Work-conserving dynamic TDM-based memory arbitration for multi-criticality real-time systems

Farouk Hebbache

To cite this version:

Farouk Hebbache. Work-conserving dynamic TDM-based memory arbitration for multi-criticality real-time systems. Hardware Architecture [cs.AR]. Université Paris Saclay (COmUE), 2019. English. NNT : 2019SACLT044 . tel-02494421

HAL Id: tel-02494421

https://pastel.hal.science/tel-02494421v1

Submitted on 28 Feb 2020

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

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

NNT : 2019SACLT044

Thèse de doctorat

Work-conserving dynamic TDM-based memory arbitration for multi-criticality real-time systems

Thèse de doctorat de l'Université Paris-Saclay préparée au CEA, List

Ecole doctorale n°580 Sciences et Technologies de l'Information et de la Communication (STIC)

Spécialité de doctorat: Programmation : modèles, algorithmes, langages, architecture

Thèse présentée et soutenue à Palaiseau, le 28/11/2019, par

FAROUK HEBBACHE

Composition du Jury :

Alain Mérigot Professeur, Université Paris-Sud (UFR Sciences)	Président
Claire Pagetti Ingénieur de recherche, ONERA (DTIM)	Rapporteur
Christine Rochange Professeur, Université de Toulouse (IRIT)	Rapporteur
Steven Derrien Professeur, Université de Rennes 1 (IRISA)	Examinateur
Frédéric Pétrot Professeur, Institut polytechnique de Grenoble (TIMA)	Examinateur
Laurent Pautet Professeur, Télécom Paris (LTCI)	Directeur de thèse
Florian Brandner Maître de conférence, Télécom Paris (LTCI)	Encadrant
Mathieu Jan Ingénieur-chercheur, CEA, List (L3S)	Encadrant

Acknowledgements

Je tiens tout d'abord à remercier grandement mon directeur de thèse, Laurent Pautet et mes deux encadrants Florian Brandner et Mathieu Jan, pour toute l'aide qu'ils m'ont apporté tout au long de cette thèse. Qu'ils soient aussi remerciés pour leurs disponibilités permanentes et pour les nombreux encouragements qu'ils m'ont prodigué.

Le travail qui est présenté dans ce mémoire a été effectué au CEA List au sein du Laboratoire L3S. Aussi, merci à messieurs E. Hamelin, O. Heron et C. Gamrat, de m'avoir accueilli et mis à ma disposition tous les moyens nécessaires aussi bien techniques que financiers, afin de mener à bien ce travail.

Merci à monsieur A. Mérigot pour avoir accepté de présider mon jury de thèse.

Merci à Madame C. Pagetti et madame C. Rochange, pour avoir accepté d'étudier mes travaux et d'en être les rapporteurs.

Merci à messieurs S. Derrien et F. Pétrot de l'attention qu'ils portent à mon travail en acceptant d'être membres du jury.

J'aimerais ensuite remercier toutes les personnes que j'ai rencontrées au L3S tout au long de ma thèse. En particulier, je tiens à remercier mes anciens collègues Daniel V., Martin Z., Philippe G., Emine L., Massinissa A., Belgacem B., Mihail A., et Briag L. pour toutes les discussions intéressantes qu'ont n'a pu avoir. Je voudrais également exprimer ma gratitude à Marie-Isablle G., secrétaire du Département DACLE, pour son aide durons ces trois dernieres années.

Un grand merci à mon épouse Katia pour son soutien inconditionnel et surtout supporté dans tout ce que j'ai entrepris.

Enfin, je voudrais exprimer ma profonde gratitude à mes parents, Ammar et Saliha, ainsi que mes frères et soeur Manaouir, Koceila, Islam et Thiziri pour leurs soutiens et leurs aides.

UNIVERSITÉ PARIS SACLAY

Résumé

Arbitrage mémoire dynamique non-oisif basé sur TDM pour des systèmes multi-criticité temps réel

par Farouk HEBBACHE

Les systèmes temps-réel doivent réagir de manière fiable, ce qui implique à la fois d'être certain du résultat produit par leurs programmes mais aussi de connaître le temps qu'ils prennent pour s'exécuter. Les pire temps d'exécution sont ainsi des données fondamentales pour la validation et la sûreté de tels systèmes temps-réel, et encore plus dans le contexte des systèmes temps-réel autonomes (robotique, voiture autonome) pour lesquels la sûreté de fonctionnement est primordiale.

Cependant calculer un pire temps d'exécution (dit WCET pour Worst-Case Execution Time), à la fois garanti (majorant strict) et pas trop pessimiste afin de réduire les coûts et la complexité de tels systèmes temps-réel, est un problème difficile à résoudre sur des architecture matérielles multi-cœurs. Une difficulté est la concurrence d'exécution entre différents programmes souhaitant accéder à une ressource partagée, typiquement une mémoire. Pour chaque requête d'accès à une telle ressource partagée émise par un programme, il faut en effet systématiquement considérer la situation la plus défavorable induite par la politique d'arbitrage utilisée. Ceci induit un pessimisme important dans la valeur du WCET obtenue et donc un faible taux d'utilisation de cette ressource partagée lors de l'exécution des programmes. Ce problème de sous-utilisation de la ressource partagée est amplifié par la multiplication de programmes non-soumis à des contraintes temporelles (dits programmes non-critiques) qui s'exécutent en parallèle des programmes temps-réel (dits programmes critiques). Les requêtes d'accès générées par ces programmes non-critiques impactent la situation la plus défavorable qui doit être considérée pour les requêtes d'accès émises par les programmes critiques et accentuent donc le pessimisme des WCET calculés.

Il est possible d'éliminer par construction toute concurrence entre les requêtes d'accès émises par les différents programmes en ayant recours à une politique d'arbitrage à multiplexage temporel (en anglais *Time-Division Multiplexing*, TDM). Selon cette politique, le temps est divisé en

créneaux temporels chacun alloué à un programme prédéfini pour un accès exclusif à la ressource partagée. Le temps d'accès à la ressource partagée par un programme peut alors être facilement borné. La bande passante offerte à un programme est indépendante des autres programmes et les WCET des programmes peuvent ainsi être déterminés.

Toutefois, le temps d'accès à la ressource partagée dépend de l'ordonnancement des créneaux temporels qui lui sont affectés. Or cet ordonnancement est généralement statique car réalisé lors de la conception du système, avant l'exécution des programmes. Il comprend généralement l'affectation d'une séquence de créneaux temporels à différents programmes, cette séquence se répétant périodiquement (on parle de période TDM). Mais les créneaux temporels d'une période TDM ne sont utilisés par les programmes que s'ils ont une requête d'accès à la ressource partagée à émettre. Lorsque cela n'est pas le cas, ces créneaux temporels sont donc inutilisés. Une politique d'arbitrage TDM basique est donc dite oisive car ces créneaux temporels ne sont pas récupérés par les autres programmes afin de diminuer leur temps d'accès à la ressource partagée.

Par ailleurs, il est admis que pour les programmes temps-réel, il peut exister une large différence entre le temps d'exécution d'une instance du programme et son WCET. L'oisiveté de la politique d'arbitrage TDM associée à cette caractéristique des programmes temps réel génère un faible taux d'utilisation de la ressource partagée. Ce problème est amplifié lorsque le nombre de programmes augmente car dans ce cas la longueur d'une période TDM est également augmentée. Un autre facteur d'amplification de ce phénomène est la présence au sein des systèmes considérés d'un nombre croissant de programmes non-critiques pour lesquels l'affectation de créneaux augmente la latence des requêtes des programmes critiques ainsi que la latence des requêtes de ces programmes non-critiques, en contradiction avec leur objectif d'avoir les meilleurs performances d'exécution en moyenne.

Les travaux menés dans le cadre de cette thèse présentent une reconstruction de la politique d'arbitrage TDM qui permet d'en réduire l'oisivité. Plutôt que de requérir à un arbitrage au niveau des créneaux TDM, notre approche dynamique fonctionne à la granularité des cycles d'horloge en exploitant du temps gagné par les requêtes précédentes. Cela permet au mécanisme d'arbitrage de réorganiser le traitement des requêtes mémoire, exploiter les latences d'accès réelles des requêtes, et ainsi d'améliorer l'utilisation de la mémoire. Nous démontrons que nos politiques d'arbitrage sont analysables car elles préservent les garanties temporelles qu'offre TDM dans le pire cas, alors que nos expérimentations montrent une meilleure utilisation de la mémoire en moyenne. De plus, une variante de notre stratégie d'arbitrage est présentée et

évaluée dans le cadre d'une implémentation matérielle à faible complexité. Les évaluations montrent que l'implémentation est efficace, tant en matière de complexité matérielle que de performance d'arbitrage concernant l'utilisation de la mémoire.

Pour finir, nous explorons l'applicabilité de nos approches dans le contexte d'un système préemptif, où nos approches peuvent induire du fait d'une préemption des délais de blocage mémoire supplémentaire en fonction de l'historique d'exécution. Ces délais de blocage peuvent induire une gigue importante et par conséquent augmenter les temps de réponse des tâches. Ainsi nous étudions des moyens de gérer et, enfin, de limiter ces délais de blocage. Trois différentes stratégies sont explorées et comparées du point de vue de leur analysabilité, de l'impact sur l'analyse des temps de réponse, et de la complexité de l'implémentation matérielle ainsi que du point de vue du comportement à l'exécution. Les expérimentations montrent que les différentes approches se comportent sensiblement de manière identique au cours de l'exécution. Cela nous permet de retenir l'approche combinant analysabilité et une faible complexité d'implémentation matérielle.

Mots-clés : *Time-Division Multiplexing, Arbitrage Dynamique, Calcul Prédictible, Systèmes Multi-Critiques, Systèmes Temps Réel, Mémoire, Préemption*

UNIVERSITÉ PARIS SACLAY

Abstract

Work-conserving dynamic TDM-based memory arbitration for multi-criticality real-time systems

by Farouk HEBBACHE

Real-time systems have to reliably react, which implies both being confident regarding the result produced by the programs and also knowing how long they take to execute. The Worst-Case Execution Times (WCETs) are thus fundamental information for the validation and reliability of such real-time systems, especially in the context of autonomous real-time systems (robotics, autonomous car, GPS) for which reliability is essential. However, computing a WCET, both guaranteed (strict upper-bound) and also not too pessimistic in order to reduce the costs and complexity of such real-time systems, is a challenging problem to solve on multi-core hardware architectures.

Multi-core architectures pose many challenges in real-time systems, which arise from contention between concurrent accesses to shared memory. Various memory arbitration schemes have been devised that address these issues, by providing trade-offs between predictability, average-case performance, and analyzability. 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.

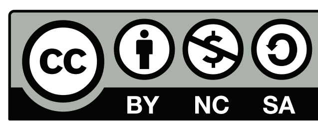
To address the aforementioned limitations of TDM, we present, in this thesis, dynamic arbitration schemes that are based on TDM. However, instead of arbitrating at the level of TDM slots, our approach operates at the granularity of clock cycles by exploiting *slack* time accumulated from preceding requests. 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. Our evaluations showed that the implementation is efficient, both in terms of hardware complexity and arbitration performance regarding the memory utilization.

Finaly, we explore the applicability of our approaches in a preemptive system model, where preemption-related memory blocking times can occur, depending on execution history. These blocking delays may induce significant jitter and consequently increase the tasks' response times. We thus explore means to manage and, finally, bound these blocking delays. Three different schemes are explored and compared with regard to their analyzability, impact on response-time analysis, implementation complexity, and runtime behavior. Experiments show that the various approaches behave virtually identically at runtime. This allows to retain the approach combining low implementation complexity with analyzability.

Keywords: Time-Division Multiplexing, Dynamic Arbitration, Predictable Computing, Multi-Criticality Systems, Real-Time Systems, Memory, Preemption

This work is licensed under a Creative Commons "Attribution-NonCommercial-ShareAlike 3.0 Unported" license.



Contents

Acknowledgements iii

Résumé v

Abstract ix

List of Publications xvii

1 Introduction 1
- 1.1 Context 1
- 1.2 Contribution Overview 2
- 1.3 Thesis Outline 4

I Background/State of the art 7

2 Real-Time Systems 9
- 2.1 Real-time Systems 10
  - 2.1.1 Definitions and Properties 10
    - Real-Time Tasks 11
    - Task Specification 12
  - 2.1.2 Worst-Case Execution Time Analysis 14
  - 2.1.3 Real-Time Scheduling 16
  - 2.1.4 Scheduling Analysis 18
- 2.2 Mixed-Criticality Systems 19
- 2.3 Conclusion and Considered System Model 20
  - 2.3.1 Task Model 20
  - 2.3.2 Scheduling Policy 21

3 Multi-Core Memory Access Interference 23
- 3.1 Memory Hierarchy 24

3.1.1 Registers 25
3.1.2 Scratchpad 25
3.1.3 Caches 26
3.1.4 Main Memory (DRAM) 27
3.2 Memory Arbitration Schemes 29
3.2.1 Fixed-Priority 30
3.2.2 First-Come First-Served 30
3.2.3 Round-Robin 31
3.2.4 Time-Division Multiplexing 32
3.2.5 Budget-Based Arbitration 34
3.2.6 TDM Arbitration and Multi-Criticality Scheduling 35
3.3 Conclusion and Assumed Hardware Architecture 36

II Contribution **37**

4 Problem Statement and Contribution Summary **39**
4.1 Thesis Problem Statement 39
4.1.1 Criticality-Aware Arbitration Schemes 40
4.1.2 Challenges with Time-Division Multiplexing 41
4.1.3 Preemption Costs for Regular TDM Arbitration 44
4.2 Contribution Outline 46

5 Dynamic TDM-based Arbitration **47**
5.1 Criticality Aware TDM-based Arbitration (TDMfs) 48
5.2 Dynamic TDM-based Arbitration Schemes 50
5.2.1 TDMdz: Deadline Driven Arbitration 50
5.2.2 TDMds: Dynamic TDM Arbitration with Slack Counters 52
5.3 Decoupling Dynamic TDM Arbitration from Slots 55
5.3.1 TDMes: Decoupled from TDM slots 55
5.3.2 TDMer: Memory Variability Awareness 58
5.4 Worst-Case Behavior 59
5.5 Experiments for Dynamic TDM-Based Arbitration Schemes 63
5.5.1 Experimental Setup 63
5.5.2 Results for Dynamic TDM-Based Arbitration Schemes 68
5.5.3 Results for Dynamic TDM with Initial Slack 71
5.5.4 Results for Varying Memory Access Latencies 74

5.6 Conclusion 77

6 **Dynamic TDM-based Arbitration Hardware Design** **79**
6.1 Architecture Overview 80
6.2 Arbitration Logic 81
6.2.1 Deadline and Slack Computation 83
6.2.2 Update Rules 84
6.3 Control Signal Generation 85
6.4 Bit-Width Considerations 86
6.5 Worst-case Behavior w.r.t. the Hardware Design 87
6.6 Experiments 93
6.6.1 Evaluation Platform 93
6.6.2 Results for Hardware Synthesis 94
6.6.3 Results for Dynamic TDM with Round-Robin Arbitration 96
6.6.4 Results for Bit-Width Constrained Slack Counters 98
6.7 Conclusion 99

7 **Multi-tasks and Preemption Support** **101**
7.1 Multi-task Preemptive Scheduling Policy 102
7.2 Preemption Costs for Dynamic TDM-based Arbitration 104
7.3 Arbitration-Aware Preemption Techniques 105
7.3.1 Scheduling with Request Waiting (`SHDw`) 106
7.3.2 Scheduling with Request Preemption (`SHDp`) 107
7.3.3 Scheduling with Criticality Inheritance (`SHDi`) 108
7.3.4 Misalignment Delays 110
7.3.5 Response-Time Analysis 111
7.4 Experiments 112
7.4.1 Experimental Setup 112
7.4.2 Results for Preemption Schemes 113
7.4.3 Results for (Preemptive) Arbitration Schemes 116
7.4.4 Results for Varying Memory Access Latencies 119
7.5 Conclusion 121

III Conclusion **123**

8 **Conclusion and Future Work** **125**

8.1 Conclusion 125
8.2 Future Work 126

Bibliography **129**

List of Figures

2.1 Overview of a task specification. 13
2.2 Task execution time distribution, figure taken from [76]. 15

3.1 Overview of a computer memory hierarchy, figure taken from [26]. 25
3.2 Overview of a DRAM organization, figure taken from [42]. 28
3.3 Regular TDM arbitration of three tasks τ_0, τ_1, and τ_2. 32

4.1 Non-work-conserving of TDM. 41
4.2 TDM slots induced delays. 42
4.3 Preemption effects w.r.t. TDM memory arbitration. 44

5.1 Criticality aware TDMfs arbitration of three tasks τ_0^c, τ_1^c and τ_2^{nc}. 49
5.2 Deadline driven TDMdz arbitration of three tasks τ_0^c, τ_1^c and τ_2^{nc}. 52
5.3 Deadline driven and slack time accumulation TDMds arbitration of three tasks τ_0^c, τ_1^c and τ_2^{nc}. 53
5.4 Reduced issue delays due to the TDMes arbiter, which operates independently from the actual TDM slot length. 57
5.5 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. 58
5.6 Empirical distributions of the request distances (in cycles) of two MiBench applications compared with the GEV distributions after fitting. 66
5.7 Comparison of the memory (MD) and processor (PD) demand for the approx. 1500 jobs of the simulation runs with 24 tasks. 67
5.8 Evolution of memory idle time, issue, and release delays over all simulations under varying utilization (lower is better). 69
5.9 Normalized sum of issue and release delays for dynamic arbitration schemes compared to TDMfs (higher is better). 70
5.10 Memory idle time considering 24 tasks with 6 critical and 18 non-critical tasks (lower is better). 71

5.11 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). 72
5.12 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). 73
5.13 Normalized sum of issue and release delays for dynamic arbitration schemes compared to `TDMfs`, with initial slack set to *Sl* (higher is better). 73
5.14 Results considering a `TDM` slot length of $Sl = 25$ cycles (lower is better). 75
5.15 Results considering a `TDM` slot length of $Sl = 100$ cycles (lower is better). 76

6.1 Overview of the hardware design of our dynamic `TDM`-based arbiter. 81
6.2 Summary of the update rules of the Deadline and Slack Computation (*DSC*) components. 84
6.3 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). 97
6.4 Evolution of memory idle time considering reduced bit-widths for the slack and deadline counters under `TDMrr` with initial slack (lower is better). 99

7.1 Hardware architecture. 103
7.2 Impact of slack accumulation on the memory blocking time for `TDMds`. 105
7.3 Memory blocking delays considering request preemption (`SHDp`). 108
7.4 Memory blocking delays considering criticality inheritance (`SHDi`). 110
7.5 Memory blocking delays across normalized system utilization for `SHDw`. 113
7.6 Maximum memory blocking delay for critical tasks across normalized system utilization. 115
7.7 Average *schedulability* success ratio through normalized system utilization. 116
7.8 Average jobs execution times with varying normalized system utilization using `SHDi`. 117
7.9 Average number of non-critical deadline misses. 118
7.10 Average results through normalized system utilization, with `TDM` slot lengths of 25 cycles. 119
7.11 Average results through normalized system utilization, with `TDM` slot lengths of 100 cycles. 120

List of Publications and Patents

Publications

[HJBP17] Farouk Hebbache, Mathieu Jan, Florian Brandner, and Laurent Pautet, **Dynamic Arbitration of Memory Requests with TDM-like Guarantees**, In *10th International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS 2017)*.

[HJBP18a] Farouk Hebbache, Mathieu Jan, Florian Brandner, and Laurent Pautet, **Shedding the Shackles of Time-Division Multiplexing**, in *39th (IEEE) Real-Time Systems Symposium, (RTSS 2018)*.

[HBJP19a] Farouk Hebbache, Florian Brandner, Mathieu Jan, and Laurent Pautet, **Arbitration-Induced Preemption Delays**, In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*.

[HBJP19b] Farouk Hebbache, Florian Brandner, Mathieu Jan, and Laurent Pautet, **Work-conserving dynamic time-division multiplexing for multi-criticality systems**, In Special Issue on Addressing the Real-Time Challenges of Multicore Architectures of the *Real-Time Systems* journal, 2019.

Patent

[HBJP18b] Farouk Hebbache, Florian Brandner, Mathieu Jan, and Laurent Pautet, **Procédé de multiplexage temporel pour l'accès à des ressources partagées**, patent submitted in October 2018. French deposit number 1860117.

Chapter 1

Introduction

Contents

1.1 Context 1
1.2 Contribution Overview 2
1.3 Thesis Outline 4

1.1 Context

A current trend in real-time embedded systems is integrating multiple tasks into a single platform. However, the tasks in domains like avionics and automotive are usually characterized with different importance, safety, or certification requirements – leading to several criticality levels. Running such tasks on a single platform creates a *Multi-Criticality* (MC) system. Hence, integrating such diverse tasks on a single platform requires complete *timing* and *spatial* isolation. This isolation prevents a critical task from being unintentionally affected by another non-critical task. To achieve spatial isolation, i.e., protecting task's state (memory and stored registers), designers usually rely on partitioning mechanisms at the platform level. On the other hand, temporal isolation guarantees a task's desired timing behavior. Achieving temporal isolation is not trivial, especially when considering a multi-core platform with shared resources, such as caches or main memory. This results in challenging issues when trying to tightly bound the tasks' worst-case execution times (WCETs) and avoid over-provisioning hardware resources. For the verification and certification of critical tasks, the most important element is the respect of their timing constraints. For non-critical tasks, however, effectively using the hardware resources is more important than isolation and predictability guarantees as long as critical tasks are not affected.

As mentioned above, multi-core architectures pose many challenges. One main challenge arises from contention between concurrent accesses to shared resources. If several cores access

shared memory, for instance, in an uncontrolled manner, interference among tasks of different criticalities cannot be avoided. Then, a non-critical task accessing the memory bus can block the access of any other concurrently executed task. Hence, each access of non-critical tasks can potentially affect the response time of critical tasks on other cores. A way of bounding the interference independently from the concurrent execution of other tasks, is to use temporal isolation [16]. It is possible to achieve temporal isolation on any hardware, but this often leads to over-provisioning of the hardware resources. This means providing more hardware resources then actually needed, which may guarantee a risk-free execution but with an underused platform. Techniques like *Time-Division Multiplexing* (TDM) achieve such a temporal partitioning, concurrent accesses here 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. 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. TDM is rarely used as a bus-arbitration policy in multi-core architectures, since it is not *work-conserving* and therefore causes hardware over-provisioning. The problem arises when the owner of a TDM slot does not (yet) have a memory request ready to be served. Under a strict TDM scheme, this slot cannot be reclaimed by another task (as under Round-Robin arbitration). This problem is further amplified as the number of cores increases, leading to longer TDM schedules. Another source of TDM pessimism stems from the length of TDM slots, expressed in clock cycles, which have to be longer than the worst-case latency of handling memory requests. Memory requests targeting a DRAM memory, however, have highly variable latencies [77], as 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 load requests are higher than memory writes, since data must be sent back to the requesting core.

1.2 Contribution Overview

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 explore TDM-based arbitration schemes that allow the arbiter, under certain conditions, to favor requests of non-critical tasks over requests from critical tasks. This is achieved by associating deadlines to the memory accesses of critical tasks, which correspond to the end of their corresponding slots under a traditional TDM scheme.

These deadlines allow the arbiter to compute the *slack time* (the relative gain compared to an execution under regular TDM) of each pending request from critical tasks in the system. If slack times permit, the arbiter can change the order in which requests are handled by re-allocating slots unused by critical tasks to non-critical tasks. This arbitration policy is called TDMds, for *dynamic TDM with slack counters*, and addresses the source of traditional TDM pessimism. Afterwards, we extend TDMds by proposing two dynamic TDM arbitration schemes (TDMes and TDMer) 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 at the granularity of clock cycles. Our experiments show that this allows improving delays suffered by traditional TDM by a factor of at least 50, and up to a factor of 300. Another contribution is a formal correctness proof of dynamic TDM-based approaches. Most notably, we prove that TDM's temporal behavior is preserved for critical tasks. Consequently, analysis results valid under TDM, such as offset analyses [41], are equally valid under our schemes.

We also propose 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. A formal proof of the worst-case behaviour for the new approach is also addressed.

The proposed dynamic TDM-based arbitration techniques encounter issues under a preemptive execution model, more precisely, the overhead induced by the arbitration scheme on a preemption. Under our dynamic TDM-based arbitration schemes, tasks may suffer from arbitration-induced preemption delays. Therefore, we define two memory delays induced by preemptions, the *memory blocking delay* and the *misalignment delay*, which may lead to significant jitter and increase task response times. Even worse, due to non-critical tasks, the memory blocking delay may be unbounded in some circumstances. We explore three different approaches to analyze the impact of these arbitration-induced preemption delays considering preemptive [56] (SHDp) and non-preemptive [6] (SHDw) memory requests. Finally, we propose a new technique (SHDi) to resolve these issues by adapting (priority or rather) *criticality inheritance* known from scheduling theory. This allows us to manage and easily bound these preemption delays. Our evaluation shows that our new approach successfully limits the worst-case preemption delays experienced at runtime under our dynamic TDM-based arbitration schemes. At the same time, we see virtually no impact on average-case performance and success rate. Note, besides, that the proposed technique is not limited to the dynamic TDM-based arbitration schemes and is also applicable to other arbitration techniques, e.g., arbitration based on fixed priorities [6].

1.3 Thesis Outline

This thesis is summarized below in terms of its organization and the content of the chapters.

- Chapter 2 presents the basics concepts, definitions, and general properties of real-time systems. This also includes an overview of the Worst-Case Execution Time (WCET) analysis problem, which is an important part of real-time system design. Afterwards a discussion on the different scheduling techniques, used in real-time systems to allocate tasks to the hardware resources, is provided. Finally, this chapter also describes mixed-criticality systems which consist of scheduling tasks with different levels of criticality on the same hardware platform, before concluding with the task model assumed in this thesis.

- Chapter 3 focuses on the hardware mechanisms that have an impact on the tasks' worst-case execution time analysis, more precisely on the memory hierarchy. Each of the different levels of the memory hierarchy can have an impact on the WCET of a real-time task. In this thesis, we focus solely on the impact of the arbitration policy that manages shared memory accesses. An overview of the various existing arbitration techniques is provided in this chapter before concluding with a description of the hardware architecture assumed in the context of this thesis.

- Chapter 4 defines the various problems that will be addressed in this thesis, specifically all the aspects that make TDM non-work-conserving.

- Chapter 5 presents our first contributions towards a work-conserving `TDM`-based arbitrations (`TDMds`, `TDMes`, and `TDMer` already introduced in the previous Section 1.2). This chapter introduces different dynamic criticality-aware `TDM`-based arbitration schemes, where the arbitration schemes are aware of the tasks' criticalities and take decisions accordingly.

- Chapter 6 presents a simple and efficient hardware implementation of a variant of our proposed arbitration schemes, aiming at reducing the hardware complexity.

- Chapter 7 briefly reviews the dynamic `TDM`-based arbitration schemes in the context of a preemptive system model. Therefore, we identify the different arbitration-induced preemption delays inherited from `TDM` and/or specific to our arbitration strategies. In this chapter, we propose various preemption models to handle the arbitration-induces preemption delays and evaluate our contributions using schedulability success ratios and memory utilization.

- Chapter 8 concludes this thesis with a summary of the various contributions and obtained results before finally discussing possible future research perspectives.

Part I

Background/State of the art

Chapter 2

Real-Time Systems

Contents

2.1 Real-time Systems 10
2.1.1 Definitions and Properties 10
2.1.2 Worst-Case Execution Time Analysis 14
2.1.3 Real-Time Scheduling 16
2.1.4 Scheduling Analysis 18
2.2 Mixed-Criticality Systems 19
2.3 Conclusion and Considered System Model 20
2.3.1 Task Model 20
2.3.2 Scheduling Policy 21

In this chapter, we are interested in defining real-time systems and the different properties underlying these types of systems. a real-time system is composed of an execution platform (computer system) and tasks running on the execution platform with temporal constraints imposed by the physical environment. Section 2.1 introduces basic concepts, definitions and general properties of real-time systems. This section also includes an overview of the Worst-Case Execution Time (WCET) analysis, which aims to provide upper-bounds for the tasks running on the computer system. The WCET analysis is an important part of real-time system design. After establishing the WCET of the real-time tasks, a scheduling policy is used to allocate the different tasks to the hardware platform on which they will run. Various real-time scheduling techniques are also covered in Section 2.1. In a real-time system, tasks may have different levels of criticality depending on the impact that a failure would have on the system and its environment. An overview of mixed-criticality scheduling is provided in Section 2.2, which consist of scheduling

tasks with different criticality levels on the same platform. Finally, as a conclusion, we describe the task model and scheduling policy considered in the context of this thesis in Section 2.3.

2.1 Real-time Systems

Real-time systems have applications in many fields of activity, such as transportation, telecommunication, robotics, space missions, multimedia systems, and industry. A real-time system is a *reactive system* in nature. It is often seen as a control environment that is associated with a computer control system. More precisely, a real-time system is in permanent interaction with its external environment, usually represented by a physical process, in order to control its behavior evolving over time. The interactions of a real-time system with its environment are designed in order to respond to events in the controller environment. More precisely, the system response time must be within a limited time, i.e., the system must be able to respond to changes in the state of its environment within this time limit. Not respecting the timing constraints of a real-time system could cause the system to behave incorrectly, causing instability, which could lead to a major system failure.

2.1.1 Definitions and Properties

Definition 2.1.1. *Real-Time System [62, 64]. A* ***real-time system*** *is defined as a control-command system in which the application's correctness depends not only on the result but also on the time at which this result is produced. If the application timing constraints are not respected, we talk about system failure.*

In a real-time system, *time* is the most important resource to manage. Tasks must be assigned and executed in such a way that they respect their timing constraints. Interactions between tasks are also time-constrained, messages are transmitted in a well-defined time interval between two real-time tasks. The environment in which a computer operates is an essential component of any real-time system. Therefore, the respect of the timing constraints of a real-time system affects its *reliability*. A failure of a real-time system can have catastrophic consequences, whether it is economic or human.

Definition 2.1.2. *Predictability [62, 3]. A system is considered predictable if a useful bound is known on temporal behavior that covers all possible initial states and state transitions.*

Predictability [62] in a real-time system depends on several factors, covering both software and hardware. In a predictable system, it must be possible to determine in advance whether all computation activities can be performed within a given time limit. To this end, the system should be *deterministic*, i.e., based on a sequence of input events, the system produces a sequence of output events, always the same output with the same input and in an order determined by the order of the input events [3]. Several factors affect the deterministic behavior of a system, including its hardware architecture, operating system and the programming language used to write the application.

Definition 2.1.3. *Composability [3, 4]. A system is considered composable if tasks cannot affect the behavior of any other task, neither its computation result nor its temporal behavior.*

In a composable system, a component of the system should not affect the temporal behaviour of another component unintentionally [9]. *Composability* is a major factor in improving the predictability of the system. However, modern hardware architectures include several advanced functions (e.g. caches), which increase the average performance of the system at the cost of highly variable timing behaviour. Since these hardware resources are most often shared and are based on historical execution information to improve performance, the execution time of an application is likely to depend on the execution history and interference of other competing applications.

Real-Time Tasks

Definition 2.1.4. *Real-Time Task [62, 64, 8]. A real-time task denoted τ_i, sometimes also called a process or a thread, is an executable entity of work which, at a minimum, is characterized by a worst-case execution time and a time constraint.*

Definition 2.1.5. *Job [64, 8]. A job is a running instance of a task on the hardware platform.*

Tasks in a real-time system are invoked/activated at regular intervals and must be executed in a limited time window. Each invocation of a task corresponds to a job that monitors the state of the system by taking input data, performs some computations and, if necessary, sends commands to modify and/or display the state of the system. A real-time task is often characterized by its criticality level, which is defined according to the severity that a task failure will have on its environment. A real-time task is also characterized by its temporal properties, distinguishing several types of task: periodic, aperiodic and sporadic. Each type normally gives rise to multiple jobs.

Definition 2.1.6. *Periodic Tasks [64, 49, 73]. Periodic tasks are real-time tasks which are activated (released) regularly at fixed rates (period). The period of a task τ_i is commonly designated by T_i. The time constraint for an instance of a periodic task is a deadline d_i that can be less than, equal to, or greater than the period. It is often assumed that the deadline equals the period (implicit deadline).*

Generally, a periodic task is used for the acquisition or the production of data at regular times. The task activities must be performed in a cyclically manner at specific rates, which can be derived from the requirements of the application. For instance, this type of task is used for monitoring purposes, where the task acquires data from sensors at regular intervals. The utilization denoted u_i, of a periodic task τ_i is the ratio of its execution time C_i over its period τ_i, i.e., $u_i = \frac{C_i}{T_i}$.

Definition 2.1.7. *Aperiodic Tasks [64, 73]. Aperiodic tasks are real-time tasks which are activated irregularly at some unknown and possibly unbounded rate. Aperiodic tasks have soft deadlines or no deadlines.*

Aperiodic tasks are event-driven, which means that these tasks behave in an unpredictable way and are governed by events from the system environment.

Definition 2.1.8. *Sporadic Tasks [64, 73]. Sporadic tasks are real-time tasks which are activated irregularly with some known bounded rate. The bounded rate is characterized by a minimum inter-arrival period, that is, a minimum interval of time between two successive activations. This is necessary (and achieved by some form of flow control) to bound the workload generated by such tasks. The time constraint is usually a deadline d_i.*

Just like aperiodic tasks, sporadic tasks are event-driven. There is no prior knowledge of the arrival times of sporadic tasks. However, they do have requirements on the task's minimum inter-arrival time. Unlike aperiodic tasks that may not have hard deadlines, sporadic tasks do have hard deadlines, i.e. when the triggering event of the task occurs, a response is required in a limited time window.

Task Specification

In real-time systems, a task τ_i is characterized, as depicted in Figure 2.1, by its priority $\prod_i$., its release time r_i, its deadline d_i, its execution time C_i, and its period T_i.

Definition 2.1.9. *Release Time [64, 73]. A release time r_i, is a point in time at which a real-time job becomes ready to (or is activated to) execute.*

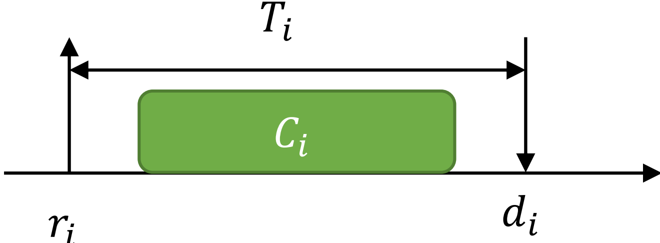


FIGURE 2.1: Overview of a task specification.

A job can be scheduled for execution at any time at or after the release time. It may or may not be executed immediately, because, for example, a higher or equal-priority task is using the processor.

Definition 2.1.10. *Deadline [64, 8]. A deadline, d_i is a point in time by which the task (job) must complete.*

Usually, a deadline d_i is an *absolute time*. Sometimes, d_i is also used to refer to a *relative deadline*. To avoid confusion we denote the relative deadline as D_i. A relative deadline of a task is the deadline measured in reference to its release time.

Definition 2.1.11. *Priority [64]. The priority i of a task indicates its order of importance for scheduling.*

A task has a priority $\prod_i$, which can be fixed or dynamically assigned. The higher the value of a task, the higher this task's priority level. Most of the time, the highest priority task instance (job) that is ready, i.e., whose activation/release date has passed and that has not yet completed, is elected to be executed by the processor. Note that in this thesis, we assume fixed priority assignment, hence the index of task i is equal to its priority $\prod_i$. Therefore, the task index i also represents its priority.

Definition 2.1.12. *Execution Time. A task's execution time is the required time for a processor to execute an instance, i.e., job, of a task τ_i.*

The execution time C_i of a task is not always constant (see Section 2.1.2). For example, a task can have different execution paths and different number of loop iterations each time the task executes. The execution paths and the number of loop iterations vary because of the changes in the input data. The upper-bound and lower-bound of a task's execution time are defined as follows:

Definition 2.1.13. *Worst-Case Execution Time (WCET) [76]. The WCET represents the longest execution time of any job of a task under any circumstances.*

Definition 2.1.14. *Best-Case Execution Time (BCET) [76]. The BCET represents the shortest execution time of any job of a task under any circumstances.*

When designing a system following the multi-tasking approach (several tasks sharing the same hardware execution platform), in addition to execution times, one has to consider the response-time R_i of a task.

Definition 2.1.15. *Response Time. The response time of a task corresponds to the interval from the task's release time to the task's completion.*

In a multi-tasking scheduling context, a task is not executed immediately when it is released because the required shared resources or the processor may be used by higher priority tasks and might thus not be available. Besides, in a preemptive scheduling context, a lower priority task can be suspended so that the processor can execute a higher priority task which is ready. As a result, a task's response time could be larger than its execution time.

2.1.2 Worst-Case Execution Time Analysis

A real-time system must be valid not only with regard to the computed results but also with regard to its time constraints. This assessment is based on bounding the temporal behavior of every task in the system. Specifically, a task's worst-case execution time (WCET) is used to ensure that its deadline is met in all cases, even in the worst case.

The task's execution time is not constant and varies according to many factors [22] that can be divided into two parts, (1) *hardware*, the architectural mechanisms implemented at the processor level, and (2) *software*, the operating system and concurrent tasks, including program inputs that can affect the path followed in the task. Many research studies have focused on obtaining an upper-bound of the worst-case execution time in the context of complex single-core architectures. This interest in obtaining a reliable and accurate upper-bound is driven by the criticality of real-time systems where a system failure can lead to severe consequences, and the cost of pessimism and imprecision during system validation, leading to low hardware resource utilization.

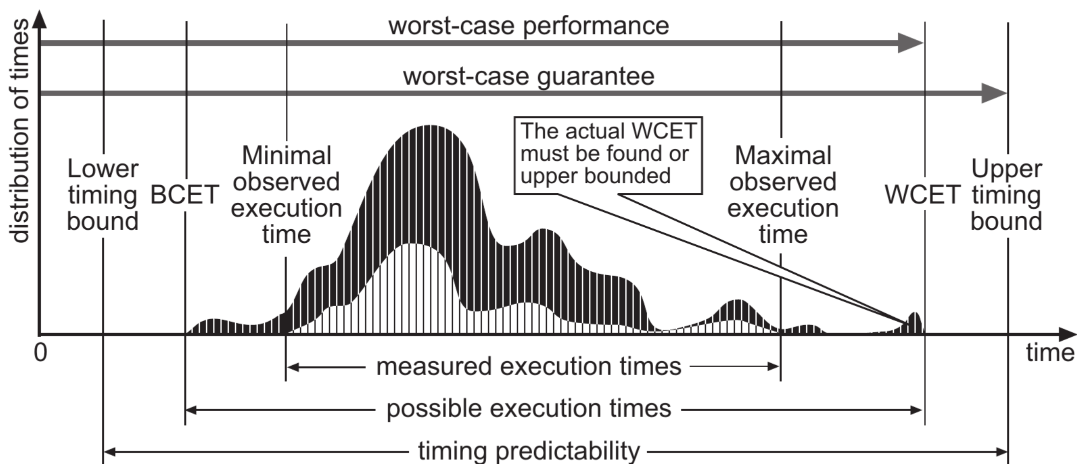


FIGURE 2.2: Task execution time distribution, figure taken from [76].

As mentioned before, WCET analysis relies on the hardware platform on which the task's jobs are running. The increase in the complexity of hardware architectures meets the performance needs of general-purpose systems, which also applies to real-time systems, albeit to a lesser extent. Hardware solutions have the benefit of being transparent to the users of the systems involved. However, more complex architectures are usually difficult to analyse, due to the large state space. Either we explore this large state space (good precisions, bad analysis time) or we use abstractions to simplify the analysis (better analysis time with less precision).

Figure 2.2 illustrates the difference between best-case, worst-case, and average execution time of a given task. The set of all execution times is shown as the upper curve, along with the best- and worst-case execution times (BCET and WCET). In most cases, the state space is too large to exhaustively explore all possible executions and thereby determine the exact worst- and best-case execution times.

There are three main families of methods for bounding the worst-case execution time of a task [76]:

Dynamic or measurement-based methods: The common method to establish execution-time bounds is to measure the *end-to-end* execution time of the task for a subset of the possible executions. For this method, the task, or parts of it, execute on the given hardware or a simulator for some set of inputs. From measured times the maximal and minimal observed execution times are derived. In general, these methods overestimate the BCET and underestimate the WCET and

so are not safe for hard real-time systems. A safety margin is thus often added to the observed maximal execution time, e.g., by multiplying by a given factor – the resulting WCET estimate may still be unsafe.

Static methods: To compute the execution time bounds of a task, static methods are not based on any execution but on the observation and analysis of its source code or binary code. This method analyze the set of all possible execution paths through the (binary) code of a task, combines this information with an abstract model of the hardware architecture, and obtains an upper bound for this combination [75]. Since all analysis steps aim to compute over-approximations, the obtained bounds are safe. However, it is challenging to proof that the formal analysis models actually match the underlying hardware implementations (which might not be publicly available, contain bugs, be poorly documented, or not be documented at all).

Probabilistic methods: Probabilistic analyses build upon the fact that an exhaustive measurement of the WCET cannot be made for all possible states of the system, and that hardware architectures are so complex that static analysis is often very difficult to perform. Burns et al. [17] proposed a statistical method, where execution time is represented by a probabilistic function. The WCET upper-bound is associated with a confidence level by which the upper-bound can be held. Therefore, the probabilistic WCET (pWCET) is defined as follows [23]: *The probabilistic worst-case execution time (pWCET) of a task describes the probability that the worst-case execution time of that task does not exceed a given value.* Probabilistic methods often require special properties on the software and hardware (e.g., independence of execution times) and are still subject to research.

2.1.3 Real-Time Scheduling

In a system with multiple resources that can be accessed simultaneously by multiple users, it is necessary to assign the available resources to the task to be executed, according to a *scheduling policy*. The scheduler is responsible for allocating tasks to processors and managing contention within each processor. In the case of real-time systems, the scheduler has to guarantee the respect of a task's timing constraints. In other words, all tasks must meet their deadlines.

There are two categories of real-time scheduling strategies: *static* and *dynamic* [21].

Static scheduling is based on an *off-line* definition of a static table, containing a sequence of all scheduling decisions regarding the task's executions. The definition of the static table requires prior knowledge of a task's behavior. This scheduling type can be applied to time-driven tasks

(e.g. periodic tasks) which offers a predictable temporal behavior. Another interesting point of static scheduling is its low runtime overhead. However, this scheduling strategy remains very inflexible, with regard to changes in the environment during execution, since it assumes that all parameters, including the release dates, have been fixed in advance.

Dynamic scheduling is an *online* approach of a task to processor allocation. The decisions are based on the current state of the system and the properties of the ready tasks, i.e., according to the task's timing constraints at runtime. This approach offers a more flexible scheduling strategy that can handle even-triggered tasks, therefore offering a higher processor utilization. However, such an approach can have high runtime cost and can be very complex to implement.

Different dynamic scheduling policies exist which mainly reside in two classes of algorithms [14]:

Fixed-priority scheduling algorithms: This approach considers that each task is assigned a priority a priori. The scheduling is done so that the task with the highest priority is assigned to the processor. The task priority can be set according to various criteria, for example, the *Rate Monotonic Scheduling* (RMS) [74, 49] policy gives the highest priority to the task with the shortest period. Another example, under *Deadline Monotonic Scheduling* (DMS) [74] where this time the task priority is assigned according to a task's deadline.

Dynamic-priority scheduling algorithms: As with fixed-priority scheduling, the scheduling decision relies on task priorities. The difference here is that the priorities dynamically change during runtime, according to various criteria. The best-known approach, *Earliest Deadline First* (EDF) [74] policy, is based on the relative deadline of the tasks. The EDF policy consists in assigning the highest priority to task with the lowest relative deadline, i.e., the smallest difference between the task absolute deadline and the current date. *Least Laxity First* (LLF) [36] is another dynamic priority approach where the priority varies according to the difference between the remaining execution time of the task instance and its relative deadline: the lower the difference is, the higher is the task priority.

Real-time scheduling may be *preemptive* or *non-preemptive*. Using a preemptive scheduling strategy, the execution of a task can be interrupted at any time by the scheduler in favor of a higher priority task. The preempted task resumes its execution after the completion of the higher priority task. On the other hand, non-preemptive scheduling does not allow task interruption.

2.1.4 Scheduling Analysis

The validation of a real-time system is based on its temporal aspect, which consists of verifying that all the jobs will always respect their deadlines. To do so, a given task set denoted Γ is analyzed and tested for *feasibility* and *schedulability* depending on the scheduling policy:

Definition 2.1.16. *Feasibility [8, 68]. Feasibility is the assessment of the ability to satisfy all timing constraints of a task set.*

Definition 2.1.17. *Feasible [8, 68]. A task set is feasible if there exists a scheduling policy guaranteeing that all timing constraints are met.*

A real-time task generates a sequence of jobs, each job has a deadline. If all instances (jobs) of the task set can be scheduled with all their deadlines met, then the task set is said to be feasible.

Definition 2.1.18. *Schedulability [8, 68]. Schedulability is the assessment of the feasibility of a task set under a given scheduling policy.*

Definition 2.1.19. *Schedulable [8, 68]. A task set is schedulable under a scheduling policy if none of its tasks, during execution, will ever miss their deadlines.*

Schedulability tests are based on the temporal characteristics of the task set. These tests verify several conditions in order to assess the schedulability of a task set. These conditions may be necessary, sufficient or exact [68] and depend on the characteristics of the systems under which they are applied.

There are three main classes of schedulability tests:

Processor utilization: The processor utilization factor relies on the ratio between the n tasks' execution times and their periods, which is computed as follow:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

For example, this test can be applied to the RM preemptive scheduling policy, the test is considered sufficient but not necessary [49], as long as the processor utilization does not exceed $n \cdot (\sqrt[n]{2} - 1)$. Different bounds can be found for different scheduling approaches (DM, EDF, LLF), these bounds and their conditions also depend on the environment and the type of the considered system. For example, in a distributed system of dependent tasks, the conditions determined by the processor utilization factor remain necessary and not sufficient.

Processor demand: This test is based on the computation of cumulative demand of task executions. It involves testing for any time interval $[t_1, t_2]$, that the maximum cumulated tasks' execution times, released and ended execution in this interval, does not exceed the length of the interval [49].

Response-time: This test is based on the tasks' Worst-Case Response Times (WCRTs) computation. It can be applied to any variant of Fixed-Priority scheduling techniques. This approach is therefore based on worst-case scenario analysis rather than on an exhaustive exploration of all possible states. In addition, this approach can be applied to more complex systems such as systems with random deadlines or dependent tasks, or distributed systems.

The test is sufficient and necessary for task sets that consist of synchronous periodic tasks. For a given task set, the computation of the WCRT, R_i of a task τ_i, is performed using the following recurrence equation [39, 7]:

$$R_i^{n+1} = C_i + B_i + \sum_{\forall j \in \mathrm{hp}(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j \tag{2.1}$$

The recurrence equations are initialized to $R_i^0 = 0$ and then iteratively reevaluated until a fixed point is reached. R_i then indicates the response time of task i, having a WCET bound C_i. In addition, the impact of preemptions by tasks with higher priority than i ($j \in \mathrm{hp}(i)$) is considered via their WCET bounds C_j and periods T_j. B_i indicates an upper bound on the time task i may be blocked (e.g., by semaphores). If all tasks meet their deadlines (i.e. $\forall i : R_i \leq D_i$) than the corresponding task set is schedulable.

2.2 Mixed-Criticality Systems

Several factors make the computation of an upper-bound for the worst-case execution time complex and most often result in a very pessimistic upper-bound. The schedulability tests are based on these pessimistic WCETs and result in low processor utilization, hence an inefficient use of the underlying hardware resources. Some approaches have been developed to take advantage of this pessimism. Vestal [71] has introduced the so-called mixed-criticality systems, where different criticality tasks are scheduled on the same platform with properties that take advantage of the WCET pessimism.

In a mixed-criticality system, a task τ_i is characterized [18] by a criticality level L_i, a set of worst-case execution times $(C_i(1), C_i(2), ..., C_i(L_i))$ depending on the criticality level, a deadline (D_i) and a period (T_i). The scheduling of a task set Γ under a mixed-criticality system relies on the *execution mode* X. This mode is related to the criticality level in which the system is operating, such that it only allows tasks with a higher criticality level to be executed, i.e. $L_i \geq X$. A mode change is triggered every time a TFE (*Timing-Failure Event*) occurs. The TFE refers to the case where a running job reaches its $C_i(X)$ and therefore it will be granted a higher worst-case execution time $C_i(X+1)$ so that it can finish executing at the cost of stopping lower criticality jobs. In a two-level criticality model ($\forall \tau_i \in \Gamma, L_i \in [1,2]$), the two execution modes work as follows:

- **Low mode ($X = 1$):** supports the execution of all tasks with criticality levels $L_i \in [1,2]$.
- **High mode ($X = 2$):** only enables tasks with a criticality level $L_i = 2$ to be executed.

Each time a TFE is triggered, tasks with a criticality lower than the execution mode are simply stopped. This approach allows the critical tasks to be executed without fear of missing their deadlines. There are alternative approaches [65, 66] that instead of stopping the lower critical tasks modify their execution conditions. These approaches are based on the *elastic task model* [19], which allows having L_i periods per task $(T_i(1), T_i(2), ..., T_i(L_i))$. Therefore, the used task period depends on the execution mode X. When a TFE is triggered, instead of stopping tasks with a lower criticality level than the execution mode, these tasks change periods (larger periods) and downgrade their execution for the benefit of more critical tasks.

2.3 Conclusion and Considered System Model

In this section, we introduce the real-time system model and assumptions on which we base our work. We present our task model in Subsection 2.3.1 before detailing our choices regarding the scheduling policy in Subsection 2.3.2.

2.3.1 Task Model

In this work, we assume a finite task set Γ consisting of independent and periodic tasks. Each task $\tau_i \in \Gamma$ is assigned a fixed priority i – a larger task index indicates higher priority. A task is characterized by the tuple $\tau_i = (C_i, T_i, D_i, M_i)$, representing the task's WCET, its period, its implicit deadline, and finally its worst-case number of memory requests respectively. Each task generates an infinite sequence of jobs at runtime, which, in turn, generate a sequence of memory

requests $\tau_{i,j}$ where $j \leq M_i$. Requests are separated by a dynamic number of processor clock cycles. This *distance* given by $dist_j$ represents the amount of computation performed between the completion of request $\tau_{i,j}$ and the issuing of the consecutive request $\tau_{i,j+1}$ or the job's termination. This allows us to model any *dynamic* job execution (even input-dependent) considering a deterministic hardware platform. In order to ensure that the *distances* between requests are independent from the execution of other tasks, we assume composable compute cores [30] and the absence of any external events that may interfere with the execution of a core.

This work focuses on two classes of tasks: *critical* tasks τ_i^c and *non-critical* tasks τ_i^{nc}. However, our model is somewhat simplified with regard to mixed-criticality systems, as we do *not* introduce different operation modes, e.g., the modes LO/HI from Vestal [71]. We currently only consider a single C_i value per task in our task model, i.e, we assume a *multi-criticality* system. We here assume that the WCETs (C_i) and deadlines (D_i) of critical tasks are firm and have to be respected under all circumstances. An execution thus fails if a critical task misses its deadline. Non-critical tasks, on the other hand, are executed in a best-effort manner by the underlying computer platform. During execution, they may exceed their WCET budget, potentially causing deadline misses – for themselves or other (critical) tasks. However, for the formal analysis based on response-time analysis (RTA) [7], we require *firm* WCETs and deadlines for non-critical tasks, due to the absence of mechanisms to handle timing failure events in the task model – note that the task model is not the main focus of this work. This is required in order to bound the response times of critical tasks.

2.3.2 Scheduling Policy

Our work focus on multi-core platforms, task scheduling on this type of architecture can be performed globally among all/a subset of the available m cores [47] or in a partitioned manner [11, 13]. Partitioned scheduling statically assigns each task onto a fixed core, while global scheduling allows tasks to migrate among cores dynamically. In principle, both of these scheduling policies could be combined with the approaches proposed here. In Chapter 5, we first assume a restricted scheduling model, where each core executes a single independent and periodic task. This aims to first evaluate our work regardless of the scheduling policy before relaxing this assumption in Chapter 7. For brevity, in Chapter 7, we limit our discussion on *partitioned scheduling* using *fixed-priorities* on each core. Critical and non-critical tasks may reside on the same core – without any restrictions on the priority assignment. Notably, non-critical tasks may have a higher priority than critical tasks. This latter chapter, hence, evaluates the preemption delays induced by our approaches.

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

Chapter 3

Multi-Core Memory Access Interference

Contents

3.1 Memory Hierarchy **24**
3.1.1 Registers 25
3.1.2 Scratchpad 25
3.1.3 Caches 26
3.1.4 Main Memory (DRAM) 27
3.2 Memory Arbitration Schemes **29**
3.2.1 Fixed-Priority 30
3.2.2 First-Come First-Served 30
3.2.3 Round-Robin 31
3.2.4 Time-Division Multiplexing 32
3.2.5 Budget-Based Arbitration 34
3.2.6 TDM Arbitration and Multi-Criticality Scheduling 35
3.3 Conclusion and Assumed Hardware Architecture **36**

In this thesis, we are interested in multi-core systems, i.e. systems with several computing cores. In fact technological advances now make it difficult to increase the frequency of each processor. The computing capability of each system is instead improved by increasing the number of processors. Multi-core systems are therefore increasingly present, even in real-time embedded systems. Single-processor systems will thus be replaced by multi-core systems. The industry will have to turn to multi-core systems. This justifies the study of multi-core systems.

Multi-core architectures pose many challenges in real-time system, which arise from the many-fold interactions between concurrent tasks during their execution. The different cores are not completely independent of each other. Several resources are shared between the computing cores, such as the main memory, the bus interconnecting the cores and the main memory, or any second- or third-level caches, in order to limit the cost of developing and manufacturing multi-core chips. As a result, access conflicts and additional latencies arise compared to a single-core architecture. These latencies must be bounded in order to be included in the worst-case execution time of the tasks. Therefore, dedicated mechanisms for partitioning and arbitrating shared resources must be used to ensure the respect of strict real-time constraints.

In a multi-core architecture, each core has its pipeline which can contain *Arithmetic and Logical Units* (ALUs), *Floating-Point Units* (FPUs), and *Load/Store* Units. The processing time for instructions using only these private resources is therefore identical to that observed on a single-core processor. However, as previously mentioned, other resources are shared by the entire platform, including some levels of the memory hierarchy. They provide a major source of temporal unpredictability for tasks that rely on them, in addition to being a bottleneck for processor performance. Precisely bounding the temporal behavior of a memory access is one of the main challenges when analyzing a task scheduled on a multi-core architecture.

This chapter focuses on the hardware mechanism that has an impact on the worst-case execution time analysis of a task more precisely on the memory hierarchy. Therefore, Section 3.1 focuses on the impact of the memory hierarchy on real-time tasks and vice-versa. Afterwards, Section 3.2 provides an overview of the arbitration mechanisms used in real-time systems. Finally, as a conclusion, a small description of the assumed hardware platform for this work, is given in Section 3.3.

3.1 Memory Hierarchy

The memory hierarchy is the main factor of the worst-case execution time analysis complexity. The memory hierarchy is composed of a series of caches that provide the link between the processor and the main memory while going through arbitration mechanisms for accessing shared memory. Each of these caches contains a partial image of the memory which is faster to access than the latter. In fact the presence or absence in the cache of data required by a sequence of instructions can result in an execution time difference of several hundred processor cycles.

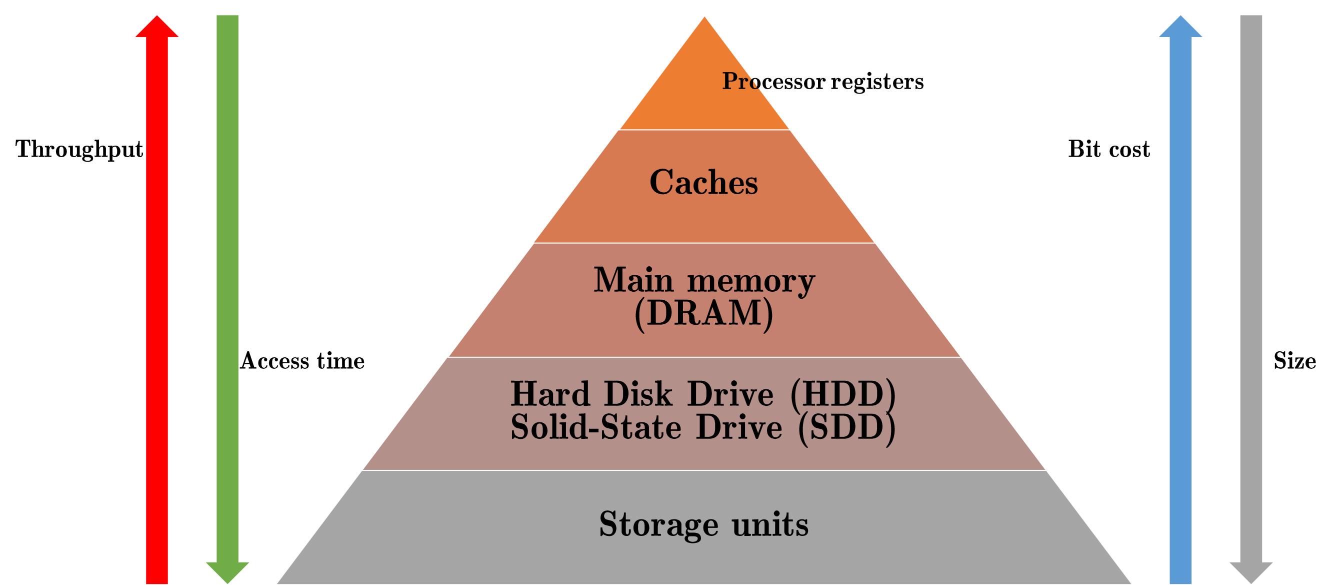


FIGURE 3.1: Overview of a computer memory hierarchy, figure taken from [26].

An overview of the memory hierarchy [26] is given in Figure 3.1. It shows that the access time of the different memory levels varies, as does the storage capacity of these levels, while the throughput and cost per bit vary as the inverse of the storage capacity. The memory hierarchy is based on the *locality of reference* [24], also known as the principle of locality, which refers to the tendency of a processor to access the same set of memory locations repeatedly over a short period of time. There are two basic types of reference locality, *temporal* and *spatial* locality. Temporal locality refers to the reuse of specific data, and/or resources, within a relatively small time duration. Spatial locality, on the other hand, refers to the use of data elements within relatively close storage locations.

3.1.1 Registers

A processor contains a number of registers [50], a register may hold an instruction, a memory address, or any kind of data. The registers work as the highest level in the memory hierarchy, and provide the fastest way to access data.

3.1.2 Scratchpad

A scratchpad memory is a local memory close to the processor core, just after the processor registers (same level as the caches). The memory has specific instructions allowing for data to be transferred directly to and from the main memory [10]. Scratchpads are often used as a local

memory alternative to the cache memory. Besides, a scratchpad allows to the task running on a core to directly manipulate its data, which can be useful for critical real-time tasks [67], unlike cache memories, which are problematic in terms of predictability. Given its properties, the programmer can ensure an execution without main memory contentions in a multi-core architecture, by directly controlling the content of the scratchpad memory through the dedicated instructions.

3.1.3 Caches

The primary purpose of a cache [63] is to bridge the latency gap between the slow main memory and faster processors. A cache contains a subset of the main memory with faster access latency and is the first one checked by the processor when it requests data from the memory. If the data requested is cached, it can be served immediately at a reduced latency compared to the main memory access, this is known as a *cache hit*. On the other hand, if the data is not in the cache, the processor request is redirected to the main memory, this is known as a *cache miss*. At any given time, the content of a cache memory depends on its technical characteristics (number and size of blocks), the memory access history performed and also the cache replacement policy used.

Replacement policy: To make room for the new memory blocks on a cache miss, the cache may have to evict one of the cached memory blocks. The heuristic used to choose which entry to evict is called the replacement policy [63]. The challenge for any replacement policy is that it must anticipate which of the cached memory blocks is least likely to be used in the future. One popular approach, *Least-Recently Used* (LRU), replaces the least recently accessed entry.

Write policy: When the processor writes a data to the cache, at some point it must also be written to main memory. The moment when this write is done distinguishes the different write policies [40]. There are two basic writing policies, *write-through* and *write-back*. In a write-through policy, the write is done synchronously both to the cache and to the main memory. Alternatively, in a write-back policy writes are not immediately forwarded to the main memory, the cache instead tracks which blocks have been modified marking them as *dirty*. The data in these locations is written back to the main memory only when that data is evicted from the cache.

Cache types: The caches can be divided into an *instruction cache*, containing the program code to be executed, and a *data cache*, or combined into a single *unified cache*. The benefit of having separated caches is to allow instructions and data to be loaded simultaneously.

Cache memory is important because it provides data to a processor much faster than main memory. It helps to reduce the memory latency and thus decreasing system response time. However, in real-time systems, because cache memory is shared between tasks and memory accesses are not always predictable, the worst-case execution time analysis gets more complex. A naive solution to this problem is to compute at any point in the program all possible cache states at runtime [53]. If this method ensures accuracy and safety, the number of states to maintain and consider tends to explode [69], its complexity in the general case is extremely high.

Several studies focus on the problem of analyzing the content of cache memories. Cache analysis is initially performed for each task in isolation using abstract interpretation. The use of an abstract representation of the possible cache states at runtime is one of the first methods [27, 51] for estimating cache content. An input state is computed for each point of the program from the output state of its predecessors until a fixed point is obtained. The behavior of each instruction with respect to the cache can then be estimated according to the content of its input state.

In a computer system that uses cache memory, when a task is preempted, memory blocks belonging to this task could be evicted from the cache. Once this task resumes, previously removed memory blocks have to be reloaded. Therefore, the *Cache Related Preemption Delay* (CRPD) denotes the delay added to the execution of the preempted task because it has to reload cache blocks evicted by the preemption [53]. In a real-time system, these CRPDs has to be taken into account when analyzing tasks' response time. Several studies [68, 5, 20] focuses on computing the cost of reloading evicted cache blocks after a preemption with regard to the worst-case response time analysis.

3.1.4 Main Memory (DRAM)

Dynamic random-access memory (DRAM) is widely used as a main memory in embedded systems, where low-cost and high-capacity memory is required. In contrast, the static random-access memory (SRAM), which is faster and more expensive than DRAM, is typically used where speed is of greater concern than cost and size, such as the higher levels in the memory hierarchy (cache memories).

DRAM organization: Figure 3.2 shows the internal organization of modern DRAMs [42, 59, 3]. The DRAMs are organized as a set of ranks, each of which consists of multiple DRAM chips. Each DRAM chip has a three-dimensional memory organization with the dimensions of *bank*, *row*, and *column*. The banks operate independently of the other banks and memory requests to

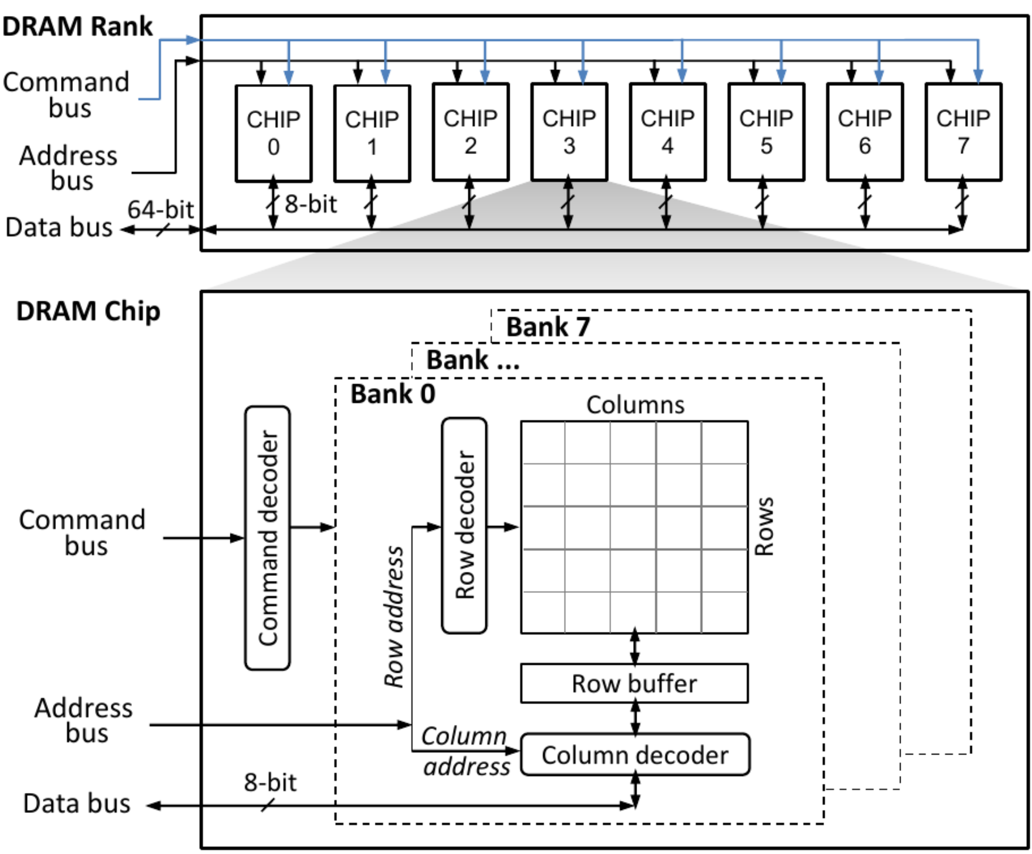


FIGURE 3.2: Overview of a DRAM organization, figure taken from [42].

different banks can be serviced in parallel. DRAM banks have a two-dimensional array of rows and columns of memory locations. To access a column in the array, the entire row containing the column is activated and transferred to the bank's *row-buffer*. The row buffer serves as a cache to reduce the latency of subsequent accesses to that row. While a row is active in the row buffer, any number of reads or writes (column accesses) may be performed. Each bank has a row buffer that can hold one open row at a time, and read and write operations are only allowed to the open row. Before opening a new row in a bank, the contents of the currently open row are copied back into the memory array. The elements in the memory arrays are implemented with a single capacitor and a resistor, a charger capacitor is defined as a one and zero when its empty. The capacitor loses its charge over time due to leakage and must be refreshed regularly to retain the stored data.

The internal DRAM architecture described above makes it challenging to use in systems with real-time requirements [3]. The DRAM memory access latency and bandwidth depend on various characteristics, which creates large variations in the time required to serve a read or a write

memory request. (1) The first source of variability is related to the row targeted by the request and the rows that are currently open in the banks. If the row targeted by the memory request is open, the request can be served immediately. On the other hand, if the request is targeting a closed row, this request must wait until the currently open row has been closed and the required row has been opened, resulting in additional latency. (2) The data bus is bi-directional and requires several clock cycles to change direction from read to write or write to read, this leads to additional latency. (3) The third source of variability is related to the capacitor refresh, which can take several clock cycles during which no data can be transferred on the data bus. Therefore, the temporal alignment of a memory request with regard to the DRAM refresh also impacts the memory access latency.

The first two sources of variability are particularly problematic since they involve a dependency on the previously issued memory requests that may have been issued by other requestors sharing the resource. This generates resource interference between requestors, where the time that the resource requires to serve a memory request depends on other requestors. This interference makes it very hard to bound the bandwidth offered by the memory and the latency of memory requests at design time, which is necessary to support firm and hard real-time requirements. When trying to provide predictable memory behavior, DRAM memories suffer from highly variable access latencies and overly pessimistic latency bounds. An alternative off-chip memory solution may be considered [32], based on Reduced Latency DRAMs (RLDRAMs). Access latencies for RLDRAMs are generally lower and, in addition, exhibit less variability. However, this type of memory is not widely used and to our best knowledge, Hassan [32] is the first to consider using this type of memory for real-time systems.

3.2 Memory Arbitration Schemes

In a multi-core architecture, the main memory is shared between several cores. Tasks running simultaneously on the different cores are subject to interference from concurrent accesses to this shared resource. This interference, therefore, impacts task execution times and an upper-bound is needed w.r.t. the maximum time required to access the main memory in order to establish the tasks' WCETs. The memory arbitration policy used to manage interference thus plays an important role in bounding tasks' WCETs and in improving the average task performance. Furthermore, proper interference management is essential to effectively use multi-core platforms.

Whenever a task requires data or instructions from the main memory, the core's bus controller, on which the task is executed, sends an access *request* to the *arbiter*. The arbiter processes the various requests according to an *arbitration policy*. The request to be granted first can be selected according to its arrival date, the static priority of the core/task that issued it, or according to an approach independent from the component issuing the request. A core can only access the main memory when the arbiter grants access to it. Several arbitration policies are discussed below.

3.2.1 Fixed-Priority

In this approach a priority level is first assigned to each element requesting access to the shared memory, this element can be a core or a task. The arbiter uses this parameter to manage the scheduling of memory requests. If several cores/tasks are requesting access to the shared memory at the same time, the request coming from the core/task with the highest priority is the first one satisfied. Such arbitration is easy to implement and is therefore widely used. However, the predictability of such an approach is complex and often very pessimistic. The time that a memory request has to wait to be processed depends on the requests sent by the other cores. Therefore, A request can be delayed indefinitely by higher priority elements [15]. Only the latencies of the element with the highest priority can be predicted regardless of the other concurrent elements. This is problematic in a real-time system, where many critical tasks are competing for the shared resource. As a result, a fixed-priority arbitration can only be safely used on a multi-core platform when only one of its cores is executing a task with strict real-time constraints. The worst-case memory access latency is then predictable and limited. On the other hand, it cannot be precisely established for the other cores/tasks [46]. Besides, this approach does not take into account the behavior of the tasks executed by the other cores, which can affect the overall performance of the platform. Altmeyer et al. [6] proposed a framework allowing to upper-bound the memory access latency with regard to contention with other competing elements. This is achieved by bounding the total number of memory accesses due to all competing elements with higher priority during a limited time interval. Their results showed that a Fixed-Priority arbitration can achieve good guaranteed performance even if the arbitration policy does not provide tightly bounded single memory access latency unlike with TDM and Round-Robin approaches (see Subsections 3.2.3 and 3.2.4).

3.2.2 First-Come First-Served

Instead of assigning a static priority to the elements seeking access to the shared memory, alternative approaches are based on a dynamic priority allocation. This priority may depend on

various characteristics, such as the *First-Come First-Served* (FCFS) policy, which assigns a priority based on the arrival date of the memory request. Under the FCFS policy, memory requests are stored in a queue, the older a request is, the sooner it will have access to the shared resource. If several requests arrive at the same time, the selection of the one that will be placed first in the queue can be random or follow a predefined policy. This approach makes it possible to treat the different cores equally. As a result, this optimizes the shared bus occupancy and improves the use of shared memory [59]. However, this type of approach does not include a memorization mechanism that would allow prioritization of requests arriving at the same time according to the history of the core that issued them. The time required to satisfy any request depends only on the number of requests received previously and not on the number of requests not yet served. For each request, a joint analysis of all tasks running on the same platform is required to determine its waiting time.

There are variants of FCFS that improve the memory access latency by taking advantage of the spatial locality of access (*First-Ready First-Come First-Served* (FR-FCSFS) [70, 37]). This mechanism is designed to ensure that when a task generates an access stream to the same memory line, requests from the other tasks can be put on hold. As in FCFS, the oldest requests are always given priority regardless of requests history to the disadvantage of tasks that issue fewer requests. In order to address this issue, several approaches were proposed [54, 52] to improve the fairness of processing memory requests by taking into account the history of memory requests issued by concurrent tasks.

3.2.3 Round-Robin

All the previously presented schemes, whether they are static or dynamic priority approaches, aim to maximize the use of the memory bus and therefore the use of main memory. However, shared memory access latencies always change dynamically depending on the behavior of other tasks running on the same platform. As a result, the computation of a fixed bound for memory access latencies is complex. Some arbitration schemes manage memory request of each core in a way that the memory access latency is independent from the interference of the other cores. The Round-Robin (RR) policy is the most straightforward approach, each core can access the memory in turn for the same quantum of time. Therefore, the maximum delay of a request issued by a core is a function of the number of cores that share the memory, and the latency of a memory access. This delay is the same for all cores, predictable and independent of the tasks running on other cores [56]. In the worst case, a memory request is processed last, the maximum processing time of a memory request is expressed by the formula $(m-1) * L + L$, where m corresponds to

the number of competing cores, and L is the access latency to the shared memory. This formula includes the accesses of all the other cores $(m-1)*L$ and the addition the access latency of the considered core. Using the RR policy, all memory accesses are subject to the same latency, which can potentially degrade the performance of shared memory usage in the case where competing tasks do not have the same bandwidth needs.

3.2.4 Time-Division Multiplexing

Similar to Round-Robin, Time-Division Multiplexing (`TDM`), in addition to being easy to implement [72] in hardware, ensures a predictable behavior by bounding access latencies and guaranteeing bandwidth to tasks independently from other tasks. To do so, `TDM` guarantees exclusive access to the shared resource, here the shared memory, in a fixed time window also called `TDM` slots. The constant length Sl of a `TDM` slot is either equal to the worst-case memory access latency or greater, in latter case it might be possible to process multiple requests in a single `TDM` slot. In the context of this thesis, we assume that the **`TDM` slot length** Sl corresponds to the worst-case memory access latency. Therefore, only one memory request at a time can be processed. Each core has a dedicated `TDM` slot that alternates over time which allows deriving a **`TDM` period** $P = m \cdot Sl$, where m refers to the number of cores.

Figure 3.3 illustrates an execution under such a `TDM` arbitration strategy, considering 3 tasks (τ_0, τ_1, τ_2) that execute on separate cores (C_0, C_1, C_2) respectively and perform concurrent memory requests to the shared memory. Each core is assigned a dedicated `TDM` slot (vertical columns, labeled C_0 through C_2 representing the cores executing the tasks) that alternate over time. The

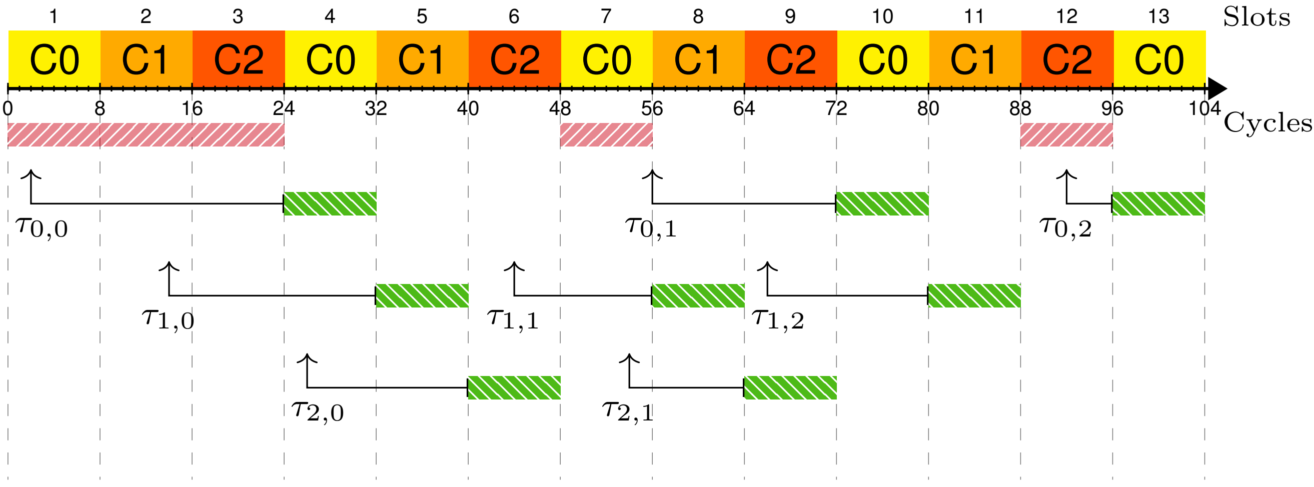


FIGURE 3.3: Regular `TDM` arbitration of three tasks τ_0, τ_1, and τ_2.

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 (see Subsection 2.3.1) are represented by sequences of memory requests as follows: $(\tau_0 : 2, 24, 12)$, $(\tau_1 : 14, 4, 2)$ and $(\tau_2 : 26, 6)$. Task τ_0, for instance, performs its first memory access $\tau_{0,0}$ after 2 clock cycles, the second access $\tau_{0,1}$ 24 clock cycles after completing the first one, and the third access $\tau_{0,3}$ another 12 cycles later.

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 [77], 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 corresponds to the *Request Inter-Task Delay* ($b - a$) and the *Request Execution Time* ($c - b$) of Paolieri, Quiñones, and Cazorla [55]. Request $\tau_{1,1}$, for instance, is issued 4 cycles after the completion of request $\tau_{1,0}$ 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 a **green hatched bar** (▨). Its request inter-task delay is thus 12 cycles, while its request execution time is equal to Sl. A regular TDM arbitration scheme assumes that the requests are granted access to the memory only at the beginning of a TDM slot, as this minimizes TDM periods. 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 $\tau_{0,2}$ is the last request to complete at the end of the 13th slot. The blocking time (the processor is stalled until the request is processed) of requests $\tau_{0,0}$, $\tau_{0,1}$, and $\tau_{1,2}$ are respectively 3.75, 3, and 1.5 slots respectively. The total blocking induced by TDM on the requests from task τ_0 is thus 66 cycles. The last memory access of task τ_0, therefore, completes after 104 cycles, since the amount of processing of task τ_0 is 38 cycles $(2 + 24 + 12)$.

The example shown in Figure 3.3 illustrates the temporal isolation among cores offered by TDM arbitration. 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 through slot offset analysis [41, 58]. However, the main issue with TDM is its non-work-conserving behavior, i.e., if a TDM slot is not used by its owning core than the slot cannot be used by another core. This non-work-conserving nature of TDM limits the performance of the underlying hardware platform, more precisely, the memory utilization.

A common approach to improve the memory utilization of TDM is by increasing the number of TDM slots according to task weights [79]. However, the TDM strategy itself is not modified, which remains heavily non-work-conserving.

Recently two software-based approaches have been proposed to improve TDM-based arbitration depending on contention [67, 45]. The first work [67] defines a task model in which tasks are split into sub-tasks consisting 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 scratchpad memory, which entails a considerable change in the underlying programming/execution model.

Kostrzewa et al. [44] propose a technique to dynamically adapt 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. Yonghui et al. [48] truly *skip* unused entries in a TDM schedule in order to allow for variable-sized TDM slots. This improves upon the non-work-conserving behavior of TDM. However, it does not preserve the TDM slot offsets analysis [41]. The behavior of a task thus relies on the memory latency/interference of other tasks.

3.2.5 Budget-Based Arbitration

Budget-Based (BB) arbitration policies allow satisfying different bandwidth requirements depending on the task needs [2]. Instead of establishing a static arbitration as in TDM, budget-based approaches simply allocate a given amount of bandwidth (*budget*) to each core. Depending on its priority level, and as long as it has not consumed its allocated budget, a core can be granted access immediately to the memory. MemGuard [80] ensures isolation between cores by implementing such an approach by 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. [1] extend MemGuard with a memory bandwidth throttling approach [80] 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 accessed per slot.

3.2.6 TDM Arbitration and Multi-Criticality Scheduling

In Section 2.2, the benefits of sharing resources between tasks with different levels of criticality were discussed. This type of approaches raises a problem: to allow resource sharing within a mixed-criticality or multi-criticality system, it is necessary to be able to provide sufficient guarantees of partitioning and separation to ensure that this type of system can share resources without compromising the temporal behavior of tasks with a higher level of criticality.

Various approaches are focusing on the problem of main memory arbitration in the context of multi-criticality task scheduling. A straightforward approach is to apply a predictable arbitration scheme like `TDM` for critical tasks [28] while allowing other schemes for non-critical tasks. However, this approach still suffers from the non-work-conserving behavior of `TDM`, hence, having a low memory utilization. Alternative approaches, like Yonghui et al. [48] which *skip* unused slots in a `TDM` period and supporting variable-sized `TDM` slots, aims to improve the memory utilization. Hassan et al. [34, 33] similarly propose a work-conserving variant of `TDM` along with a technique to generate harmonic `TDM` schedules accommodating critical and non-critical tasks. However, these approaches do not preserve the relative alignment of the program execution with the `TDM` schedule, slot offset analyses [41, 58] thus cannot be applied.

Paolieri et al. [56] propose a multi-core platform for mixed-criticality systems that is equipped with a hierarchical Round-Robin arbiter, which always prioritizes critical tasks. However, their arbiter is not able to exploit task criticalities to improve memory utilization or to reduce the average execution time of non-critical tasks.

Other approaches track slack-time gain by critical tasks compared to their WCET so that non-critical tasks can benefit from it. Kritikakou et al. [45] 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 interrupted and then resumed after the completion of critical tasks. Kostrzewa et al. [43] 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.

TABLE 3.1: Table comparing the various arbitration policies.

Policy	Predictability	Starvation	Dynamic	Work-conserving	Precise WCET analysis
FCFS	X	X	√	√	X
FP	X	√	X	√	X
RR	√	X	√	√	X
BB	√	X	√	X	X
TDM	√	X	X	X	√

3.3 Conclusion and Assumed Hardware Architecture

In this thesis, we assume a hardware platform consisting of m cores that are connected via a central arbiter to shared main memory. The memory requests dynamically generated by the jobs at runtime thus compete for this shared resource. We assume that each core is equipped with internal caches. Memory requests thus represent transfers of cache blocks resulting from cache misses. In order to ensure that the *distances* (see Subsection 2.3.1) between requests are independent from the execution of other tasks, we assume composable compute cores [30] and the absence of any external events that may interfere with the execution of a core. The interference between the independent tasks consequently stems from accesses to the shared memory only and, in particular, depends on the employed memory arbiter. For simplicity, we assume that all cores, the memory bus/arbiter, and the memory itself operate at the same clock speed. We thus generally refer to time in *clock cycles*.

With regard to the arbitration schemes presented in Section 3.2, Table 3.1 illustrates a comparison between the different traditional memory arbitration approaches based on different criteria. Our goal is to have a work-conserving arbitration policy targeting multi-criticality real-time systems, the TDM policy seems to be a good starting point. Even if TDM is not dynamic and non-work-conserving, we choose TDM because it is easy to implement, and more importantly predictable with safe and easy to compute worst-case memory access bounds [41, 58]. Therefore in this work, we focus exclusively on extending the TDM arbitration scheme.

Part II

Contribution

Chapter 4

Problem Statement and Contribution Summary

Contents

4.1 Thesis Problem Statement **39**
4.1.1 Criticality-Aware Arbitration Schemes 40
4.1.2 Challenges with Time-Division Multiplexing 41
4.1.3 Preemption Costs for Regular TDM Arbitration 44
4.2 Contribution Outline **46**

4.1 Thesis Problem Statement

In Section 2.2, we illustrated the benefits of scheduling tasks with different levels of criticality on the same platform. This approach aims to increase resource utilization. However, most multi-criticality systems take decisions only at the task scheduling level. Our aim in this work is, hence, to develop a criticality-aware memory arbitration in order to improve memory utilization, while keeping temporal guarantees for critical tasks. To achieve this goal, the arbitration scheme needs to be aware of task criticalities and dynamically take arbitration decisions accordingly. To address this challenge, the criticality-aware arbitration scheme must be able to offer a predictable behavior with regard to the desired temporal guarantees of critical tasks. On the other hand, the arbiter should also provide enough flexibility regarding the arbitration of memory requests from tasks with different criticality, to exploit the main memory efficiently. In this work, we are not considering mixed-criticality systems but rather multi-criticality systems, where we have critical and non-critical tasks on the same platform. Therefore, we are not interested in tasks with several levels of criticality and a WCET per level as illustrated in Section 2.3.1.

4.1.1 Criticality-Aware Arbitration Schemes

Section 3.2 illustrated different techniques to manage memory interference. One important aspect of the various arbitration policies is that each of them comes with different properties. `TDM` is suitable for providing temporal isolation among cores, while Round-Robin guarantees a fair treatment among all requestors. Fixed-priority and budget-based approaches are suitable when differentiated treatment needs to be applied to the cores. This means that an arbitration policy needs to be selected according to the requirements of the tasks running on the different cores. Considering a multi-criticality system, in addition to tasks priorities, different requirements are needed depending on the tasks' criticalities. For instance, critical tasks require temporal guarantees w.r.t. their timing constraints. Non-critical tasks, on the other hand, should be executed when possible and achieve the best possible performance. Therefore, the memory arbitration scheme should take into account these different requirements. One may suggest that the fixed-priority and budget-based approaches are appropriate for such a system. However, we noted in Subsection 3.2.1 that the priority-based approaches require information on all tasks competing to access the shared memory, hence lacking timing composability. The budget-based approach (see Subsection 3.2.5), on the other hand, lacks flexibility, tasks executed by a core are suspended when the budget of memory requests, periodically assigned to the core, is depleted even if the memory is currently unused. Another approach is to improve the memory utilization of TDM (see Subsection 3.2.4 and Subsection 3.2.6) by increasing the number of TDM slots according to task weights [79]. Others apply strict TDM arbitration to critical tasks [28] while allowing other schemes for non-critical tasks. In both cases, the TDM strategy itself is not modified, which remains heavily non-work-conserving.

All these aforementioned techniques aim to improve the main memory utilization while keeping temporal guarantees on the tasks' executions. However, these approaches are either not efficient in exploiting the main memory, or the arbitration between critical and non-critical tasks often results in prioritization of critical tasks over non-critical ones [56, 45]. The latter focuses on the preservation of critical tasks' temporal guarantees, but on the other hand, this has a negative impact on the average performance of non-critical tasks.

The memory access delay of schemes built on a more dynamic arbitration (Priority-Based, FCFS), can only be bounded if all interference among possible concurrent accesses is known. Therefore, the delay of an access and thus the WCET of a task is no longer computable for a given, single task on a given platform in isolation, but only if the full *set* of tasks running on the platform is known. This means that the timing composability is lost, and assuming a worst-case

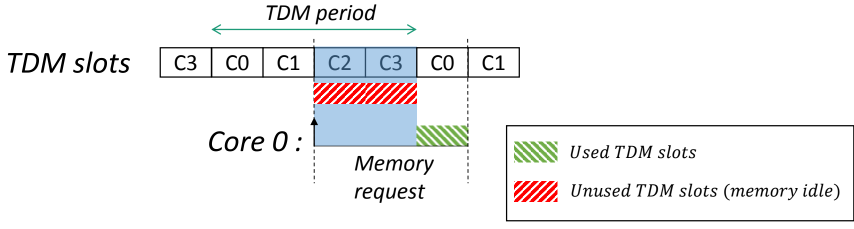


FIGURE 4.1: Non-work-conserving of TDM.

hypothesis for these strategies leads to overly pessimistic results. With regard to the analysis of arbitration schemes, Pitter and Schoeberl [57] compared the predominant arbitration methods. They consider TDM arbitration to be the most composable and predictable method. The basic property in favor of TDM is, that TDM allows to statically determine the delay of a request, once we know the point in time when the request is issued [41, 58].

Our goal is to have a real-time work-conserving arbitration policy for multi-criticality systems. To this end, the TDM policy seems to be a good starting point because it is easy to implement, composable, and predictable, with good worst-case memory access bounds. Therefore, in this work, we focus exclusively on TDM-based arbitration schemes.

4.1.2 Challenges with Time-Division Multiplexing

(1) TDM Schedule: The access latency of memory request when using TDM depends on the scheduling of TDM slots, even if they are unused. Such unused slots appear when an owner of a TDM slot does no (yet) have a memory request ready to be served. 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. Figure 4.1 illustrates such non-work-conserving behavior. Here the memory is shared by 4 cores, the arbitration policy being TDM, each core has a dedicated TDM slot to exclusively access the shared memory (C0 to C3). On this example, core 0 is issuing a memory request and has to wait for its dedicated TDM slot $C0$ to be granted access to the memory. However, the slots $C2$ and $C3$ are unused since no memory request is issued from the owners of those TDM slots. Therefore, the TDM arbitration policy constrains the requests to be processed during their dedicated TDM slots even if the memory is idle as illustrated in Figure 4.1.

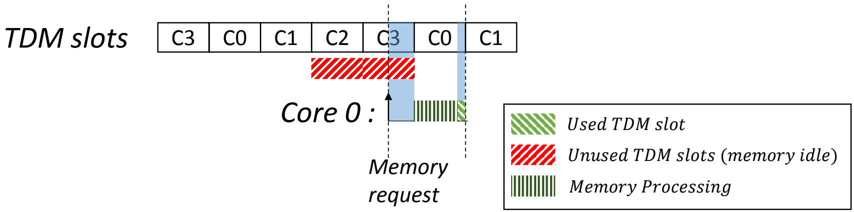


FIGURE 4.2: TDM slots induced delays.

(2) Fixed Slot Arbitration: Another issue of TDM stems from the use of fixed TDM slots as memory accesses are granted only at the beginning of TDM slots (see Subsection 3.2.4). Hence, when a request is issued within an unused slot, the access has to wait for the beginning of the upcoming TDM slot. Figure 4.2 shows an example where the request from core 0 has to wait for the start of its following TDM slot to be granted access to the memory. In this example, when the request from core 0 is issued, the current unused slot C3 cannot be exploited by its owner, because the memory access latency requires the full TDM slot. Therefore, even if a request is issued from the owner of slot C3 at that instant, the request has to wait for its following dedicated TDM slot (one TDM period later). However, since the owner of C3 cannot exploit the remaining time of its slot, the owner of the following slot C0 could use the remaining time of C3 to eagerly process C0's memory request. Here, the request from core 0 could start directly at its issue date. In this case, it is always safe to immediately start processing the request, an overflow into the next TDM slot is not an issue. However, due to the static scheduling of the TDM slots, the strict TDM arbiter does not allow such an eager processing. We can quantify the overhead caused by TDM's inefficiency by counting the number of (bus) clock cycles that processor cores have to wait:

Definition 4.1.1. *During the execution of a task under TDM, 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. As illustrated, in Figure 4.1, by the pending memory request within the unused slot C3.*

(3) TDM Slot Length: The length of TDM slots, expressed in clock cycles, represent another source of inefficiency. The TDM slot length has to be longer than the worst-case latency of handling memory requests, due to the fact that all memory requests have to complete within the duration of a TDM slot. Memory requests targeting a DRAM memory, however, have highly variable latencies [77]. The temporal behavior of the DRAM depends, for instance, on memory refresh operations or whether the accessed memory page changed, see SubSection 3.1.4 for more details. Besides, the access latencies of memory load requests are higher than those of memory writes,

since data must be sent back to the requesting core. Figure 4.2 illustrates the aforementioned issue, where the difference between the TDM slot length and the actual access latency, illustrated by the memory processing bar, indicates the delay induced by the fixed length of TDM slots. Therefore, a strict TDM arbitration, targeting a memory with variable latencies, introduces considerable pessimism when considering that all memory accesses take the worst-case latency. Again, we can quantify the induced overhead by counting the number (bus) clock cycles that the processor cores have to wait:

Definition 4.1.2. *During the execution of a task under TDM, 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 unused TDM slot. In Figure 4.1, if a memory request is pending during the time between the end of the memory's processing and the end of the TDM slot (*C0*), hence, this time difference is considered as a release delay.*

(4) Hardware Complexity: One of the main advantages of the TDM arbitration scheme is that it is easy to implement in hardware (see Section 3.2.4). To overcome the aforementioned limitations, we need to explore more sophisticated arbitration schemes. However, the hardware implementation costs represent one of the major drawbacks of sophisticated arbitration schemes, such as priority-based or budget-based arbitration. The challenge is to address the limitations of TDM without breaking the bank in terms of hardware implementation costs.

(5) TDM and Criticality Awareness: The TDM limitations presented above are all related to the non-work-conserving nature of the arbitration and the inefficient use of the main memory. A possible solution to TDM's non-work-conserving is to explore the ideas of multi-criticality scheduling. The level of a task's criticality should not only be used by task schedulers, but also by the memory arbiters. The challenge with this approach can be divided into two levels:

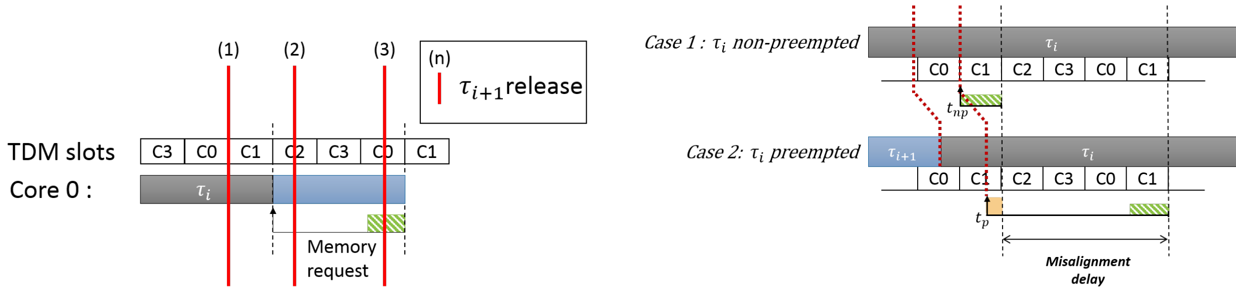
- **(5.a) Request Level:** Memory request arbitration considering task criticalities. By keeping the same temporal behavior offered by traditional TDM, i.e., guaranteeing a bounded memory access latency, for critical tasks. On the other hand, it is important to be able to have the best possible use of the hardware resource (memory utilization). The latter is not among TDM's strengths, given that it is not a work-conserving approach.
- **(5.b) Task Level:** Developing a sophisticated arbitration scheme may have an impact on tasks' executions with regard to a multi-task preemptive scheduling. Therefore, the overall system analysis (e.g. response time analysis) needs to take into account all possible arbitration-induced delays.

4.1.3 Preemption Costs for Regular TDM Arbitration

In multi-task scheduling, one must consider the preemption-related delays that need to be bounded and taken into consideration by the schedulability test. For this work, we focus on preemption costs caused by the arbitration policy and ignore other costs due to the scheduler invocation, context switching, pipeline flushes, or Cache-Related Preemption Delays (CRPD).

Subfigure 4.3a depicts three preemption scenarios of a task τ_{i+1} that preempts a lower-priority task τ_i at different release dates (red vertical lines). Both tasks are executing on core $C0$, hence, own slot $C0$. The first case (1) refers to a preemption occurring while the CPU performs computations (gray area). The task than can be preempted right away (ignoring potential pipeline stalls). The second case (2) refers to a situation where the preemption occurs while the CPU stalls, waiting to access the shared memory. While it would be possible to abort the pending memory request and immediately preempt τ_i [56], this requires modifications to the processor pipeline. In the last case (3), the preemption occurs while the memory processes a memory request. It would theoretically be possible to also abort the request at this stage – requiring modifications to the processor pipeline and throughout the entire memory hierarchy. A simpler alternative for cases (2) and (3), both in terms of hardware and timing analysis, would be to avoid aborting the request and simply wait for its completion [6].

Clearly, all three cases may induce preemption-related delays that need to be bounded and taken into consideration by the schedulability test (in addition to classical CRPDs). The worst-case delay experienced for case (1) is *trivial* and only depends on the characteristics of the processor pipeline. Case (3) similarly is analyzable and can be bounded by the worst-case memory latency, e.g., a TDM slot length [6]. The analysis of case (2) is more complex since the behavior



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

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

FIGURE 4.3: Preemption effects w.r.t. TDM memory arbitration.

potentially depends on the other tasks in the system and the memory arbitration policy. We first analyze the timing behavior for regular TDM and later extend this analysis to our work in Chapter 7.

Definition 4.1.3. *Under a system with fixed-priority preemptive scheduling, the* ***memory blocking delay*** *(MB) denotes the number of clock cycles that a higher-priority task τ_{i+1} is blocked from executing on its core after its release, due to a pending memory request by a lower-priority task τ_i. This memory blocking delay is illustrated by case (2) in Subfigure 4.3a.*

However, the blocking time does not cover all preemption-related delays of regular TDM. Recent work [41, 58] proposed sophisticated WCET analyses, which exploit the relative alignment of the program execution with regard to the TDM schedule, without preemption. A preemption may impact the program's relative alignment – unless task scheduling itself is aligned with the TDM period.

Definition 4.1.4. *The* ***misalignment delay*** *(MA) denotes the number of additional clock cycles that the first memory access of a task takes, w.r.t. the worst case considered by the WCET analysis, when resuming after a preemption. As illustrated in Subfigure 4.3b.*

Subfigure 4.3b depicts such a misalignment delay under regular TDM for a request of a critical task τ_i^c running on core C1, hence, owning TDM slot C1. Case 1 (top) illustrates an execution without preemption, where a request is issued right at the beginning of the task's TDM slot at time instant t_{np}. The program's alignment w.r.t. the TDM schedule is ideal and the request is processed immediately. Case 2 (bottom) shows the *same* execution of τ_i^c after a preemption by τ_{i+1}. In the absence of other side-effects, such as CRPDs, the same computations are performed by τ_i^c up to its first memory request (as indicated by the red dotted lines), which now is issued at time instant t_p. However, the task's alignment w.r.t. the TDM schedule was slightly shifted due to the preemption. The request thus has to wait longer than expected by the WCET analysis, which assumed an execution without preemption.

Given the fact that we aim to work with TDM-based arbitration schemes, we have to study the impact of our approaches on these delays (*memory blocking delay* and *misalignment delay*). A detailed overview of such impacts are presented in Chapter 7.

4.2 Contribution Outline

We have presented a summary of the various challenges regarding the memory arbitration schemes, as a reminder, we are exclusively interested in the TDM arbitration policy. TDM offers very good temporal properties, but due to its non-work-conserving nature, it results in low resource utilization. In this work, our aim is to improve memory utilization without loosing the good temporal properties offered by TDM. To do this, we propose a work-conserving dynamic and criticality aware TDM-based arbitration scheme.

Our approach is intended to resolve the different challenges presented above, therefore the contributions are organized as follow. We first present dynamic TDM-based arbitration schemes in Chapter 5 dealing with the challenge of criticality awareness at the memory request level and the scheduling of TDM slots, addressing challenges **(1)** and **(5.a)** from Section 4.1.2. We also extend the approach to overcome the limitations regarding the fixed slot arbitration and the pessimism of TDM slot length, i.e., resolving challenges **(2)** and **(3)**. The approach no-longer performs arbitration at the granularity of slots, but at the level of clock cycles. The chapter also covers a demonstration of the equivalence between the proposed approach and the regular TDM arbitration w.r.t. the worst-case behavior. The chapter ends with a description of the experimental setup and an evaluation of the proposed approaches in terms of memory utilization and efficiency.

After addressing the limitations of TDM, Chapter 6 presents a simple and efficient hardware implementation of a variant of our proposed arbitration schemes aiming to reduce the hardware complexity related to challenge **(4)**.

The remaining challenge is the task level impact of such a sophisticated dynamic arbitration scheme, c.f., challenge **(5.b)**. Chapter 7 briefly reviews the dynamic TDM-based arbitration schemes proposed in this thesis, and identifies the different arbitration-induced preemption delays inherited from TDM and/or specific to our arbitration strategies w.r.t. the identified preemption delays introduced in Section 4.1.3. We propose in this chapter various preemption models to handle these delays and evaluate our contributions using schedulability success ratios and memory utilization.

Chapter 5

Dynamic TDM-based Arbitration

Contents

5.1 Criticality Aware TDM-based Arbitration (TDMfs) 48
5.2 Dynamic TDM-based Arbitration Schemes 50
5.2.1 TDMdz: Deadline Driven Arbitration 50
5.2.2 TDMds: Dynamic TDM Arbitration with Slack Counters 52
5.3 Decoupling Dynamic TDM Arbitration from Slots 55
5.3.1 TDMes: Decoupled from TDM slots 55
5.3.2 TDMer: Memory Variability Awareness 58
5.4 Worst-Case Behavior 59
5.5 Experiments for Dynamic TDM-Based Arbitration Schemes 63
5.5.1 Experimental Setup 63
5.5.2 Results for Dynamic TDM-Based Arbitration Schemes 68
5.5.3 Results for Dynamic TDM with Initial Slack 71
5.5.4 Results for Varying Memory Access Latencies 74
5.6 Conclusion 77

In a multi-core architecture, the contention between concurrent accesses to shared memory represents a major challenge in real-time systems. We observed in Subsection 3.2.4, 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*. The problem is further amplified as the number of cores increases, leading to longer TDM schedules.

In this chapter, we address two of the limitations presented in Subsection 4.1.2 namely *TDM schedule* and *multi-criticality at request level*, points **(1)** and **(5.a)** respectively, by exploring dynamic TDM-based arbitration schemes. Under certain conditions, the arbiter allows to favor requests of non-critical tasks over requests from critical tasks. This is achieved by associating deadlines to the memory accesses of critical tasks, which correspond to the end of their corresponding slot under a regular TDM scheme. The arbitration is, therefore, based on an *Earliest-Deadline First* (EDF) strategy. The main motivation of applying EDF is to prioritize non-critical requests over critical requests whenever possible in order to improve the memory utilization. Besides, a deadline driven arbitration scheme leads to requests completing earlier compared to their deadlines. As a result, a *slack time* can be derived and used to provide more dynamic memory arbitration scheme, while maintaining the temporal guarantees ensured by TDM.

The rest of the chapter is organized as follows. Section 5.1 introduces our technique to integrate task criticality into memory request arbitration. Section 5.2 presents the TDM-based approach driven by deadlines and slack accumulation which provides a dynamic dimension to the TDM scheduling. Thereafter, an approach is presented to solve issues related to the very nature of TDM namely *fixed slots arbitration* and *TDM slot length* pessimism in Section 5.3.1. 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 memory requests. The equivalence of the proposed approach and the regular TDM arbitration w.r.t. the worst-case behavior is demonstrated in Section 5.4. Section 5.5 discusses performance evaluation of the approach, before concluding the chapter in Section 5.6.

5.1 Criticality Aware TDM-based Arbitration (TDMfs)

In this chapter, we explore criticality aware memory request arbitration on a multi-core architecture consisting of m cores. As already discussed in the Section 2.3, For now we assume a restricted scheduling model, where each core executes a single independent and periodic task τ_i, $1 \leq i \leq m$ (a more realistic system model is considered in chapter 7). To recall, we consider in our model, as stated in Section 2.3.1, two classes of tasks: *critical* and *non-critical* tasks. We 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 means to bound the worst-case execution times of these tasks. Non-critical tasks, on the other hand, may miss their deadlines. We do not demand strict worst-case execution time bounds for them in this chapter. 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.

The *non-work-conserving* of TDM stems from various limitations (see Section 4.1.2 for more details). One of them called *TDM schedule* occurs when the owner of a TDM slot does not (yet) have a memory request ready to be served. Under a regular TDM scheme, this slot cannot be reclaimed by another task (as for instance under Round Robin). We will first address this issue through the integration of the criticality-awareness into TDM. This first approach called TDMfs (**f**ree **s**lots) aims to exploit the TDM unused slot in favor of non-critical memory request.

The following example assumes an architecture with 3 cores (C_0, C_1, C_2) executing respectively two critical tasks τ_0^c and τ_1^c and one non-critical task τ_2^{nc} and performs concurrent memory requests to the shared memory. The task set is similar to the one illustrated in Section 3.2.4, the main difference relies on the criticality awareness of the memory arbitration scheme. The TDMfs arbitration scheme is aware of task criticality and applies a different arbitration compared to regular TDM. The main difference here relies on the TDM slot allocation, thus only critical tasks have dedicated TDM slots and non-critical tasks can only reclaim unused slots left by critical tasks. Here, each critical task has dedicated TDM slots (vertical columns, labeled C_0 and C_1 representing the cores executing the critical tasks), which alternate every 8 cycles (corresponding to the TDM slot length Sl) resulting in a TDM period $P = 16$ cycles.

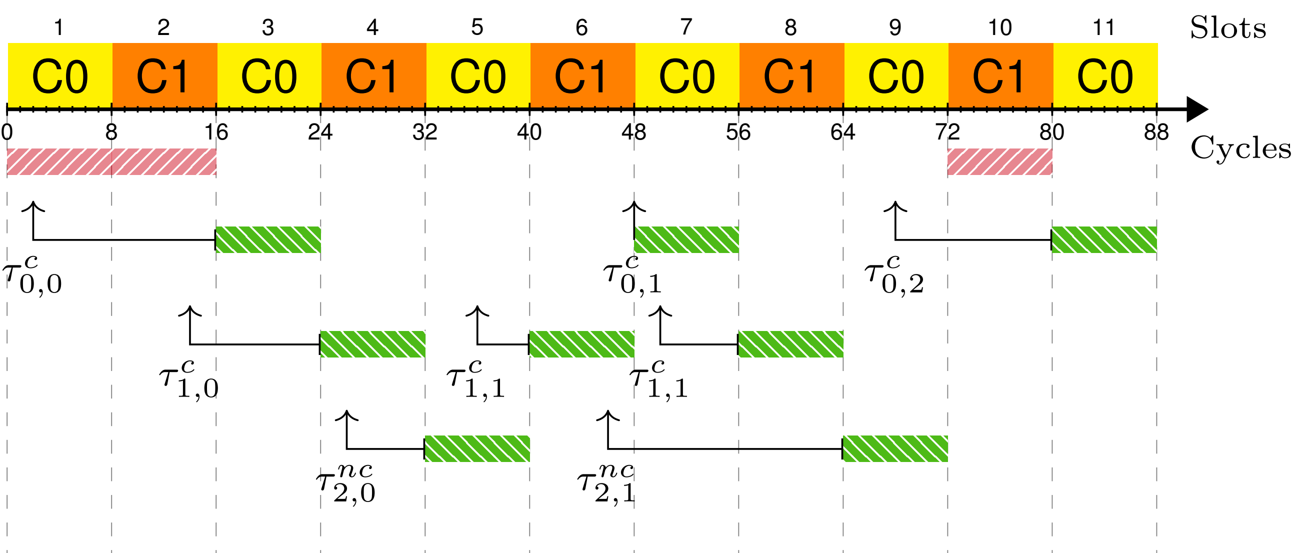


FIGURE 5.1: Criticality aware TDMfs arbitration of three tasks τ_0^c, τ_1^c and τ_2^{nc}.

Figure 5.1 shows a TDMfs arbitration. Requests from critical tasks, here, are granted access during their dedicated TDM slots as for regular TDM arbitration scheme. The requests from non-critical tasks ($\tau_{2,0}^{nc}$ and $\tau_{2,1}^{nc}$) can only reclaim unused slots. The first request $\tau_{2,0}^{nc}$ is granted access at slot 5 which is normally dedicated to requests from tasks scheduled on core C0 (τ_{0}^{c}). The arbiter checks if their is any pending requests from the current owner of the TDM slot if not, than it grants access to a pending non-critical request. The arbitration among pending non-critical requests can be chosen arbitrarily, the main idea here is to always grant access critical requests at their dedicated TDM slots as if they were arbitrated using a regular TDM approach. Non-critical tasks memory request, on the other hand, are scheduled in a best effort manner, taking advantage of the unused slots.

5.2 Dynamic TDM-based Arbitration Schemes

In this section, we present a dynamic TDM-based arbitration schemes. The goal is to have a work-conserving arbitration policy with the same guarantees offered by TDM for critical tasks.

5.2.1 TDMdz: Deadline Driven Arbitration

One way of interpreting TDM is to associate deadlines with requests. For regular TDM this deadline corresponds to the end of the next TDM slot of a critical task. For simplicity, we assign the end of the upcoming next TDM slot as the deadline for requests from non-critical tasks. The deadlines ensure that requests of critical tasks are in the worst-case completed at the same instant as an execution under a regular TDM scheme. If a critical and a non-critical task have the same request deadline, the critical task wins. If a non-critical request misses its deadline, we simply push the deadline a slot length into the future. Deadlines are unique among critical tasks, the TDM arbiter thus can ensure that requests complete exactly at their deadlines.

Definition 5.2.1. *Critical Request Deadline. Considering TDM arbitration, a deadline d_j of the jth memory request issued by a critical tasks corresponds to the end date its dedicated TDM slot.*

Definition 5.2.2. *Non-Critical Request Deadline. The deadline of non-critical requests under dynamic TDM-based arbitration 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.:*

$$d_j = \left\lceil \frac{a_j}{Sl} + 1 \right\rceil \cdot Sl$$

Algorithm 1 Deadline computation for critical tasks.

```
1: function DEADLINE(Arrival, CurrentPeriod, SlotOffset)
2:     Deadline = CurrentPeriod + SlotOffset + SlotLength − 1
3:     if Arrival − CurrentPeriod > SlotOffset then
4:         Deadline = Deadline + TDMPeriod
5:     return Deadline
```

Now, every request carries a deadline and a more dynamic `TDM`-based arbiter could very well chose to execute requests before their actual deadlines. For this it suffices to order the requests in a priority queue by their deadlines. On a tie the priority queue prioritizes the critical over non-critical tasks.

The arbiter represents requests as pairs (a_j, d_j), consisting of the arrival date and deadline. Requests are immediately issued to the arbiter's priority queue upon arrival and ordered by their deadline (prioritizing critical tasks over non-critical). At the beginning of each `TDM` slot the arbiter selects a request $\tau_{i,j}$ with the highest priority (i.e., lowest deadline) and assigns a completion date $C(\tau_{i,j})$ to it. The deadline of all non-critical requests that missed their deadline is incremented by the slot length Sl. If the queue is empty the `TDM` slot is unused and no request is completed. An execution is valid if for every request $\tau^c_{i,j} = (a_j, d_j)$ of a critical task $C(\tau^c_{i,j}) \leq d_j$ holds.

Algorithm 1 shows how to compute the deadline d_j of a critical request under the `TDMdz` scheme. The algorithm takes three arguments, the arrival date (`Arrival`), the start of the current `TDM` period (`CurrentPeriod`), and the offset of the task's own `TDM` slot with regard to the beginning of a `TDM` period (`SlotOffset`). In addition, the constant length of the `TDM` slot and `TDM` period is needed (`SlotLength` and `TDMPeriod` respectively). For non-critical tasks the deadline is the end date of the current `TDM` slot plus the `TDM` slot length. The deadline of request $\tau^c_{0,0}$, in Figure 5.2, is computed w.r.t. its arrival date 2 cycles after the start of the task. Hence, the arrival date $a_0 = 2$ (`Arrival = 2`) cycles. The current period start is at cycle 0 (`CurrentPeriod = 0`) w.r.t. to request arrival date. The slot offset of τ^c_0's `TDM` slot w.r.t. the beginning of a `TDM` period is 0 (`SlotOffset = 0`). Therefore, following the Algorithm 1, the deadline computation of request $\tau^c_{0,0}$ takes into account its slot offset, the beginning of the current `TDM` period, and the `TDM` slot length which are respectively 0, 0, and 8, therefore its deadline is $d_0 = 0 + 0 + 8 - 1 = 7$ (Line 2). Since the request's arrival offset ($2 - 0 = 2$) is too large (Line 4), the request misses τ^c_0's slot at offset 0 and is delayed by a TDM period (+16). Therefore, the deadline of request $\tau^c_{0,0}$ is $d_0 = 7 + 16 = 23$.

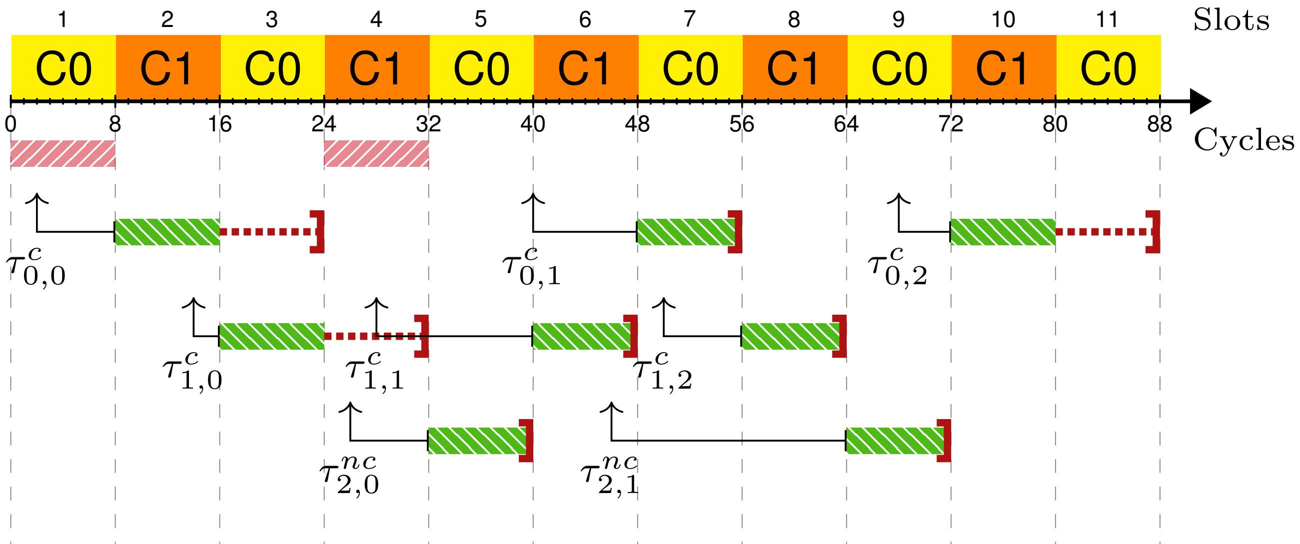


FIGURE 5.2: Deadline driven `TDMdz` arbitration of three tasks τ^c_0, τ^c_1 and τ^{nc}_2.

Figure 5.2 shows all 3 tasks executing under such an arbitration scheme, which we call `TDMdz` (**d**eadline-driven and **z**ero slack). Critical requests may now complete before their deadlines, which is indicated by a doted line (⋅]). In this example requests $\tau^c_{0,0}$, $\tau^c_{0,1}$ and $\tau^c_{1,1}$ each complete 1 `TDM` slot earlier than their deadlines. Similar to `TDMfs`, non-critical tasks reclaims unused `TDM` slots ($\tau^{nc}_{2,1}$ in slot 9), but may also delay critical requests ($\tau^{nc}_{2,0}$ wins over $\tau^c_{1,1}$ in slot 5).

5.2.2 TDMds: Dynamic TDM Arbitration with Slack Counters

A closer comparison between Figure 5.1 and Figure 5.2 reveals that requests $\tau^c_{0,0}$, $\tau^c_{1,0}$ and $\tau^c_{0,2}$ each completes ahead of the original execution under `TDMfs` arbitration. In all cases, the WCET analysis will always take into account the worst case scenario, i.e., memory requests issued from critical tasks are always scheduled at their dedicated `TDM` slot. An effective way to capitalize on this critical task lead compared to its worst-case, is to be able to leverage it for non-critical tasks memory requests. This observation gives rise to an extension of `TDMdz` that tracks and accumulates slack time with regard to an execution under regular `TDM`. This can be done by subtracting the completion date of a request from its deadline. Furthermore, the slack (Δ) accumulation ensures a more accurate deadline computation. By storing the advance made by previous requests, we can compute requests deadlines that will always match the completion dates under regular `TDM` arbitration.

Definition 5.2.3. *Slack Time. The slack time corresponds to the difference between the memory request completion under dynamic `TDM`-based arbitration and its deadline. The later referring to the completion*

date under regular `TDM`*, the slack time is therefore the time gained by the dynamic arbitration compared to regular* `TDM`*.*

Definition 5.2.4. *Slack Counter. Under dynamic* `TDM`*-based arbitration, the slack counter of a critical task τ_i^c after the completion of its jth memory request is given by: $\Delta_j = d_j - c_j$.*

The key idea of the *dynamic* `TDM` *with slack counters* (`TDMds`) arbitration scheme is to interpret `TDM` scheduling as driven by deadlines. Under regular `TDM` each request completes precisely at the end of the request owner's next `TDM` slot, which can be seen as deadline. The deadlines of `TDMds` similarly correspond to the end of `TDM` slots. However, instead of systematically delaying requests until their respective deadlines, requests are processed dynamically in any order – as long as deadlines are met. This allows to compute the *slack time* of critical tasks, i.e., by how much the task's last request completed earlier w.r.t. the request's deadline. The slack is stored in dedicated counter and allows to prioritize non-critical requests, i.e., spend the slack of critical tasks in favor of non-critical tasks. Note, however, that slack accumulated within a job of a task is naturally not preserved for subsequent jobs. Slack counters are consequently reset at job start. Also note that deadlines are ensured for critical requests only, while non-critical requests are processed in a best-effort manner. The dynamic processing allows non-critical tasks, for instance, to reclaim otherwise unused `TDM` slots (`TDMfs`).

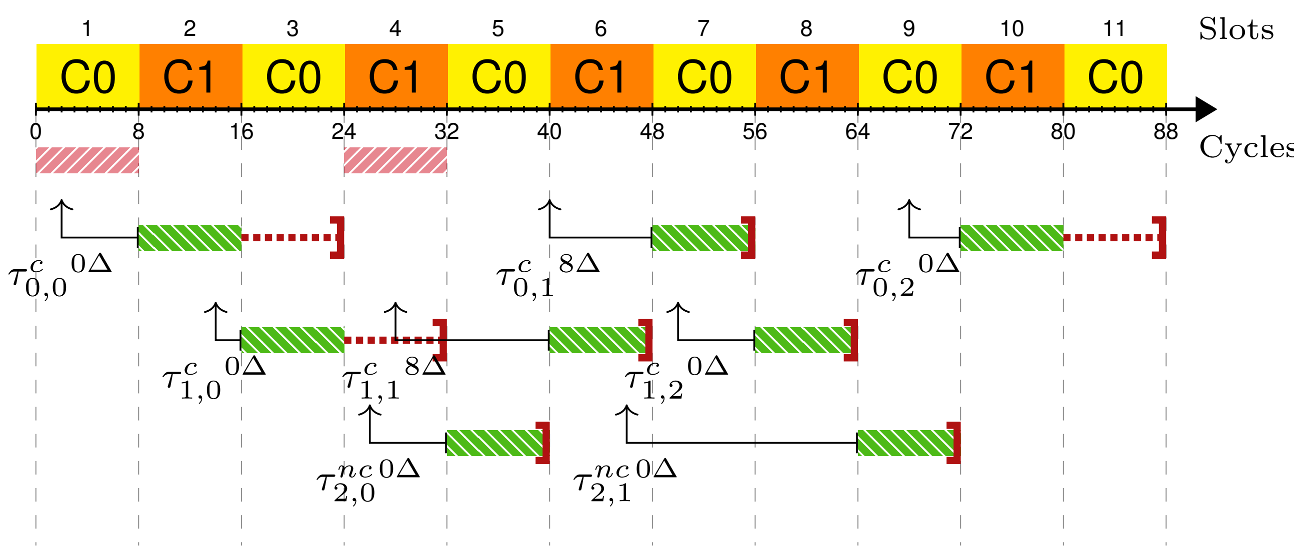


FIGURE 5.3: Deadline driven and slack time accumulation `TDMds` arbitration of three tasks τ_0^c, τ_1^c and τ_2^{nc}.

Figure 5.3 illustrates the resulting arbitration scheme, TDMds. Tasks may now accumulate slack (illustrated by request superscript $x\Delta$) which gives more freedom to the arbiter to choose which request to handle next. A critical request may arrive earlier than expected under a regular TDM scheme if the request's owner accumulated some slack before. The request's deadline would then also appear earlier than under regular TDM if one would simply consider the request's issue date to compute the deadline, i.e., the end of the next TDM slot after the issue date. Under TDMds the deadline of a new critical request is instead computed from a delayed issue date, which is derived by adding the previously accumulated slack to the request's original issue date. A memory request delayed issue date is computed to determine its arrival date if the arbitration used was regular TDM. Given enough slack this may push the request's deadline into the future, i.e., past the next TDM slot, and provide additional freedom to the arbiter. This also ensures that the deadlines under TDMds **exactly** correspond to the deadlines/completion dates that would be observed under regular TDM. This holds for **any** dynamic execution of **any** program. For each critical request, this can be visualized on Figures 5.2, 5.3, 5.4, and 5.5 compared to requests completion dates on Figure 5.1.

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

As illustrated by the example above, the use of deadlines and slack counters allows the TDMds arbiter, compared to TDMfs, to improve the utilization of the memory. This is visible on the reduced number of unused slots considering the same task set. However, issues stemming from the very nature of TDM remain, related to the use of fixed TDM slots as the request arbitration only occurs at the beginning of slots and the TDM slot length pessimism as described in Subsection 4.1.2.

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 5.3 shows three such requests, namely $\tau^c_{0,0}$, $\tau^c_{1,1}$, and $\tau^{nc}_{2,0}$. The arbitration of TDMds is limited to TDM slots and thus cannot immediately grant

request $\tau_{2,0}^{nc}$ 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.

5.3 Decoupling Dynamic TDM Arbitration from Slots

The use of deadlines and slack times allows the arbiter to improve the memory utilization. However, issues stemming from the very nature of TDM remain, both are related to the use of fixed TDM slots namely issue and release delays (see definitions 4.1.1 and 4.1.2). Therefore, to overcome these limitations, we describe how the arbitration can be decoupled from TDM slots in order to reduce issue delays by considering slack counter. Afterward, a slight variation is presented to address release delays.

5.3.1 TDMes: Decoupled from TDM slots

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 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 2. 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 (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 5.4 illustrates the resulting arbitration under `TDMes` for the task set from before $(\tau_0^c, \tau_1^c, \tau_2^{nc})$. Requests can now start early, if the conditions described before are met. This is the case for request $\tau_{2,0}^{nc}$, 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 core $C0$ executing task τ_0^c, whose slack counter is 8 (stemming from access $\tau_{0,0}^c$). At the moment when $\tau_{2,0}^{nc}$ is issued, the arbiter thus needs to verify the second condition of Algorithm 2. Task τ_0^c could potentially issue a request in the next cycle (27), which

Algorithm 2 Condition to apply *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
```

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 $\tau_{2,0}^{nc}$ access to the memory, as shown in the figure. The same result can be obtained by comparing the distance $(32 - 26 = 6$ cycles) to the next TDM slot with τ_0^c's slack counter (8 cycles), since $6 < 8$. The same situation arises for request $\tau_{2,1}^{nc}$. The arbiter handles request $\tau_{2,1}^{nc}$ 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 τ_0^c's slack counter. The remaining requests in the example, except for $\tau_{0,0}^c$, fall into the first condition of Algorithm 2, i.e., the owner of the request is also the owner of the subsequent TDM slot. For instance, request $\tau_{1,0}^c$ is processed within slot 3, as core $C1$ executing τ_1^c owns slot 4. The early-start optimization cannot be applied to $\tau_{0,0}^c$, since task τ_1^c, 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 5.3), the TDMes policy is again more efficient. The memory is almost always busy and the last request completes at cycle 73 (as opposed to 80 under TDMds and 88 under TDMfs). The approach also has an impact on the non-critical task execution time that ends earlier at cycle 68 compared to cycle 72 for TDMds. As non-critical tasks memory requests are processed earlier, this allows more requests to be processed while pushing critical tasks memory requests to their deadlines.

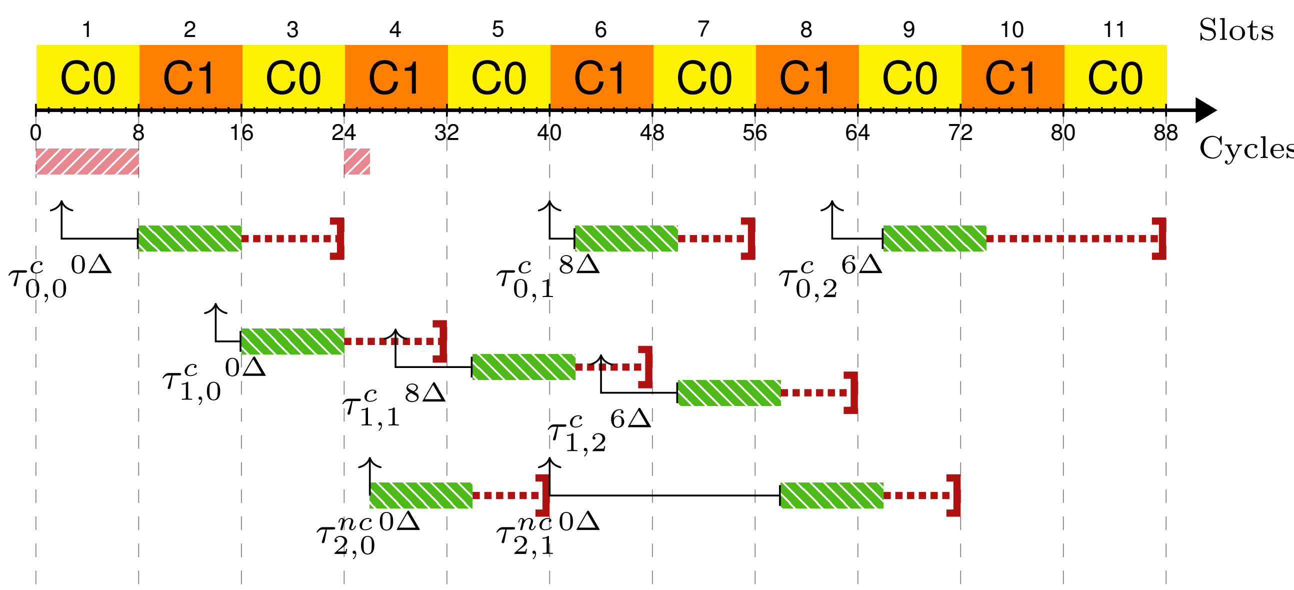


FIGURE 5.4: Reduced issue delays due to the TDMes arbiter, which operates independently from the actual TDM slot length.

5.3.2 TDMer: Memory Variability Awareness

Up to now, the actual behavior of the memory to handle requests (load or store) was irrelevant to the memory arbiter. Which simply rely 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. This may introduce considerable pessimism in the form of *release delays*, as the actual memory latency typically varies from access to access depending on the internal state of the underlying memory technology, as described in Subsection 3.1.4.

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.

Figure 5.5 again shows an execution trace for the task set $(\tau_0^c, \tau_1^c, \tau_2^{nc})$ under the TDMer scheme from Subsection 3.2.4. The main difference concerns the duration of the memory processing time, which is now illustrated by green hatched bars of variable length (▨). Request $\tau_{0,0}^c$, for

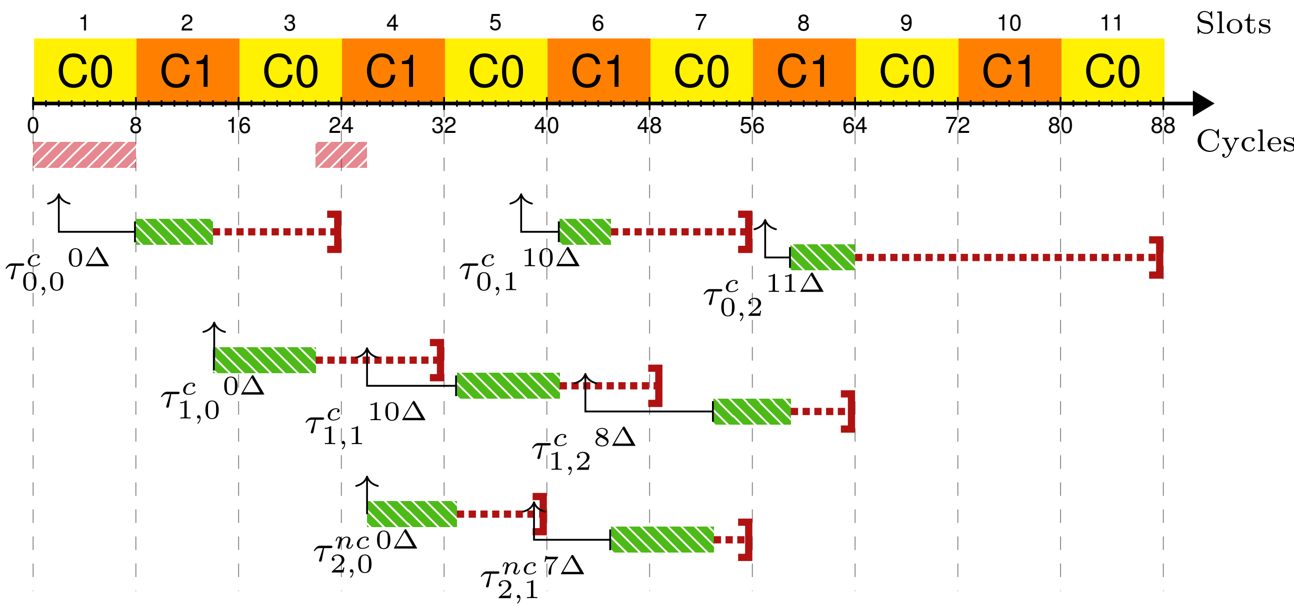


FIGURE 5.5: Elimination of release delays under TDMer arbitration, which considers the actual latency of memory access. Some of the *eliminated* release delays may simply be transformed into issue delays.

instance, now completes 2 cycles earlier than before. This entails several changes. Firstly, the slack counter of task τ_0^c increases by an additional 2 cycles, which now amounts to 10 cycles (cf. ${\tau_{0,0}^c}^{10\Delta}$). Secondly, request $\tau_{1,0}^c$ 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 $\tau_{0,1}^c$ completes in TDM slot 6 instead of TDM slot 7. This allows the arbiter to change the order of requests $\tau_{2,1}^{nc}$ and $\tau_{1,2}^c$, since $\tau_{1,2}^c$'s deadline is far enough in the future and task τ_0^c has sufficient slack (11). 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 Subsection 5.3.1). In this case, the early-release optimization does not actually improve the memory utilization and release delays are simply transformed into issue delays.

In comparison to TDMes (Figure 5.4), again an improvement is achieved, despite a slight increase in the memory's total idle time (cf. the red hatched bars (▨). The last request ($\tau_{0,2}^c$) completes at cycle 64, as opposed to cycle 73 for TDMes. In particular the non-critical task τ_2^{nc} gained from the altered schedule and completes its last memory request 13 cycles earlier. Only task τ_1^c 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 deadline for critical requests in Figures 5.3 through 5.5 match.

The improvements are even greater when comparing with the original TDM-based execution (Figure 3.3), which completed after 104 cycles. TDMer yields an improvement of a 62.5% – while, in the worst case, preserving a strict separation between critical and non-critical tasks. Note, that the requests completion dates in Figure 3.3 are not comparable to the completion dates/deadlines shown in the other figures, due to the presence of a third TDM slot for $C2$.

Both approaches, TDMes and TDMer, improve the memory utilization, while converging to regular TDM in the worst case (see Section 5.4). This allows to preserve properties that make TDM popular – including results obtained from advanced TDM based program analysis [41, 58].

5.4 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 5.4.1. *(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 regular `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 [41, 58].

In order to show the correctness of Theorem 5.4.1 we 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 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. As described in Subsection 3.2.4, Each of the n critical tasks has its own slot with a length of Sl, resulting in a repetitive `TDM` schedule with a period $P = n \cdot Sl$. For each memory request issued by a critical task τ_i^c the arbiter is assumed to store or compute the following information: the request's arrival date (aka. issue date) a_j, completion date c_j, and deadline d_j as well as the start date of the current `TDM` period Sp, and the offset of the task's `TDM` slot $O(\tau_i^c)$ (with regard to Sp).

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

Definition 5.4.1. *`TDM` Request Deadline. Considering `TDM` arbitration, the deadline d_j of the jth request issued by a critical task τ_i^c is given by:*

$$d_j = \begin{cases} Sp + O(\tau_i^c) + Sl & \text{if } a_j \leq Sp + O(\tau_i^c) \\ Sp + O(\tau_i^c) + P + Sl & \text{else.} \end{cases}$$

Lemma 5.4.1. *Given Definition 5.4.1, the deadline of a critical request corresponds to the end date of its dedicated `TDM` slot. The deadline d_j is consequently always equal to the completion date c_j under regular `TDM` arbitration.*

Proof. The deadline d_j always corresponds to the end of τ_i^c's TDM slot, as each of the arguments used in the computation is, by definition, a multiple of the TDM slot length Sl (Sp, $O(\tau_i^c)$, and P). The formula simply distinguishes two cases (1) when the request is issued at or before the start of τ_i^c'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 5.4.2. *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 $O(\tau_i^c)$ and $O(\tau_k^c)$ have to be different when $i \neq k$. □

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., $c_j \leq d_j$) can be used to implement a TDM-based arbitration scheme that preserves Theorem 5.4.1, e.g., an implementation based on the earliest-deadline-first strategy like TDMer (see Theorem 5.4.2). However, since our approach 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 (cf. Definition 5.2.4) for each critical task. These slack counters are updated after every completion of a critical memory request. The slack is then used to compute a delayed issue date for the next request of τ_i^c. 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 5.4.3. *The deadlines for critical requests under TDMer correspond to the same deadlines as under regular TDM.*

Proof. Considering Definition 5.2.4, we can show by induction that a request's issue date $a_j^{\mathtt{TDM}}$ under regular TDM always corresponds to the delayed issue date under TDMer. This last date can be computed from the original issue date $a_j^{\mathtt{TDMer}}$ and the slack counter value Δ_{j-1}, i.e., $a_j^{\mathtt{TDM}} = a_j^{\mathtt{TDMer}} + \Delta_{j-1}$.

Induction base $j = 0$: $\Delta_0 = 0$ the issue date under TDM and TDMer are naturally the same, i.e. $a_0^{\mathtt{TDM}} = a_0^{\mathtt{TDMer}}$.

Induction step: Assuming a composable architecture that ensures that the delay $dist_j$ between the j-th and $(j-1)$th memory request is constant (cf. Section 3.3), we obtain:

$$dist_j = a_j^{\mathtt{TDM}} - c_{j-1}^{\mathtt{TDM}} = a_j^{\mathtt{TDMer}} - c_{j-1}^{\mathtt{TDMer}}$$

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

$$\begin{aligned}\Delta_{j-1} &= d_{j-1}^{\texttt{TDMer}} - c_{j-1}^{\texttt{TDMer}} = d_{j-1}^{\texttt{TDM}} - c_{j-1}^{\texttt{TDMer}} = c_{j-1}^{\texttt{TDM}} - c_{j-1}^{\texttt{TDMer}} = \\ &= c_{j-1}^{\texttt{TDM}} + dist_j - c_{j-1}^{\texttt{TDMer}} - dist_j = \\ &= c_{j-1}^{\texttt{TDM}} + (a_j^{\texttt{TDM}} - c_{j-1}^{\texttt{TDM}}) - c_{j-1}^{\texttt{TDMer}} - (a_j^{\texttt{TDMer}} - c_{j-1}^{\texttt{TDMer}}) = \\ &= a_j^{\texttt{TDM}} - a_j^{\texttt{TDMer}} \\ &\implies a_j^{\texttt{TDM}} = a_j^{\texttt{TDMer}} + \Delta_{j-1}\end{aligned}$$

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

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. The deadlines of non-critical (cf. Definition 5.2.2) requests may obviously collide with deadlines of critical tasks. The arbitration policy thus has to take these collisions into account.

Definition 5.4.2. *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 5.4.3. *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 2.*

Theorem 5.4.2. *TDMer ensures that any critical request completes before its deadline, i.e., $c_j \leq d_j$, in addition to Theorem 5.4.1.*

Proof. Assume that request j of a critical task τ_i^c is the first critical request in the system that missed its deadline, i.e., its $c_j > d_j$. This means that the memory was busy processing another request at the beginning of τ_i^c's TDM slot.
First assume the case that the request l of another task τ_k^c being processed by the memory was granted access to the memory at instant t after the issue date of request j, i.e., $a_j \leq t$. This implies that the deadline d_l is either smaller or equal to d_j, otherwise j 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 5.4.2), which again would exclude its admission to its weaker priority. It follows that $d_l < d_j$ and, due to Lemma 5.4.1, $d_l \leq d_j - Sl$. This, however, would imply that the request l missed its deadline as it was still being processed at the beginning of τ_i^c's slot. This contradicts the assumption that j 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 j, i.e., $t < a_j$. This implies that the early-start optimization was applied. More precisely the second condition (Line 6), as l cannot originate from τ_i^c. However, this is impossible since $t < a_j + \Delta_{j-1} < d_j - Sl$ has to hold while the early-start admission test at the same time requires $a_j + \Delta_{j-1} > d_j - Sl$. □

It is thus impossible that any critical request misses its deadline, which, in addition, is equal to that of an execution under `TDM` (Lemma 5.4.3). Non-critical requests are, as expected, potentially subject to deadline misses.

5.5 Experiments for Dynamic TDM-Based Arbitration Schemes

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.

5.5.1 Experimental Setup

Our simulation framework is based on the task model presented in Section 2.3.1. However, for now we assume a restricted scheduling model, where each core executes a single task. 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 (see Section 5.1), (2) `TDMdz` and `TDMds`, a dynamic `TDM`-based arbitration policy respecting `TDM` slots (see Section 5.2.2), (3) `TDMes`, a dynamic approach decoupled from `TDM` slots (see Section 5.3.1), and (4) `TDMer`, a refinement of `TDMes` addressing release delays (see Section 5.3.2).

All experiments are based on randomly generated synthetic task sets obtained via *UUniFast* [25] 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 our system is constrained 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. Note that it does not matter whether the first task period T_1 is chosen randomly or is fixed, as in our case. 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, $hp = \text{LCM}_{1 \leq i \leq n}(T_i)$ – here hp can be at most 1200 ms. 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. $dist_j$ in Section 2.3.1). 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 [29] in order to calibrate the traffic generator. The MiBench benchmarks were first executed individually on the Patmos architecture [60] 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

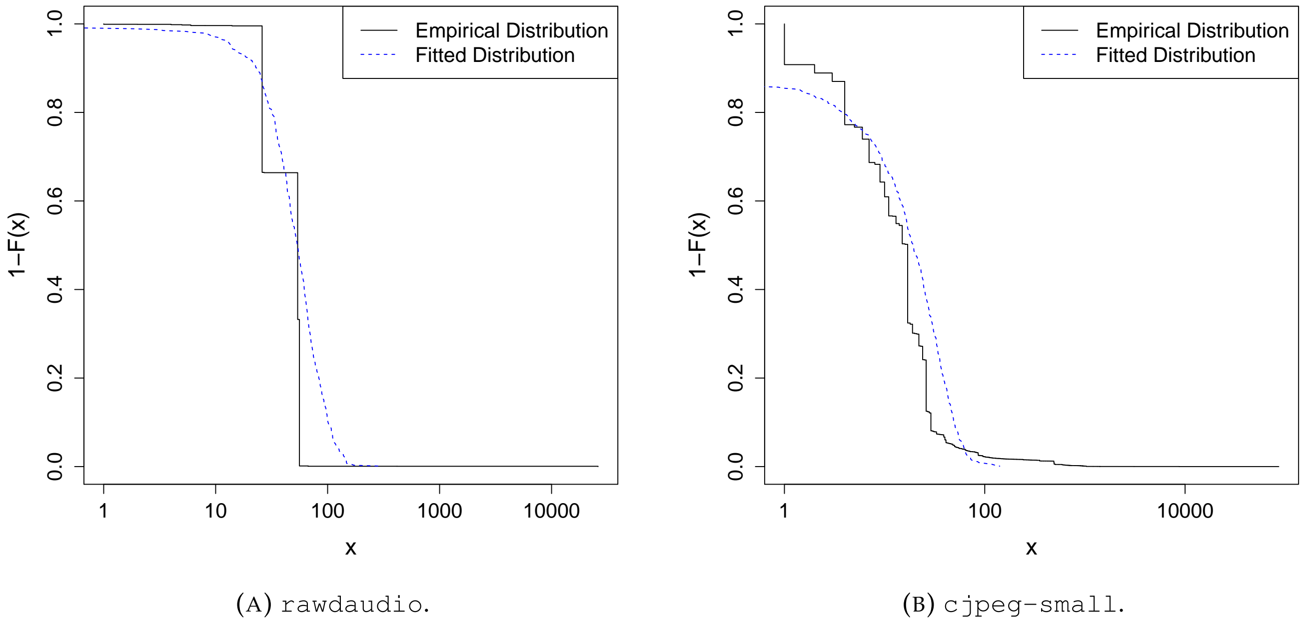
byte *stack cache* with block size 4, a 32 KB write-through *data cache* with *LRU* replacement on 4 sets and 32 byte blocks. Memory latencies are ignored during this trace collection.

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) [31] 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. $dist_j$ in Section 2.3.1) between them. Note, however, that the generated memory accesses have to be consistent with the task's worst-case execution time C_i. 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.

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* [78] package of the R statistical computing environment, in order to fit the parameters μ, σ, and ξ to the trace data in Subsection 5.5.1. 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 5.6 shows the inverted empirical distribution functions of the request distances (in cycles) from two application traces (`rawdaudio` and `cjpeg-small`) and compares them to



(A) `rawdaudio`.

(B) `cjpeg-small`.

FIGURE 5.6: Empirical distributions of the request distances (in cycles) of two MiBench applications compared with the GEV distributions after fitting.

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. Assuming that the number of memory request for a job k is given by $NbrAcc_i^k$, this allows us to characterize the job's behavior. To do so, we compare the processor demand PD_i^k and memory demand MD_i^k, which together must not exceed the task's WCET, i.e., $PD_i + MD_i \leq C_i$. These parameters emerge from the generated memory sequence of a job as follows:

$$PD_i^k = \sum_{k=0}^{NbrAcc_i^k} dist_j$$

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

Here, $dist_j$ refers to the distance between the j-th and $(j-1)$-th memory access of a job of task τ_i. Figure 5.7 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

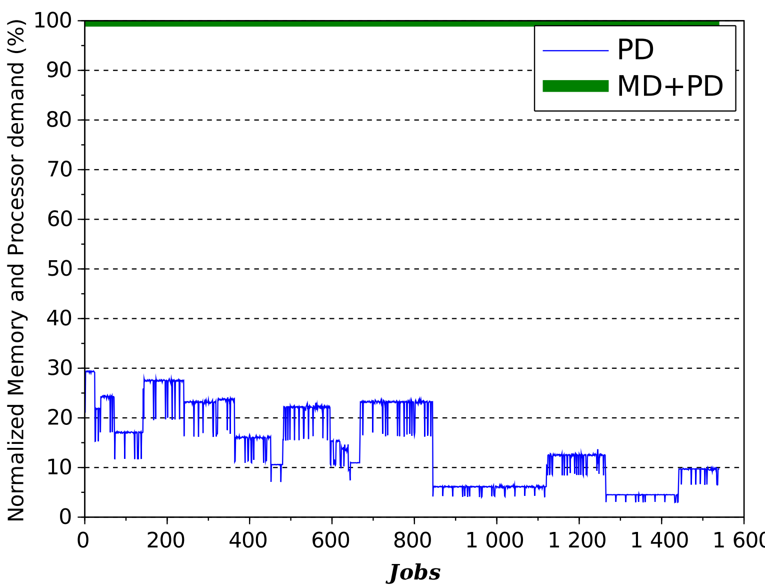


FIGURE 5.7: Comparison of the memory (MD) and processor (PD) demand for the approx. 1500 jobs of the simulation runs with 24 tasks.

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.

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 [12, 38], 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 sets hyper-period (up to 1200ms) – 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

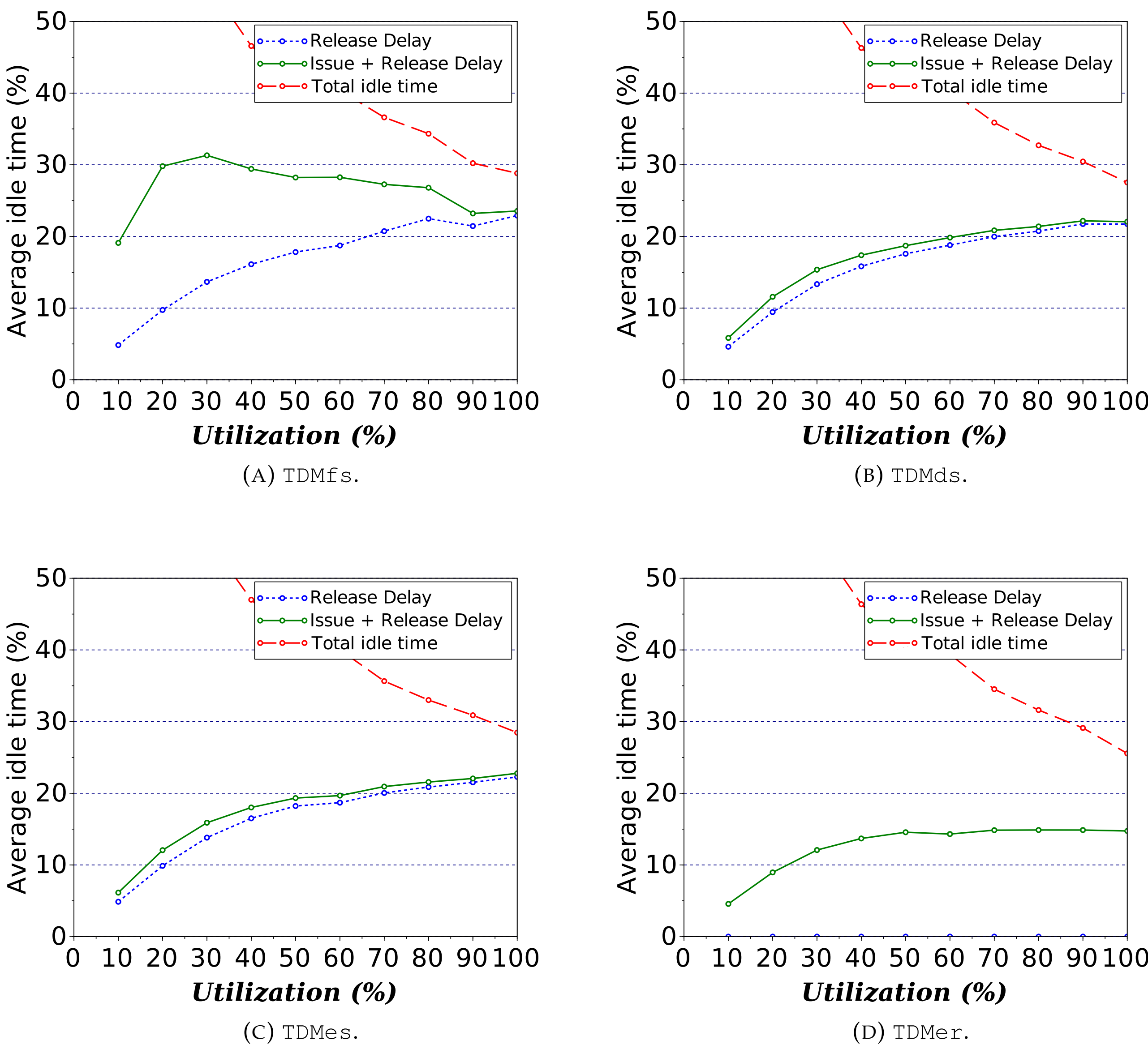
at 100 Mhz was implemented in the FPGA and performed memory accesses in isolation via the multi-port memory controller provided by the SoC (the remaining components of the SoC were deactivated). At any moment a single memory access was in-flight during these measurements. Depending on the internal state 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.

5.5.2 Results for Dynamic TDM-Based Arbitration Schemes

The first set of figures represents an overall comparison of the various `TDM`-based approaches over all simulation runs. Figure 5.8 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"). 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. We choose to trim Figure 5.8 in order to achieve a better visualization of the impact of our arbitration policies on the issue and release delays.

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. This was expected, as seen in the example shown in Figure 5.5. 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



(A) TDMfs.

(B) TDMds.

(C) TDMes.

(D) TDMer.

FIGURE 5.8: Evolution of memory idle time, issue, and release delays over all simulations under varying utilization (lower is better).

immediate next TDM slot. It appears that this can be explained due to the absence of slack at the beginning of jobs (see Section 5.5.3).

Figure 5.9 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

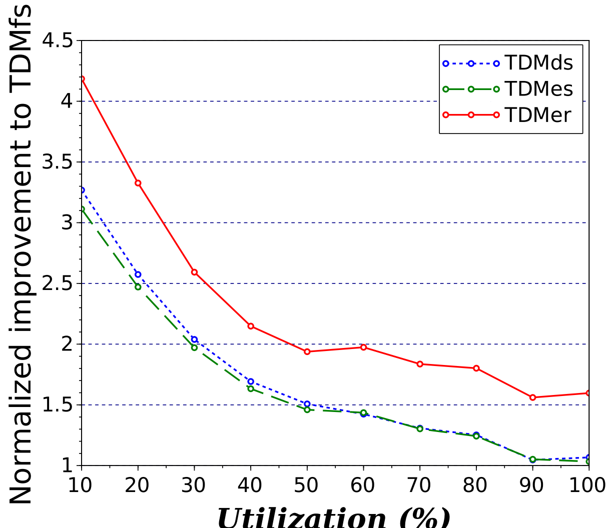
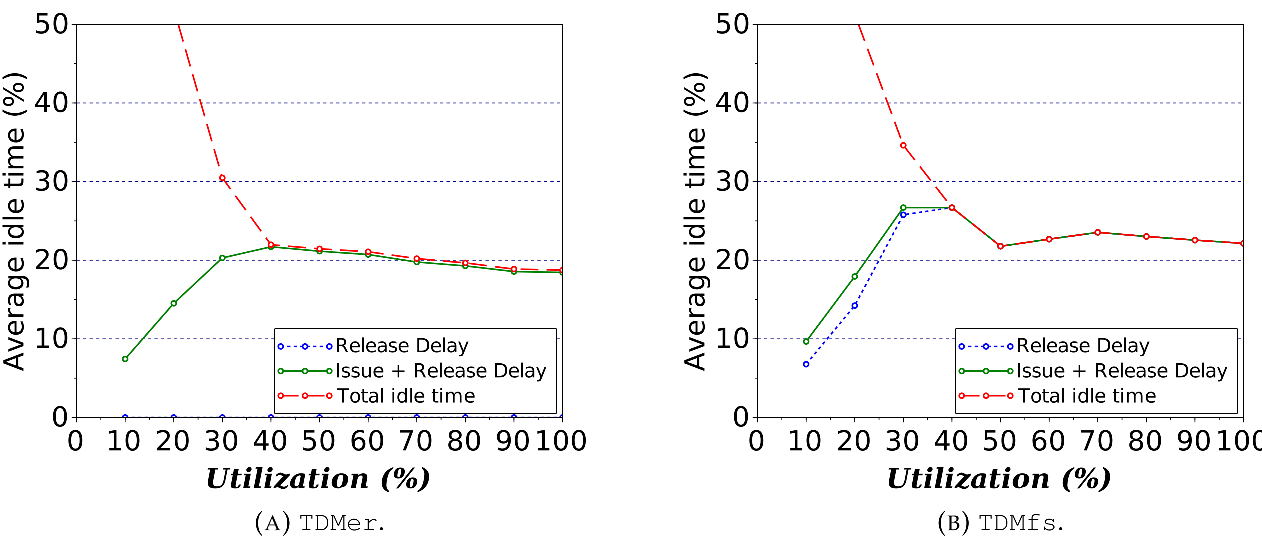


FIGURE 5.9: Normalized sum of issue and release delays for dynamic arbitration schemes compared to `TDMfs` (higher is better).

trend. However, under a 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%) 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 5.8d 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



(A) TDMer.

(B) TDMfs.

FIGURE 5.10: Memory idle time considering 24 tasks with 6 critical and 18 non-critical tasks (lower is better).

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 5.10 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.

5.5.3 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 5.8, a non-negligible portion of these release delays are merely transformed into issue delays (see Subsection 5.3.2). This is particularly visible for the configuration with 24 tasks (Figure 5.10). 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 counters

values of critical tasks are often too low to apply the early-start optimization (cf. Algorithm 2). 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. 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 5.5.1. 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 regular `TDM`. This additional *virtual* memory accesses can then be taken into consideration for the correctness proof from Section 5.4, e.g., by adapting the base case accordingly.

Figure 5.11 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

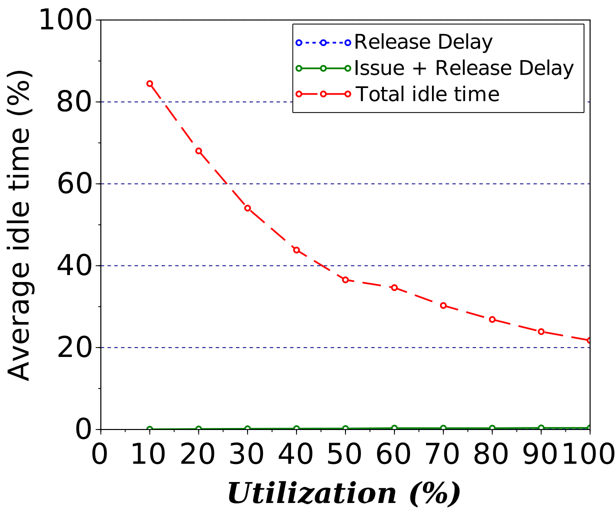


FIGURE 5.11: Evolution of memory idle time, issue, and release delays over all simulations with varying utilization under `TDMer` with slack counters initialized to Sl at job start (lower is better).

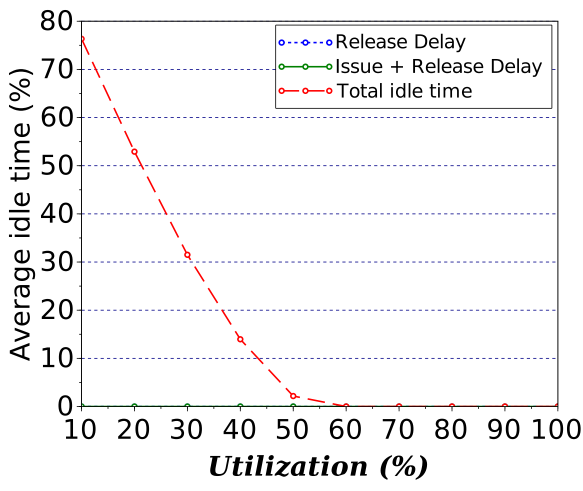


FIGURE 5.12: Memory idle time considering 24 tasks with 6 critical and 18 non-critical tasks, under `TDMer` with initial slack set to *Sl* (lower is better).

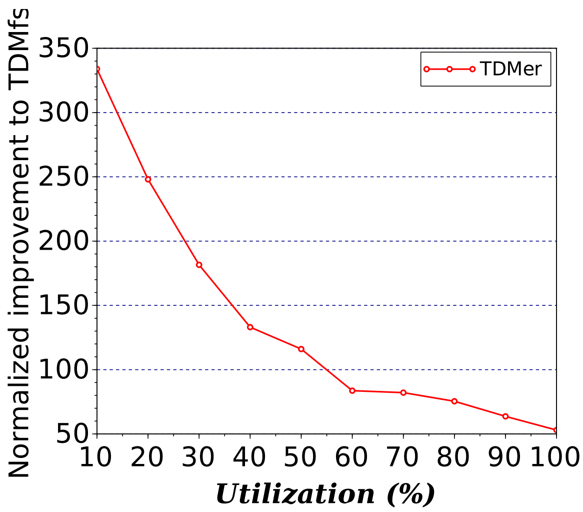


FIGURE 5.13: Normalized sum of issue and release delays for dynamic arbitration schemes compared to `TDMfs`, with initial slack set to *Sl* (higher is better).

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 5.12. In particular, we observe that the arbitration policy no longer limits memory 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 5.13 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` with initial slack is successful in eliminating release delays and significantly reducing issue delays, even when the memory bandwidth is close

to saturation, as shown in Figure 5.9. 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.

5.5.4 Results for Varying Memory Access Latencies

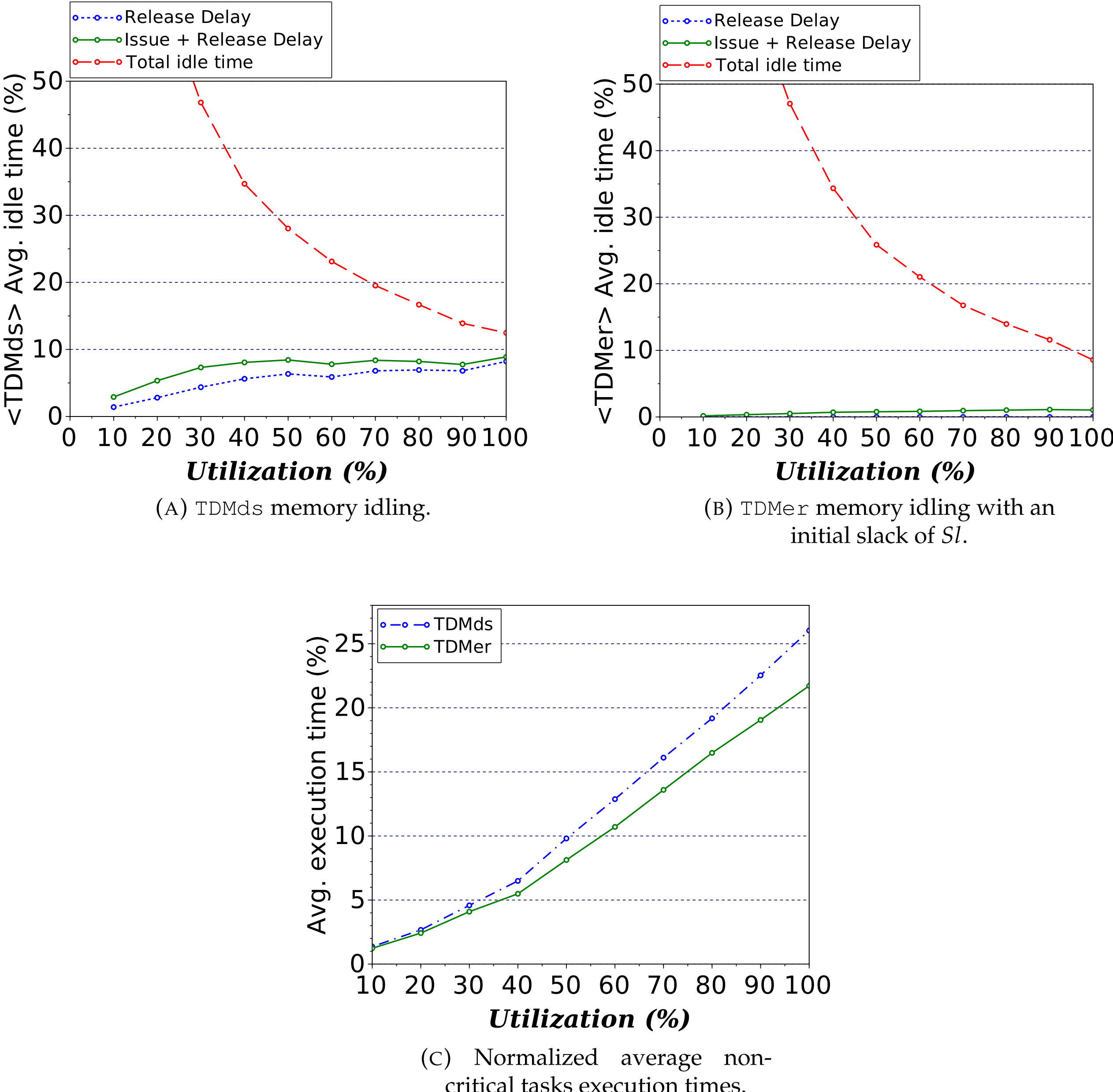
In the previous experiments we assumed a fixed `TDM` slot length of 40 cycles, corresponding to the memory access latency of a DDR3 memory. In Subsection 3.1.4, we showed that DDR DRAMs, when trying to provide predictable memory behavior, suffer from highly variable access latencies and overly pessimistic latency bounds. Alternative solutions targeting real-time systems are also described Subsection 3.1.4, 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, and fixing the minimum memory access latency at 21 cycles. Therefore, a longer slot length means more variability in memory accesses. Varying the `TDM` slot length impacts the generated memory profiles, as described in Subsection 5.5.1, 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 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 5.5.1). 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 5.14 and 5.15 summarize the obtained results from these simulation runs. Subfigures 5.14a and 5.15a 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



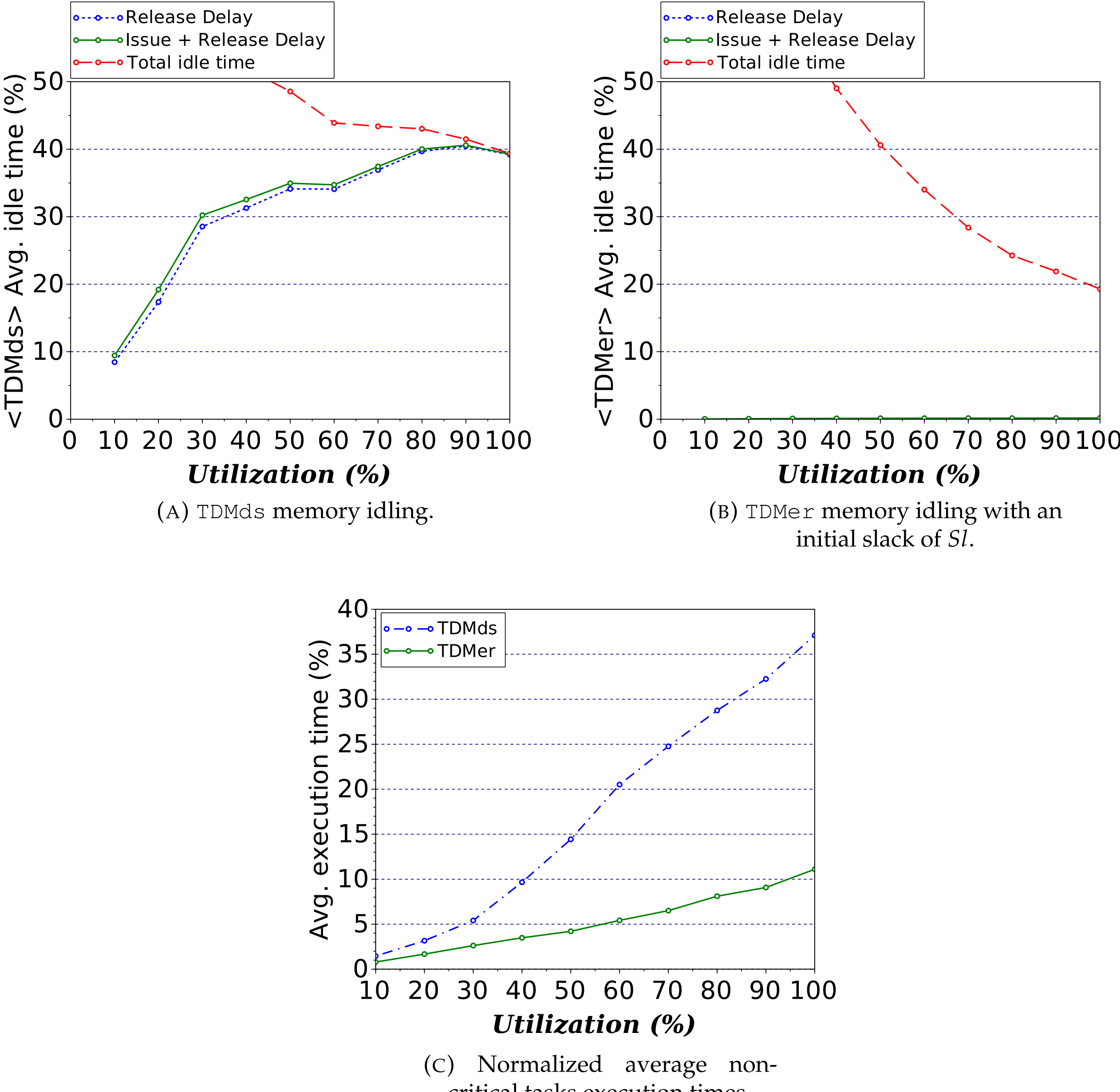
(A) TDMds memory idling.

(B) TDMer memory idling with an initial slack of Sl.

(C) Normalized average non-critical tasks execution times.

FIGURE 5.14: Results considering a TDM slot length of $Sl = 25$ cycles (lower is better).

overhead. The `TDMer` approach, with initial slack, also in this configuration successfully eliminates these delays, as depicted in Subfigure 5.14b – 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 5.15a). Subfigure 5.15b, shows



(A) `TDMds` memory idling.

(B) `TDMer` memory idling with an initial slack of Sl.

(C) Normalized average non-critical tasks execution times.

FIGURE 5.15: Results considering a `TDM` slot length of $Sl = 100$ cycles (lower is better).

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. The results are considerably worse for TDMfs than those for TDMds– the combined issue and release delays under TDMfs 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 $Sl = 100$ and $Sl = 25$ respectively.

Subfigure 5.14c shows a breakdown of the normalized non-critical tasks execution times w.r.t. the total trace length for TDMds, 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 5.15c), 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 high variability 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.

5.6 Conclusion

This chapter presents dynamic TDM-based arbitration schemes of a multi-criticality system, where each core execute a single task which can be critical or non-critical. We first presented a criticality aware arbitration TDMfs taking memory requests arbitration decisions with regard to tasks criticality, sharing the memory resource between critical and non-critical tasks. Thereafter a deadline driven arbitration with slack accumulation TDMds was introduced. This approach aims to address the challenge of the TDM schedule (point (**1**) of Subsection 4.1.2), one of the sources of TDM non-work-conserving. TDMds allow a more flexible and dynamic memory arbitration by re-allocating over a time interval (linked to the request deadlines) the empty slots. However, issues stemming from the very nature of TDM remain, related to the use of fixed TDM slots as the request arbitration only occurs at the beginning of slots and the TDM slot length pessimism as described in

point (**2**) of Subsection 4.1.2. The *early-start* optimization `TDMes` resolves the limitation regarding the fixed slot arbitration on regular `TDM` namely *issue delay*, instead of taking decision at the granularity of slots our approach now operates at the granularity of clock cycles by exploiting slack time accumulated from preceding requests. The other limitation, regarding the `TDM` slot length pessimism and the high variability of memory access latencies (cf. point (**3**) of Subsection 4.1.2), was addressed using the `TDMer` arbitration scheme. `TDMer` combines all the previously presented arbitration properties in addition to eliminating the *release delay* by shedding the artificial constraint of waiting the entire duration of a `TDM` slot before *releasing* the memory and allowing the next request to be processed. 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. Our evaluation reveals considerable gains, in particular, when approaching high system utilization. Our dynamic approach, in addition to eliminating the main limitations of `TDM` and therefore improving the memory utilization, converges towards regular `TDM` in the worst-case for critical tasks (see Section 5.4). Consequently, `TDMer` supports the same worst-case execution time analysis techniques as of regular `TDM`. After validating our approach that addresses the different limitations of `TDM`, the next step is to implement the hardware version and determine the hardware cost that such dynamic approach has.

Chapter 6

Dynamic TDM-based Arbitration Hardware Design

Contents

6.1 Architecture Overview . . . 80
6.2 Arbitration Logic . . . 81
6.2.1 Deadline and Slack Computation . . . 83
6.2.2 Update Rules . . . 84
6.3 Control Signal Generation . . . 85
6.4 Bit-Width Considerations . . . 86
6.5 Worst-case Behavior w.r.t. the Hardware Design . . . 87
6.6 Experiments . . . 93
6.6.1 Evaluation Platform . . . 93
6.6.2 Results for Hardware Synthesis . . . 94
6.6.3 Results for Dynamic TDM with Round-Robin Arbitration . . . 96
6.6.4 Results for Bit-Width Constrained Slack Counters . . . 98
6.7 Conclusion . . . 99

In this chapter, we discuss means to implement a variant of the dynamic TDM-based arbitration scheme, described in Chapter 5, in hardware. Such an implementation faces four main challenges. Firstly, the use of a priority queue to implement the EDF policy for critical requests is expected to 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 of `TDMer` 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.

We first provide an overview of the interactions among these components in Section 6.1. Subsequently, in Section 6.2 and Section 6.3, we discuss relevant components in more detail – along with simple and efficient solutions addressing the aforementioned challenges. A formal analysis with regard to the worst-case behavior is demonstrated in Section 6.5. Section 5.5 discuss the evaluation of the hardware design regarding hardware costs results, along with a comparison of `TDMer` and the hardware variant using various metrics. Finally, we conclude the chapter in Section 5.6.

6.1 Architecture Overview

Figure 6.1 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

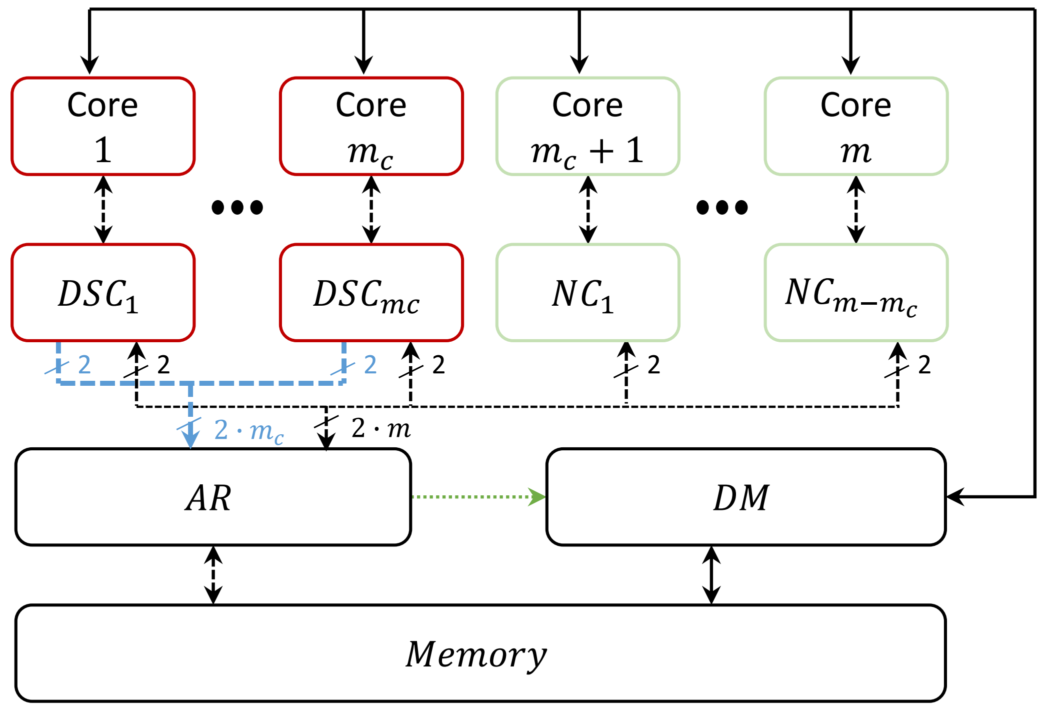


FIGURE 6.1: Overview of the hardware design of our dynamic TDM-based arbiter.

communicate deadline (PM_i) and slack (ES_i) information of the respective critical core. 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.

6.2 Arbitration Logic

TDMer relies on a 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 5.4, which only refers to EDF in Definition 5.4.3 and Theorem 5.4.2.

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 2), 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 5.3.1), 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 5.2, 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 6.6. 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 6.5. In the next section, we thus explain how these control signals are derived by the DSC components.

6.2.1 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 5.4, the deadline computation is related to the start date of the current TDM period Sp and the offset of the task's TDM slot $O(\tau_i^c)$ (w.r.t Sp). Based on this observation, we use a dedicated counter register Dl_i to model the current deadline of a request from a critical task τ_i^c. 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

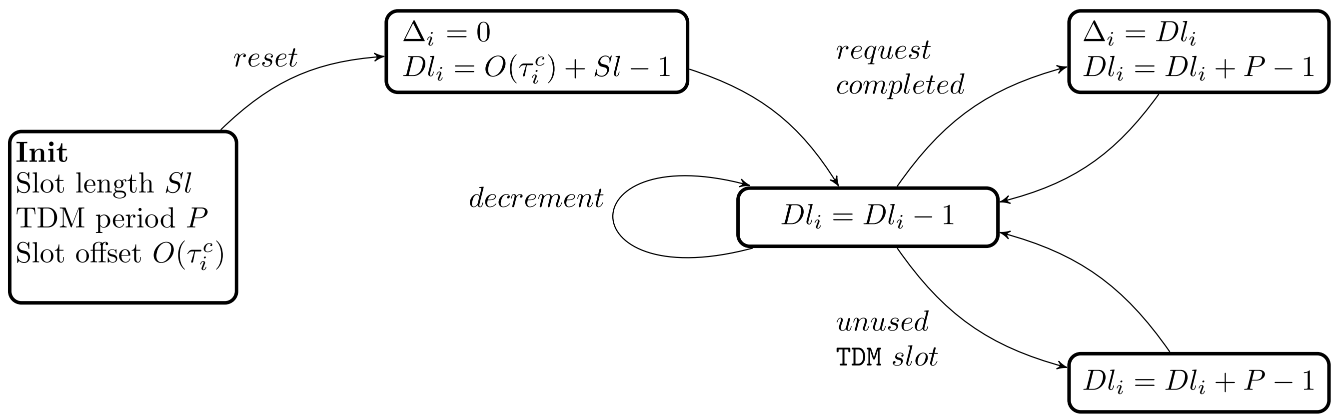


FIGURE 6.2: Summary of the update rules of the Deadline and Slack Computation (DSC) components.

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 6.2. The first rule is applied on a system reset (left). The second rule updates Dl_i when a task's slot is *unused* under regular 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.

6.2.2 Update Rules

System Reset: During a system reset, each DSC_i module initializes its deadline counter for its respective critical task τ_i^c depending on its offset $O(\tau_i^c)$ within the TDM schedule, as illustrated in Figure 6.2 (left). At this moment, obviously no request from τ_i^c 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^c 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 regular TDM. When the beginning of this TDM slot under regular TDM has passed and no request from

τ_i^c 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^c in the future. Such a deadline is one TDM period ahead, i.e., equals to $Dl_i + P - 1$, as illustrated by Figure 6.2 (bottom right). The slack counter does not change during this update.

Under regular TDM an *unused* slot can be detected by comparing the beginning of the τ_i^c's next slot with the *delayed issue date*, i.e., the current instant plus the slack counter Δ_i (see Subsection 5.2.2). In our hardware implementation these absolute dates are represented as relative dates w.r.t. the current instant. The beginning of τ_i^c'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^c 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^c 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 regular TDM, the slot associated with the current deadline would have been used. The next possible deadline for any future request issued by τ_i^c falls into the next TDM period and is simply given by $Dl_i + P - 1$. Both updates are illustrated by Figure 6.2 (top right).

6.3 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^c 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^c'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^c 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 2.

A similar argument can be made for the PM_i signal at the beginning of a TDM slot owned by a task τ_i^c. 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^c, 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.

6.4 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^c'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.

6.5 Worst-case Behavior w.r.t. the Hardware Design

TDMrr brings some modifications to our dynamic schemes presented in Chapter 5, 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 5.4 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 6.5.1. *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 6.5.1. *Under* `TDMrr` *the relative deadlines* Dl_i *of a critical task* 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 6.2.2. 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 6.5.1. *At any time instant t only one of the $PM_i^{@t}$ signals of all critical tasks $\tau_i \in \Gamma$ can be asserted, i.e., $PM_i^{@t} = 1 \wedge PM_j^{@t} = 1 \Longrightarrow i = j$.*

Proof. This follows immediately from Lemma 6.5.1 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 6.5.2. *The absolute deadline under `TDMrr` $d_k^{\mathtt{TDMrr}}$ 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 6.2.2). Other update rules are not possible up to the completion of the request. □

Lemma 6.5.2. *Given an absolute deadline $d_k^{\mathtt{TDMrr}}$ of a critical request r_k with an arrival date $a_k^{\mathtt{TDMrr}}$, `TDMrr` guarantees that the request completes at or before this deadline: $a_k^{\mathtt{TDMrr}} < c_k^{\mathtt{TDMrr}} \leq d_k^{\mathtt{TDMrr}}$.*

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

The absolute deadline $d_k^{\mathtt{TDMrr}}$ of the request has to be equal to the absolute deadline of task τ_i at the moment of the request's arrival, which follows from Corollary 6.5.2:

$$d_k^{\mathtt{TDMrr}} = d_i^{@a_k} = O(\tau_i) + Sl - 1 + x_k \cdot P, \text{ where } x_k \in \mathbb{N}^0$$

Therefore, $c_k^{\mathtt{TDMrr}} > O(\tau_i) + Sl - 1 + x_k \cdot P$, 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 6.5.1):

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

Consequently PM_i had to be asserted at this instant (cf. Subsection 6.3). 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 6.5.1 and Corollary 6.5.1 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 6.5.3. *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 $d_0^{\mathtt{TDM}} = O(\tau_i) + Sl - 1$. Under TDMrr the situation is more complex, due to the constant updates of the Dl_i register. However, the absolute deadline (cf. Definition 6.5.1) has to be valid, for all time instants t while the request is pending at the arbiter (see Lemma 6.5.2):

$$\forall t \in \mathbb{N}^0, a_0^{\mathtt{TDM}} \leq t < c_0^{\mathtt{TDMrr}} \leq d_0^{\mathtt{TDMrr}} : d_0^{\mathtt{TDMrr}} = d_i^{@t}$$

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 6.2.2). The value $d_i^{@t}$ does not change while a request is pending (see Corollary 6.5.2), we thus have to show that the request completes in time, i.e., $c_0^{\mathtt{TDMrr}} \leq d_0^{\mathtt{TDMrr}}$, which follows from Lemma 6.5.2.

Case (2): The request misses the first TDM slot ($a_0 > O(\tau_i)$):
The deadline under regular TDM is $d_0^{\mathtt{TDM}} = O(\tau_i) + x_0 \cdot P + (Sl - 1)$, 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 $a_0^{\mathtt{TDM}} > d_0^{\mathtt{TDM}} - P + Sl - 1$, we can derive the number of missed TDM slots as follows:

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

Under regular TDM and TDMrr, the first request arrival date is the same for both, i.e., $a_0^{\mathtt{TDM}} = a_0^{\mathtt{TDMrr}}$. 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 6.2.2) 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 $d_0^{\mathtt{TDM}} - P - (Sl - 1)$.
Therefore, at $t = d_0^{\mathtt{TDM}} - P - (Sl - 2)$ 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:

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

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

until the request's completion (see Corollary 6.5.2).

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 $Dl_i^{@c_0^{\mathtt{TDMrr}}}$, i.e., $\Delta_i = Dl_i^{@c_0^{\mathtt{TDMrr}}}$.

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

Case (1): Under TDM, request r_k arrives before the next TDM slot ($a_k^{\mathtt{TDM}} < P - (Sl-1)$):

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., $d_k^{\mathtt{TDM}} = d_{k-1}^{\mathtt{TDM}} + P$ and $dist_k = a_k^{\mathtt{TDM}} - d_{k-1}^{\mathtt{TDM}} < P$.

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

It remains to show that this deadline does not change up to the request's arrival at $a_k^{\mathtt{TDMrr}}$. 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 $Dl_i^{@c_{k-1}^{\mathtt{TDMrr}}} \geq P$.

Case (2): Under TDM, request r_k arrives after the next TDM slot ($a_k^{\mathtt{TDM}} \geq P - (Sl-1)$):

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, resulting in:

$$d_k^{\mathtt{TDM}} = d_{k-1}^{\mathtt{TDM}} + x_k \cdot P, \text{ where } x_k = \left\lfloor \frac{dist_k + Sl - 2}{P} \right\rfloor + 1$$

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 6.5.1. *(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 under TDMrr completes no later than the same execution under strict TDM.*

Proof. The correctness of Theorem 6.5.1 is ensured by by Lemma 6.5.2 by guaranteeing that request completes at or before its deadline, i.e., $c_k^{\mathtt{TDMrr}} \leq d_k^{\mathtt{TDMrr}}$. Lemma 6.5.3 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 6.5.3, 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.

6.6 Experiments

In this section we evaluate the hardware design of the arbitration logic described in Section 6.1. 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.

6.6.1 Evaluation Platform

The hardware design described in Section 6.1 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 [60]. 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 5.5, as input, which allows an automatic verification against the cycle-accurate software simulator that was already used in the other experiments.

6.6.2 Results for Hardware Synthesis

After careful validation, we instantiated and synthesized several different versions of our hardware design in order to find out the hardware cost of our approach `TDMrr`. 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 6.1 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 6.6.4 for a justification). The table also indicates the maximum operating frequency. Note that the reported numbers only concern 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, the design occupies approximately 10 500 ALMs. The unoptimized 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.

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

TABLE 6.1: Implementation details of our hardware design on an Intel Cyclone V SE SoC-FPGA considering 24-bit deadline and slack counters.

Cores		ALMs		Registers		Frequency (MHz)
m	m_c	Number (%)	Per m_c	Number (%)	Per m_c	
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	1 424 (3.40%)	89	814 (0.97%)	51	98.18

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 from 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 by when normalizing 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 6.2 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 6.1 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

TABLE 6.2: 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	1 228	686	96.08

number of cores the critical path is more impacted by the computation of the PM_i and ES_i signals at 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 is 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.

6.6.3 Results for Dynamic TDM with Round-Robin Arbitration

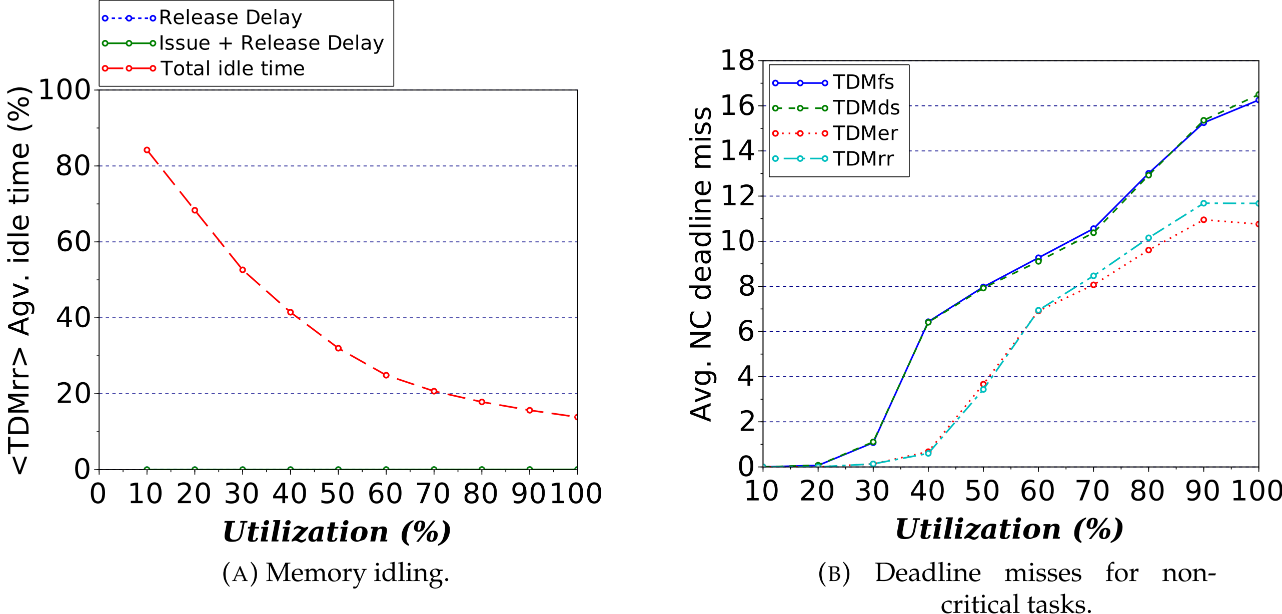
The designed variant of the dynamic `TDM`-based arbitration policy `TDMrr`, applies a different deadline and slack computation strategy. More importantly, instead of using an EDF arbitration policy with priority queues, `TDMrr` uses a Round-Robin arbitration policy over *all* pending memory requests, 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 5.5.

Figure 6.3a shows a breakdown of the average memory idle time for `TDMrr` over all simulation runs. Recall that, as for Figure 5.11 from Section 5.5.3, the plot shows stack 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, the performance of `TDMrr` appears to be slightly better than that of `TDMer` starting with a system utilization of about 50%, when comparing Figure 6.3a and Figure 5.11. `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`.

This, however, comes at a price. Figure 6.3b depicts the average number of deadline misses of non-critical tasks. Due to the aforementioned difference between `TDMer` and `TDMrr`, we can notice a difference in terms of deadline miss between `TDMer` and `TDMrr`. This trend appears to coincide with the trend regarding the memory idling from Figure 6.3a. 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



(A) Memory idling.

(B) Deadline misses for non-critical tasks.

FIGURE 6.3: 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).

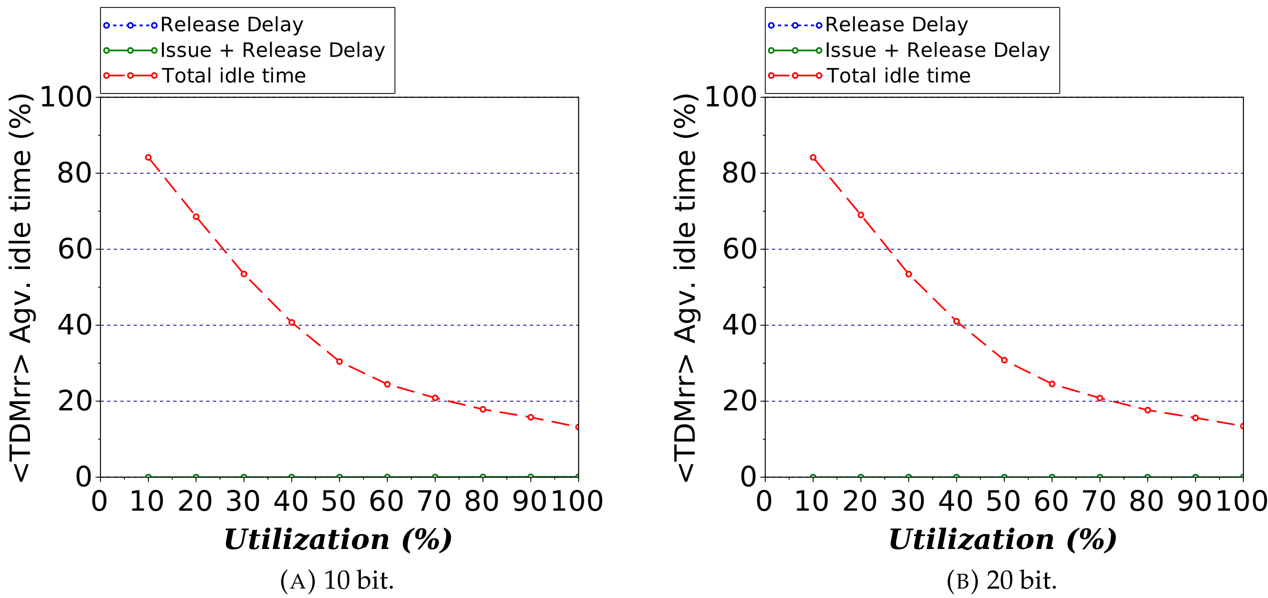
the ratio between critical and non-critical tasks (50%/50% vs. 25%/75%, see Subsection 5.5.1). Configurations with at most 4 cores do not cause sufficient memory load, while configurations with more than 16 cores quickly cause overload. The arbitration policy in these situations makes little difference. 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 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.

6.6.4 Results for Bit-Width Constrained Slack Counters

Section 6.4 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 consider 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 widths above 24 bits thus cannot impact the arbiter's performance negatively. 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 5.5.1).



(A) 10 bit.

(B) 20 bit.

FIGURE 6.4: Evolution of memory idle time considering reduced bit-widths for the slack and deadline counters under TDMrr with initial slack (lower is better).

Figure 6.4 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.

6.7 Conclusion

In this chapter, our aim was to adopt a simple and low-cost approach in terms of hardware resources. To do so, we had to make some modifications to the TDMer version, the first being the replacement of the EDF priority queue by a less complex and yet efficient policy between pending requests using a Round-Robin approach. Our evaluations showed that the implementation

of dynamic TDM-based arbitration policy (TDMrr) is efficient, both in terms of hardware complexity and arbitration performance regarding the memory utilization. The second modification was related to the computation of deadlines and slack, rather than using absolute dates in our hardware approach we use relative dates. This allowed to reduce the number of bits needed to store these values in dedicated registers. As these registers were limited by their number of bits, it was necessary to guarantee that in the worst-case this would not have a negative impact on the correct arbitration of critical requests, a proof of the worst-case behavior is described in Section 6.5. Finally, using additional simulations we showed that the proposed solution combines the advantages of dynamic TDM-based scheduling with an efficient and simple hardware realization. The remaining challenge of this work is related to the task level impact of our dynamic TDM-based arbitration scheme. More precisely, we need to study the arbitration-induced preemption delays that are linked to our approach.

Chapter 7

Multi-tasks and Preemption Support

Contents

7.1 Multi-task Preemptive Scheduling Policy . . . 102
7.2 Preemption Costs for Dynamic TDM-based Arbitration . . . 104
7.3 Arbitration-Aware Preemption Techniques . . . 105
7.3.1 Scheduling with Request Waiting (SHDw) . . . 106
7.3.2 Scheduling with Request Preemption (SHDp) . . . 107
7.3.3 Scheduling with Criticality Inheritance (SHDi) . . . 108
7.3.4 Misalignment Delays . . . 110
7.3.5 Response-Time Analysis . . . 111
7.4 Experiments . . . 112
7.4.1 Experimental Setup . . . 112
7.4.2 Results for Preemption Schemes . . . 113
7.4.3 Results for (Preemptive) Arbitration Schemes . . . 116
7.4.4 Results for Varying Memory Access Latencies . . . 119
7.5 Conclusion . . . 121

To overcome TDM limitations presented in Subsection 4.1.2, we described in Chapter 5 novel dynamic TDM-based arbitration schemes. We envisioned that the task's criticality level should not only be used by the task scheduler, but also by the memory arbiter. The arbiter associates a deadline with each memory request of a critical task, which corresponds to the end of its corresponding slot under a strict TDM scheme. The deadline then allows to compute the *slack time* of a critical task, i.e., the amount of time that the task's last request completed before the request's deadline. This slack time then can be exploited by the arbiter, under certain conditions, to favor

requests of non-critical tasks over requests from critical tasks, to freely reorder memory requests, and to schedule memory requests at the granularity of clock cycles – instead of being confined to TDM slots. The resulting TDMer arbiter significantly reduces the memory idle time, compared to a regular TDM arbiter.

However, the proposed dynamic TDM-based arbitration techniques TDMds and TDMer face issues under a preemptive execution model. In this chapter, we address the impact of such sophisticated arbitration scheme on task level regarding the multi-task scheduling and preemption costs in particular. Therefore, we define two memory delays induced by preemptions, the *memory blocking delay* and the *misalignment delay*, which may lead to significant jitter and increase task response times. Even worse, due to non-critical tasks, the memory blocking delay may be unbounded in some circumstances. We explore three different approaches to analyze the impact of these arbitration-induced preemption delays considering preemptive [56] (SHDp) and non-preemptive [6] (SHDw) memory requests. Finally, we propose a new technique (SHDi) to resolve these issues by adapting (priority or rather) *criticality inheritance* known from scheduling theory. This allows us to manage and easily bound these preemption delays. Our evaluation shows that our new approach successfully limits the worst-case preemption delays experienced at runtime under our dynamic TDM-based arbitration schemes.

The remainder of this chapter is organized as follows. Section 7.1 describes the considered multi-task scheduling policy w.r.t. the criticality aware arbitration schemes. In Section 7.2 we identify the different delays caused by preemptions in dynamic TDM-based arbitration strategies, while Section 7.3 proposes three preemption models to handle these delays. In Section 7.4, we evaluate our contributions and demonstrate the improvements in terms of blocking delay reductions and success rates on randomly generated task sets. Finally, Section 7.5 concludes the chapter.

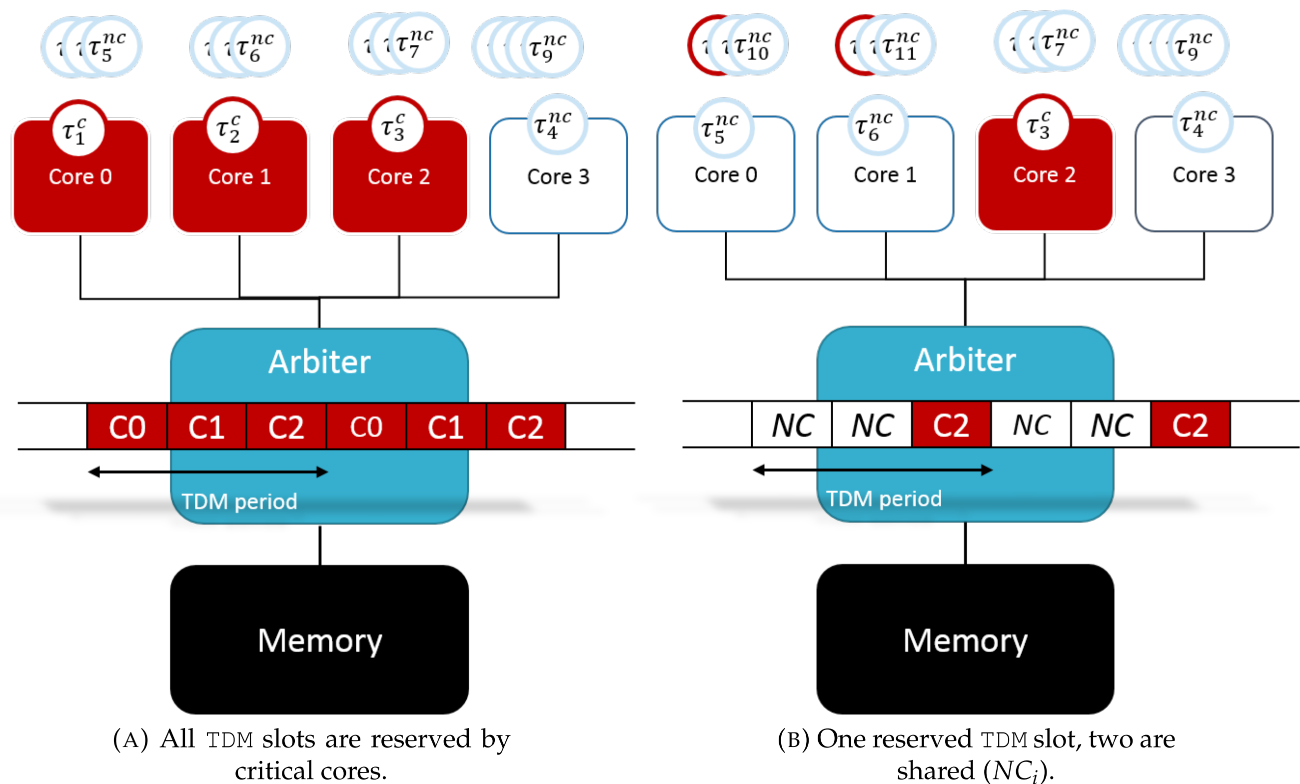
7.1 Multi-task Preemptive Scheduling Policy

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

This raises the question on how TDM slots are assigned among cores/tasks. We propose a pragmatic solution for this work, but other alternatives are obviously possible. Since each core that

executes at least one critical task may require a `TDM` slot at some moment, we assign one `TDM` slot to each such core. However, the slot is only reserved exclusively for that core when it actually executes a critical task, i.e., the core is considered critical. Cores that do not execute a critical task at a specific moment are themselves non-critical. The `TDM` slots that are not reserved, because their owning cores are currently executing non-critical tasks, are marked as NC_i and shared among all running tasks on all cores in the system. This strategy is illustrated by Figure 7.1, showing a system with 4 cores. The `TDM` schedule consists of three slots (C0, C1, and C2), since only three of the cores may execute critical tasks (red). In Subfigure 7.1a, these three cores execute critical tasks and are thus themselves critical. In Subfigure 7.1b, on the other hand, only a single critical task executes. Consequently two of the `TDM` slots are marked NC_i and shared by all running tasks.

Note, that non-critical tasks on core 3 may suffer from starvation on the depicted platform. For the formal analysis we require that at least one `TDM` slot is marked NC_i whenever a non-critical task executes in order to rule starvation out. We thus define a larger `TDM` period for non-critical tasks using an application-specific constant k (e.g., k represents the number of non-critical cores



(A) All `TDM` slots are reserved by critical cores.

(B) One reserved `TDM` slot, two are shared (NC_i).

FIGURE 7.1: Hardware architecture.

divided by the number of NC_i slots when applying FIFO arbitration among non-critical requests): $P^{nc} = k \cdot P$.

Also, recall that we assume that preemptions are pre-programmed through a timer-like hardware component (see section 2.3.2). Using these components, we can control under which conditions preemptions are actually triggered, e.g., to block the current preemption until an ongoing memory request has completed or to preempt a pending request.

7.2 Preemption Costs for Dynamic TDM-based Arbitration

We now investigate the issues raised by our dynamic TDM-based arbitration described in Chapter 5. We chose to not consider the TDMrr approach in this chapter as it is the result of a technical choice made to reduce the hardware costs of the design. Besides, someone else may choose to apply another implementation choices according to other criteria (e.g. use another policy than Round-Robin). The results obtained in Chapter 6 shows that the behavior with regard to the memory utilization is virtually the same for TDMer and TDMrr. These are the reasons that led us to consider only the approaches described in Chapter 5, i.e. TDMfs, TDMds and TDMer.
As a reminder, in this work, we only focus on preemption costs caused by the arbitration policy and ignore other costs due to the scheduler invocation, context switching, pipeline flushes, or Cache-Related Preemption Delays (CRPD). From here on, we use the term *preemption cost* to refer to the costs related to the *arbitration policy only*.

The dynamic TDM-based arbitration schemes described in Chapter 5 inherit the memory blocking and misalignment delays from regular TDM. Figure 7.2 shows 3 tasks executing on 2 cores that share slots C0 and C1 within a TDM period under the TDMds arbitration scheme. The mapping between tasks and cores is indicated by matching colors (yellow and orange). Critical tasks τ_0^c and τ_1^c are executed on the same core, which results in a preemption of task τ_0^c by τ_1^c ( ). Non-critical task τ_2^{nc} executes alone on the other core. A core has a dedicated TDM slot when it executes a critical task (τ_1^c, τ_0^c), while the slots of cores executing non-critical tasks (here only τ_2^{nc}) are *shared* by the running non-critical tasks (see the next section).

Lets assume that tasks cannot be preempted while performing a memory access [6]. Task τ_0^c then blocks the higher-priority task τ_1^c after its release in TDM slot 4. The preemption's blocking time, highlighted by the blue cross-hatched bar ( ), appears to be larger than the worst-case memory access latency under regular TDM. The latency of request $\tau_{0,2}^c$ amounts to two entire TDM

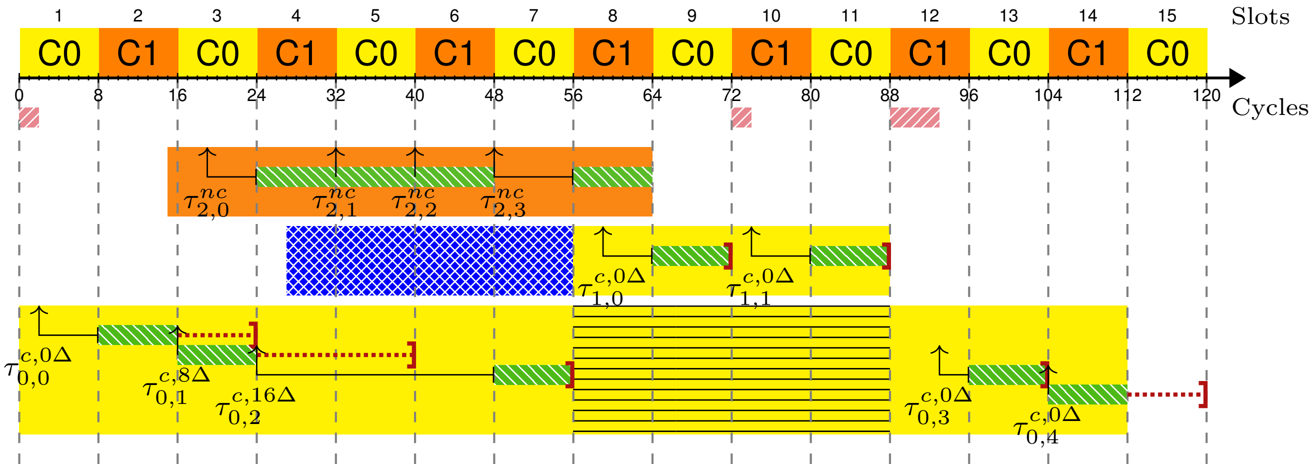


FIGURE 7.2: Impact of slack accumulation on the memory blocking time for `TDMds`.

periods and τ_1^c starts executing only after its completion. The reason for this long delay is the high slack counter value ($\Delta 16$) for request $\tau_{0,2}^c$, combined with the interference of the non-critical task τ_2^{nc} running on the other core. The non-critical task *consumes* all the slack of the critical request $\tau_{0,2}^c$, delaying its completion, and thus also delaying the preemption. The situation is potentially even worse when a non-critical task is preempted during a memory request. Recall that non-critical requests are executed in a best-effort manner. The blocking time caused by such a request may even be unbounded.

The example illustrates that preemption-induced delays need to be taken into consideration during system analysis. Note that the same reasoning (see Section 4.1.3) used for regular `TDM` to establish the misalignment delay bound applies for all dynamic `TDM`-based approaches considered here. Next, we investigate three different approaches to integrate the memory blocking delay due to preemptions for our dynamic `TDM`-based arbitration schemes by extending response-time analysis. In addition, we compare the approaches w.r.t. their implementation complexity and actual runtime behavior.

7.3 Arbitration-Aware Preemption Techniques

Let us assume that memory requests are never aborted – cf. cases (2) and (3) from Subfigure 4.3a. In this case, preemptions have to be delayed until a potentially ongoing request completes. Under regular `TDM` arbitration, this corresponds to the usual worst-case memory access

latency w.r.t. the TDM period P and TDM slot length Sl, which can be bounded by [6]:

$$\text{MB}^{\text{TDM}} = P + Sl - 1 \tag{7.1}$$

As demonstrated in Section 7.2, this bound is not valid for TDMds and TDMer, due to additional delays caused by the slack counters. While it certainly appears feasible to determine bounds on the slack counter values of tasks, it can be expected that these bounds would be rather pessimistic. In this section, we first explore this option to simply *wait* [6] due to its uncomplicated hardware implementation and later refer to it as SHDw (for *ScHeDuling wait*). Next, we explore an alternative option (SHDp, for *ScHeDuling preempt*), which assumes that ongoing requests can be *preempted* as long as they are not yet processed by the main memory [56] – cf. case (2) from Subfigure 4.3a. Finally, we propose a new solution (SHDi, for *ScHeDuling inheritance*) that tries to limit the impact of the slack counters by imposing a (new) deadline on an ongoing request when the core's timer signals a preemption by a critical task, i.e., the request of the preempted task *inherits* the criticality of the preempting task. Thereafter, we show how these models can be integrated into a response time analysis.

7.3.1 Scheduling with Request Waiting (SHDw)

This strategy simply waits that an on-going memory request finishes [6]. Compared to regular TDM, the memory blocking time can however be considerably larger due to the slack counters. Consequently, a timing analysis technique is required to allow bounding the maximum slack counter value of a critical task τ_i^c. A trivial bound Δ_i^{max} can be computed by multiplying a task's worst-case number of memory requests (M_i) with the maximum slack possibly accumulated per request: $M_i \cdot (P - Sl)$ (TDMds) or $M_i \cdot (P + Sl - 1 - l)$ (TDMer), where l indicates the minimum memory latency. The resulting bound appears highly pessimistic, but more precise bounds are out of the scope of this work.

To bound the memory blocking time of a critical task τ_i^c two cases need to be considered. Firstly, the blocking delay of preempting some lower-priority non-critical tasks (lpnc(i)) has to be considered via $\text{MB}_i^{nc,\text{SHDw}}$ – which is similar to Equation 7.1 for regular TDM. Secondly, the blocking delay of preempting another lower-priority critical task (lpc(i)) has to be considered via $\text{MB}_i^{c,\text{SHDw}}$. Here, the maximum slack counter values (Δ_l^{max}) over all lower-priority critical tasks

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

$$\mathrm{MB}_i^{c,\mathtt{SHDw}} = \begin{cases} 0 & \text{if } \mathrm{lpc}(i) = \emptyset, \\ P + Sl - 1 + \max\limits_{l \in \mathrm{lpc}(i)} \Delta_l^{max} & \text{else} \end{cases}$$
$$\mathrm{MB}_i^{nc,\mathtt{SHDw}} = \begin{cases} 0 & \text{if } \mathrm{lpnc}(i) = \emptyset, \\ P^{nc} + Sl - 1 & \text{else} \end{cases}$$
$$\mathrm{MB}_i^{\mathtt{SHDw}} = \max(\mathrm{MB}_i^{c,\mathtt{SHDw}}, \mathrm{MB}_i^{nc,\mathtt{SHDw}}) \tag{7.2}$$

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

7.3.2 Scheduling with Request Preemption (SHDp)

An alternative approach is to preempt ongoing memory requests. We consider that requests can only be preempted while pending at the arbiter, but not while being processed by the memory [56]. Consequently, preemptions are still delayed when a request is currently processed by the memory (cf. case 3 of Subfigure 4.3a). The memory blocking delay for critical and non-critical tasks then can trivially be bounded by the worst-case memory latency, which in turn is bounded by Sl:

$$\mathrm{MB}_i^{\mathtt{SHDp}} = Sl - 1 \tag{7.3}$$

However, the preempted task later has to reissue the memory request that was preempted, which causes additional delays that need to be accounted for in its response time. A trivial bound for this reissuing delay is the TDM period P, i.e., the maximum latency of a pending request after the preemption. Note that there is no need to consider the slack counter value, since it is already covered by the WCET. Furthermore, the misalignment delay always applies to the preempted/reissued request. The misalignment delay thus already covers this overhead (see Subsection 7.3.4).

Figure 7.3 shows an execution of the same task set introduced in Subsection 7.2 using TDMds and the SHDp preemption scheme. This time request $\tau_{0,2}^c$, waiting to access the memory, is immediately preempted by task τ_1^c and is reissued once τ_0^c resumes execution. The request $\tau_{0,2}^c$ thus

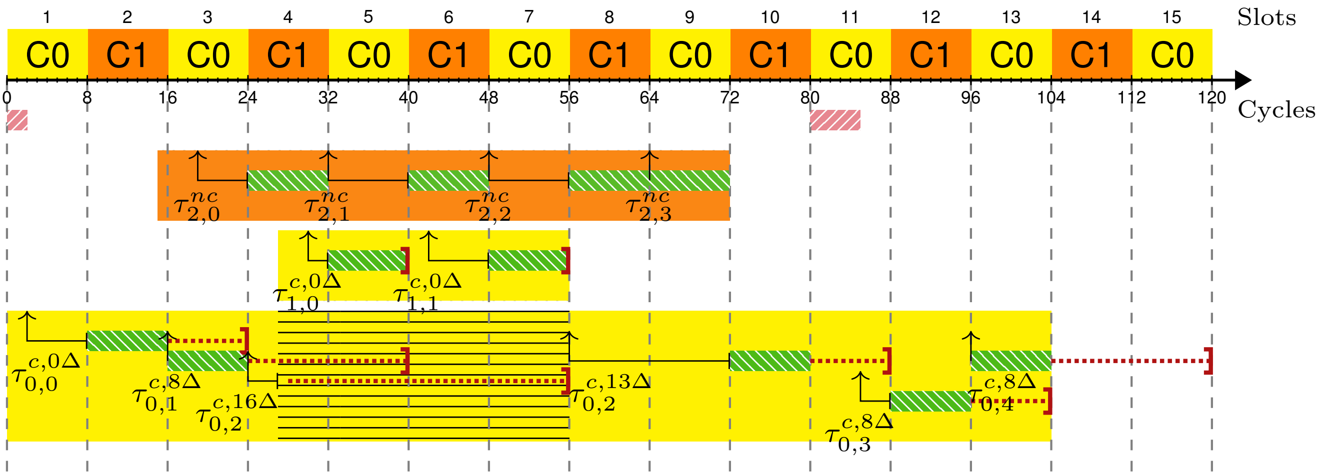


FIGURE 7.3: Memory blocking delays considering request preemption (SHDp).

appears twice in the figure, once before the preemption (aborted) at cycle 27 and once thereafter (reissued). Note that the slack counter, due to the request's preemption, diminished from 16Δ to 13Δ.

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

7.3.3 Scheduling with Criticality Inheritance (SHDi)

The aim of this approach is to provide means to control the impact of the slack counters and the interference from non-critical tasks on the memory blocking delay of preempting tasks. The dynamic TDM-based arbiters considered here are all based on the notion of *deadlines*. The idea of **criticality inheritance** (inspired from priority inheritance scheduling technique [61]) is to *impose* a new deadline on a pending request at the moment when a critical task is released, regardless of

the criticality of the preempted task. The preemption is still delayed – as before under the SHDw scheme. However, the blocking time is bounded by the newly imposed deadline.

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

At runtime two scenarios can be distinguished, depending on the criticality of the preempted task. Firstly, if a critical task is preempted it is clear that its pending critical request already carries a deadline. Replacing this deadline is easy, it suffices to signal to the memory arbiter that the recomputation of the deadline is needed using the current cycle, i.e., the release of the preempting task. Secondly, non-critical requests do not carry a deadline and may be held in a structure separated from critical requests (e.g., a FIFO for TDMer). The request thus needs to be taken out of this structure and reissued as a critical request with the appropriate deadline. Afterwards, the core has to reclaim its TDM slot, which up to now has been marked non-critical, i.e. NC_i. These operations only concern the arbiter and do not impact the processor pipeline or other parts of the memory hierarchy.

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

$$\mathrm{MB}_i^{\texttt{SHDids}} = P + Sl - 1 + t_{id} \tag{7.4}$$

$$\mathrm{MB}_i^{\texttt{SHDier}} = P + 2 \cdot Sl - 1 + t_{id} \tag{7.5}$$

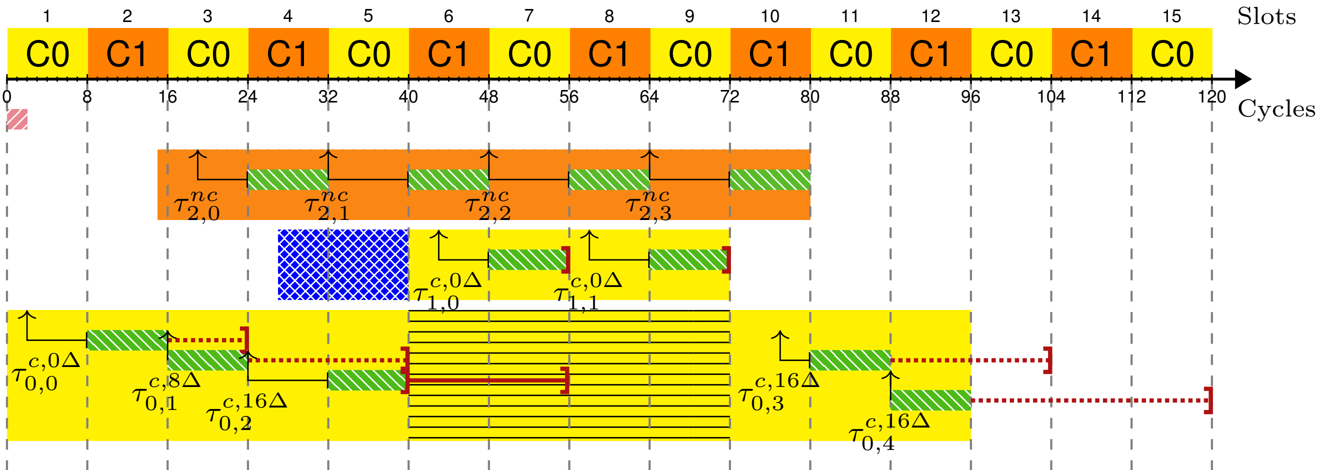


FIGURE 7.4: Memory blocking delays considering criticality inheritance (SHDi).

Note that the second *Sl* in Equation 7.5 stems from the non-zero slack counter, which ensures that all requests can access memory without clashes. We currently do not apply criticality inheritance when the preempting task is non-critical (deadlines could easily be imposed for preemptions by non-critical tasks). Non-critical tasks currently implicitly follow the SHDw strategy.

Figure 7.4 again shows an execution of the same task set as in the previous examples, this time under TDMds combined with SHDi. Critical task τ_1^c is released while critical request $\tau_{0,2}^c$ is pending. The pending request has its original deadline at the end of TDM slot 7 (⊣). Request $\tau_{0,2}^c$ is subject to a limited interference from only request $\tau_{2,0}^{nc}$ which completes before $\tau_{0,2}^c$ can be processed. The preemption imposes a new deadline at the end of the immediate next TDM slot (slot 5). The arbiter thus has to prevent the interference from the other core and has to assign the next slot to the preempted task – effectively bounding the memory blocking delay.

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

7.3.4 Misalignment Delays

Regular TDM as well as the dynamic TDM-based schemes all suffer from misalignment delays, highlighted in Section 4.1.3 and Section 7.2, for the same reasons. The delay appears when the task's misalignment *at the first* memory access after a preemption is larger w.r.t. the task's own TDM slot as determined by the WCET analysis [41, 58]. In the worst case the associated memory

request misses the task's TDM slot by a single cycle, i.e., the issue date (TDM) or delayed issue date (dynamic TDM) miss the slot by a cycle. The request consequently completes in the worst case in the next TDM period, resulting in the following bound for all the TDM-based approaches:

$$\mathrm{MA}^{\mathtt{TDM}} = \mathrm{MA}^{\mathtt{TDMds}} = \mathrm{MA}^{\mathtt{TDMer}} = P \tag{7.6}$$

The delay can be larger for non-critical tasks ($P^{nc} = k \cdot P$). Also note that the bound is smaller than the worst-case memory access latency of regular TDM ($P + Sl - 1$), since the memory wait time of the first TDM slot is accounted for in the WCET (highlighted in beige in Subfigure 4.3b).

7.3.5 Response-Time Analysis

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

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

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

$$R_i^{n+1} = (C_i + \mathrm{MB}_i) + \sum_{\forall j \in \mathrm{hp}(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil ((C_j + \mathrm{MB}_j) + \mathrm{MA}) \tag{7.7}$$

For the SHDw and SHDp schemes, independently of the TDM policy, but also for SHDi when combined with TDMds, the term MB_j can be safely removed from this equation. The latency of the ongoing memory request, owned by the preempted task, is indeed counted twice: once for the preempting task (MB_j) and the preempted task (C_i). This is illustrated by Figure 7.4, were the blue area (), representing the MB_j term for task τ_1^c, ends at the deadline of request $\tau_{0,2}^c$ and thus the C_i of τ_0^c. For TDMer under SHDi, the imposed deadline may be greater than the original deadline of the ongoing request, since the imposed deadline is recomputed with a slack counter

of Sl. The term MB_j is thus pessimistic and a possible refinement is to subtract the minimal memory latency l from this term.

7.4 Experiments

We use simulation to evaluate the runtime behavior of the various preemption mechanisms. In order to determine the impact of the preemption mechanism on the memory blocking delay and, therefore, on the memory utilization w.r.t. the used arbitration scheme. Besides, we also evaluate the tasks' schedulability according to the used arbitration scheme. Before discussing the results, we provide details on the experimental setup.

7.4.1 Experimental Setup

We developed a simulation engine that is able to simulate a dynamic execution trace of an entire multi-core platform. Traces are specified according to the system/hardware model from Section 2.3.1 and Section 7.3. The engine includes a fixed-priority preemptive task scheduler, and various memory arbitration schemes (`TDMfs`, `TDMds`, `TDMer`), which can be combined with the aforementioned preemption mechanisms (`SHDw`, `SHDp`, `SHDi`).

The engine can be configured in terms of the number of (non-)critical cores, `TDM` slots in a period (at least 1 per critical core), and (non-)critical tasks. The task scheduler respects user-defined core affinities that can be assigned freely to tasks. However, for our experiments each task is assigned to a single fixed core only (partitioned scheduling). Tasks are represented by a sequence of jobs, which, in turn, represent dynamic execution traces consisting of memory accesses that are separated by a fixed amount of computation time (cf. Subsection 2.3.1). This allows to simulate the same execution trace of a task set using different platform configurations and compare the results. Recall that the framework does not model the actual computations, only the *time* needed for computations. The same method is used for the task sets generation as in Section 5.5.1. The tasks are therefore bases on the *UUniFast* algorithm [25] allowing to randomly generate a task set. In addition to a *traffic generator* providing synthetic memory access patterns representing *cache misses* of some random dynamic program execution. Regarding the `TDM` schedule, the simulation runs consider a similar `TDM` slot length of 40 cycles as in Section 5.5.1. For `TDMer`, which is able to exploit memory requests completing faster than the `TDM` slot length, we simulate a varying latencies using a uniform random distribution in the range $[21, 40]$ clock cycles.

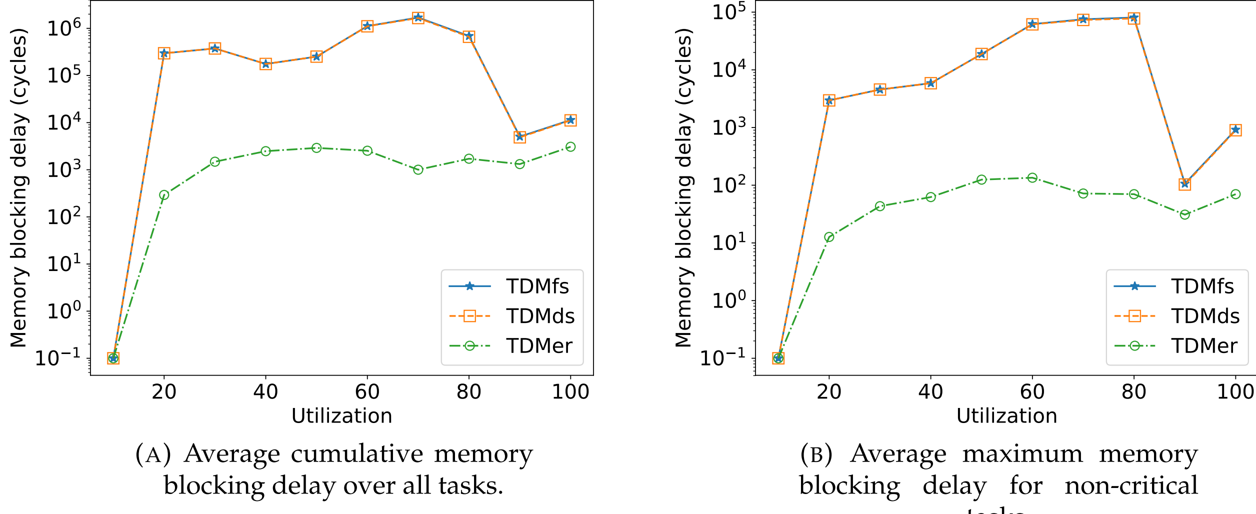
Simulation Setup

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

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

7.4.2 Results for Preemption Schemes

We start by analyzing the memory blocking delays on the simulated task sets with the three preemption mechanisms considered. Subfigure 7.5a shows the cumulative average memory blocking delay over an entire simulation run for the SHDw preemption mechanism considering the TDMfs, TDMds, and TDMer arbitration policies. Each point represents an average value over all schedulable task sets at the corresponding normalized utilization. As expected, the cumulative overhead can become very large going up to $1.6 \cdot 10^6$ cycles for both TDMfs and TDMds



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

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

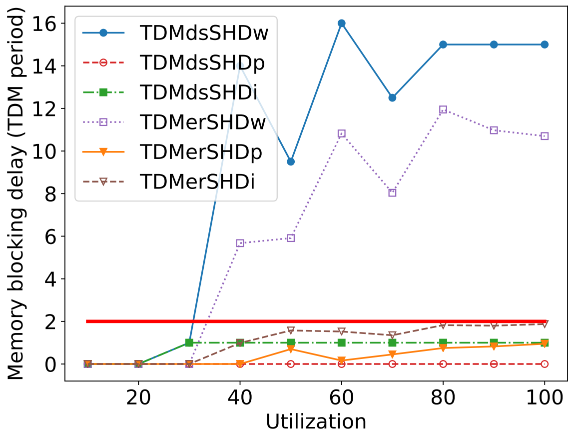
FIGURE 7.5: Memory blocking delays across normalized system utilization for SHDw.

(showing virtually identical results), while only reaching $3 \cdot 10^3$ cycles for `TDMer`– despite the fact that preemptions are rare events. On average, there are 7 preemptions per run, with a maximum number of 288 preemptions in rare cases. We observed a task set executing on 16 cores experiencing **502**ms (an average of **31**ms per core) of blocking delays within **590**ms of total execution time (assuming a clock speed of 100 Mhz). The other approaches, `SHDp` and `SHDi`, have significantly lower overhead compared to `SHDw`.

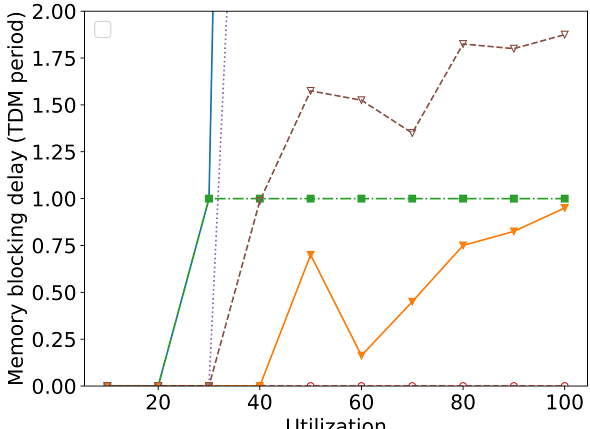
Since non-critical tasks execute in a best-effort manner, their memory blocking delays are expected to potentially become quite large, in particular, when preempting another non-critical task. Subfigure 7.5b thus highlights the average maximum memory blocking delay experienced by non-critical tasks across utilization levels for `SHDw` in combination with either `TDMfs`, `TDMds`, or `TDMer`. As can be seen, individual preemptions consistently take thousands of cycles, which corresponds to about **18**ms (maximum observed memory blocking delay). As these events are still rare, some volatility in the simulations is visible through the large drop at 90% utilization. Note that typically non-critical tasks represent the majority of the computation and memory load of the generated task sets. Consequently, non-critical tasks experience most of the preemptions as well as the associated memory blocking delays.

Finally, Subfigure 7.6a shows the maximum memory blocking delays experienced by critical tasks considering all three preemption mechanisms in combination with `TDMds` and `TDMer`. The delays are normalized w.r.t. the `TDM` period in order to avoid penalizing simulation configurations with shorter periods. These delays are representative of the upper bounds defined in Subsection 7.3. Note that the results for `TDMfs` are virtually identical to those for `TDMds` and are thus not shown.

As expected, **the `SHDp` scheme presents very low memory blocking delays**, as can be seen in more detail in Subfigure 7.6b. Under `TDMds` this preemption mechanism leads to no noticeable memory blocking. This is explained due to the non-work-conserving nature of this arbitration technique, which leads to long memory wait times and consequently increases the probability of preempting a pending request. The situation is different for `TDMer`. Due to its high efficiency, the probability of preempting a pending request is much lower. It is instead more likely to preempt a request that is currently processed by the memory, resulting in moderate memory blocking delays for `SHDp`. Note that these delays may never exceed a single `TDM` period, notably for configurations with a single critical core. **The highest memory blocking delays are observed for `SHDw` under `TDMds`**. The memory delay amounts to up to 16 `TDM` periods for a configuration with



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



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

FIGURE 7.6: Maximum memory blocking delay for critical tasks across normalized system utilization.

2 cores, where both cores are critical (i.e., $P = 80$ cycles). The TDMer arbiter fares slightly better, with a maximum of about 12 TDM periods. This demonstrates the high overhead experienced even by critical tasks when using SHDw.

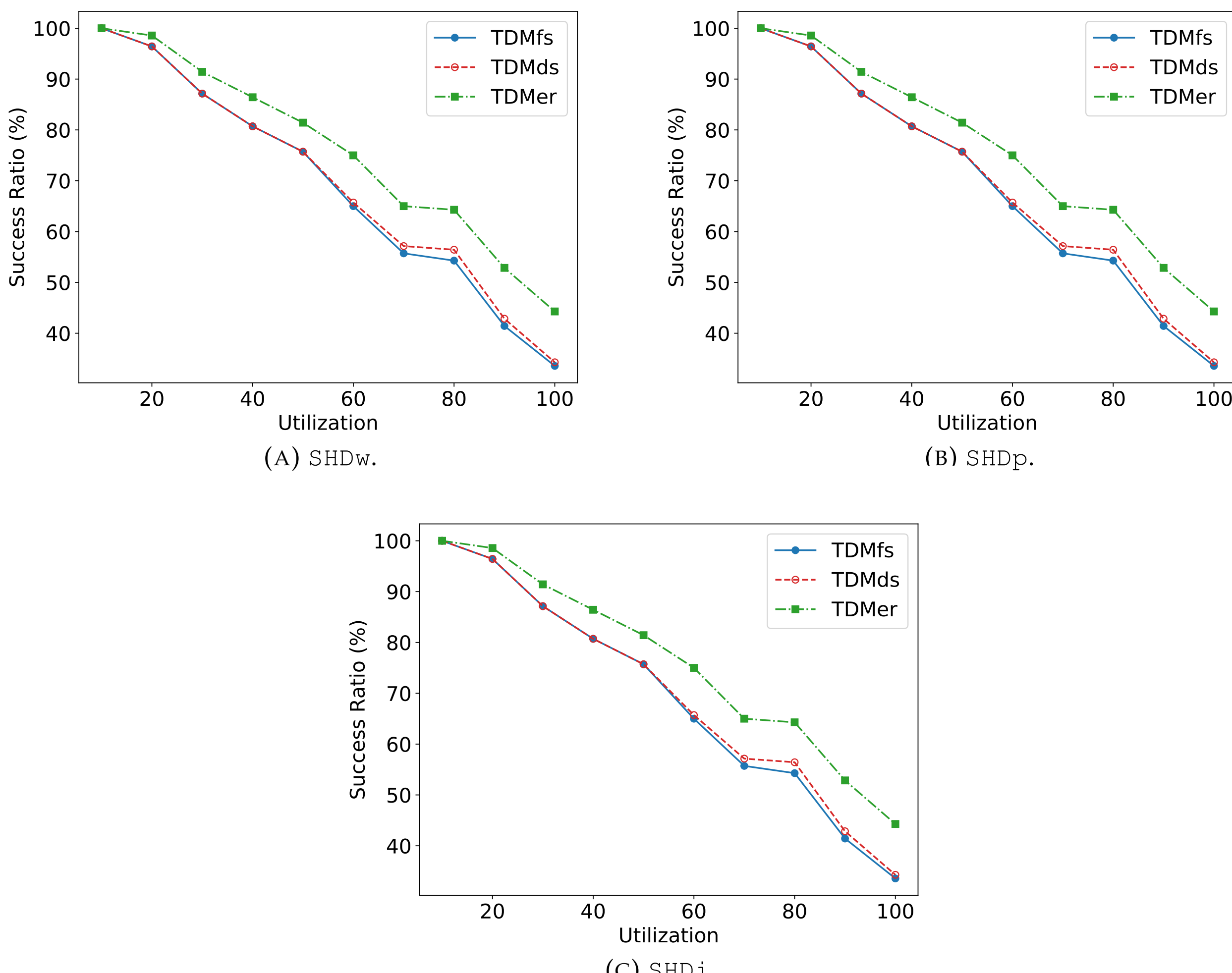
The SHDi scheme, as expected, falls in-between the two other schemes. In combination with TDMds the memory blocking delay is at most one TDM period. However, starting from 50% utilization, we can notice that TDMer exhibits slightly worse results compared to TDMds. Nevertheless, the preemption cost for TDMer always stays below 2 TDM periods (highlighted by Subfigure 7.6b) – notably for configurations with a single critical core (cf. Equation 7.5).

These results confirm the intuitive expectation that the overhead induced by **the three preemption schemes is strictly increasing from SHDp over SHDi to SHDw**. However, this is not always the case due to the extra costs induced by the slack counters under TDMer (a counter-example was encountered for a larger slot length of $Sl = 100$, while evaluating the impact of varying the memory access latency see Subsection 7.4.4). Recall that the newly imposed deadline during a preemption is computed considering a minimum slack of a single TDM slot length (Sl) in order to avoid clashes on the immediate next TDM slot (see Subsection 7.3.3). Accounting for this additional runtime overhead $\varepsilon \leq Sl$, we can derive a relationship between the various preemption schemes: $\mathrm{MB}_i^{\texttt{SHDp}} \leq \mathrm{MB}_i^{\texttt{SHDi}} \leq \mathrm{MB}_i^{\texttt{SHDw}} + \varepsilon$.

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

7.4.3 Results for (Preemptive) Arbitration Schemes

We now evaluate the success rate of our preemptive arbitration policies. The success ratio refers to the number of task sets that were *schedulable* for each level of utilization, i.e., simulation ended its execution without any deadline miss for critical tasks. Subfigures 7.7a, 7.7b, and 7.7c depict this success rate for our 3 arbitration policies under `SHDw`, `SHDp`, and `SHDi` respectively



(A) `SHDw`.

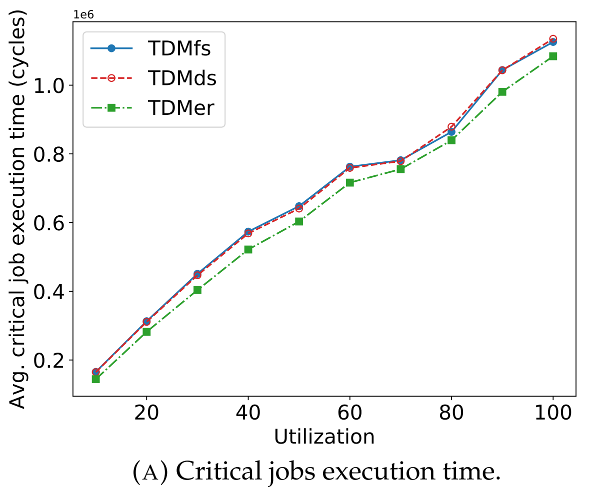
(B) `SHDp`.

(C) `SHDi`.

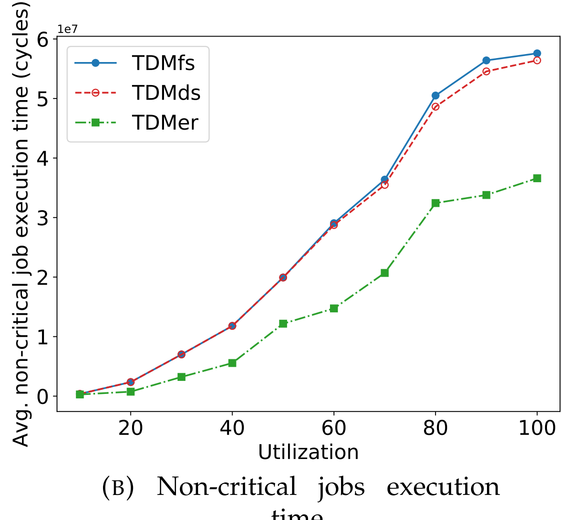
FIGURE 7.7: Average *schedulability* success ratio through normalized system utilization.

(each point represents an average value over 124 runs). Results are shown considering the same task set for all combinations. Comparing the impact of the preemption schemes across all utilization levels, little difference among them is visible. This can be explained by the number of preemptions, which is small compared to the total execution time of each simulation. As seen in Section 7.4.2, individual preemptions may however cause considerable overhead.

Overall, there was no significant difference in the success ratio of TDMfs and TDMds, except for a normalized utilization above 70%. In that case, we can notice better results for TDMds, regardless of the preemption technique. This is different from the results obtained in Subsection 5.5.2, which shows an improved memory utilization for TDMds compared to TDMfs. In Subsection 5.5.2, the observed improvement reached up to a factor of 3.3 in terms of memory utilization at low total system utilization, which then leveled off considerably at higher load. It appears that, TDMds improves the memory utilization mostly for situations where system load is not jeopardizing *schedulability*, explaining the small impact on success rates. However, regardless of the preemption techniques, TDMer shows better results at almost all utilization levels. This can be explained by the decoupling from TDM slots and the fine-grained dynamic memory arbitration. This effectively renders the TDMer scheme work-conserving. These results confirm those previously (see Chapter 5) obtained considering a restricted task model (one task per core), where the observed memory utilization improvements for TDMer were considerable, even at high utilization levels. TDMer thus has a relevant impact on the success rates.



(A) Critical jobs execution time.



(B) Non-critical jobs execution time.

FIGURE 7.8: Average jobs execution times with varying normalized system utilization using SHDi.

As the most reasonable choice for a preemption scheme is SHDi, Subfigures 7.8a and 7.8b depict the average execution time of critical and non-critical tasks respectively using this scheme. **The relative gain by TDMer is particularly visible for the average execution time of non-critical tasks at high utilization levels**. For instance, at 100% system utilization TDMer achieves an improvement of 36%. we can clearly conclude that TDMer improves the execution times of critical and non-critical tasks, and thus can reduce the probability of missing the deadline of critical tasks. Additionally, as preemptions are rare events, the runtime impact of the different preemption schemes is small compared to the impact of the arbitration policy.

Figure 7.9 shows a breakdown of the average number of non-critical tasks deadline misses, while varying normalized system utilization. Note that we only consider schedulable synthetic task sets (no critical tasks deadline miss during the execution). As expected, one can notice that the number of deadline misses increase with the normalized utilization. **TDMer presents better results by reducing the number of deadline misses compared to the other arbitration schemes**. By reducing the average execution times of non-critical tasks as depicted in Figure 7.8b, the chances for non-critical tasks to respect their deadlines increases, which is confirmed by the similar trends between the two aforementioned figures.

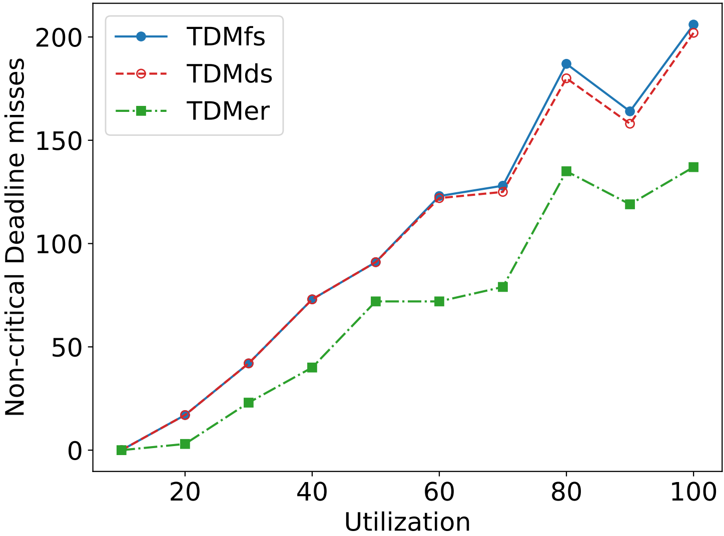
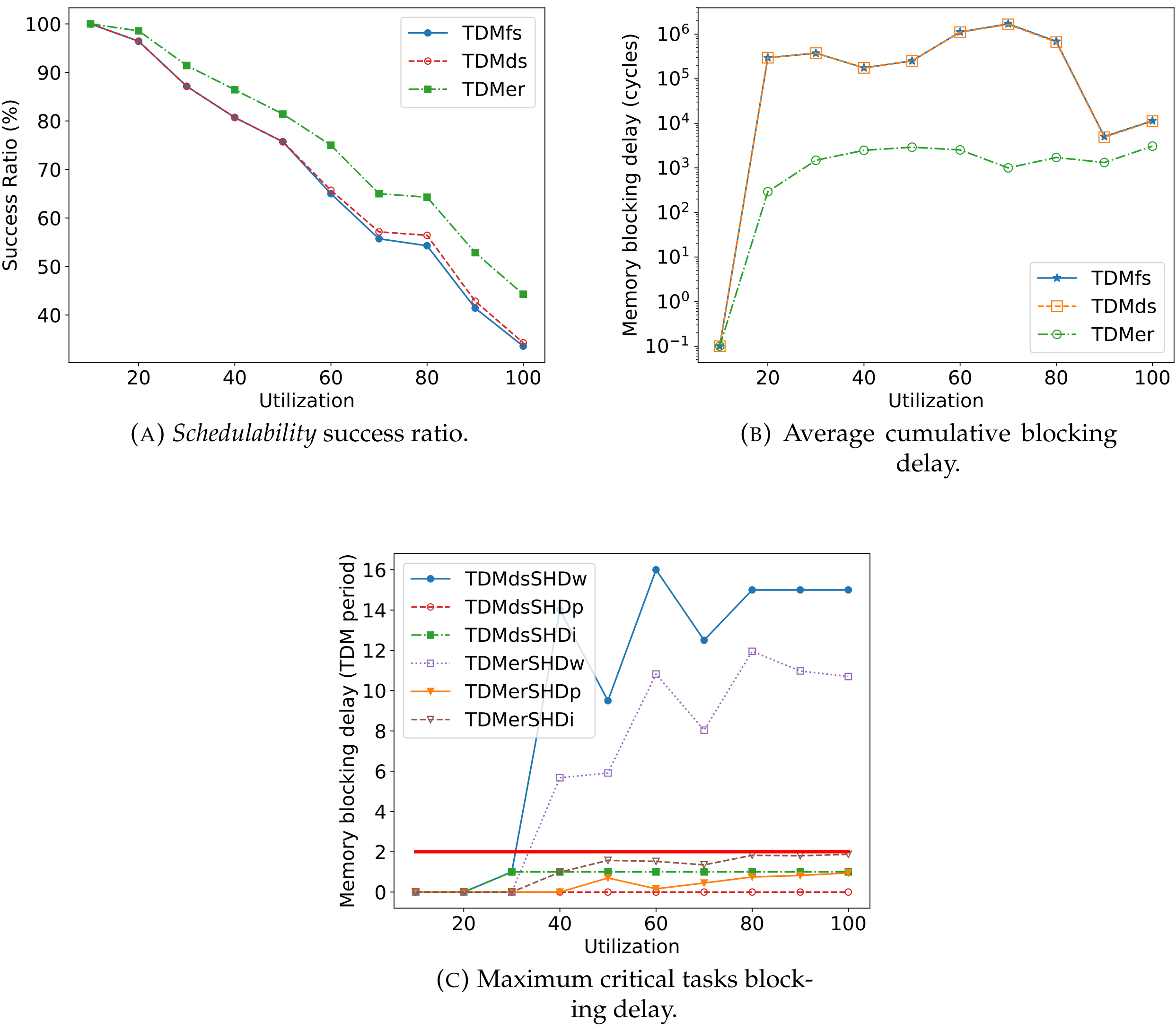


FIGURE 7.9: Average number of non-critical deadline misses.

7.4.4 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 DDR3 memory. In the following experiments, we are interested in evaluating the impact that varying the memory access latency might have on our arbitration schemes. Therefore, we slightly modified our experimental setup by varying the TDM slot length *Sl*. Similar to the experiments done in Subsection 5.5.4, we used two additional configurations with slot lengths of 25 and 100 cycles respectively, while keeping a minimum access latency of 21 cycles. A value of 25 cycles represents a memory with small access latency variability, while



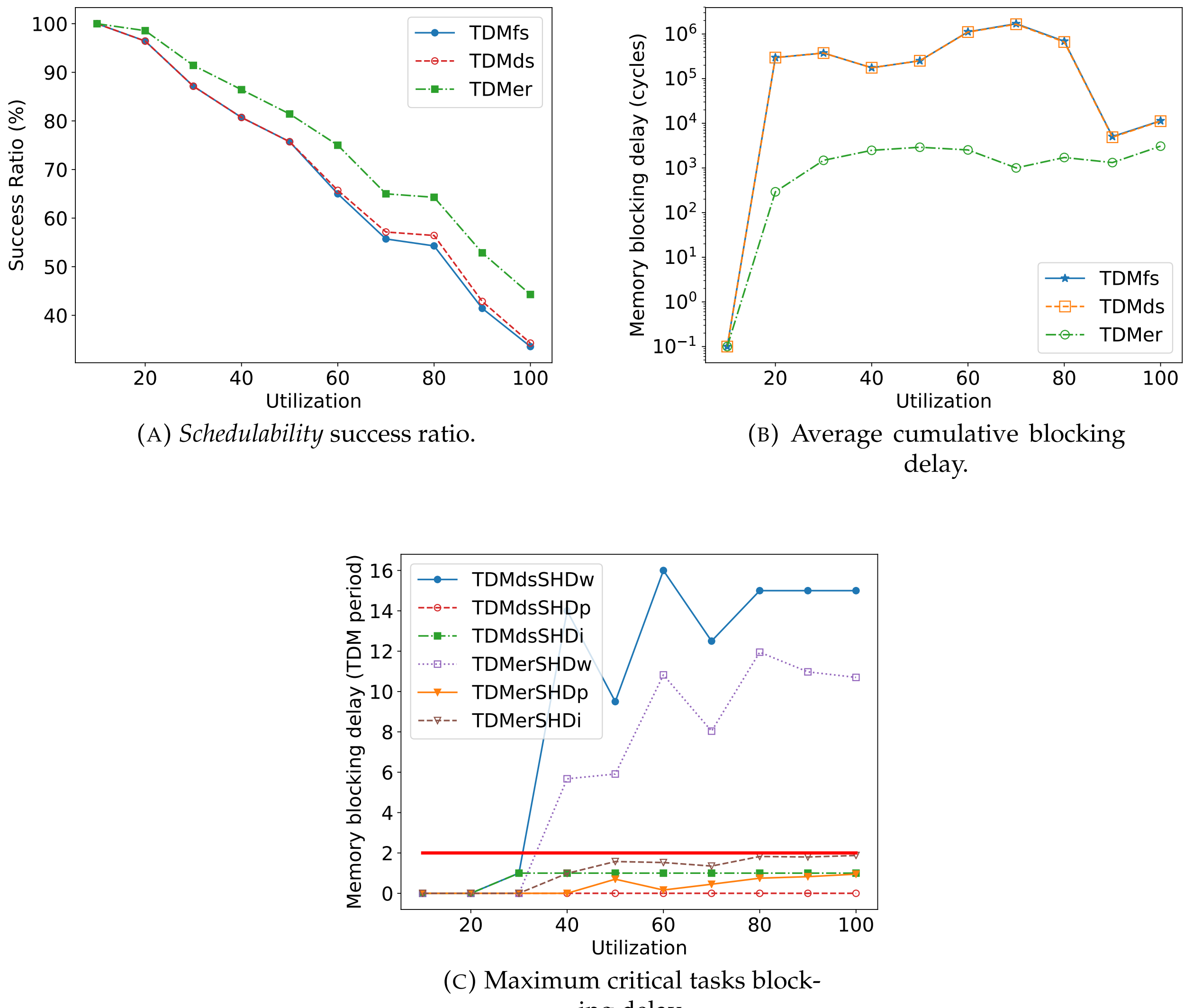
(A) *Schedulability* success ratio.

(B) Average cumulative blocking delay.

(C) Maximum critical tasks blocking delay.

FIGURE 7.10: Average results through normalized system utilization, with TDM slot lengths of 25 cycles.

a value of 100 cycles represents a high latency variability. Changing the value of Sl impacts the generated memory profiles (described in Subsection 5.5.1), as the number of memory accesses that fit into the task's WCETs (C_i) derived by UUniFast depends on Sl. Recall that our traffic generator takes the worst-case memory access latency for each newly generated request into account, which is bounded by $P + Sl - 1$ cycles. *Varying the* TDM *slot length thus also impacts the generated task sets*. Note that both the TDM period P and the TDM slot length Sl are impacted in our modified setup. The results presented in this section are therefore not directly comparable.



(A) *Schedulability* success ratio.

(B) Average cumulative blocking delay.

(C) Maximum critical tasks blocking delay.

FIGURE 7.11: Average results through normalized system utilization, with TDM slot lengths of 100 cycles.

We performed the same number of simulation runs as in the previous experiments. Figure 7.10 and 7.11 summarize the obtained results for $Sl = 25$ cycles and $Sl = 100$ cycles respectively. Subfigures 7.10a and 7.11a shows a breakdown of the average schedulability success ratios from all runs using the `SHDi` preemption mechanism. When $Sl = 25$ cycles, we can see that the improvement of `TDMer` compared to `TDMfs` is lower than the improvement obtained when $Sl = 40$ cycles. When increasing the access latency variability, i.e. $Sl = 100$ cycles, the improvement of `TDMer` over `TDMfs` is even greater than when $Sl = 40$ cycles. The improvement under `TDMer`, compared to `TDMfs`, appears to be proportional to the memory access latency variability. This is particularly visible at high loads. It is explained by the fact that non-critical tasks can potentially exploit the considerable memory idle time caused by the high variability of the memory access latencies, unlike the other arbitration schemes that have to respect the `TDM` slots.

The same trends are visible in Subfigures 7.10b and 7.11b w.r.t. the cumulative memory blocking time under `SHDw` preemption scheme. The drop at 20% utilization of `TDMer` in Subfigure 7.11b also confirms the volatility in the simulations due to the rare preemption events. Subfigures 7.10c and 7.11c show the normalized maximum values of the memory blocking delays w.r.t. the `TDM` period for critical tasks. These results are similar to those obtained with the original slot length ($Sl = 40$ cycles). The memory blocking delays, considering `SHDi`, still remains below 2 `TDM` periods.

The results confirm the previously defined upper-bounds for the various preemption mechanisms and that **our dynamic `TDM`-based arbitration scheme `TDMer` is performing well even with low latency memories**, confirming the results obtained in Section 5.5.4. But the gain is even more significant when using memories with highly varying latencies. So, regardless of the memory type, using our arbitration policy in combination with the `SHDi` preemption scheme allows to improve the use of the shared memory and to increase the overall performance of the system. In addition, our approach guarantees to respect the timing constraints required by critical tasks for real-time systems.

7.5 Conclusion

The work presented in this chapter extends the approaches on dynamic `TDM`-based memory arbitration schemes described in Chapter 5 by adding support for a preemptive execution model.

We propose means to manage, and finally bound the delays the two sources of arbitration-induced preemption delays the *memory blocking delay* and the *misalignment delay* identified in Subsection 4.1.3. While bounding the misalignment delay is straightforward, limiting the memory blocking delay is more involved. We thus propose formal bounds for two obvious preemption schemes based on waiting and preemptable memory requests (`SHDw` and `SHDp`). Additionally, we propose an alternative scheme (`SHDi`), which imposes new deadlines for critical requests when preempted and leverages *criticality inheritance* when a critical task is blocked by a non-critical request. The experimental results showed that the preemption mechanisms exhibit little difference at runtime, which allows us to select the best approach according to other criteria, such as low implementation complexity and analyzability. The new `SHDi` approach appears to offer the best trade-off in terms of these criteria. The evaluation also confirms the good performance offered by our work-conserving dynamic `TDM`-based arbitration technique (`TDMer`), which has a noticeable impact on the success ratio.

Part III

Conclusion

Chapter 8

Conclusion and Future Work

Contents

8.1 Conclusion 125
8.2 Future Work 126

This chapter concludes this thesis by providing a contribution summary before proceeding to a discussion on possible future work.

8.1 Conclusion

The work carried out within this thesis aimed at developing a criticality-aware and work-conserving arbitration policy. To this end, we first explore the various existing arbitration techniques used in real-time systems. It turned out that the TDM arbitration policy was a good starting point, even if TDM is not criticality-aware. However, TDM provides strict guarantees for critical tasks running on the platform. However, this policy is not work-conserving and leads to low memory utilization. In Section 4.1.2, we analyzed the various issues (namely *issue* and *release* delays) that make `TDM` a non-work-conserving arbitration policy and, in addition, identified challenges that have to be overcome when developing a dynamic `TDM`-based arbitration for multi-critical systems.

Chapter 5 introduces various approaches to overcome the problems that render `TDM` non-work-conserving. From this chapter the `TDMer` approach arises, which is the most efficient in terms of memory utilization while keeping the strict temporal guarantees offered by `TDM`. `TDMer` is a dynamic `TDM`-based arbitration policy. Instead of arbitrating at the level of `TDM` slots, `TDMer` 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. Our evaluation reveals considerable gains, in particular when approaching high system utilization. TDMer allows reducing delays suffered by traditional TDM by a factor of at least 50, and up to a factor of 300. Another contribution is a formal correctness proof of the dynamic TDM-based approaches. Most notably, we prove that TDM's temporal behavior is preserved for critical tasks. Consequently, analysis results valid under TDM, such as offset analyses [41], are equally valid under our schemes.

In Chapter 6, we proposed a hardware implementation of a slightly simplified variant of the TDMer approach denoted TDMrr. Using additional simulations we showed that the proposed solution is efficient, both in terms of hardware complexity and arbitration performance regarding the memory utilization.

The remaining challenge of this work is related to the task level impact of our dynamic TDM-based arbitration policy specifically the arbitration induced preemption delays. 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. In Chapter 7, we extended our system model to allow several tasks to execute on a single core and even allow 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 consider the resulting delays during schedulability analysis. We propose formal bounds for two obvious preemption schemes based on waiting and preemptable memory requests (SHDw and SHDp). Additionally, we propose an alternative scheme (SHDi), which imposes new deadlines for critical requests when preempted and leverages *criticality inheritance* when a critical task is blocked by a non-critical request. The SHDi approach appears to offer the best trade-off in terms of implementation complexity and analyzability concerning the upper-bound of preemption-related memory blocking delay. Also note that, slack counters, in case of a preemption, are part of the execution context of a task and need to be saved/restored accordingly.

8.2 Future Work

The dynamic TDM-based arbitration policies proposed in this thesis comes with a set of restrictions. First, we limited the discussions to uniform TDM schedules, where all TDM slots have the same length and each critical task is assigned to 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 [35, 79]. 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 2) then needs to be extended 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 the dynamic TDM-based arbitration policies currently assume independent periodic tasks. Interactions between dependent tasks may, however, impact the timing behavior. For instance, a task may wait for another task (e.g., the 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 preserving 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.

In Chapter 7, the SHDi approach was evaluated w.r.t. to the dynamic TDM-based arbitration schemes presented in Chapter 5. However, it could be applied to other schemes like fixed priority. The upper-bound proposed by Altmeyer et al. [6] covering the number of memory accesses is incomplete w.r.t. our definition of memory blocking delays. When a higher-priority task τ_i preempts a lower-priority task τ_k, Altmeyer et al. only consider the blocking delay due to a pending memory request of τ_k on τ_i on the same core. However, in a multi-core setting, requests from other cores may also block τ_i. Notably, tasks on other cores with a priority j, $k \leq j \leq i$. This situation is not taken into account in [6]. However, our SHDi preemption mechanism could limit the overhead by inheriting the priority of task τ_i to the pending request of τ_k.

The presented dynamic TDM-based arbitration schemes assume unbounded slack accumulation. However, a hardware implementation requires internal counter registers for slack accumulation. Hence, the amount of slack that can be accumulated at runtime is constrained by the bit-width of those internal registers as shown in Section 6.4. Consequently, the memory blocking delay upper-bound for SHDw (equation 7.2) can be simplified w.r.t. the hardware constrained upper-bound of the slack counters. A small upper-bound on slack results in low hardware complexity and a small overhead of the memory blocking delay w.r.t. the tasks' WCET. However, this

also limits the flexibility of our arbitration schemes on slack consumption. Recall that the present approach considers multi-criticality scheduling and ultimately our approach aims to fully support mixed-criticality (MC) scheduling. Therefore, as a research perspective, the slack accumulated at the arbitration level should also be used to add more flexibility to the MC scheduling. Hence, to do so we need to define a reasonable slack bound that offers the maximum flexibility with minimum hardware costs.

Bibliography

[1] A. Agrawal et al. "Contention-Aware Dynamic Memory Bandwidth Isolation with Predictability in COTS Multicores: An Avionics Case Study". In: *29th Euromicro Conference on Real-Time Systems (ECRTS)*. Vol. 76. LIPIcs. Schloss Dagstuhl, 2017, 2:1–2:22.

[2] B. Akesson et al. "Real-Time Scheduling Using Credit-Controlled Static-Priority Arbitration". In: *2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. 2008, pp. 3–14. DOI: 10.1109/RTCSA.2008.21.

[3] Benny Akesson and Kees Goossens. *Memory Controllers for Real-Time Embedded Systems: Predictable and Composable Real-Time Systems*. 1st. Springer Publishing Company, Incorporated, 2011. ISBN: 144198206X, 9781441982063.

[4] Benny Akesson et al. "Composability and Predictability for Independent Application Development,Verification, and Execution". In: *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*. Ed. by Michael Hübner and Jürgen Becker. New York, NY: Springer New York, 2011, pp. 25–56. ISBN: 978-1-4419-6460-1. DOI: 10.1007/978-1-4419-6460-1_2.

[5] Sebastian Altmeyer, Robert I. Davis, and Claire Maiza. "Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems". In: *Real-Time Systems* 48.5 (2012), pp. 499–526. ISSN: 1573-1383. DOI: 10.1007/s11241-012-9152-2.

[6] Sebastian Altmeyer et al. "A Generic and Compositional Framework for Multicore Response Time Analysis". In: *Proceedings of the International Conference on Real Time and Networks Systems*. RTNS '15. ACM, 2015, pp. 129–138. ISBN: 978-1-4503-3591-1. DOI: 10.1145/2834848.2834862.

[7] N. Audsley et al. "Applying new scheduling theory to static priority pre-emptive scheduling". In: *Software Engineering Journal* 8 (5 1993), pp. 284–292.

[8] Neil Audsley et al. "Real-Time System Scheduling". In: *Predictably Dependable Computing Systems*. Ed. by Brian Randell et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 41–52. ISBN: 978-3-642-79789-7.

[9] Andrea Baldovin, Enrico Mezzetti, and Tullio Vardanega. “A Time-composable Operating System”. In: *12th International Workshop on Worst-Case Execution Time Analysis*. Ed. by Tullio Vardanega. Vol. 23. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2012, pp. 69–80. ISBN: 978-3-939897-41-5. DOI: 10.4230/OASIcs.WCET.2012.69.

[10] Rajeshwari Banakar et al. “Scratchpad memory: a design alternative for cache on-chip memory in embedded systems”. In: *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No.02TH8627)* (2002), pp. 73–78.

[11] S. Baruah and N. Fisher. “The partitioned multiprocessor scheduling of sporadic task systems”. In: *26th IEEE International Real-Time Systems Symposium (RTSS’05)*. 2005, 9 pp.–329. DOI: 10.1109/RTSS.2005.40.

[12] S. K. Baruah, A. Burns, and R. I. Davis. “Response-Time Analysis for Mixed Criticality Systems”. In: *32nd Real-Time Systems Symposium (RTSS)*. Vienna, Austria, 2011, pp. 34–43.

[13] Sanjoy Baruah and N. Fisher. “The partitioned multiprocessor scheduling of deadline-constrained sporadic task systems”. In: *IEEE Transactions on Computers* 55.7 (2006), pp. 918–923. ISSN: 0018-9340. DOI: 10.1109/TC.2006.113.

[14] Sanjoy K. Baruah. “Dynamic- and Static-priority Scheduling of Recurring Real-time Tasks”. In: *Real-Time Systems* 24.1 (2003), pp. 93–128. ISSN: 1573-1383. DOI: 10.1023/A:1021711220939.

[15] Roman Bourgade. “Worst-case execution time analysis of real-time tasks executed on a multicore architecture”. Theses. Université Paul Sabatier - Toulouse III, Oct. 2012. URL: https://tel.archives-ouvertes.fr/tel-00746073.

[16] Dai Bui et al. “Temporal Isolation on Multiprocessing Architectures”. In: *Proceedings of the 48th Design Automation Conference*. DAC ’11. San Diego, California: ACM, 2011, pp. 274–279. ISBN: 978-1-4503-0636-2. DOI: 10.1145/2024724.2024787.

[17] A. Burns and S. Edgar. “Predicting computation time for advanced processor architectures”. In: *Proceedings 12th Euromicro Conference on Real-Time Systems. Euromicro RTS 2000*. 2000, pp. 89–96. DOI: 10.1109/EMRTS.2000.853996.

[18] Alan Burns and Robert I. Davis. “A Survey of Research into Mixed Criticality Systems”. In: *ACM Comput. Surv.* 50.6 (Nov. 2017), 82:1–82:37. ISSN: 0360-0300. DOI: 10.1145/3131347.

[19] G. C. Buttazzo, G. Lipari, and L. Abeni. “Elastic task model for adaptive rate control”. In: *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*. 1998, pp. 286–295. DOI: 10.1109/REAL.1998.739754.

[20] Chang-Gun Lee et al. "Analysis of cache-related preemption delay in fixed-priority preemptive scheduling". In: *17th IEEE Real-Time Systems Symposium*. 1996, pp. 264–274. DOI: 10.1109/REAL.1996.563723.

[21] Sheng Cheng, John A. Stankovic, and Krithivasan Ramamritham. *Scheduling Algorithms for Hard Real-Time Systems–A Brief Survey*. Tech. rep. Amherst, MA, USA, 1987.

[22] A. Colin and S. M. Petters. "Experimental evaluation of code properties for WCET analysis". In: *RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003*. 2003, pp. 190–199. DOI: 10.1109/REAL.2003.1253266.

[23] Liliana Cucu-Grosjean. "Independence - a misunderstood property of and for probabilistic real--time systems". In: *Alan Burns 60th Anniversary*. Ed. by N. Audsley and S. Baruah. York, United Kingdom, Mar. 2013. URL: https://hal.inria.fr/hal-00920504.

[24] Peter J. Denning. "The Locality Principle". In: *Commun. ACM* 48.7 (July 2005), pp. 19–24. ISSN: 0001-0782. DOI: 10.1145/1070838.1070856.

[25] P. Emberson, R. Stafford, and R.I. Davis. "Techniques For The Synthesis Of Multiprocessor Tasksets". In: *Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*. 2010, pp. 6–11.

[26] Daniel ETIEMBLE and François ANCEAU. "Hiérarchie mémoire : les caches". In: *Techniques de l'ingénieur Architectures matérielles* base documentaire : TIB308DUO.ref. article : h1002 (2012). fre. URL: https://www.techniques-ingenieur.fr/base-documentaire/technologies-de-l-information-th9/architectures-materielles-42308210/hierarchie-memoire-les-caches-h1002/.

[27] Christian Ferdinand and Reinhard Wilhelm. "Efficient and Precise Cache Behavior Prediction for Real-Time Systems". In: *Real-Time Systems* 17.2 (1999), pp. 131–181. ISSN: 1573-1383. DOI: 10.1023/A:1008186323068.

[28] M. D. Gomony et al. "A Globally Arbitrated Memory Tree for Mixed-Time-Criticality Systems". In: *IEEE Trans. Comput.* 66.2 (Feb. 2017), pp. 212–225. ISSN: 0018-9340.

[29] M. R. Guthaus et al. "MiBench: A free, commercially representative embedded benchmark suite". In: *Proc. of the Int. Workshop on Workload Characterization*. 2001, pp. 3–14.

[30] S. Hahn, J. Reineke, and R. Wilhelm. "Towards Compositionality in Execution Time Analysis: Definition and Challenges". In: *SIGBED Rev.* 12.1 (2015), pp. 28–36. ISSN: 1551-3688.

[31] J. Hansen, S. Hissam, and G. A. Moreno. "Statistical-Based WCET Estimation and Validation". In: *Intl. Workshop on Worst-Case Execution Time Analysis (WCET'09)*. Vol. 10. OASIcs. Schloss Dagstuhl, 2009, pp. 1–11. ISBN: 978-3-939897-14-9.

[32] M. Hassan. "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)*. 2018, pp. 495–505. DOI: 10.1109/RTSS.2018.00062.

[33] M. Hassan and H. Patel. "Criticality- and Requirement-Aware Bus Arbitration for Multi-Core Mixed Criticality Systems". In: *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2016, pp. 1–11. DOI: 10.1109/RTAS.2016.7461327.

[34] M. Hassan, H. Patel, and R. Pellizzoni. "A framework for scheduling DRAM memory accesses for multi-core mixed-time critical systems". In: *21st IEEE Real-Time and Embedded Technology and Applications Symposium*. 2015, pp. 307–316. DOI: 10.1109/RTAS.2015.7108454.

[35] Mohamed Hassan, Hiren Patel, and Rodolfo Pellizzoni. "PMC: A Requirement-Aware DRAM Controller for Multicore Mixed Criticality Systems". In: *ACM Trans. Embed. Comput. Syst.* 16.4 (May 2017), 100:1–100:28. ISSN: 1539-9087. DOI: 10.1145/3019611.

[36] J. Hildebrandt, F. Golatowski, and D. Timmermann. "Scheduling coprocessor for enhanced least-laxity-first scheduling in hard real-time systems". In: *Proceedings of 11th Euromicro Conference on Real-Time Systems. Euromicro RTS'99*. 1999, pp. 208–215. DOI: 10.1109/EMRTS.1999.777467.

[37] Ibrahim Hur and C. Lin. "Adaptive History-Based Memory Schedulers". In: *37th International Symposium on Microarchitecture (MICRO-37'04)*. 2004, pp. 343–354. DOI: 10.1109/MICRO.2004.4.

[38] Mathieu Jan, Lilia Zaourar, and Maurice Pitel. "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, 2013, pp. 43–48.

[39] M. Joseph and P. Pandya. "Finding Response Times in a Real-Time System". In: *The Computer Journal* 29.5 (Jan. 1986), pp. 390–395. ISSN: 0010-4620. DOI: 10.1093/comjnl/29.5.390.

[40] Norman Jouppi. "Cache Write Policies and Performance." In: vol. 21. May 1993, pp. 191–201. DOI: 10.1109/ISCA.1993.698560.

[41] T. Kelter et al. "Static Analysis of Multi-core TDMA Resource Arbitration Delays". In: *Real-Time Syst.* 50.2 (Mar. 2014), pp. 185–229. ISSN: 0922-6443.

[42] H. Kim et al. "Bounding memory interference delay in COTS-based multi-core systems". In: *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2014, pp. 145–154. DOI: 10.1109/RTAS.2014.6925998.

[43] A. Kostrzewa, S. Saidi, and R. Ernst. "Slack-based Resource Arbitration for Real-time Networks-on-chip". In: *Proceedings of the Conference on Design, Automation & Test in Europe*. DATE '16. EDA, 2016, pp. 1012–1017. ISBN: 978-3-9815370-6-2.

[44] A. Kostrzewa et al. "Flexible TDM-based Resource Management in On-chip Networks". In: *Int. Conf. on Real Time and Networks Systems*. Lille, France: ACM, 2015, pp. 151–160. ISBN: 978-1-4503-3591-1.

[45] A. Kritikakou et al. "Distributed run-time WCET controller for concurrent critical tasks in mixed-critical systems". In: *Int. Conf. on Real-Time Networks and Systems*. 2014.

[46] John P. Lehoczky and Lui Sha. "Performance of Real-time Bus Scheduling Algorithms". In: *Proceedings of the 1986 ACM SIGMETRICS Joint International Conference on Computer Performance Modelling, Measurement and Evaluation*. SIGMETRICS '86/PERFORMANCE '86. Raleigh, North Carolina, USA: ACM, 1986, pp. 44–53. ISBN: 0-89791-184-9. DOI: 10.1145/317499.317538.

[47] H. Leontyev and J. H. Anderson. "Generalized Tardiness Bounds for Global Multiprocessor Scheduling". In: *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*. 2007, pp. 413–422. DOI: 10.1109/RTSS.2007.33.

[48] Y. Li, B. Akesson, and K. Goossens. "Architecture and Analysis of a Dynamically-scheduled Real-time Memory Controller". In: *Real-Time Syst.* 52.5 (Sept. 2016), pp. 675–729. ISSN: 0922-6443.

[49] C. L. Liu and James W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment". In: *J. ACM* 20.1 (Jan. 1973), pp. 46–61. ISSN: 0004-5411. DOI: 10.1145/321738.321743.

[50] Sparsh Mittal. "A Survey of Techniques for Designing and Managing CPU Register File". In: *Concurrency and Computation Practice and Experience* 29 (July 2016). DOI: 10.1002/cpe.3906.

[51] Frank Mueller. "Static Cache Simulation and Its Applications". In: *PhD Dissertation* (Oct. 1994).

[52] Onur Mutlu and Thomas Moscibroda. "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors". In: *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)* (2007), pp. 146–160.

[53] Hemendra Singh Negi, Tulika Mitra, and Abhik Roychoudhury. "Accurate Estimation of Cache-related Preemption Delay". In: *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS '03. Newport Beach, CA, USA: ACM, 2003, pp. 201–206. ISBN: 1-58113-742-7. DOI: 10.1145/944645.944698.

[54] Kyle J. Nesbit et al. "Fair Queuing Memory Systems". In: *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 39. Washington, DC, USA: IEEE Computer Society, 2006, pp. 208–222. ISBN: 0-7695-2732-9. DOI: 10.1109/MICRO.2006.24.

[55] M. Paolieri, E. Quiñones, and F. J. Cazorla. "Timing Effects of DDR Memory Systems in Hard Real-time Multicore Architectures: Issues and Solutions". In: *ACM Trans. Embed. Comput. Syst.* 12.1s (Mar. 2013), 64:1–64:26. ISSN: 1539-9087.

[56] Marco Paolieri et al. "Hardware Support for WCET Analysis of Hard Real-time Multicore Systems". In: *Proceedings of the International Symposium on Computer Architecture*. ISCA '09. ACM, 2009, pp. 57–68. ISBN: 978-1-60558-526-0. DOI: 10.1145/1555754.1555764.

[57] Christof Pitter and Martin Schoeberl. "A Real-time Java Chip-multiprocessor". In: *ACM Trans. Embed. Comput. Syst.* 10.1 (Aug. 2010), 9:1–9:34. ISSN: 1539-9087. DOI: 10.1145/1814539.1814548.

[58] Hamza Rihani et al. "WCET Analysis in Shared Resources Real-time Systems with TDMA Buses". In: *Proceedings of the 23rd International Conference on Real Time and Networks Systems*. RTNS '15. ACM, 2015, pp. 183–192. ISBN: 978-1-4503-3591-1. DOI: 10.1145/2834848.2834871.

[59] S. Rixner et al. "Memory access scheduling". In: *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*. 2000, pp. 128–138. DOI: 10.1145/339647.339668.

[60] M. Schoeberl et al. "Towards a Time-predictable Dual-Issue Microprocessor: The Patmos Approach". In: *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*. OASICS. 2011, pp. 11–21.

[61] L. Sha, R. Rajkumar, and J. P. Lehoczky. "Priority inheritance protocols: an approach to real-time synchronization". In: *IEEE Transactions on Computers* 39.9 (1990), pp. 1175–1185. ISSN: 0018-9340. DOI: 10.1109/12.57058.

[62] K. G. Shin and P. Ramanathan. "Real-time computing: a new discipline of computer science and engineering". In: *Proceedings of the IEEE* 82.1 (1994), pp. 6–24. ISSN: 0018-9219. DOI: 10.1109/5.259423.

[63] Alan Jay Smith. "Cache Memories". In: *ACM Comput. Surv.* 14.3 (Sept. 1982), pp. 473–530. ISSN: 0360-0300. DOI: 10.1145/356887.356892.

[64] John A. Stankovic, Krithi Ramamritham, and Marco Spuri. *Deadline Scheduling for Real-Time Systems: Edf and Related Algorithms*. Norwell, MA, USA: Kluwer Academic Publishers, 1998. ISBN: 0792382692.

[65] H. Su and D. Zhu. "An Elastic Mixed-Criticality task model and its scheduling algorithm". In: *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2013, pp. 147–152. DOI: 10.7873/DATE.2013.043.

[66] H. Su, D. Zhu, and D. Mossé. "Scheduling algorithms for Elastic Mixed-Criticality tasks in multicore systems". In: *2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications*. 2013, pp. 352–357. DOI: 10.1109/RTCSA.2013.6732239.

[67] R. Tabish et al. "A Real-Time Scratchpad-Centric OS for Multi-Core Embedded Systems". In: *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2016, pp. 1–11.

[68] Hai Nam Tran. "Cache memory aware priority assignment and scheduling simulation of real-time embedded systems". Theses. Université de Bretagne occidentale - Brest, Jan. 2017. URL: https://tel.archives-ouvertes.fr/tel-01773862.

[69] Antti Valmari. "The state explosion problem". In: *Lectures on Petri Nets I: Basic Models: Advances in Petri Nets*. Ed. by Wolfgang Reisig and Grzegorz Rozenberg. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 429–528. ISBN: 978-3-540-49442-3. DOI: 10.1007/3-540-65306-6_21.

[70] Mary K. Vernon and Udi Manber. "Distributed Round-Robin and First-Come First-Serve Protocols and Their Application to Multiprocessor Bus Arbitration". In: *ISCA*. 1988.

[71] S. Vestal. "Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance". In: *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*. 2007, pp. 239–243. DOI: 10.1109/RTSS.2007.47.

[72] Jean Walrand and Pravin Varaiya. "CHAPTER 2 - Network Services and Layered Architectures". In: *High-Performance Communication Networks (Second Edition)*. Ed. by Jean Walrand and Pravin Varaiya. Second Edition. San Francisco: Morgan Kaufmann, 2000, pp. 39 –102. ISBN: 978-1-55860-574-9. DOI: https://doi.org/10.1016/B978-0-08-050803-0.50007-1.

[73] Jiacun Wang. *Real-Time Embedded Systems*. Quantitative software engineering series. Hoboken, BJ, USA: Wiley, 2017. ISBN: 9781118116173. DOI: 10.1002/9781119420712.

[74] K.C. Wang. "Embedded Real-Time Operating System". In: Springer, Cham, 2017. ISBN: 978-3-319-51517-5. DOI: 10.1007/978-3-319-51517-5.

[75] Reinhard Wilhelm. "Determining Bounds on Execution Times". In: *Handbook on Embedded Systems,R. Zurawski, Ed. CRC Press, Boca Raton, FL. 14–1, 14–23* (2005).

[76] Reinhard Wilhelm et al. "The Worst-case Execution-time Problem – Overview of Methods and Survey of Tools". In: *ACM Trans. Embed. Comput. Syst.* 7.3 (May 2008), 36:1–36:53. ISSN: 1539-9087. DOI: 10.1145/1347375.1347389.

[77] Z. P. Wu, Y. Krish, and R. Pellizzoni. "Worst Case Analysis of DRAM Latency in Multi-requestor Systems". In: *Real-Time Systems Symposium (RTSS)*. Vancouver, Canada, 2013, pp. 372–383.

[78] D. Wuertz, T. Setz, and Y. Chalabi. "Rmetrics - Modelling Extreme Events in Finance". In: 2017. URL: http://www.rmetrics.org.

[79] M.-K. Yoon, J.-E. Kim, and L. Sha. "Optimizing Tunable WCET with Shared Resource Allocation and Arbitration in Hard Real-Time Multicore Systems". In: *Real-Time Systems Symp.* Vienna, Austria, 2011, pp. 227–238. ISBN: 978-0-7695-4591-2.

[80] H. Yun et al. "MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms". In: *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2013, pp. 55–64.

ÉCOLE DOCTORALE

Sciences et technologies de l'information et de la communication (STIC)

université PARIS-SACLAY

Titre : Arbitrage mémoire dynamique non-oisif basé sur TDM pour des systèmes multi-criticité temps réel

Mots clés : Time-Division Multiplexing, Arbitrage Dynamique, Calcul Prédictible, Systèmes Multi-Critiques, Systèmes Temps Réel, Mémoire

Résumé : Les architectures multi-cœurs posent de nombreux défis dans les systèmes temps réel, qui découlent des conflits entre les accès simultanés à la mémoire partagée. Parmi les politiques d'arbitrage mémoire disponibles, le multiplexage temporel, en anglais Time-Division Multiplexing (TDM), assure un comportement prédictible en limitant les latences d'accès et en garantissant une bande passante aux tâches indépendamment des autres tâches. Pour ce faire, TDM garantit un accès exclusif à la mémoire partagée dans une fenêtre temporelle fixe. L'approche TDM, cependant, fournit une faible utilisation des ressources car elle *oisive*. De plus, elle est très inefficace pour les ressources ayant des latences d'accès très variables, comme le partage de l'accès à une mémoire DRAM. La longueur constante d'une fenêtre TDM est donc très pessimiste et entraîne une sous-utilisation de la mémoire. Pour pallier ces limitations, nous présentons des mécanismes d'arbitrage dynamique basés sur TDM. Cependant, plutôt que d'arbitrer au niveau des fenêtres TDM, notre approche fonctionne à la granularité des cycles d'horloge en exploitant les temps morts accumulés par les requêtes précédentes. Cela permet à l'arbitre de réorganiser les requêtes mémoire, d'exploiter les latences d'accès réelles des requêtes, et donc d'améliorer l'utilisation de la mémoire. Nous démontrons que nos politiques sont analysables car elles préservent les garanties de TDM dans le pire des cas, alors que nos expériences montrent une amélioration significative de l'utilisation de la mémoire.

Title : Work-conserving dynamic TDM-based memory arbitration for multi-criticality real-time systems

Keywords : Time-Division Multiplexing, Dynamic Arbitration, Predictable Computing, Multi-Criticality Systems, Real-Time Systems, Memory

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.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France