as long as the blocking can be bounded by a duration (as opposed to blocking up to an instant). Otherwise, it suffices to reset the task's slack counter.

However, the most fundamental restriction is the one-to-one mapping between tasks and cores. The main issue here stems from the fact that requests issued to the arbiter may take a considerable amount of time to complete, which would delay interrupts and, consequently, preemptions. We recently extended our system model to address this issue (Hebbache et al., 2019). The new work allows several tasks to execute on a single core and even allows to mix critical and non-critical tasks on a given core. This in turn necessitates means to preempt ongoing transfers or to limit the blocking delay that preempting tasks may suffer via hardware support. Both options are explored and techniques are proposed that allow to take the resulting delays into consideration during schedulability analysis. Slack counters, in either case, are part of the execution context of a task and need to be saved/restored accordingly. The number of DSC_i components, required to track the slack of a all tasks in a task set, is consequently limited by the number of cores in the system. Also, cores may dynamically emit critical and non-critical requests depending on whether they execute a critical or non-critical task.

Finally, we plan to develop a technique to exploit the ability to reorder requests (e.g., for DDR memory command scheduling) and to exploit alternative forms of slack.

References

Agrawal A, Fohler G, Freitag J, Nowotsch J, Uhrig S, Paulitsch M (2017) Contention-aware dynamic memory bandwidth isolation with predictability in COTS multicores: An avionics case study. In: 29th Euromicro Conference on Real-Time Systems (ECRTS), Schloss Dagstuhl, LIPIcs, vol 76, pp 2:1–2:22

Baruah SK, Burns A, Davis RI (2011) Response-time analysis for mixed criticality systems. In: 32nd Real-Time Systems Symposium (RTSS), Vienna, Austria, pp 34–43

Burns A, Davis RI (2017) A survey of research into mixed criticality systems. ACM Comput Surv 50(6):82:1–82:37, DOI 10.1145/3131347

Emberson P, Stafford R, Davis R (2010) Techniques for the synthesis of multiprocessor tasksets. In: Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS), pp 6–11

Fohler G (1995) Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In: Proc. Real-Time Systems Symposium (RTSS), Pisa, Italy, pp 152–161

Gomony MD, Garside J, Akesson B, Audsley N, Goossens K (2017) A globally arbitrated memory tree for mixed-time-criticality systems. IEEE Trans Comput 66(2):212–225

Guthaus MR, Ringenberg JS, Ernst D, Austin TM, Mudge T, Brown RB (2001) Mibench: A free, commercially representative embedded benchmark suite. In: Proc. of the Int. Workshop on Workload Characterization, pp 3–14

Hahn S, Reineke J, Wilhelm R (2015) Towards compositionality in execution time analysis: Definition and challenges. SIGBED Rev 12(1):28–36

Hansen J, Hissam S, Moreno GA (2009) Statistical-Based WCET Estimation and Validation. In: Intl. Workshop on Worst-Case Execution Time Analysis (WCET'09), Schloss Dagstuhl, OASIcs, vol 10, pp 1–11

Hassan M (2018) On the off-chip memory latency of real-time systems: Is ddr dram really the best option? In: 2018 IEEE Real-Time Systems Symposium (RTSS), pp 495–505, DOI 10.1109/RTSS.2018.00062

Hassan M, Patel H, Pellizzoni R (2017) PMC: A requirement-aware DRAM controller for multicore mixed criticality systems. ACM Trans Embed Comput Syst 16(4):100:1–100:28, DOI 10.1145/3019611

Hebbache F, Jan M, Brandner F, Pautet L (2017) Dynamic arbitration of memory requests with tdm-like guarantees. In: International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS'17)

Hebbache F, Jan M, Brandner F, Pautet L (2018) Shedding the shackles of time-division multiplexing. In: 2018 IEEE Real-Time Systems Symposium, RTSS 2018, Nashville, TN, USA, December 11-14, 2018, pp 456–468, DOI 10.1109/RTSS.2018.00059, URL https://doi.org/10.1109/RTSS.2018.00059

Hebbache F, Jan M, Brandner F, Pautet L (2019) Arbitration-induced preemption delays. In: 31th Euromicro Conference on Real-Time Systems (ECRTS), Schloss Dagstuhl, LIPIcs, vol 133, too appear

Jan M, Zaourar L, Pitel M (2013) Maximizing the execution rate of low-criticality tasks in mixed criticality systems. In: Proc. of the 1st Intl. Workshop on Mixed Criticality Systems (WMC), Vancouver, Canada, pp 43–48

Jun M, Bang K, Lee HJ, Chang N, Chung EY (2007) Slack-based bus arbitration scheme for soft real-time constrained embedded systems. In: Asia and South Pacific Design Automation Conf., pp 159–164

Kelter T, Falk H, Marwedel P, Chattopadhyay S, Roychoudhury A (2014) Static analysis of multi-core TDMA resource arbitration delays. Real-Time Syst 50(2):185–229

Kostrzewa A, Saidi S, Ecco L, Ernst R (2015) Flexible TDM-based resource management in on-chip networks. In: Int. Conf. on Real Time and Networks Systems, ACM, pp 151–160

Kostrzewa A, Saidi S, Ernst R (2016) Slack-based resource arbitration for real-time networks-on-chip. In: Proceedings of the Conference on Design, Automation & Test in Europe, EDA, DATE '16, pp 1012–1017

Kritikakou A, Pagetti C, Roy M, Rochange C, Faugère M, Girbal S, Gracia Pérez D (2014) Distributed run-time WCET controller for concurrent critical tasks in mixed-critical systems. In: Int. Conf. on Real-Time Networks and Systems

Li Y, Akesson B, Goossens K (2016) Architecture and analysis of a dynamically-scheduled real-time memory controller. Real-Time Syst 52(5):675–729

Paolieri M, Quiñones E, Cazorla FJ (2013) Timing effects of ddr memory systems in hard real-time multicore architectures: Issues and solutions. ACM Trans Embed Comput Syst 12(1s):64:1–64:26

Rihani H, Moy M, Maiza C, Altmeyer S (2015) WCET analysis in shared resources real-time systems with TDMA buses. In: Proceedings of the 23rd International Conference on Real Time and Networks Systems, ACM, RTNS '15, pp 183–192, DOI 10.1145/2834848.2834871

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

Tabish R, Mancuso R, Wasly S, Alhammad A, Phatak SS, Pellizzoni R, Caccamo M (2016) A real-time scratchpad-centric os for multi-core embedded systems. In: Real-Time and Embedded Technology and Applications Symposium (RTAS), pp 1–11

Vestal S (2007) Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In: 28th IEEE International Real-Time Systems Symposium (RTSS 2007), pp 239–243, DOI 10.1109/RTSS.2007.47

Wu ZP, Krish Y, Pellizzoni R (2013) Worst case analysis of dram latency in multi-requestor systems. In: Real-Time Systems Symposium (RTSS), Vancouver, Canada, pp 372–383

Wuertz D, Setz T, Chalabi Y (2017) Rmetrics - Modelling Extreme Events in Finance. URL http://www.rmetrics.org

Yoon MK, Kim JE, Sha L (2011) Optimizing tunable WCET with shared resource allocation and arbitration in hard real-time multicore systems. In: Real-Time Systems Symp., Vienna, Austria, pp 227–238

Yun H, Yao G, Pellizzoni R, Caccamo M, Sha L (2013) Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In: Real-Time and Embedded Technology and Applications Symposium (RTAS), pp 55–64