FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Shared Resource Contention-Aware Schedulability Analysis of Hard Real-Time Systems

Jatin Arora



Doctoral Program in Electrical and Computer Engineering

Supervisor: Prof. Dr. Eduardo Manuel Medicis Tovar

Co-Supervisor: Dr. Cláudio Roberto Ribeiro Maia

Co-Supervisor: Prof. Dr. Luís Miguel Pinho de Almeida

September 8, 2023

© Jatin Arora: September, 2023

Shared Resource Contention-Aware Schedulability Analysis of Hard Real-Time Systems

Jatin Arora

Doctoral Program in Electrical and Computer Engineering

Approved by:

President: Prof. José Nuno Moura Marques Fidalgo
Referee: Prof. Renato Mancuso
Referee: Prof. Isabelle Puaut
Referee: Prof. Pedro Alexandre Guimarães Lobo Ferreira Souto
Supervisor: Prof. Eduardo Manuel Medicis Tovar

September 8, 2023

Dedicated this special feat to my parents, the most important pillars of my life, who stood by me through every failure and triumph.

Abstract

Modern commercial-off-the-shelf (COTS) multicore processors were introduced to provide raw computing power and to build energy-efficient and cost-effective solutions. As a consequence, they have been used in most modern computing systems. However, the adoption of multicore platforms in *hard real-time systems*, i.e., systems that run applications with *stringent time requirements*, is still under the scrutiny of the real-time systems research community. One of the main challenges that hinder the use of COTS multicore platforms in hard real-time systems is their unpredictability, which originates from the *sharing* of different *hardware resources* such as shared caches, interconnect (e.g., memory bus), and the main memory. Specifically, a task executing on one core of a multicore platform has to compete with other co-running tasks (tasks running on other cores) to access these *shared resources*. This contention between tasks to access shared resources is formally known as the *shared resource contention*. Shared resource contention is problematic as it can negatively influence the temporal behavior of tasks in a *non-deterministic manner*. To safely determine whether all tasks running in the system can execute without violating their respective timing constraints, one of the most important factors is to analyze and derive a safe bound on the maximum shared resource contention that tasks can suffer.

The main objective of this dissertation is to build novel solutions to accurately quantify the shared resource contention that can be suffered by tasks due to the sharing of resources such as the memory bus and the main memory.

Among all the existing solutions that eliminate/allow analyzing the shared resource contention, the *phased execution model* has been identified as a good solution that enables a more precise analysis of the shared resource contention. Specifically, the idea of the phased execution model is to divide the task execution into distinct *computation* and *memory phases* such that a shared resource, e.g., main memory, can be accessed by the tasks only during their memory phases. However, shared resource contention can still occur even when tasks comply with the phased execution model, e.g., when memory phases of tasks running on different cores execute concurrently.

In this dissertation, we start by modeling and analyzing the maximum *memory bus contention*, i.e., contention due to the sharing of *memory bus* that can be suffered by tasks under the phased execution model. We then evaluate the impact of the bus arbitration policy on the bus contention that tasks can suffer by varying the bus arbitration policy. Results show that the use of a fairer bus arbitration scheme along with the phased execution model can lead to a tighter bound on the memory bus contention.

We also investigate the relationship between the memory bus and the cache memories. Specifically, we show that the bus contention strongly relates to the number of bus requests which further depends on the content of cache memories. Building on this, we propose the *holistic bus contention analysis* that analyzes the cache memories to bound the maximum number of *cache misses* and integrate them in the computation of the maximum bus contention that phased tasks can suffer.

Furthermore, we improve the bounds on the *main memory contention* of phased tasks considering dynamic random access-based memory systems. We first identify the pessimism in the

existing analysis that lies in the overestimation of the memory contention that can be caused by the write memory requests. We then propose a memory contention analysis that accurately quantifies the memory contention that can be suffered by tasks. We also show how the memory address mapping of tasks can impact the maximum memory contention that tasks can suffer.

Finally, we focus on the *memory-centric scheduling* that schedules the memory phases of all tasks running on the system such that multiple tasks cannot access the main memory concurrently to avoid shared resource contention. However, tasks can still be delayed, for example, if the memory scheduler is currently serving a memory phase of a task on another core. We first identify the sources of pessimism in the recent memory-centric scheduling-based analysis. We then provide insights on how such pessimism can be addressed. Building on this, we propose an improved memory-centric scheduling-based analysis that addresses the pessimism of the existing analysis.

Keywords: Real-time systems, Hard real-time systems, Multicore processors, Shared resources, Bus contention, Memory contention, Timing analysis.

Resumo

Processadores multinúcleo comercialmente disponíveis foram introduzidos para fornecer maior poder de computação e para criar soluções económicas e eficientes em termos de energia. Como consequência, eles têm sido usados na maioria dos sistemas de computação modernos. No entanto, a adoção de plataformas multinúcleo em sistemas de tempo real rígidos, ou seja, sistemas que executam aplicações com requisitos de tempo rigorosos, continuam a ser tema de investigação atual da comunidade de sistemas de tempo real. O principal desafio que dificulta o uso destes processadores multinúcleo em sistemas de tempo real rígido é sua imprevisibilidade, que se origina no uso partilhado de diferentes recursos de hardware, como memórias cache, interconexão (por exemplo, barramento de memória) e memória principal. Mais especificamente, uma tarefa executada num núcleo de uma plataforma multinúcleo precisa competir com outras tarefas co-executadas (tarefas executadas em outros núcleos) para aceder a estes recursos compartilhados. Essa disputa entre tarefas para aceder aos recursos compartilhados é formalmente conhecida como contenção de recurso compartilhado. A contenção de recursos compartilhados é problemática, pois pode influir negativamente o comportamento temporal das tarefas de maneira não determinística. Para determinar com segurança se todas as tarefas em execução no sistema podem ser executadas sem violar as suas respetivas restrições de tempo, é necessário analisar a contenção máxima de recursos compartilhados que as tarefas podem sofrer.

O principal objetivo desta dissertação é desenvolver novas soluções para, com precisão, quantificar o impacto que a contenção de recursos partilhados, como o barramento de memória e a memória principal, tem na execução de tarefas.

Dentre todas as soluções existentes que eliminam/permitem analisar a contenção de recursos compartilhados, o modelo de execução em fases tem sido identificado como uma solução promissora que possibilita uma análise mais precisa deste tipo de contenção. Mais especificamente, a ideia do modelo de execução em fases é dividir a execução da tarefa em distintas fases de execução e de acesso à memória, de modo que um recurso compartilhado, por exemplo, memória principal, possa ser acedido pelas tarefas apenas durante as suas fases de acesso à memória. No entanto, a contenção de recursos compartilhados ainda assim pode ocorrer mesmo quando as tarefas obedecem ao modelo de execução em fases, por exemplo, quando as fases de memória de tarefas executadas em diferentes núcleos são executadas simultaneamente.

Neste trabalho, começamos por modelar e analisar a contenção máxima do barramento de memória, ou seja, o atraso devido à partilha do barramento de memória que pode ser sofrido por tarefas sob o modelo de execução em fases. Em seguida, avaliamos o impacto sofrido pelas tarefas ao aplicarmos variações da política de arbitragem do barramento. Os resultados mostram que o uso de um esquema de arbitragem de barramento mais justo em conjunto com o modelo de execução em fases pode levar a um limite com menos variações na contenção do barramento de memória.

Também investigamos a relação entre o barramento de memória e as memórias cache. Especificamente, mostramos que a contenção do barramento está fortemente relacionada ao número de requisições de barramento, que depende ainda mais do conteúdo das memórias cache. Com base

nisso, propomos a análise holística de contenção de barramento que analisa as memórias cache para limitar o número máximo de faltas de cache e integrá-las no cálculo da contenção máxima de barramento que as tarefas em fases podem sofrer.

Além disso, melhoramos os limites da contenção de memória principal de tarefas em fases considerando sistemas de memória dinâmicos baseados em acesso aleatório. Primeiro identificamos o pessimismo na análise existente que reside na superestimação da contenção de memória que pode ser causada pelas solicitações de escrita de memória. Em seguida, propomos uma análise de contenção de memória que quantifica, com precisão, a contenção de memória que as tarefas podem sofrer. Também mostramos como o mapeamento do endereço de memória das tarefas pode impactar na contenção máxima de memória sofridas pelas tarefas.

Por fim, focamos no escalonamento centrado na memória que escalona as fases de memória de todas as tarefas em execução no sistema de forma que duas tarefas não acedam a memória principal simultaneamente para evitar a contenção de recursos compartilhados. No entanto, as tarefas ainda podem ser atrasadas, por exemplo, se o escalonador de memória estiver a atender a uma fase de memória de uma tarefa em outro núcleo. Primeiro identificamos as fontes de pessimismo na recente análise baseada em escalonamento centrada na memória. Em seguida, fornecemos intuições sobre como esse pessimismo pode ser abordado. Com base nisso, propomos uma análise aprimorada baseada em agendamento centrado na memória que aborda o pessimismo existente na análise.

Palavras-chave: Sistemas de tempo real, Sistemas de tempo real rígido, Processadores multi-núcleo, Recursos partilhados, Contenção de barramento, Contenção de memória, Análise temporal

Acknowledgments

The roller coaster journey of PhD research would not be completed without the support of different individuals. First and foremost, I would like to express my heartfelt gratitude to my supervisor, **Prof. Eduardo Tovar**, for his unwavering support throughout my PhD journey. I am truly grateful to him for consistently encouraging my proactive approach toward research and participation in various activities. Being the director of CISTER, he also ensured excellent hosting conditions.

I also extend my sincere appreciation to my co-supervisor, **Dr. Cláudio Maia**, for his invaluable support and thorough review of the majority of the work conducted during my PhD studies. His expertise has provided crucial feedback from both analytical and practical perspectives, which helped me in carrying out the research work presented in this dissertation.

I am immensely grateful to **Dr. Syed Aftab Rashid** for his invaluable feedback on most of the work developed during my PhD studies. I had the privilege of collaborating with him during the third year of my PhD. Due to his excellent & timely feedback, we collaborated on almost all works. In fact, he is the main person who helped me for the past one year. Among others, his expert advice on efficiently writing manuscripts and poster preparation was extremely useful.

Furthermore, I would like to express my sincere gratitude to **Dr. Geoffrey Nelissen** for first supervising me in the early stage of PhD and later continuing to collaborate with us after moving to TU/e, Eindhoven. I am especially grateful to him for introducing me to the relevant research problems which helped me in identifying the potential path for my PhD research.

Thanks to **Prof. Luis Almeida** for reviewing this dissertation and for helping with FEUP's administration. Thanks to my dear friend **Giann Nandi** for translating the abstract into Portuguese. Special thanks to the Jury members for their time and efforts in the evaluation of this dissertation.

I am thankful to all the colleagues/friends at **CISTER** for their friendship and good times, especially to David, Giann, Konstantinos, Yilian, Javier, Harrison, Jingjing, Ishfaq, Miguel, Gowhar, Enio, Radha, José, Lukas, Shashank, Mubarak, Patrick, Yousef, Ali, Reydel, Saeid, Luis Javier, Abdul, Shardul, Pedro Santos, Ramiro, Filipe, Alam, Yimin, and Mohammad. I would also like to thank Sandra, Cristiana, Marwin, Inês, and Benilde for their invaluable administrative support.

Most importantly, I want to express my deepest appreciation to my family members for their unwavering love and support. I am especially grateful to my parents, **Mr. Raj Kumar** and **Mrs. Darshana Rani**, for their love, support, and guidance that have profoundly shaped both my personal and professional life. I am also thankful to my siblings/cousins for their love and support, particularly my elder brother **Sumit Arora** who has been the biggest source of inspiration for me. It would have been impossible to achieve this feat without their love, support, and motivation.

Special thanks to all my Indian friends, especially the "Brown Munde Squad", i.e., Pankaj, Sachin, Gaurish and Satinder, for always being the best, most reliable, and funniest friends.

This work was partially supported by FCT (Fundação para a Ciência e Tecnologia) under the individual doctoral grant 2020.09532.BD.

Jatin Arora

List of Publications

The following list of publications reflects the results achieved during the development of this dissertation.

Journal Publications

- **Jatin Arora**, Cláudio Maia, Syed Aftab Rashid, Geoffrey Nelissen, and Eduardo Tovar, "Schedulability Analysis for 3-Phase Tasks with Partitioned Fixed-Priority Scheduling" in **Journal of Systems Architecture** 131 (2022), 102706. DOI: doi.org/10.1016/j.sysarc.2022.102706
- **Jatin Arora**, Cláudio Maia, Syed Aftab Rashid, Geoffrey Nelissen, and Eduardo Tovar, "Bus-Contention Aware WCRT Analysis for the 3-Phase Task Model Considering a Work-Conserving Bus Arbitration Scheme" in **Journal of Systems Architecture** 122 (2022), 102345. (**Best Paper Award at ICESS 2021**) DOI: doi.org/10.1016/j.sysarc.2021.102345
- **Jatin Arora**, Syed Aftab Rashid, Geoffrey Nelissen, Cláudio Maia, and Eduardo Tovar, "Memory Contention-aware WCRT Analysis for the 3-Phase Task Model" in **ACM Transactions on Embedded Computing Systems** (under review).

Conference Publications

- **Jatin Arora**, Syed Aftab Rashid, Geoffrey Nelissen, Cláudio Maia, and Eduardo Tovar, "Improved Bus Contention Analysis for 3-Phase Tasks" in 29th IEEE **RTCSA**, Niigata, Japan, 2023. To appear.
- **Jatin Arora**, Syed Aftab Rashid, Cláudio Maia, Geoffrey Nelissen, and Eduardo Tovar, "Work-in-Progress: A Holistic Approach to WCRT Analysis for Multicore Systems", in 43rd IEEE **RTSS**, Houston, TX, USA, 2022, pp. 511-514, doi: 10.1109/RTSS55097.2022.00054.
- **Jatin Arora**, Syed Aftab Rashid, Cláudio Maia, and Eduardo Tovar, "Analyzing Fixed Task Priority based Memory Centric Scheduler for the 3-Phase Task Model", in 28th IEEE **RTCSA**, Taipei, Taiwan, 2022, pp. 51-60, doi: 10.1109/RTCSA55878.2022.00012.
- **Jatin Arora**, Cláudio Maia, Syed Aftab Rashid, Geoffrey Nelissen, and Eduardo Tovar, "Bus-Contention Aware Schedulability Analysis for the 3-Phase Task Model with Partitioned Scheduling" in 29th **RTNS**, 2021, pp. 123–133, doi.org/10.1145/3453417.3453433.
- **Jatin Arora**, Cláudio Maia, Syed Aftab Rashid, Geoffrey Nelissen, and Eduardo Tovar, "Work-in-Progress: WCRT Analysis for the 3-Phase Task Model in Partitioned Scheduling," in 41st IEEE **RTSS**, 2020, pp. 407-410, doi: 10.1109/RTSS49844.2020.00050.

Workshop Publications

- **Jatin Arora**, Syed Aftab Rashid, Geoffrey Nelissen, Cláudio Maia, and Eduardo Tovar, "Memory Contention Analysis for 3-Phase Tasks" in **JRWRTC**, 2023, co-located with **RTNS** 2023. http://rtns2023-jrwrtc-memory-contention-analysis-for-3-phase-tasks.pdf.
- **Jatin Arora**, Cláudio Maia, and Syed Aftab Rashid, "Open Questions for the Bus-Blocking Problem in the 3-Phase Task Model under Partitioned Scheduling" in the **CAPITAL** Workshop, 2021. http://cister-labs.pt/docs/CAPITAL-Workshop-2021.pdf.

PhD Forum and Extended Abstracts

- **Jatin Arora**, Eduardo Tovar, and Cláudio Maia, "Shared Resource Contention Aware Schedulability Analysis for Multiprocessor Real-Time Systems" in the **PhD Forum**, **DATE**, 2023 (**PhD Forum Best Poster Prize**). http://cister-labs.pt/docs/DATE-PhD-forum-2023.pdf.
- **Jatin Arora**, Cláudio Maia, Syed Aftab Rashid, and Eduardo Tovar, "Open Issues in Analyzing the Schedulability for the 3-Phase Task Model using Partitioned Scheduling" in the symposium of Electrical and Computer Engineering of **DCE**, 2021 doi: doi.org/10.24840/978-972-752-276-7.

Other Publications

- **Jatin Arora**, and Patrick Meumeu Yomsi, "Wearable Sensors Based Remote Patient Monitoring using IoT and Data Analytics" in U.Porto Journal of Engineering, Volume 5, Issue 1, pp 34-45, 2019. DOI: doi.org/10.24840/2183-6493-005.001-0003

Contents

1 Introduction **1**
- 1.1 Multicore Platforms . . . 2
- 1.2 Phased Execution Model . . . 5
- 1.3 Shared Resource Contention and the 3-Phase Task Model . . . 6
- 1.4 Thesis Scope and Contributions . . . 9
- 1.5 Thesis Structure . . . 11

2 Background and Related Work **13**
- 2.1 Background . . . 13
  - 2.1.1 Task Characterization . . . 14
  - 2.1.2 Task Scheduling . . . 15
  - 2.1.3 Worst-Case Timing Analysis . . . 16
  - 2.1.4 Hardware Platform Characterization . . . 20
  - 2.1.5 Phased Execution Models . . . 28
- 2.2 Related Work . . . 32
  - 2.2.1 Related Work for the Generic Task Model . . . 32
  - 2.2.2 Related Work for the Phased Execution Model . . . 35
- 2.3 Chapter Summary . . . 39

I Bus Contention Analysis for the 3-Phase Task Model **41**

3 Bus Contention-Aware Schedulability Analysis for the 3-Phase Task Model **43**
- 3.1 System Model . . . 44
  - 3.1.1 Task Model . . . 44
  - 3.1.2 Execution Model . . . 45
  - 3.1.3 Memory Access Models . . . 46
- 3.2 Bus Blocking Analysis for the Dedicated Memory Access Model (DMAM) . . . 47
  - 3.2.1 Properties of the DMAM . . . 47
  - 3.2.2 Bounding the Number of Bus Blockings for the DMAM . . . 49
  - 3.2.3 Maximum Bus Blocking Computation for the DMAM . . . 51
  - 3.2.4 Bus Contention Analysis for all Remote Cores . . . 56
- 3.3 Bus Blocking Analysis for the Fair Memory Access Model (FMAM) . . . 57
  - 3.3.1 Useful Properties for the FMAM . . . 58
  - 3.3.2 Bounding the Number of Bus Blockings for the FMAM . . . 60
  - 3.3.3 Maximum Bus Blocking Computation for the FMAM . . . 61
- 3.4 Schedulability Analysis . . . 64
- 3.5 Experimental Evaluation . . . 67

3.5.1 Case Study . . . 67
3.5.2 Experiments using Synthetic Tasks . . . 71
3.6 Chapter Summary . . . 75

4 Evaluating the Impact of Bus Arbitration Policy on Bus Contention 77
4.1 System Model . . . 78
4.1.1 Memory Bus Model . . . 78
4.1.2 Execution Model . . . 78
4.2 Motivational Example . . . 79
4.3 Bus Contention Analysis for RR-based Bus Arbitration Policy . . . 80
4.3.1 Step 1: Bounding the Maximum Number of Bus Slots required by the Local/Remote Core . . . 80
4.3.2 Step 2: Bounding Maximum Bus Contention . . . 82
4.4 Accurately Estimating the Impact of Lower Priority Blocking . . . 86
4.5 Schedulability Analysis . . . 89
4.6 Experimental Evaluation . . . 90
4.7 Chapter Summary . . . 95

5 Cache-aware Bus Contention Analysis 97
5.1 System Model . . . 98
5.2 Background . . . 99
5.3 Persistence-aware Cache Analysis for 3-Phase Tasks . . . 100
5.3.1 Upper Bounding Memory Access Requests by the Local Core . . . 101
5.3.2 Upper Bounding Memory Access Requests by the Remote Core . . . 103
5.4 Cache-aware Bus Contention Analysis . . . 104
5.4.1 Cache-aware Bus Contention Analysis for the RR Bus Arbitration Policy 104
5.4.2 Cache-aware Bus Contention Analysis for the FCFS Bus Arbitration Policy 106
5.5 Worst Case Response Time Analysis . . . 107
5.6 Experimental Results . . . 109
5.7 Chapter Summary . . . 114

II Memory Centric Scheduling 115

6 Fixed Task Priority-based Memory Centric Scheduling 117
6.1 System Model . . . 118
6.1.1 Task Model . . . 118
6.1.2 Task Priority (TP) based Memory Centric Scheduler . . . 119
6.2 Motivational Example . . . 119
6.3 Analyzing Fixed Task Priority-based Memory Centric Scheduler . . . 121
6.3.1 Bounding Intra-Core Interference . . . 121
6.3.2 Bounding Intra-Core Blocking . . . 122
6.3.3 Bounding Inter-Core Memory Interference . . . 122
6.3.4 Bounding Inter-Core Memory Blocking . . . 123
6.4 WCRT Analysis . . . 127
6.5 Analyzing the Impact of Preemption Point Selection . . . 128
6.5.1 Bounding Intra-Core Blocking . . . 129
6.5.2 Bounding Inter-Core Memory Blocking . . . 129
6.6 Experimental Evaluation . . . 130

6.7 Chapter Summary . . . 134

III Memory Contention Analysis 135

7 Memory Contention Analysis for 3-Phase Tasks 137
7.1 System Model . . . 139
7.1.1 Task Model . . . 139
7.1.2 Main Memory Model . . . 140
7.2 Background . . . 142
7.3 Proposed Memory Contention Analysis for 3-phase tasks . . . 144
7.3.1 Memory Address Mapping . . . 144
7.3.2 Memory Contention Analysis for Random Mapping . . . 146
7.3.3 Memory Contention Analysis for Bank Level Contiguous Mapping . . . 150
7.4 WCRT Analysis . . . 154
7.4.1 WCRT Analysis for Random Mapping . . . 154
7.4.2 WCRT Analysis for Bank Level Contiguous Mapping . . . 156
7.5 Experimental Evaluation . . . 156
7.5.1 Case Study . . . 157
7.5.2 Experiments using Synthetic Tasks . . . 159
7.6 Chapter Summary . . . 163

8 Conclusion and Future Work 165
8.1 Summary of Contributions . . . 165
8.2 Thesis Validation . . . 167
8.3 Future Work . . . 167
8.3.1 Improving Preciseness of Bound on Shared Resource Contention . . . 167
8.3.2 Task to Core Mapping Strategies . . . 168
8.3.3 Holistic Frameworks . . . 169

References 171

A Unbounded Priority Inversion Problem 183

List of Figures

1.1 Main sources of shared resource contention in multicore processors 3
1.2 Shared resource contention problem in the 3-phase task model 6

2.1 Memory Hierarchy in COTS multicore platform 20
2.2 Organization of DRAM 26
2.3 Contention-free system level offline schedule for PREM tasks 29
2.4 Contention-free system level offline schedule for 3-phase tasks 30

3.1 Bus blocking caused by a remote core for each bus blocking suffered at the local core 48
(a) Scenario 1 48
(b) Scenario 2 48
(c) Scenario 3 48
3.2 Maximum number of bus blockings when $\tau_j \in lp_{i,l}$ executes at the start of $W_{i,l}$ . . 50
3.3 Maximum number of bus blockings when $\tau_h \in hp_{i,l}$ executes at the start of $W_{i,l}$. 50
3.4 Maximum bus blocking for $N_{\pi_l}(W_{i,l}) > N_{\pi_r}(W_{i,l})$ 52
3.5 Possible scenarios when $N_{\pi_l}(W_{i,l}) = N_{\pi_r}(W_{i,l})$ 53
(a) Possible scenario 1 53
(b) Possible scenario 2 53
3.6 Maximum bus blocking for DMAM and FMAM 57
(a) Bus blocking under DMAM 57
(b) Bus blocking under FMAM 57
3.7 Bus blocking suffered by a pair of one R and one A-phase on the local core . . . 59
(a) Scenario 1 of Property 3.4 59
(b) Scenario 2 of Property 3.4 59
3.8 Maximum number of bus blockings suffered by the local core during $W_{i,l}$ when $lp_{i,l} = \emptyset$ 60
3.9 Maximum number of bus blockings suffered by the local core during $W_{i,l}$ when $lp_{i,l} \neq \emptyset$ 61
3.10 Varying core utilization 69
3.11 Varying the number of cores and core utilization 70
(a) Varying core utilization for $m = 2$ 70
(b) Varying core utilization for $m = 8$ 70
(c) Varying core utilization for $m = 16$ 70
3.12 Varying core utilization 72
3.13 Varying the number of cores and core utilization 73
(a) Varying core utilization for $m = 2$ 73
(b) Varying core utilization for $m = 8$ 73

(c) Varying core utilization for $m = 16$ . . . 73
3.14 Varying the tasks' Memory Demand (MD) . . . 74
(a) Varying MD for 20% core utilization . . . 74
(b) Varying MD for 30% core utilization . . . 74
(c) Varying MD for 40% core utilization . . . 74
3.15 Varying the tasks' period Range and core utilization . . . 74
(a) 100 to 1000 task period range . . . 74
(b) 100 to 2000 task period range . . . 74
(c) 100 to 5000 task period range . . . 74

4.1 WCRT of tasks under (a) FCFS and (b) RR bus arbitration policy . . . 79
(a) Example scenario for the FCFS bus arbitration policy . . . 79
(b) Example scenario for the RR bus arbitration policy . . . 79
4.2 Example scenario to derive maximum bus contention for case 2 . . . 84
4.3 Scenario 1 when task $\tau_j \in lp_{i,l}$ cause blocking to task τ_i during $W_{i,l}$ . . . 87
4.4 Scenario 2 when task $\tau_z \in lp_{i,l}$ cause blocking to task τ_i during $W_{i,l}$ . . . 87
4.5 Varying core utilization . . . 91
4.6 Varying the number of cores and core utilization . . . 92
(a) Number of cores (m) = 2 . . . 92
(b) Number of cores (m) = 4 . . . 92
(c) Number of cores (m) = 8 . . . 92
4.7 Varying core utilization for different MD configurations . . . 93
(a) Very Low MD . . . 93
(b) Low MD . . . 93
(c) High MD . . . 93
(d) Very High MD . . . 93
4.8 Varying core utilization for different tasks' period range . . . 94
(a) Short (S) Tasks' Period Range . . . 94
(b) Medium (M) Tasks' Period Range . . . 94
(c) Long (L) Tasks' Period Range . . . 94
4.9 Varying slot size SS . . . 94

5.1 Varying core utilization and number of cores for the FCFS bus arbitration policy . 110
(a) FCFS bus policy, m=2 . . . 110
(b) FCFS bus policy, m=4 . . . 110
(c) FCFS bus policy, m=8 . . . 110
5.2 Varying core utilization and number of cores for the RR bus arbitration policy . . 111
(a) RR bus policy, m=2 . . . 111
(b) RR bus policy, m=4 . . . 111
(c) RR bus policy, m=8 . . . 111
5.3 Varying Memory Demand (MD) for the FCFS bus arbitration policy . . . 112
(a) FCFS bus policy, VL MD . . . 112
(b) FCFS bus policy, L MD . . . 112
(c) FCFS bus policy, H MD . . . 112
(d) FCFS bus policy, VH MD . . . 112
5.4 Varying Memory Demand (MD) for the RR bus arbitration policy . . . 113
(a) RR bus policy, VL MD . . . 113
(b) RR bus policy, L MD . . . 113
(c) RR bus policy, H MD . . . 113

(d) RR bus policy, VH MD . . . 113
5.5 Varying number of cache sets . . . 113
(a) Varying cache sets for the FCFS bus policy . . . 113
(b) Varying cache sets for the RR bus policy . . . 113

6.1 Inter-core memory interference . . . 120
(a) PP-based MCS [Schwäricke et al., 2020] . . . 120
(b) TP-based MCS . . . 120
6.2 Maximum number of inter-core memory blockings suffered on the local core π_l during $W_{i,l}$ . . . 124
6.3 Maximum number of inter-core memory blockings for non-preemptive E-phases 129
6.4 Varying core utilization and number of cores . . . 132
(a) Varying core utilization for m=2 . . . 132
(b) Varying core utilization for m=4 . . . 132
(c) Varying core utilization for m=8 . . . 132
6.5 Varying core utilization for different MD configurations . . . 133
(a) VL MD . . . 133
(b) L MD . . . 133
(c) H MD . . . 133
(d) VH MD . . . 133
6.6 Varying core utilization for different task period ranges . . . 133
(a) Task period range of 100-1000 . . . 133
(b) Task period range of 100-2000 . . . 133
(c) Task period range of 100-5000 . . . 133

7.1 Illustration of the platform model . . . 141
7.2 Example scenario for random mapping . . . 144
7.3 Example scenario for bank level contiguous mapping . . . 146
7.4 Varying the core utilization and number of cores . . . 158
(a) Number of cores $m = 2$ . . . 158
(b) Number of cores $m = 4$ . . . 158
(c) Number of cores $m = 8$ . . . 158
7.5 Varying the core utilization and number of cores . . . 160
(a) Number of cores $m = 2$ . . . 160
(b) Number of cores $m = 4$ . . . 160
(c) Number of cores $m = 8$ . . . 160
7.6 Varying the Memory Demand (MD) . . . 161
(a) Low MD . . . 161
(b) Medium MD . . . 161
(c) High MD . . . 161
7.7 Varying the task period range . . . 162
(a) Short (S) task period range . . . 162
(b) Medium (M) task period range . . . 162
(c) Long (L) task period range . . . 162

A.1 Unbounded priority inversion problem in the TP-based MCS using global memory preemptions . . . 183

List of Tables

2.1 JEDEC timing constraints for DDR3-1333H [JEDEC, 2008]. 27

3.1 Table of Symbols 46
3.2 Benchmark parameters used in the experiments. 68

7.1 Table of Symbols 142

List of Abbreviations

A-Phase	Acquisition Phase
BL	Burst Length
CAST	Certifications Authorities Software Team
COTS	Commercial Off-The-Shelf
CPRO	Cache Persistence Reload Overhead
CPU	Central Processing Unit
DRAM	Dynamic Random Access Memory
DMAM	Dedicated Memory Access Model
ECB	Evicting Cache Block
EDF	Earliest Deadline First
E-Phase	Execution Phase
FCFS	First-Come First Serve
FIFO	First-in First-out
FEUP	Faculdade de Engenharia da Universidade do Porto
FR-FCFS	First-Ready First-Come First Serve
FMAM	Fair Memory Access Model
FP	Fixed Priority
FPNP	Fixed Priority Non Preemptive
FSB	Front-side Bus
I/O	Input/Output
ILP	Integer Linear Programming
LLC	Last Level Cache
MD	Memory Demand
MCS	Memory Centric Scheduling
PCB	Persistent Cache Block
PP	Processor Priority
PREM	PRedictable Execution Model
RAM	Random Access Memory
ROM	Read Only Memory
R-Phase	Restitution Phase
RR	Round Robin
SAG	Schedule Abstraction Graph
SOTA	State-of-the-Art
SS	Slot Size
SCE	Single Core Equivalence
TDMA	Time-Division Multiple Access
TP	Task Priority
WCET	Worst-Case Execution Time
WCRT	Worst-Case Response Time

Chapter 1

Introduction

Embedded systems have become an essential part of our day-to-day life. These systems can be found in a wide range of applications such as mobile phones, printers, digital cameras, automobiles, missile systems, health monitoring systems, etc. An embedded system can be characterized by its functional behavior, i.e., the logical correctness of the system. There is a special class of embedded systems in which, apart from the functional behavior, the *temporal behavior* of the applications running on the system is critical. The temporal behavior of an application determines the amount of time required by the application to perform the given set of operations. In the literature, these types of embedded systems are referred to as ***Real-Time Embedded Systems*** or ***Real-Time Systems***. Real-time systems are defined as systems in which the correctness of the system depends not only on the logical results, i.e., functional behavior, but also on the timing on which the results are produced, i.e., temporal behavior [Stankovic, 1988]. In particular, a specific *time bound* is associated with each application that specifies the time by which an application should complete all the required operations. If an application cannot complete all the required operations within the specified time bound then it is considered that the given application cannot fulfill its timing constraints. Depending on the type of real-time system, there can be some negative consequences if an application cannot fulfill the given timing constraints. On the basis of the level of negative consequences that can happen when applications cannot comply with their timing constraints, the real-time systems can be broadly classified into *hard real-time systems*, *soft real-time systems*. Hard real-time systems run applications with *stringent timing requirements* and the consequences of not meeting given timing constraints by applications can be catastrophic. A common example is the car airbag system in which activating the airbag within a given time is important; otherwise, it may have serious negative consequences. Applications of hard real-time systems can be found in various sectors where *temporal behavior* is *critical*, e.g., automotive, avionics, aerospace, railways, etc. [Cecere et al., 2016, Koo and Kim, 2018]. Soft real-time systems also run applications with timing requirements, but the consequences of not meeting given timing constraints only affect the quality of service (QoS). A common example is the lag in the audio/video in multimedia applications, which can degrade QoS. A given application running on the real-time system is typically composed of a set of *tasks* that are responsible for performing a set of required operations

in a timely manner. So, from now onward we will use the term task to refer to runtime units of applications.

Given the strict timing requirements, it is crucial to ensure that tasks running on real-time systems, particularly hard real-time systems, meet their timing constraints. This implies that it is necessary to provide *timing guarantees* on tasks running on the real-time system. To provide the timing guarantees on tasks, we need to determine the maximum time that can be consumed by tasks while performing the required operations. This is commonly achieved through the analysis of the *worst-case timing behavior* of tasks. Two metrics are used for this purpose. One that considers the task under analysis running in isolation and another that considers the task under analysis running with other tasks. Specifically, the first metric is characterized by the *maximum time* taken by the task to complete its *execution* in the worst-case scenario, and is commonly computed by considering the task under analysis is running in isolation, i.e., the only task in the system. The second metric is characterized by the *longest time* taken by the task from *release to completion* in the worst-case scenario, and is computed by considering that the task under analysis is running with other tasks in the system. On the basis of the worst-case timing behavior and timing constraints of a given task, we can determine either by analysis or during system execution whether a given task is *schedulable*, i.e., a task can complete the set of operations on the system without violating its timing constraints. Similarly, the same process can be performed for all tasks in the system to determine whether *all tasks* in the system are schedulable or not. This type of analysis is commonly known as *schedulability analysis* which is performed to determine whether all tasks in the system can meet their timing constraints. Consequently, the schedulability analysis is of extreme importance especially for hard real-time systems as it allows the system designer to determine whether the given set of tasks can execute on the system without violating their respective timing constraints. Different factors can influence the worst-case timing behavior and schedulability analysis. These factors include the number of tasks, their properties, operating system policies, and the architecture of the hardware computing platform. In the next section, we start by discussing the hardware computing platforms and their impact on the worst-case timing behavior of tasks. Since the focus of this dissertation is multicore computing platforms, we will mainly discuss multicore processors.

1.1 Multicore Platforms

Traditionally, embedded system application development was a complex and time-consuming process, involving custom-built hardware and software. To reduce costs and speed up the production cycle, the industry has embraced the use of commercial off-the-shelf (COTS) components, which are readily available and offer faster and cost-effective product development [Dasari et al., 2013]. Single-core processors were the go-to choice for computing systems, but they had limitations. These processors had to run at high frequencies to meet the computational demands of complex applications, but it was impossible to operate them above their specified maximum frequency. Operating single-core processors at high frequencies resulted in increased heat dissipation and energy

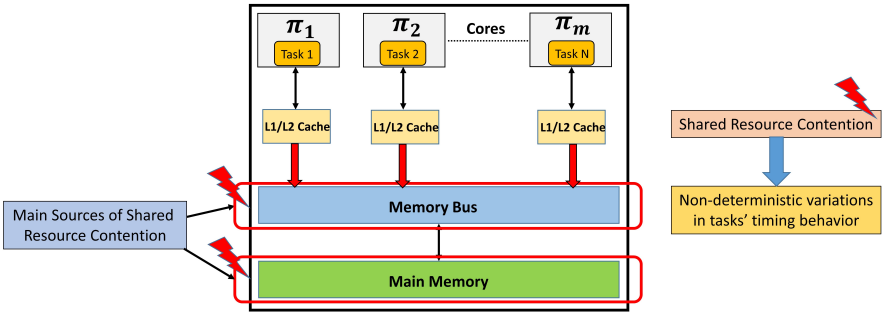


Figure 1.1: Main sources of shared resource contention in multicore processors

consumption.

To overcome the physical limitations of single-core processors, *multicore processors* were introduced. Unlike single-core processors, multicore processors have multiple cores integrated on the same die, which allows tasks running on different cores to execute in parallel to each other and concurrently. Multicore processors offer several advantages over single-core processors such as raw computing power, energy efficiency, cost-effectiveness, etc. which is why they have been adopted by most systems including soft real-time systems. However, the adoption of multicore platforms in hard real-time systems is still under the scrutiny of the real-time systems research community.

The main challenge that hinders the use of COTS multicore platforms in hard real-time systems is their *unpredictability*, which originates from the *sharing* of *hardware resources*, e.g., shared caches, the interconnect (e.g., memory bus), and the main memory, among all the cores. A task executing on one core of a multicore platform has to compete with other co-running tasks (tasks running on other cores) to access these *shared resources*. This contention between tasks to access shared resources is formally known as the *shared resource contention*. The shared resource contention can negatively impact the temporal behavior of tasks in a *non-deterministic* manner. This poses a significant challenge in the worst-case timing analysis of tasks. Specifically, as shown in Figure 1.1 when tasks running on multiple cores access the memory bus and main memory concurrently, the contention can be suffered by tasks at the memory bus and the main memory. This is the reason that the *memory bus* and the *main memory* have been identified as the *major sources* of shared resource contention in the literature [Rosen et al., 2007, Schranzhofer et al., 2010, Andersson et al., 2010, Schliecker and Ernst, 2010, Chattopadhyay et al., 2010, Chattopadhyay and Roychoudhury, 2011, Kelter et al., 2011, Dasari et al., 2011, Yao et al., 2012, Dasari and Nelis, 2012, Heechul Yun et al., 2013, Kelter et al., 2014, Yun et al., 2014, Yao et al., 2016a, Kim et al., 2014, Yun et al., 2015, Mancuso et al., 2015, Dasari et al., 2015, Rihani et al., 2015, Jacobs et al., 2015, Kim et al., 2016, Jacobs et al., 2016, Davis et al., 2017, Maia et al., 2017, Hassan and Pellizzoni, 2018, Rashid et al., 2020, Casini et al., 2020, Hassan and Pellizzoni, 2020, Schwäricke et al., 2020, Senoussaoui et al., 2022a, Thilakasiri and Becker, 2023a]. The main memory is typically much larger than the local memory of the core, such as L1 cache, allowing it to store all the necessary data and instructions for all tasks in the system. However, accessing the main memory incurs considerably higher latency compared to the faster cache memories. Consequently, when a

requested memory block is not available in the cache, all tasks need to access the main memory to read from or write the required data or instructions. Thus, in scenarios in which multiple tasks running on different cores require concurrent access to the main memory, contention may occur. This phenomenon is referred to as *memory contention*, i.e., contention due to the sharing of main memory. Furthermore, all the cores in the system commonly access the main memory through the shared interconnect which is usually referred to as shared *memory bus*. Due to such sharing, tasks running on multiple cores can compete to access the memory bus in order to perform the required memory operations on the main memory. This competition can lead to the problem of *bus contention*, i.e., contention due to the sharing of the memory bus.

It has been shown in the literature [Zhuravlev et al., 2010, Nowotsch and Paulitsch, 2012, Nélis et al., 2016, Radojković et al., 2012] that the problem of shared resource contention can significantly impact the temporal behavior of tasks. For example, it has been observed in [Nélis et al., 2016] that the shared resource contention leads to a slowdown of 8x in the execution time of tasks compared to their execution time in isolation. Similarly, an even higher slowdown of 14x was observed in [Radojković et al., 2012]. This problem of bus/memory contention in multicore systems is not only recognized by academia but also by industries and certification authorities. As an example, the aviation authorities of Europe, Asia and North and South America, together known as the "Certifications Authorities Software Team (CAST)", published a position paper named CAST32-A [CAST-32A, 2016] where they express their concerns related to the use of multicore processors in hard real-time systems such as avionics systems. Recently, these requirements have been integrated into the Acceptable Means of Compliance (ACMs) documents by the European Union Aviation Safety Agency (EASA) [AMC-20, 2022].

Analyzing the bus/memory contention can be extremely challenging because it depends on the low-level arbitration mechanisms employed by the memory bus and memory controller which is not always disclosed by the manufacturers. Due to such low-level arbiters, memory requests arriving from different cores can be reordered by the memory bus and the main memory which makes it difficult to predict the time by which memory requests of tasks will be served. Due to this temporal unpredictability posed by the bus/memory contention in multicore systems, analyzing the worst-case timing behavior of tasks becomes extremely challenging. As the worst-case timing behavior of tasks is a prerequisite for performing the schedulability analysis, it is of utmost importance to accurately quantify the shared resource contention suffered by tasks in order to perform their worst-case timing analysis and schedulability analysis. *This dissertation primarily focuses on analyzing bus/memory contention for tasks executing on multicore platforms.*

To address the problem of shared resource contention in multicore systems, industry, and academia have conducted extensive research (see the survey for details [Maiza et al., 2019]). These efforts of the real-time systems research community have led to the development of various solutions built on top of software/hardware mechanisms to facilitate the worst-case timing analysis of tasks. Among all these existing approaches, the concept of *phased execution model* [Pellizzoni et al., 2011, Durrieu et al., 2014] has been identified as a promising solution. We will now discuss the phased execution model in detail in the next section.

1.2 Phased Execution Model

The concept of *phased execution models* [Pellizzoni et al., 2011, Durrieu et al., 2014] was introduced to circumvent the temporal unpredictability posed by the shared resource accesses in multicore systems. The main idea of this model is to divide the execution of a task into *computation* and *memory phases* such that the task only accesses the shared resources, i.e., main memory and memory bus, during its memory phase and no main memory accesses are allowed during the computation phase. This concept first became a reality in the form of the *PRedictable Execution Model (PREM)* [Pellizzoni et al., 2011]. In the PREM, a task first executes its memory phase to prefetch all the required data/instructions from the main memory and store it in the core local memory (e.g., L1 cache). Then the task is executed by the core during the computation phase using the preloaded data/instructions in the local memory, without accessing the shared memory bus/main memory. In the PREM model, when one task on a given core is executing its memory phase, another task running on a different core can simultaneously perform its computation phase, without suffering the shared resource contention. This allows for parallel execution of PREM tasks on different cores without any shared resource contention.

The PREM was then generalized to the *3-phase task model* (also known as the AER model) [Durrieu et al., 2014] in which the execution of each task is divided into *three phases*, namely *Acquisition*, *Execution*, and *Restitution*. During the acquisition phase (also called the *A-phase*), the task's data/instructions are loaded from the main memory via the memory bus into the core local memory. During the execution phase (also called the *E-phase*), the task is executed by the core as per the preloaded data/instructions by the task. Finally, in the restitution phase (i.e., *R-phase*), the processed data is written back to the main memory via the memory bus. This categorizes the A- and R-phases into *memory phases* in which accesses to the main memory via the memory bus are performed, and the E-phase into *computation phase* in which the task does not access the memory bus/main memory. The 3-phase task execution model has been preferred for multiprocessor scheduling due to its *predictable* nature, as 1) it schedules the shared resource accesses of tasks only during specified time intervals, i.e., memory phases; and 2) it ensures predictable memory access patterns, where read memory accesses occur during the A-phase, and write memory accesses occur during the R-phase. This information can be useful in building fine-grained analysis of the shared resource contention that tasks can suffer. Consequently, the 3-phase task execution model has received much attention from the industry and academia [Durrieu et al., 2014, Becker et al., 2016, Maia et al., 2016, Tabish et al., 2016, Maia et al., 2017, Pagetti et al., 2018, Fort and Forget, 2019, Gracioli et al., 2019, Tabish et al., 2019, Soliman et al., 2019, Koike et al., 2020, Rivas et al., 2019, Casini et al., 2020, Schuh et al., 2020, Meunier et al., 2022, Thilakasiri and Becker, 2023a, Kloda et al., 2023, Tabish et al., 2023].

Although the PREM and the 3-phase task model are similar, unlike the PREM the 3-phase task model uses the R-phase at the completion of the E-phase of the task which can be useful, for example, to perform predictable inter-core communication [Tabish et al., 2019]. Furthermore, the 3-phase task model has been identified as suitable for industrial applications where temporal

predictability is crucial, e.g., predictable flight control system [Durrieu et al., 2014, Pagetti et al., 2018]. Considering the utility of phased execution models to solve the shared resource contention problem, in this dissertation, we focus on this model, and in particular, we use the ***3-phase task execution model***.

1.3 Shared Resource Contention and the 3-Phase Task Model

Using the 3-phase task model, it is possible to schedule tasks such that when a task is executing its A/R-phase, tasks running on other cores can execute their E-phases concurrently without suffering shared resource contention. Building on this idea, a few works [Becker et al., 2016, Pagetti et al., 2018] have proposed frameworks to generate a system level *offline schedule* such that multiple tasks cannot execute their memory phases concurrently to *avoid* shared resource contention. However, these offline scheduling-based approaches have some *limitations*. For instance, an offline schedule may not be enforced in some scenarios, e.g., due to the event-triggered/sporadic nature of tasks. Furthermore, scalability is an issue for such approaches, as the system-level offline schedule may not be valid and needs to be reconstructed if some changes take place in the system, e.g., variation in the length of the memory phases, addition of a new task, etc.

When tasks are scheduled by the scheduler on the CPU based on specific properties of tasks, it can potentially lead to a scenario in which tasks running on multiple cores execute their memory phases concurrently. This can lead to the problem of *shared resource contention* in the 3-phase task model. As an example, Figure 1.2 shows a system on which 3-phase tasks execute on a dual-core platform, i.e., marked as core 1 and core 2. On each core, two tasks are mapped, i.e., tasks τ_h and τ_i on core 1 and tasks τ_u and τ_k on core 2. We can see in Figure 1.2 that when task τ_h on core 1 requires access to the shared memory bus/main memory to execute its A-phase, it suffers shared resource contention due to execution of the A-phase of τ_u of core 2. Similarly, the task τ_i on core 1 also suffers shared resource contention from τ_k of core 2. Assuming that task τ_i is the task under analysis, we can see in Figure 1.2 that the timing behavior of task τ_i is significantly affected due to the shared resource contention suffered by tasks τ_i and τ_h on core 1 even in a simple case of a system with two cores and four tasks.

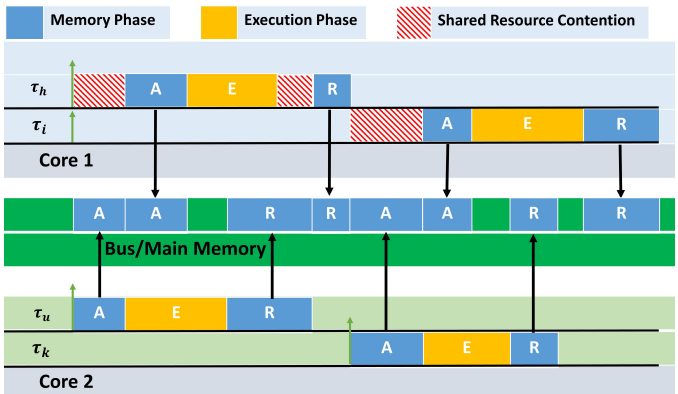


Figure 1.2: Shared resource contention problem in the 3-phase task model

Considering the importance of this problem, a few works [Maia et al., 2017, Thilakasiri and Becker, 2023a] have focused on analyzing the shared resource contention for the 3-phase task model. Specifically, these works consider the *memory bus* as the major source of shared resource contention because the memory bus is responsible for connecting all cores with the main memory. These works focus on analyzing the maximum *bus contention* for tasks scheduled on the multi-core platform using *global scheduling*. As a consequence, the solutions presented in the existing works do not address the bus contention problem from the *partitioned scheduling* perspective. In contrast to global scheduling which allows tasks to execute and migrate to any core in the system, partitioned scheduling statically maps tasks to cores and does not allow tasks to migrate to other cores. As a consequence, in partitioned scheduling, one can determine the specific set of tasks that will execute on each core in the system. However, addressing the bus contention problem considering partitioned scheduling can be extremely challenging because of the extra degree of non-determinism in comparison to global scheduling. For example, using global scheduling, tasks that execute on the system are commonly scheduled using the global scheduler that decides the order and time on which each task executes on the system. In such a scenario, the degree of non-determinism is smaller since the scheduler is aware of all the currently executing tasks in the system, and all tasks in the ready queue. On the contrary, in partitioned fixed-priority scheduling, there is a per-core scheduler that decides the order and time to execute each task running on that core based on specific properties of tasks. In such a scenario, when scheduling tasks on a given core, the scheduler may not be aware of currently executing co-running tasks on other cores and their current status, i.e., whether the co-running task is executing the memory phase or computation phase. This brings a higher degree of non-determinism and uncertainty as it becomes more difficult to determine the specific set of co-running tasks that cause shared resource contention to the task under analysis. *This highlights the importance and complexities involved in analyzing the bus contention for the 3-phase task model considering partitioned scheduling.*

Furthermore, while deriving the bound on bus contention, these works [Maia et al., 2017, Thilakasiri and Becker, 2023a] neglect the fact that memory bus contention strongly relates to the number of bus/memory requests generated by tasks, which, in turn, depends on the content of the cache memories during the execution of those tasks. Specifically, these existing works assume that the worst-case number of bus/memory requests will be generated during all the memory phases of all tasks, irrespective of the already existing content in the cache memory. Considering that each memory phase of every task will access all memory blocks from the main memory without analyzing the maximum number of cache misses can be pessimistic. This is because there can be a scenario in which some memory blocks that once loaded into the cache (i.e., the core local memory) will never be evicted or invalidated by the task itself. As a consequence, such memory blocks available in the cache memory can be *reused* by tasks to potentially reduce the number of bus/memory accesses of tasks. As the bus contention depends on the bus/memory accesses, bounding and integrating cache misses can provide a tighter bound on the bus contention that 3-phase tasks can suffer. It means that assuming worst-case cache misses without analyzing the cache memories can result in overestimating the bus contention that can be suffered by tasks

potentially resulting in underutilization of the computing platform. *This highlights the importance of a holistic approach for bus contention analysis that analyses & integrates the maximum number of cache misses while computing the number of bus/memory access and bus contention.*

Another work [Casini et al., 2020] that focuses on the shared resource contention suffered by 3-phase tasks considers the *main memory* as the major source of shared resource contention. Specifically, the work considers an architecture that facilitates a point-to-point connection between each core and the main memory. As a consequence, each core can issue memory requests to the main memory at the same time, potentially resulting in *memory contention*. Even though the memory contention analysis presented in [Casini et al., 2020] focuses on partitioned fixed-priority scheduling, it has some limitations. For example, the existing work can overestimate the memory contention that can be suffered by the task under analysis due to the write requests, i.e., R-phase memory requests. This overestimation can yield pessimistic bounds on the memory access times and memory contention suffered by tasks which in turn lead to pessimistic bounds on the worst-case timing analysis. This pessimism can potentially result in underutilization of the computing platform, thus, limiting the usage of available resources of the computing platform. To accurately quantify the maximum memory contention that can be suffered by tasks, there is a need to tightly bound the memory contention that 3-phase tasks can suffer. Furthermore, memory address mapping of tasks can be leveraged to accurately quantify the memory contention that can be suffered by tasks. *This highlights the importance of building fine-grained memory contention analysis for 3-phase tasks by accurately quantifying the memory contention caused by write memory requests and considering the memory address mapping of tasks.*

Another line of research uses *Memory Centric Scheduling* (MCS) [Yao et al., 2012, Yao et al., 2016a, Rivas et al., 2019, Schwäricke et al., 2020] to schedule the memory phases of all tasks executing on all the cores. The goal of MCS is to use the *global memory-centric scheduler* to prevent multiple tasks from accessing shared resources simultaneously. Specifically, the global memory-centric scheduler schedule at most one memory phase at a time at the system level. This ensures that there can be at most one memory phase accessing the shared resources at a time, avoiding any shared resource contention. However, tasks can still be delayed, e.g., if the core is ready to schedule a memory phase of the task but the global memory scheduler is busy scheduling the memory phase of a co-running task. This delay is commonly referred to as *memory interference* and it depends on the specific properties of tasks and the behavior of the memory-centric scheduler. The initial research in this area focused on a Time Division Multiple Access (TDMA) based MCS [Yao et al., 2012] that assigns static TDMA slots to the cores. Each core schedules its memory accesses, i.e., memory phases, during its assigned TDMA slots. The most recent work on MCS [Schwäricke et al., 2020] adopts a fixed *Processor Priority* (PP) based scheduler in which global fixed priority is assigned to each core. The PP-based MCS schedules the memory phases of tasks on the basis of the *priority of the cores* on which tasks execute. Although these works provide important solutions, they can potentially overestimate the memory interference, e.g., TDMA-based MCS is built on top of TDMA which is a *non-work conserving* policy; and PP-based MCS schedule memory accesses on the basis of priority of cores and does *not* take into account *task priorities*. *This*

highlights the importance of improving the existing MCS-based solutions to reduce the memory contention suffered by tasks by taking into account task priorities.

Despite the efforts of the real-time system research community to solve the problem of shared resource contention using the 3-phase task model, there are several problems that have not yet been completely addressed. In this dissertation, we will be focusing on a few of those problems/questions as stated below:

P1. How to accurately quantify the bus contention suffered by 3-phase tasks considering partitioned scheduling?

P2. What is the impact of bus arbitration policy on bus contention suffered by 3-phase tasks?

P3. How to holistically analyze the bus contention for the 3-phase task model considering the relationship between the memory bus and cache memories?

P4. How to improve existing memory-centric schedulers to tightly bound the memory interference of tasks?

P5. How to accurately quantify the main memory contention for the 3-phase task model?

P6. What is the impact of main memory address mapping on the memory contention suffered by tasks?

1.4 Thesis Scope and Contributions

Modern multicore platforms typically share various hardware resources such as caches, memory bus, and main memory, among multiple cores. Due to such sharing, tasks running on different cores may compete to access these resources resulting in *shared resource contention*. A *safe* upper bound on the shared resource contention is a prerequisite for the *worst-case timing analysis* of *3-phase tasks* executing on a multicore computing platform. The bound on the shared resource contention should be as precise as possible in order to avoid the underutilization of system resources. Building on this, we define the high-level goal of this dissertation as follows:

The high-level goal of this dissertation is to provide solutions to accurately quantify the shared resource contention between 3-phase tasks due to the sharing of two resources, i.e., the memory bus, and the main memory, either independently or in conjunction with each other.

To achieve this goal, this dissertation makes the following contributions:

1. **Accurate quantification of the bus contention for the 3-phase task model**

   The first contribution addresses problems **P1.**, and **P2.** and is presented in Chapters 3 and 4. In Chapter 3, we propose the bus contention analysis for the 3-phase task model considering partitioned scheduling and the First-Come-First-Served (FCFS) bus arbitration policy. In

particular, we analyze the bus contention considering two different memory access models that can be suited for different applications. In Chapter 4, we show how the bounds on the bus contention can be improved by considering a fairer bus arbitration policy such as the Round-Robin (RR). We show that if the blocking caused by a lower priority task in the multicore platform is computed in a manner similar to that of uniprocessors, it can yield unsafe bounds. We show how to correctly quantify the maximum blocking that can be caused by a lower priority task under the 3-phase task model executing on a multicore platform. The worst-case timing analysis of tasks is then performed by integrating the maximum bus contention that tasks can suffer. Finally, the schedulability analysis is performed.

2. **Holistic bus contention analysis for the 3-phase task model**

   The second contribution addresses problem **P3.** and is presented in Chapter 5. In this contribution, we present a *holistic* overview of the relationship between the memory bus and cache memories. We show that the bus contention strongly depends on the cache misses and considering the worst-case cache misses without analyzing the cache memory may lead to pessimistic bounds on the bus contention. In particular, we use the notion of *cache persistence*, i.e., memory blocks that once loaded into the cache by the task can be reused by its subsequent jobs without the need to access the main memory. This can tightly bound the number of cache misses which in turn reduces the bus/main memory accesses that can be generated during the memory phases. A tighter bound on the bus contention is then derived by incorporating the number of cache misses. Evaluations show that the cache-aware bus contention analyses can provide significantly tighter bounds in comparison to their respective cache-oblivious counterparts.

3. **Analysis of the fixed-task priority-based memory-centric scheduler**

   The third contribution addresses problem **P4.** and is presented in Chapter 6. We propose a memory-centric scheduler that schedules the memory phases of tasks executing on the system according to *priority of tasks*. We show that the proposed memory-centric scheduler can reduce the memory interference that can be suffered by tasks in comparison to the existing Processor Priority (PP) based memory-centric scheduler that schedules the memory phases based on the priority of cores on which tasks execute. The proposed memory-centric scheduler considers fixed-priority limited preemptive scheduling in contrast to the fixed-priority non-preemptive scheduling used in the PP-based MCS. Furthermore, we investigate the impact of preemption point selection, i.e., whether to allow preemption anytime during the E-phase as in [Yao et al., 2012] or only at the start/end of the E-phase, on the memory interference suffered by tasks. Evaluations show that memory interference can be reduced using the proposed MCS compared to the PP-based MCS, which results in improving schedulability.

4. **Accurate quantification of the main memory contention for the 3-phase task model**

   The fourth contribution addresses problems **P5.** and **P6.** and is presented in Chapter 7. We first identify the sources of pessimism in the existing work [Casini et al., 2020] that analyzes the main memory contention for the 3-phase task model. We then provide insights on how to address the pessimism in the existing analysis. We also show that memory contention strongly relates to the memory address mapping of tasks, i.e., how the data required by memory phases of tasks are mapped to the address space of the main memory. Building on this, we propose a fine-grained memory contention analysis for the 3-phase task model by leveraging different information such as the type of memory request, memory address mapping of tasks, memory controller arbitration policy, etc. The worst-case timing analysis of tasks is then performed by integrating the maximum memory contention that 3-phase tasks can suffer. Finally, the schedulability analysis is performed.

1.5 Thesis Structure

The remainder of this dissertation is structured as follows:

Chapter 2 provides the essential background required to understand the work developed in this dissertation and discusses related works that align with the problems addressed in this dissertation.

The main contributions of this dissertation are divided into *three parts*.

Part I focus on the first and second contributions and it consists of Chapters 3, 4 and 5.

In Chapter 3, we formally present the bus contention analysis for the 3-phase task model considering partitioned fixed-priority scheduling and the FCFS bus arbitration policy. Chapter 4 presents a fine-grained bus contention analysis for the 3-phase task model considering a round-robin bus arbitration policy. It also presents an algorithm to accurately quantify the blocking caused by lower priority tasks executing on the same core of the multicore platform under partitioned fixed-priority non-preemptive scheduling.

Chapter 5 discusses the holistic nature of shared resources and shows the interdependence between the number of cache misses and memory bus requests. We analyze the upper bound on the number of cache misses generated during the memory phases and integrate them into the bus contention analysis.

Part II focus on the third contribution and it consists of Chapter 6.

Chapter 6 presents the schedulability analysis of the proposed MCS. It also shows the impact of preemption point selection on the memory interference suffered by tasks.

Part III focus on the forth contribution and it consists of Chapter 7.

Chapter 7 presents the memory contention analysis for the 3-phase task model. It also shows how a tighter bound on the memory contention can be obtained by leveraging memory address mapping of tasks.

Finally, Chapter 8 concludes the dissertation and provides some future research directions.

Chapter 2

Background and Related Work

In this chapter, we introduce the relevant background concepts and existing works related to the problems addressed in this dissertation. Specifically, the relevant background concepts are presented in Section 2.1, and existing works that are related to problems addressed in this dissertation are discussed in Section 2.2.

2.1 Background

This section discusses the relevant background concepts that are important for understanding the work developed in this dissertation. We start by categorizing the real-time systems.

As discussed earlier, it is of extreme importance that tasks running on real-time systems must complete the required set of operations without violating their respective timing constraints. Depending on the type of real-time systems, the violation of timing constraints of tasks can cause some negative consequences. Depending on the level of negative consequences that may occur due to not meeting the timing constraints of tasks, the real-time systems can be broadly categorized as follows.

- ***Hard Real-Time Systems:*** Hard real-time systems are those systems in which violating the timing constraints of tasks can lead to catastrophic consequences. A common example is car airbag systems in which activating the airbag within a given time bound is essential; otherwise, it can cause serious negative consequences to the driver and passengers. Other applications of hard real-time systems include flight control systems, missile control systems, etc.

- ***Soft Real-Time Systems:*** Soft real-time systems are those systems in which violating the timing constraints of tasks does not have catastrophic consequences but can negatively impact the quality of service. A common example is multimedia applications in which violating the timing constraints of tasks may result in a lag in video/audio.

This dissertation focus on hard real-time systems.

Now we will discuss the characterization of tasks in the following section.

2.1.1 Task Characterization

A system can be composed of a set of tasks, usually denoted by Γ. A task τ_i is said to be *arrived* when it is ready to execute on the system. A task can typically execute several times on a system and each execution of the task is known as *job*. A task τ_i can be characterized by *minimum inter-arrival time* or *period* T_i that determines the time difference between the release of its two consecutive jobs. Based on the minimum inter-arrival time/period T_i, a task τ_i can be categorized into three different classes.

- ***Periodic Task:*** A task τ_i is said to be periodic if it releases a job exactly after every T_i time units, after a specified initial offset.
- ***Sporadic Task:*** A task τ_i is said to be sporadic if the time difference between the release of any of its consecutive jobs is at least T_i time units.
- ***Aperiodic Task:*** A task τ_i is said to be aperiodic when there are no timing constraints between the release of its any two consecutive jobs.

This dissertation assume tasks are sporadic.

Each job of a given task τ_i must complete its execution by a specified time bound known as *deadline* usually denoted by D_i. A task is said to meet its timing constraints if it completes its execution by its deadline; otherwise, a deadline miss occurs. Specifically, deadlines are associated with every job of the task and the task is said to miss its deadline if any of its jobs cannot complete its execution within the *relative deadline*. The relative deadline is the time difference between the deadline D_i and the release of a given job of task τ_i. Tasks deadlines can be classified into three types as follows.

- ***Implicit deadline task:*** A task τ_i is said to have implicit deadline if its deadline D_i is equal to its period/minimum inter-arrival time T_i, i.e., $D_i = T_i$.
- ***Constrained deadline task:*** A task τ_i is said to have a constrained deadline if its deadline D_i is less than or equal to its period/minimum inter-arrival time T_i, i.e., $D_i \leq T_i$.
- ***Arbitrary deadline task:*** When the deadline D_i does not have any specific relationship with T_i, i.e., D_i can be anywhere, the task τ_i is said to have a arbitrary deadline.

This dissertation considers tasks with constrained deadlines.

When k^{th} job of the task τ_i arrives in the system, it is referred to as the arrival time of k^{th} job of the task τ_i and is denoted by $a_{i,k}$. Similarly, when k^{th} job of the task τ_i starts executing on the

system, it is referred to as the start time of k^{th} job of the task τ_i and is denoted by $s_{i,k}$. Finally, when k^{th} job of the task τ_i finishes its execution, it is referred to as the finish time of k^{th} job of the task τ_i, denoted by $f_{i,k}$.

The time required to serve one job of a task τ_i on a given computing platform is referred to as the execution time of one job of task τ_i. From the worst-case timing behavior perspective, the task is characterized by its ***Worst-Case Execution Time*** (WCET) which is the maximum time taken by *any job* of task τ_i to complete its execution in isolation (i.e., running alone). The WCET of task τ_i is commonly denoted as C_i. Similarly, the response time R_i of a task τ_i, when it executes on a given platform along with a set of tasks, is defined as the time difference between the release and completion of a given job of task τ_i. From the worst-case timing behavior perspective, the expression ***Worst-Case Response Time*** (WCRT) is defined as the maximum time difference between the release and completion of *any job* of task τ_i, usually denoted by R_i^{max}, considering the concurrent execution of all tasks in the system.

Once the WCET C_i and the minimum inter-arrival time/period T_i of task τ_i is known, we can compute the *task utilization* U_i. The task utilization U_i represents the fraction of processor time spent executing task τ_i for C_i time units after every T_i time units. Formally, the task utilization U_i is given by $U_i = C_i/T_i$. Similarly, the *core utilization* represents the fraction of processor time spent executing the set of all tasks running on that core such that each task τ_i executes for C_i time units and releases a job after every T_i time units. Let Γ_p represent the taskset assigned to core π_p, the core utilization of π_p is given by $\sum_{\tau_i \in \Gamma_p} U_i$.

2.1.2 Task Scheduling

In computing systems, a *scheduler* is responsible for scheduling the set of tasks assigned to the system, i.e., responsible for deciding the order and time at which tasks will execute on the CPU. The behavior of the scheduler depends on the *scheduling policy*. Scheduling policies can be broadly categorized into *fixed-priority-based scheduling*, *dynamic priority-based scheduling*, and *offline scheduling*. In fixed-priority-based scheduling, a fixed priority is assigned to each task in the system at design time, and task priorities cannot change at run time. On the basis of the set of ready tasks and their priorities, tasks are scheduled on the CPU by the scheduler. As an example of fixed-priority-based scheduling, assume that there are two tasks τ_h and τ_i ready at time t such that the priority of τ_h is higher than that of τ_i. In this case, task τ_h will be scheduled as it is the highest priority task among all ready tasks. Fixed-priority scheduling can be further categorized into the following types.

- ***Fixed-priority preemptive scheduling:*** In fixed-priority preemptive scheduling, the currently executing task can be preempted at any time by a higher priority task. The preempted task then resumes its execution when there are no pending higher priority tasks.

- ***Fixed-priority non-preemptive scheduling:*** In fixed-priority non-preemptive scheduling, task preemptions are not allowed. Therefore, once a task starts its execution, it cannot be

preempted by any task released on the core. The scheduler can only schedule another task once the ongoing task completes its execution.

- ***Fixed-priority limited preemptive scheduling:*** In fixed-priority limited preemptive scheduling, the task execution is divided into preemptible and non-preemptible sections. A task can only be preempted by a higher priority task when it is executing in its preemptible section. This type of scheduling is useful when tasks can be preempted generally but require certain operations to be performed non-preemptively.

In all fixed priority-based scheduling approaches, task priorities must be assigned at the design time. The most common priority assignment algorithms are Rate Monotonic (RM) [Liu and Layland, 1973] and Deadline Monotonic (DM) [Audsley, 1990]. RM assigns task priorities based on task periods such that the shorter the task period, the higher the task priority. This means that tasks with shorter periods will be assigned higher priorities than tasks with longer periods. The RM algorithm is based on the observation that tasks with shorter periods have higher arrival rates and therefore need to be scheduled more frequently than tasks with longer periods. Similarly, DM assigns task priorities based on the relative deadline of tasks such that the shorter the deadline, the higher the task priority. This means that tasks with shorter deadlines will have higher priorities than tasks with longer deadlines.

In dynamic priority scheduling, the priority is assigned to tasks at run time based on various criteria. *Earliest Deadline First* (EDF) is one of the most common dynamic priority scheduling policies. In EDF, the scheduler checks all ready tasks and schedules a task with the earliest deadline. After the completion of the ongoing task, the next task to be executed will be one whose deadline is the earliest among all ready tasks. EDF can also be preemptive and non-preemptive.

Unlike fixed priority and dynamic priority scheduling, in offline scheduling, a static schedule is constructed at design time which defines the order and time on which each task will execute on the CPU. This type of scheduling is more predictable as the time and order of execution of all tasks are known beforehand. However, such type of scheduling may not be applicable in some scenarios, e.g., when tasks are of an event-triggered nature.

This dissertation mainly considers fixed-priority non-preemptive scheduling except Chapter 6 that considers fixed-priority limited preemptive scheduling.

2.1.3 Worst-Case Timing Analysis

As discussed earlier, the worst-case timing behavior of tasks is analyzed through the worst-case timing analysis. The first step to perform the worst-case timing analysis is the computation of the WCET of tasks. The WCET analysis for a given task is performed by considering the underlying computing platform and its architecture. Commonly, the bound on the WCET is typically computed in *isolation*, i.e., the task cannot be interfered by any other task and it has exclusive access to all required resources such as CPU, memory bus, caches, main memory, etc. There are different techniques to bound the WCET of tasks. These techniques can be broadly categorized into static

analysis, measurement-based analysis, or hybrid analysis [Wilhelm et al., 2008]. In the static analysis, the WCET is determined at the design time by taking into account different factors such as the program execution paths of the task, recursions, processor speed, pipeline mechanisms, memory access latency, etc. The upper bound on the WCET for a given task is computed by considering the maximum time required to execute any job of that task considering all possible execution paths, recursions, memory accesses, memory access latency, etc. In the measurement-based analysis, the task is executed on the system in isolation several times, i.e., thousands or even millions of times, and the WCET is derived by the distribution of the measurements. The hybrid WCET analysis uses both static and measurement-based techniques to derive the WCET of tasks. Having bounded the WCET of all tasks in the system, the next step is to perform the schedulability analysis as described in the subsequent subsection.

2.1.3.1 Schedulability Analysis

The *schedulability analysis* determines whether all tasks in the taskset can complete their execution without any deadline miss at run time. The schedulability analysis can be performed using different methods [Buttazzo, 2011], e.g., CPU utilization-based test, CPU demand-based test, WCRT analysis, etc. However, since we focus on fixed-priority scheduling, we use the traditional *WCRT-based schedulability analysis* [Joseph and Pandya, 1986, Lehoczky, 1990, Bril et al., 2007]. WCRT analysis of task τ_i determines the maximum time taken by any job of task τ_i from its release to completion on a given platform considering the given taskset. Having bounded the WCRT of a task, it is possible to determine whether the task is said to be *schedulable*. A task is schedulable only if its WCRT is less than or equal to its relative deadline. This implies that a task can execute on the system without missing its deadline considering all the possible scenarios. Similarly, the taskset is said to be schedulable if all tasks in the taskset are schedulable, i.e., each task can complete its execution without a deadline miss. To perform the WCRT analysis, the set of tasks, their priorities, WCET, minimum inter-arrival times/periods, computing system architecture, and the scheduling policy should be known in advance. We will now discuss the WCRT analysis for fixed-priority scheduling considering single-core processors.

WCRT analysis for fixed-priority preemptive scheduling: It has been proven in [Liu and Layland, 1973, Joseph and Pandya, 1986] that for fixed-priority preemptive scheduling of tasks with constrained deadlines and their priorities assigned using RM/DM, the WCRT is observed considering the *critical instant*. Specifically, critical instant is a scenario that maximizes the response time of the task under analysis. Considering fixed-priority preemptive scheduling in which tasks have constrained deadlines, the WCRT of τ_i is suffered by its first job initiated with the critical instant.

The response time of task τ_i is computed using the first positive solution on the fixed-point iteration of the following equation.

$$R_i = C_i + \sum_{\tau_h \in hep_i \setminus \tau_i} \left\lceil \frac{R_i}{T_h} \right\rceil \times C_h \tag{2.1}$$

where R_i is the response time of task τ_i; C_i is the WCET of task τ_i; C_h is the WCET of task τ_h; T_h is the minimum inter-arrival time of task τ_h; and hep_i is the set of all tasks with a priority higher or equal to than that of τ_i.

Due to fixed-priority preemptive scheduling, task τ_i can suffer interference due to the execution of all jobs released by a task τ_h which has higher or equal priority (except τ_i itself) than that of τ_i. The interference caused by τ_h is maximized when it releases a job every T_h time units and every job executes for C_h time units. During any time window of length R_i, task τ_h can release at most $\left\lceil \frac{R_i}{T_h} \right\rceil$ jobs. Thus, $\left\lceil \frac{R_i}{T_h} \right\rceil \times C_h$ maximizes the interference caused by task τ_h on task τ_i during a time window of length R_i. Extending this to all tasks in hep_i except task τ_i, the maximum interference that can be suffered by task τ_i is upper bounded by $\sum_{\tau_h \in hep_i \setminus \tau_i} \left\lceil \frac{R_i}{T_h} \right\rceil \times C_h$. Finally, C_i is added in Equation 2.1 to consider the contribution of the WCET of τ_i when computing its response time. Due to preemptive scheduling, any lower priority task cannot affect the response time of τ_i.

Note that R_i appears on both sides of Equation 2.1 which means Equation 2.1 is recursive and a fixed-point computation on R_i can be used to find a solution by initiating $R_i = C_i + \sum_{\tau_h \in hep_i \setminus \tau_i} C_h$. The computation of R_i is stopped if $R_i > D_i$ which means that the response time of τ_i is greater than its relative deadline, thus, the task τ_i is not schedulable.

Since the WCRT R_i^{max} of τ_i is suffered by its first job under fixed-priority preemptive scheduling, the WCRT of τ_i is also given by R_i, i.e., $R_i^{max} = R_i$.

WCRT analysis for fixed-priority non-preemptive scheduling: For the fixed-priority non-preemptive scheduling of tasks with constrained deadlines and their priorities assigned using RM/DM, it has been proven in [Bril et al., 2007] that the WCRT can be suffered by any job of task τ_i that executes during the longest *level-i busy window*. The definition of the level-i window is given as follows:

Definition 2.1. [Level-i busy window (from [Lehoczky, 1990])] A level-i busy window is a time interval (a,b) in which the pending workload of tasks with priorities higher or equal to that of task τ_i is positive for all $t \in (a,b)$ and 0 at the boundaries a and b.

Therefore, to compute the WCRT of task τ_i, the first step is to compute the length of the level-i busy window. We will now discuss the formal computation of the length of the level-i busy window.

The length of the level-i busy window is denoted by W_i and is given by the first positive solution to the fixed-point iteration of the following equation.

$$W_i = C_{lp_i}^{max} + \sum_{\tau_h \in hep_i} \left\lceil \frac{W_i}{T_h} \right\rceil \times C_h \tag{2.2}$$

where W_i is the length of the level-i busy window; C_h is the WCET of task τ_h; T_h is the minimum inter-arrival time of task τ_h; hep_i is set of all tasks with a priority higher than or equal (including τ_i) to that of τ_i; and lp_i is set of all tasks with a priority lower than that of τ_i.

Due to fixed-priority non-preemptive scheduling, task τ_i can be delayed if a lower priority task τ_j started its execution before the release of τ_i. Since we cannot estimate the specific lower

priority task that will cause blocking, the blocking is maximized by considering a lower priority task with the maximum WCET among all tasks in lp_i, denoted by $C_{lp_i}^{max}$, and given by $C_{lp_i}^{max} = \max_{\tau_j \in lp_i}\{C_j\}$. Furthermore, to maximize the length of the level-i busy window, we need to account for the workload generated by all tasks in hep_i (which includes τ_i). The workload generated by all tasks in hep_i is maximized by considering that every task $\tau_h \in hep_i$ releases a job every T_h time units and each job of τ_h executes for C_h time units. Hence, the maximum workload that can be generated by all tasks in hep_i within the level-i busy window is upper-bounded by $\sum_{\tau_h \in hep_i} \left\lceil \frac{W_i}{T_h} \right\rceil \times C_h$.

Note that W_i appears on both sides of Equation 2.2 which means Equation 2.2 is recursive and a fixed-point computation on W_i can be used to find a solution by initiating $W_i = C_{lp_i}^{max} + \sum_{\tau_h \in hep_i} C_h$.

Having bounded the length of the level-i busy window W_i, the next step is to analyze the *maximum number of jobs* that task τ_i can execute within the level-i busy window W_i. The maximum number of jobs that can be executed by task τ_i within the level-i busy window W_i is denoted by K_i, where K_i is given by the following equation.

$$K_i = \left\lceil \frac{W_i}{T_i} \right\rceil \tag{2.3}$$

Having bounded the maximum number of jobs that task τ_i can execute within the level-i busy window K_i, we can compute the response time of task τ_i. To compute the response time of the k^{th} job of task τ_i, we first need to compute its *latest start time*, as it cannot be preempted or delayed by any other task once it starts its execution.

Let $\tau_{i,k}$ denote the k^{th} job of task τ_i executing during W_i, then the latest start time of $\tau_{i,k}$ is denoted by $s_{i,k}$ and is given by the first positive solution to the fixed-point iteration of the following equation.

$$s_{i,k} = C_{lp_i}^{max} + (k-1) \times C_i + \sum_{\tau_h \in hep_i \setminus \tau_i} \left\lceil \frac{s_{i,k}}{T_h} \right\rceil \times C_h \tag{2.4}$$

The start time of $\tau_{i,k}$ can be delayed due to the blocking caused by a lower priority task. The blocking is maximized by $C_{lp_i}^{max}$ and can be computed similarly to that in Equation 2.2. Furthermore, previous jobs of the task τ_i can delay the start of $\tau_{i,k}$. Thus, $(k-1) \times C_i$ upper bounds the contribution of k-1 jobs of τ_i. Finally, the maximum interference that can be caused by a higher priority task τ_h during W_i is upper bounded by $\left\lceil \frac{s_{i,k}}{T_h} \right\rceil \times C_h$. Extending this to all tasks in hep_i except τ_i, the maximum interference that can be suffered by $\tau_{i,k}$ is upper bounded by $\sum_{\tau_h \in hep_i \setminus \tau_i} \left\lceil \frac{s_{i,k}}{T_h} \right\rceil \times C_h$.

In Equation 2.4, $s_{i,k}$ appears on both sides which means that Equation 2.4 is recursive and a fixed-point computation on $s_{i,k}$ can be used to find a solution by initiating $s_{i,k} = C_{lp_i}^{max} + C_i + \sum_{\tau_h \in hep_i \setminus \tau_i} C_h$. The computation of $s_{i,k}$ is stopped if $s_{i,k} > D_i \times k$ which means that the latest start time of $\tau_{i,k}$ is greater than its relative deadline, thus, task τ_i is not schedulable.

Having bounded the latest start time of $s_{i,k}$, the response time $R_{i,k}$ of $\tau_{i,k}$ is given by the following equation.

$$R_{i,k} = s_{i,k} + C_i - (k-1) \times T_i \tag{2.5}$$

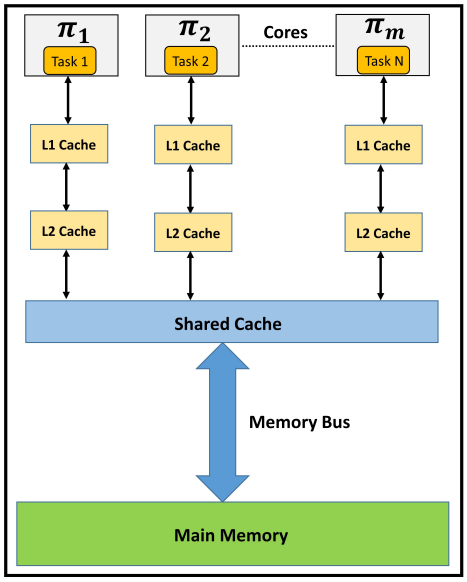


Figure 2.1: Memory Hierarchy in COTS multicore platform

Finally, the WCRT of task τ_i can be computed by maximizing Equation 2.5 over all the jobs of τ_i that can execute within the level-i busy window, i.e., from 1 to K_i

$$R_i^{max} = \max_{k \in [1,K_i]} \{R_{i,k}\} \tag{2.6}$$

For the ***fixed-priority limited-preemptive scheduling***, the WCRT analysis of tasks can be performed using the same procedure described for fixed-priority non-preemptive scheduling [Bril et al., 2007]. However, the blocking caused by a lower priority task τ_j to task τ_i can be computed by considering the longest non-preemptible section of τ_j instead of its WCET. Furthermore, task τ_i can suffer interference from higher priority tasks until the completion of its last preemptible section.

For all the fixed-priority scheduling explained above, a taskset Γ is said to be schedulable only if the WCRT R_i^{max} of each task τ_i in the taskset is less than or equal to its relative deadline D_i, and the utilization of the core is less than or equal to the core's capacity, i.e., 1.

2.1.4 Hardware Platform Characterization

In this section, we discuss the characterization of the hardware platform. Specifically, we will discuss the platform characterization of multicore systems. The memory hierarchy in the COTS multicore platform is shown in Figure 2.1. In the following sections, we will discuss in detail various hardware components in multicore platforms[1]. We start by discussing processing *cores*, which is one of the most important parts since they are the fundamental components of the system.

[1]Note that there can be many different components in multicore systems depending on the platform but we mainly discuss the components that are relevant in the context of this dissertation.

2.1.4.1 Processors

A processor or processing core is the Central Processing Unit (CPU), i.e., the main electronic circuitry within the system, that processes and executes tasks. As per the given instructions of the task, the core can perform various operations such as arithmetic, logical, controlling, load/store from/to the memory, I/O operations, etc. In single-core processors, there is one core in the system, and all the resources such as caches, interconnects, memory, I/O, etc. are accessible only by that core. In multicore processors, there can be multiple cores fabricated on the same chip such that each core is independent and can execute a set of tasks running on it. As discussed earlier, in multicore processors, the hardware resources such as caches, memory bus, main memory, I/Os, etc. are shared among multiple cores running in the system. This implies that a given core may not have exclusive access to all the shared resources.

Multicore systems can be further classified as *homogeneous* or *heterogeneous*. In the homogeneous multicore platforms, the computing capacity of all the cores in the system is similar, i.e., architecture, microarchitecture, frequency, etc. of all the cores are identical. This implies that the WCET of a task computed on one core is also valid for any other core in the system. A common example of a homogeneous multicore platform is the P4080 platform [P4080, 2011] which comprises eight high-performance e500mc cores. In heterogeneous systems, the computing capacity of all the cores may not be the same. Thus, the bound on the task's WCET computed on one core may not be valid for another core in the system. A common example of a heterogeneous multicore platform is Xilinx's Zynq Ultrascale+ [Boppana et al., 2015] platform that comprises four Arm Cortex-A53 cores and two Arm Cortex-R5F cores.

This dissertation considers homogeneous multicore platforms.[2]

As discussed earlier, in uniprocessor scheduling every job of a task executes on the same core. However, in multiprocessor scheduling, a task can potentially execute its jobs on different cores on the basis of the availability of the cores. This phenomenon of shifting tasks to different cores at run time is known as *migration*. Depending on whether task migration is allowed or not, multiprocessor scheduling can be broadly categorized into the following types [Davis and Burns, 2011].

- ***Partitioned scheduling:*** In partitioned scheduling, tasks are assigned to cores at design time, and tasks assigned to each core are not allowed to migrate to any other core in the system.

- ***Global scheduling:*** In global scheduling, all tasks in the system are kept in a global queue and can be scheduled on any processor that is available. Upon preemption, a task can migrate to another core to resume its execution.

[2] Although this dissertation assumes multicore platform with identical cores, the work developed in this dissertation also holds even if cores are not identical, i.e., cores offer different performance.

- ***Semi-partitioned scheduling:*** Semi-partitioned scheduling is a mix of both global and partitioned scheduling. Specifically, a subset of tasks are assigned to cores and are not allowed to migrate to other cores whereas the rest of the task can be scheduled on any core in the system.

This dissertation focuses on the partitioned scheduling.

2.1.4.2 Cache Memories

Memories are essential hardware components of an embedded system to store the data/instructions required by tasks to perform any operation. An embedded system typically has a hierarchy of memories. The memory hierarchy, type, and size depend on the given hardware platform but commonly embedded systems have off-chip Random Access Memories (RAM), Read Only Memory (ROM), and cache memories. We will now briefly explain the cache memories and their organization.

Cache memories are crucial in enhancing the performance and efficiency of multicore platforms. In a multicore platform, cache memories offer high-speed storage for frequently accessed data and instructions with low latency. This feature can significantly reduce the time spent waiting for data to be retrieved from the main memory, which can be time-consuming. The cache can be *independent*, i.e., Data (D) cache, and Instruction (I) cache, or *unified*, i.e., the cache can hold both data and instructions. Cache memories come in various sizes and levels, with each level serving a different purpose. The first level cache, known as the L1 cache, is the smallest and fastest cache memory. L1 cache works as the local memory of a core since each core in the system can have its own L1 cache. The second level cache or L2 cache is larger and slower than the L1 cache but still provides faster access than the L3 cache. Typically, each core has its own L2 cache. Multicore platforms can have a third-level cache, also known as the Last-Level Cache (LLC), which is shared among all the cores in the system. Furthermore, cache memories that are shared among multiple cores, i.e., LLC, are usually partitioned among cores such that each core has its non-overlapping cache partition. The cache partitioning can be achieved by means of hardware features, e.g., Intel's Cache Allocation Technology [Shivappa., 2014], ARM's Lockdown by master [PL310, 2008], or by software techniques such as cache coloring [Kessler and Hill, 1992, Liedtke et al., 1997].

In hardware platforms, the unit for cache access is known as *cache line*. Caches are usually partitioned into different sets of equal size and are called *cache set*. A cache set may comprise a single cache line or multiple cache lines. To transfer data from the main memory to the cache, the system breaks the information into *memory blocks*. These blocks represent the smallest units of data that can be loaded from the memory at a time. The first step is to map each memory block onto a cache set; once a block is mapped, it is then placed into one of the cache lines within that set. The number of memory blocks that can be stored in each cache set is referred to as the *associativity* of the cache and such a cache is called *set-associative* cache. The set associative cache can be broadly categorized into the following types.

- ***Direct mapped cache:*** In the direct mapped cache, the associativity is 1 which means that each cache set consists of a single cache line. This implies that a memory block can reside in exactly one line.

- ***Fully-associative cache:*** In the fully-associative cache, the associativity is equal to the number of sets in the cache. This implies that a memory block can reside in any cache line.

Whenever a task needs to access the required data/instructions, it first accesses cache memories. If the required data/instructions are available in the cache memory, it results in *cache hit* and the task can execute using the data/instructions available in the caches without the need to access the main memory. On the contrary, if the required data/instructions are not available in the cache memory, it results in *cache miss*. Upon a cache miss, the data/instructions are fetched from the next level of cache, if exists. If the data/instructions are not available in the LLC, the main memory request is initiated to fetch the data/instructions from the main memory.

Content in the cache memories should be consistent with the main memory. This is typically achieved through the *write policies* of the cache. The write policies of the cache memories can be broadly categorized as follows.

- ***Write-through cache:*** In the write-through cache policy, whenever a cache line is updated, the corresponding data is updated to the main memory at the same time.

- ***Write-back cache:*** In the write-back cache policy, a cache line is only written to the main memory if that cache line is *evicted*, i.e., the cache line is being modified by updating the cache line with new data.

Furthermore, the *cache replacement policy* determines the order in which the existing content on cache lines will be replaced by new data of tasks. Some of the most common cache replacement policies are *random*, and *Least Recently Used* (LRU). In the random cache replacement policy, the cache line can be randomly replaced. This implies that it may not be possible to predict the pattern in which the cache will replace the existing content to make space for the new content. In the case of the LRU cache replacement policy, the cache checks the *age* of cache lines, i.e., the time of its last access. Then the cache lines with the highest age, i.e., cache lines that were accessed older than all the other cache lines, are evicted to store the new data in the cache lines.

This dissertation considers direct mapped cache that uses write-back cache policy. This dissertation is valid for a single level of cache memory as well as multiple levels of caches with a non-inclusive policy. The shared cache is assumed to be partitioned among each core such that each core has its non-overlapping cache partition.

2.1.4.3 Memory Bus

The *memory bus* is mainly responsible for connecting the cores with the main memory. In other words, the memory bus is the communication channel that accesses the required data/instructions

from the main memory and stores them in the cache memories. The term *system bus* or *Front-Side Bus* (FSB) [Dasari et al., 2013] is also used to denote the memory bus. As the memory bus is a communication channel, it can be characterized by its *bandwidth*, which determines the amount of data that the bus can transfer per time unit.

Depending on the type of the required memory operation, i.e., read or write, the memory bus performs *bus transactions*. In the case of a read request, the given core sends the memory request via the memory bus to access the memory block from the requested address locations of the main memory. The memory bus then forwards this request to the main memory. Once the requested memory block is served by the main memory, the memory bus transfers the required data to the cache memories of the core, i.e., local cache, and shared cache partition. In case of a write request, the given core transfers the data to the main memory via the memory bus by specifying the required memory addresses of the main memory. Bus transactions can be performed in several different ways [Dasari, 2014]. We broadly divide them into two categories as follows.

- ***Atomic transaction:*** An atomic bus transaction is modeled as an invisible pair of request-reply transactions which means that once the bus starts serving a request transaction, it remains busy until the response transaction. In other words, a bus remains busy until all requested memory blocks have been transferred from the main memory to the core's local memory to serve a given memory request. This implies that the bus accepts any subsequent request only after the completion of the previous memory request. This ensures that at most one request arrives at the main memory at a time. Although atomic bus transactions facilitate simple and predictable operations, i.e., by ensuring that at most one memory request is pending at the main memory at a time, memory bus bandwidth can be underutilized, i.e., memory bus remains busy even when the bus is not transferring data.

- ***Split transaction:*** In the split bus transaction, a bus request is modeled as two different transactions 1) the bus request transaction, i.e., request initiated from the core to access the memory bus, and 2) the bus response transaction, i.e., to serve the required data/code from the main memory to the requested core via the memory bus. Specifically, the bus remains busy only while performing a transaction, and can accept a new request once the current transaction is completed. As a consequence, once the bus completes a bus request transaction for a given core, it can perform bus request transactions for other cores even if the response transactions of previous requests are pending. This can efficiently utilize the bus but can decrease the predictability since there can be multiple pending memory requests at the main memory, and a request can be reordered by the main memory based on its arbitration policies.

This dissertation considers a single-channel shared memory bus that connects all the cores to the main memory and uses atomic bus transaction protocol. This dissertation considers the FCFS and RR bus arbitration policies.

Bus Arbitration Policy: In the memory bus, the bus arbitration policy is responsible for determining the order in which memory requests from all the cores will be served by the memory bus. Consequently, the memory requests from cores can be reordered by the bus depending on its bus arbitration policy. The following are the most commonly used bus arbitration policies in COTS multicore platforms [Davis et al., 2017].

- ***Time Division Multiple Access (TDMA):*** In the TDMA bus arbitration policy, a static *bus access slot* or *bus slot* is assigned to each core at the design time. A given core can only access the bus during its assigned slot. TDMA is a non-work-conserving policy such that each core reserves its bus slot disregarding the bus requests issued by tasks on that core. Consequently, TDMA is a predictable bus arbitration policy but can potentially underutilize the bus.

- ***Round-Robin (RR):*** RR policy also uses the notion of bus slots. However, the bus slots are not statically assigned to cores. Specifically, when a core issues bus requests and if there are no pending bus requests from other cores, the bus will start serving the bus requests of the core until the maximum capacity, i.e., bus slot size. At this point in time, the bus will serve the requests from all the other cores with pending bus requests such that each core can access the bus for no longer than the size of the bus slot. However, if other cores do not have any pending requests, the bus will continue serving the bus requests of the same core until there is a pending request from another core. This implies that RR is a work-conserving policy such that 1) a core does not reserve the bus if there is no pending bus request; and 2) a core releases the bus if the time to serve all its pending bus requests is less than the bus slot size. Furthermore, RR is a fairer bus arbitration policy because the bus slot size is the same for each core in the system resulting in fairly allocating the memory bus among all cores.

- ***First-Come First-Served (FCFS):*** In the FCFS bus arbitration policy, there is a *global queue* in which bus requests from all cores are enqueued. The global queue is sorted in first-in first-out (FIFO) order. This implies that older requests are prioritized by the bus over newer bus requests.

- ***Fixed Processor Priority (FPP):*** In the FPP bus arbitration policy, a *fixed priority* is assigned to *each core* in the system at the design time which cannot change at the run time. The bus requests issued by tasks that execute on higher priority cores are prioritized by the bus over the bus requests issued by tasks that execute on lower priority cores.

- ***Fixed Task Priority (FTP):*** In the FTP bus arbitration policy, a *fixed global priority* is assigned to *each task* in the system at the design time which cannot change at the run time. The bus requests issued by higher priority tasks are prioritized by the bus over the bus requests issued by lower priority tasks.

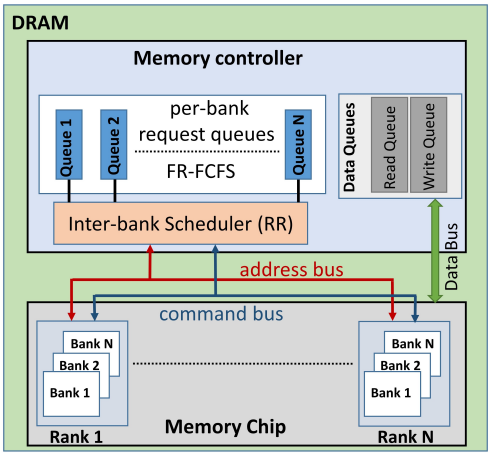


Figure 2.2: Organization of DRAM

2.1.4.4 Main Memory

The main memory is global memory which can be accessed by all the cores in the system. The main memory can store the data required by all tasks in the system. Furthermore, the main memory is the largest in terms of size compared to the cache memories. However, the main memory access times can be much slower than those of cache memories. COTS multicore processors typically use Dynamic Random Access Memory (DRAM).

As shown in Figure 2.2, the DRAM is composed of 1) *memory controller* that is responsible for determining the order in which the memory requests from all the cores will be served; 2) *memory chip*, an array of memory cells that stores the data required by tasks; 3) *command bus*, the interconnect through which memory controller issues all commands to the memory chip, and the *data bus*, the interconnect through which data is transferred from/to memory controller to/from the memory chip.

The DRAM is organized into multiple *ranks* in which each rank is composed of multiple *banks*. Each bank is further composed of rows and columns that store the data. Each memory bank has a *row buffer* that stores the data accessed during the most recent access to that bank. Furthermore, a row of the bank is said to be *activated* if it was accessed during the most recent access to that bank. If a memory request targets the same row of the bank as the activated row, it results in a *row-hit* as the data is available in the row buffer and can directly be accessed from the activated row. On the contrary, if a memory request targets a row that is different from the one that is activated, it results in a *row-miss*.

In DRAM, typically there are three commands that are issued during a memory operation. These commands are: 1) PRE (PREcharge) command that moves back the current content of the row buffer to its corresponding row in the DRAM bank; 2) ACT (ACTivate) command that activates the requested row of the bank; and 3) CAS command that performs the intended read/write operation on the activated row.

Based on the state of the row buffer and the requested row of the bank, the following are the possible sequence of commands that the memory controller can issue to serve a single memory

request [Kim et al., 2014, Yun et al., 2015, Hassan and Pellizzoni, 2018]:

1. If the requested row is the same as the activated row, i.e., row-hit, then only the CAS command is issued to perform the intended read/write operation on the activated row.

2. If the bank does not have an activated row, i.e., the row buffer is empty, then the memory controller first issues the ACT command to activate the requested row followed by the CAS command to perform the intended read/write operation on the activated row.

3. If the bank has an activated row different from the requested row, then the memory controller first issues the PRE command to move back the current content of the row buffer to its corresponding row in the bank. The ACT command is then issued to activate the requested row. Finally, the CAS command is issued to perform the intended read/write operation on the activated row.

Each of the commands mentioned above is issued according to the JEDEC standard [JEDEC, 2008] which defines all the timing constraints that need to be satisfied while performing a memory operation on the DRAM. As shown in Table 2.1, the JEDEC timing constraints are divided into *intra-bank* timing constraints and *inter-bank* timing constraints. The intra-bank timing constraints can be defined as timing constraints applied between the commands issued to the same bank. The inter-bank timing constraints can be defined as timing constraints applied between commands of the same type (PRE, ACT, or CAS) issued to any bank.

Parameters	**Description**	**Cycles**
Intra-bank constraints		
tRCD	ACT to CAS delay	9
tRL	RD to Data Start	9
tRP	PRE to ACT delay	9
tWL	WR to Data Start	8
tRAS	ACT to PRE delay	24
tRC	ACT to ACT (same bank)	33
tWR	Data End of WR to PRE	10
tRTP	Read to PRE delay	5
Inter-bank constraints		
tCCD	CAS to CAS delay	4
tRTW	RD to WR delay	6
tWTR	WR to RD delay	5
tRRD	ACT to ACT (different bank in same rank)	4
tB	Data bus transfer	4
tFAW	Four bank activation window	20

Table 2.1: JEDEC timing constraints for DDR3-1333H [JEDEC, 2008].

The memory controller determines the order in which memory requests from all the cores will be served by the DRAM. The memory controller issues all the commands to the memory chip to perform the required operation. These commands are communicated from the memory controller

to the memory chip through the shared command bus that connects the memory controller to all the banks. The requested data is served via the shared data bus that connects the memory controller to all the banks. The memory requests targeting each bank are enqueued into *per-bank queues*. Typically, the memory controller of COTS multicore processors uses *open-row* policy to improve the overall throughput of the requests. In the open-row policy, the row-hit memory requests are prioritized over row-miss requests. To achieve this goal, COTS multicore systems typically use the *FR-FCFS* (First-Ready First-Come-First-Served) policy [Rixner et al., 2000, Nesbit et al., 2006, Kim et al., 2014, Hassan and Pellizzoni, 2018, Casini et al., 2020, Hassan and Pellizzoni, 2020] to sort each per-bank queue which prioritizes 1) row hits over row misses; and 2) older requests over newer requests. Furthermore, the FR-FCFS is often implemented with *thresholding* [Kim et al., 2014, Hassan and Patel, 2016, Hassan and Patel, 2018, Hassan and Pellizzoni, 2018] that defines an upper bound on the maximum number of new requests that can delay an older request due to request reordering when accessing the same bank. Each per-bank queue is then exposed to the *inter-bank scheduler* that is responsible for scheduling memory requests from all the per-bank queues. Typically, the inter-bank scheduling policy is *RR (Round-Robin)* [Yun et al., 2014, Hassan and Pellizzoni, 2018, Casini et al., 2020] such that the inter-bank scheduler serves one request per turn from each per-bank queue.

COTS multicore processors typically use *write batching* to prioritize read memory requests over write memory requests since writes do not stall the processor pipeline [Yun et al., 2015, Hassan and Pellizzoni, 2018, Casini et al., 2020, Hassan and Pellizzoni, 2020]. The write requests are then served in batches [Chatterjee et al., 2012] to improve the turnaround time of the data bus as serving a set of memory requests of the same type is more efficient [Ecco and Ernst, 2017]. For example, if a write memory request is served after the read request, the tRTW inter-bank timing constraint is applicable. The most common method of implementing write batching is the *watermarking technique* [Chatterjee et al., 2012]. Specifically, when there are pending read requests, write requests can only be served by the memory controller if the number of write requests in the write buffer exceeds the *watermarking threshold*, and then at least one batch of write requests will be served by the memory controller.

2.1.5 Phased Execution Models

The notion of *phased execution models* [Pellizzoni et al., 2011, Durrieu et al., 2014] was introduced to reduce the temporal unpredictability posed by shared resource accesses of tasks executing on multicore systems. The main idea is to divide the execution of each task into distinct *computation phase* and *memory phase(s)* so that a task only accesses the shared resources, i.e., memory bus/main memory, during its memory phase; and no main memory accesses are generated by the task during its computation phase. Since tasks may not be compatible with this model, re-factoring tasks' code may be required to generate phased execution model-compatible tasks.

2.1.5.1 PRedictable Execution Model (PREM)

To transform the idea of phased execution models into reality, the *PRedictable Execution Model* (PREM) [Pellizzoni et al., 2011] was introduced. In the PREM, the execution of each task is divided into two intervals, namely, *predictable interval* and *compatible interval*. In the predictable interval, the execution of the task is divided into two phases, namely *Memory* (M) and *Execution* (E). During the memory phase, a task prefetches all its data/instructions from the main memory and stores it in the core's local memory (e.g., L1 cache). Note that a task may also need to write back some data from the cache to the main memory during the memory phase. For instance, if the set of cache lines are dirty on which the task will load the new data, the task first writes back the dirty cache lines to the main memory and then loads the new data on those cache lines. After the completion of the memory phase, all the data/instructions required for the task execution are now loaded into the core's local memory. At this point in time, the core executes the execution phase of the task using the preloaded data in the core's local memory without the need to access the main memory. Note that each phase and complete predictable interval executes non-preemptively. Once the predictable interval is completed, tasks can execute their compatible interval in which tasks can access shared resources at any time, and code refactoring is not required. Although the concept of the PREM was initially introduced for single-core processors [Pellizzoni et al., 2011], it was leveraged by a plethora of works for multiprocessor scheduling [Alhammad and Pellizzoni, 2014b, Alhammad and Pellizzoni, 2014a, Yao et al., 2012, Yao et al., 2016a, Wasly and Pellizzoni, 2014, Melani et al., 2015, Soliman and Pellizzoni, 2019, Schuh et al., 2020, Schwäricke et al., 2020, Rashid et al., 2022, Senoussaoui et al., 2022b, Senoussaoui et al., 2022a]. These works leveraged the PREM model such that a task execution is equivalent to the predictable interval, i.e., each task is divided into M and E-phases. Leveraging the PREM, a system-level offline schedule can be constructed using M and E-phases to eliminate shared resource contention. An example is shown in Figure 2.3 in which the system has two cores, i.e., Core 1 and Core 2, and two PREM tasks are mapped to each core, i.e., task τ_h and τ_i on Core 1 and task τ_u and τ_k on Core 2. An offline/time-triggered schedule is built using PREM tasks such that at most one task executes its memory phase, i.e., M-phase, at a time. Due to scheduling PREM tasks in such a manner, tasks

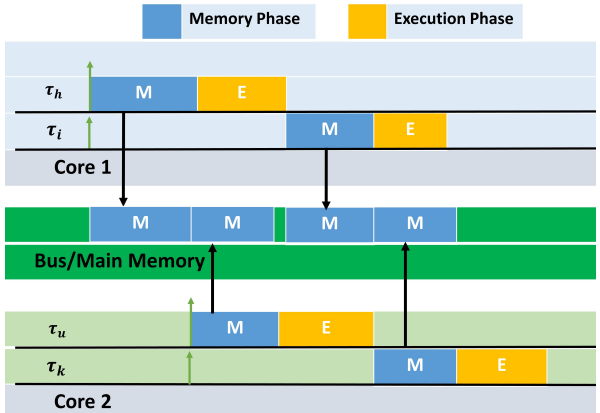


Figure 2.3: Contention-free system level offline schedule for PREM tasks

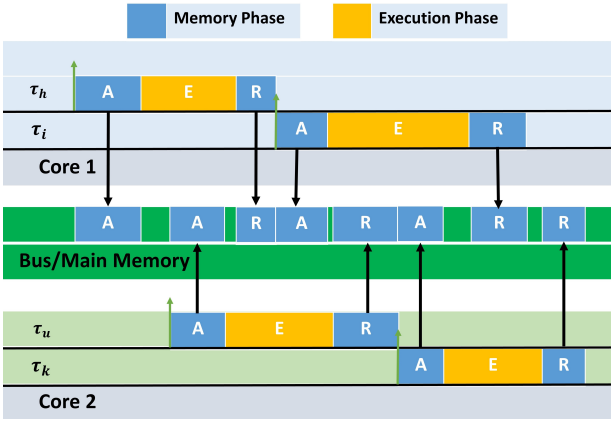


Figure 2.4: Contention-free system level offline schedule for 3-phase tasks

do not suffer any shared resource contention.

2.1.5.2 3-Phase Task Model

The concept of the PREM was then generalized to the *3-phase task execution model* [Durrieu et al., 2014] in which task execution is divided into three phases, namely, *Acquisition* (A), *Execution* (E), and *Restitution* (R). The 3-phase task model is also known as the AER model, the Read-Execute-Write (REW) model or the 3-phase PREM. In the 3-phase task model, the A- and R-phases are considered memory phases, i.e., the time intervals in which the task can fetch and write-back data from/to the main memory via the memory bus, and the E-phase is the computation phase, i.e., the time interval in which the task only performs computations using the preloaded data and does not issue any main memory request. When a task is released, it executes its A-phase to prefetch all the required data/instructions from the main memory and store it in the core's local memory, e.g., L1/L2 cache. It then executes its E-phase by accessing the data/instructions that are preloaded in the core's local memory, without the need to access bus/main memory. Finally, the task writes the modified data back to the main memory during the R-phase. The task is said to be completed at the completion of its R-phase. Performing the R-phase at the end of the task to write back all the modified data can be useful for the purposes of synchronization, inter-core communication, and data dependencies. Note that this is slightly different than the PREM in which the task performs write-backs and memory prefetches during a single memory phase and the task is considered to be completed at the end of the E-phase. Consequently, a plethora of works has leveraged the 3-phase task model for multiprocessor scheduling [Durrieu et al., 2014, Becker et al., 2016, Maia et al., 2016, Tabish et al., 2016, Maia et al., 2017, Rouxel et al., 2017, Pagetti et al., 2018, Tabish et al., 2019, Soliman et al., 2019, Koike et al., 2020, Rouxel et al., 2019, Rivas et al., 2019, Casini et al., 2020, Schuh et al., 2020, Thilakasiri and Becker, 2023a, Kloda et al., 2023, Tabish et al., 2023, Thilakasiri and Becker, 2023b]. Similarly to the PREM, the 3-phase task model can also be used to build system-level offline schedule such that at most one task on the system executes its memory phase at a time. An example is shown in Figure 2.4 in which the system has two cores, i.e., Core 1 and Core 2, and two 3-phase tasks are mapped to each core, i.e., task τ_h and τ_i on Core

1 and task τ_u and τ_k on Core 2. An offline/time-triggered schedule is built using 3-phase tasks such that at most one task executes its memory phase, i.e., A- or R-phase, at a time. Due to scheduling 3-phase tasks in such a manner, tasks do not suffer any shared resource contention.

2.1.5.3 Tools for Generating Phased Execution Model Compatible Tasks

Generating tasks that comply with phased execution models may require refactoring of tasks code. The initial work on PREM [Pellizzoni et al., 2011] generated PREM-compatible tasks manually, i.e., through the efforts and expertise of the programmer. Thus, the complexity of converting legacy code to PREM-compliant code can increase with the size of the code and the type of application. Consequently, the development cycle can be complex and time-consuming, due to the time and complexity involved in converting legacy code to PREM-compliant code. To solve this issue, several tools [Mancuso et al., 2014, Matějka et al., 2018, Fort and Forget, 2019, Forsberg et al., 2018, Soliman and Pellizzoni, 2019, Soliman et al., 2019, Schuh et al., 2020, Forsberg et al., 2021] were proposed by the real-time systems research community to automatically convert legacy code to PREM/3-phase task model compliant code. Thanks to the availability of these toolsets, it is possible to automatically convert legacy code to generate tasks that are compatible with phased execution models. We will now briefly discuss some of the works in this direction.

Mancuso et al. [Mancuso et al., 2014] proposed Light-PREM, which is a tool that automatically refactors legacy applications and transforms them into PREM-compatible code. Specifically, Light-PREM relies on memory profiling, i.e., memory access tracing, to construct the memory phases. Furthermore, Light-PREM is compiler-independent and is implemented at the source code level. From a high-level overview, Light-PREM performs the following steps: 1) access collection; 2) chunk detection; 3) handle detection; 4) graph construction; 5) relative expression construction; and 6) prefetch aggregation.

Matějka et al. [Matějka et al., 2018] proposed a compiler-based tool to convert legacy code into PREM-compatible code. Specifically, their approach is based on the LLVM infrastructure [Lattner and Adve, 2004] and automatically converts C/C++ code into PREM-compatible code. In their approach, the compiler performs several passes, i.e., identifying portions of the code, splitting them into multiple predictable intervals based on the size of the core's local memory, generating code for prefetch and writeback phases, etc., in order to generate the PREM-compliant code automatically.

Soliman et al. [Soliman and Pellizzoni, 2019] proposed a framework to automatically generate code compatible with the PREM model. Their solution is largely agnostic to the programming language being used since it is based on the intermediate representation of the LLVM compiler infrastructure [Lattner and Adve, 2004]. Specifically, their framework can generate PREM compatible *segments* such that each segment is composed of one memory phase and one execution phase. The segmentation can be useful when the local memory of the core is not large enough to store all the data required during the memory phase of a task. Consequently, PREM tasks can be segmented such that a task comprises multiple PREM segments, and the core's local memory is large enough to store the data required by any segment of the task. The authors then extended

their work to generate 3-phase tasks compatible segments in [Soliman et al., 2019]. Similarly, a few other approaches were proposed with the goal of generating tasks compatible with the phased execution models [Fort and Forget, 2019, Forsberg et al., 2018, Forsberg et al., 2021].

2.2 Related Work

In this section, we will discuss some of the existing works that are related to the problems addressed in this dissertation. This section is divided into two subsections in which Section 2.2.1 discusses the related works considering the *generic task model*, i.e., memory accesses are allowed anytime during task execution, and Section 2.2.2 discusses the related work considering *phased execution models*, i.e., memory accesses are allowed only during the memory phase(s) of tasks.

2.2.1 Related Work for the Generic Task Model

The problem of temporal unpredictability posed by the shared resource contention in multicore systems is not new and is a well-known problem (see surveys [Maiza et al., 2019] and [Lugo et al., 2022]). Consequently, a plethora of works have focused on the problem of shared resource contention in multicore systems considering the *generic task model* [Schranzhofer et al., 2010, Andersson et al., 2010, Schliecker and Ernst, 2010, Rosen et al., 2007, Chattopadhyay et al., 2010, Chattopadhyay and Roychoudhury, 2011, Kelter et al., 2011, Kelter et al., 2014, Dasari et al., 2011, Dasari and Nelis, 2012, Heechul Yun et al., 2013, Yun et al., 2014, Mancuso et al., 2015, Kim et al., 2014, Kim et al., 2016, Yun et al., 2015, Dasari et al., 2015, Rihani et al., 2015, Jacobs et al., 2015, Jacobs et al., 2016, Davis et al., 2017, Wu et al., 2016, Hassan and Pellizzoni, 2018, Hassan and Pellizzoni, 2020, Rashid et al., 2020]. These approaches can be broadly divided into two categories: 1) approaches that focus on the problem of ***memory bus contention***[3] [Schranzhofer et al., 2010, Andersson et al., 2010, Rosen et al., 2007, Chattopadhyay et al., 2010, Chattopadhyay and Roychoudhury, 2011, Kelter et al., 2011, Kelter et al., 2014, Schliecker and Ernst, 2010, Dasari et al., 2011, Dasari and Nelis, 2012, Dasari et al., 2015, Rihani et al., 2015, Jacobs et al., 2015, Jacobs et al., 2016, Davis et al., 2017, Rashid et al., 2020]; and 2) approaches that focus on the problem of ***main memory contention***[4] [Kim et al., 2014, Kim et al., 2016, Heechul Yun et al., 2013, Yun et al., 2014, Yun et al., 2015, Wu et al., 2016, Ecco and Ernst, 2017, Hassan and Pellizzoni, 2018, Hassan and Pellizzoni, 2020].

2.2.1.1 Bus Contention-based Approaches

In this section, we will discuss the existing approaches that focus on the problem of ***bus contention*** in multicore systems considering the generic task model.

[3]Note that some of these works used terms such as system bus contention, FSB contention, shared resource contention, memory contention (i.e., black box modeling of memory) which is referred to as memory bus contention or bus contention in this dissertation.

[4]These approaches assume DRAM as the main memory and considers low-level arbitration mechanism used by the DRAM controller.

Several approaches build solutions on top of the TDMA bus arbitration policy [Schranzhofer et al., 2010, Rosen et al., 2007, Chattopadhyay et al., 2010, Chattopadhyay and Roychoudhury, 2011, Kelter et al., 2011, Kelter et al., 2014] in which static TDMA slots are assigned to the cores. These approaches analyze the maximum delay that tasks can suffer due to the allocation of the TDMA slots to other cores in the system. This delay is then integrated into the WCET/WCRT of tasks to analyze the schedulability of tasks.

Anderson et al. [Andersson et al., 2010] consider a multicore platform in which tasks are scheduled using partitioned non-preemptive scheduling and the platform uses a work-conserving bus arbitration policy. Furthermore, their work assumes that the shared cache is partitioned among all the cores. The upper bound on the bus contention is then derived by considering the maximum number of bus requests that can be issued by all tasks in the system. The upper bound on the maximum bus contention suffered by tasks is then integrated into their WCET.

Dasari et al. [Dasari et al., 2011, Dasari and Nelis, 2012, Dasari et al., 2015] built several solutions to analyze the maximum bus contention suffered by tasks on multicore systems. Their initial works [Dasari et al., 2011, Dasari and Nelis, 2012] consider a multicore platform that uses a work-conserving bus arbitration policy such as the RR. Their work tightly upper bounds the number of bus requests using the notion of tasks' memory request profiles. The bus contention analysis is then derived considering partitioned fixed-priority non-preemptive scheduling. In [Dasari et al., 2011], the upper bound on the bus contention is integrated into the WCRT of tasks. Similarly, the upper bound on the bus contention is integrated into the WCET of tasks in [Dasari and Nelis, 2012]. Finally, the authors propose a general framework [Dasari et al., 2015] to analyze the bus contention considering a variety of bus arbitration policies and integrate the maximum bus contention suffered by tasks into their WCET.

Davis et al. [Davis et al., 2017] proposed an extensible framework to compute the WCRT of tasks executing on multicore systems. The main idea is to analyze the shared resource contention suffered by tasks due to the sharing of caches, memory bus, and main memory. The upper bound on the shared resource contention suffered by tasks due to the sharing of various shared resources is then integrated into their WCRT. Specifically, their work analyzes the bus contention that can be suffered by tasks considering various bus arbitration policies such as TDMA, RR, FCFS, FPP, and FTP and integrates the respective bounds into the WCRT of tasks. The work also shows the impact of bus arbitration policies on the bus contention suffered by tasks for the generic task model.

Rashid et al. [Rashid et al., 2020] extended the work of [Davis et al., 2017] by proposing cache-aware bus contention analysis. The authors show that the bus contention strongly depends on the LLC misses and not considering the actual number of LLC misses while computing the number of bus requests can overestimate the bus contention. Specifically, the authors use the notion of cache persistence [Rashid et al., 2016] that considers reusable cache lines, i.e., memory blocks that once loaded into the cache can be reused by the task without the need of loading them from the main memory. The tighter bound on the LLC misses is then computed and integrated into the computation of the number of bus requests issued by tasks. The bus contention analysis is then derived by considering various bus arbitration policies. The results in [Rashid et al., 2020]

shows that bound on the bus contention of tasks can be significantly improved by considering the interdependence of the memory bus on the cache.

2.2.1.2 Memory Contention-based Approaches

In this section, we will discuss the existing approaches that focus on the problem of ***main memory contention*** in multicore systems considering the generic task model. Specifically, this section discusses approaches that analyze main memory contention by modeling the main memory as a *white-box*, i.e., organization of the memory, and arbitration mechanisms of the memory controller are considered. These approaches consider DRAM as the main memory which is commonly used in COTS multicore platforms.

Kim et al. [Kim et al., 2014] presented the first approach for the main memory contention analysis that relies on white-box modeling. Specifically, the authors take into account the behavior of low-level arbiters of the memory controllers of DRAM. These details include the structure of DRAM which can be composed of multiple ranks and each rank can be composed of multiple banks. Furthermore, banks are divided into rows and columns to store the data required by tasks. The initial work [Kim et al., 2014] considered a single rank-based DRAM but was then generalized to consider multiple ranks in their extended work [Kim et al., 2016]. The work considers *bank partitioning* such that each core has a set of banks assigned to that specific core. Finally, the maximum memory contention suffered by tasks is derived by considering the intra-bank contention and inter-bank contention. The bound on memory contention takes into account the intra-bank and inter-bank timing constraints defined by the JEDEC standard [JEDEC, 2008] (see Table 2.1).

Yun et al. [Yun et al., 2015] propose the memory contention by considering architectures in which each core can issue multiple memory requests at a time. Furthermore, their work considers bank partitioning such that each core has a set of private banks so that tasks can only suffer inter-bank memory contention. The authors show that the bound on inter-bank contention provided in [Kim et al., 2014] can be pessimistic. Building on this, their analysis accurately quantifies the maximum inter-bank contention that can be suffered by tasks. Unlike the work in [Kim et al., 2014], their work [Yun et al., 2015] considers the memory controller that employs *write batching* in which read memory requests are prioritized over write memory requests. The write requests are then served in batches [Chatterjee et al., 2012] to improve the turnaround time of the data bus. Specifically, their analysis upper bounds the main memory contention suffered by tasks using the request-driven approach, i.e., considers memory requests issued by the task under analysis, and the job-driven approach, i.e., considers memory requests issued by tasks executing on other cores.

Hassan et al. [Hassan and Pellizzoni, 2018] built a general framework for memory contention analysis by considering 144 possible configurations of the memory controller. These configurations were based on several factors such as whether the system uses bank partitioning or not, one/multiple memory requests per core at a time, with/without write batching, RR/fixed priority inter-bank scheduling policies, etc. Similarly to the previous works, this work also bound the memory contention suffered by tasks considering the inter-bank and intra-bank timing constraints. The authors provide an upper bound on the main memory contention for 81 configurations out

of 144 configurations and declared the remaining 63 platform configurations unbounded. In their subsequent work, Hassan et al. [Hassan and Pellizzoni, 2020] extended their work by providing an upper bound on the main memory contention considering all the 144 configurations of the memory controller. In this work [Hassan and Pellizzoni, 2020], the authors also provide a bound on the number of row hit requests for configurations that do not use write batching.

Apart from the above-mentioned categories that particularly focus on bus and/or memory contention analysis, there are some works that build system-level solutions to minimize/eliminate shared resource contention suffered by tasks. In this line, Mancuso et al. [Mancuso et al., 2015] proposed the Single Core Equivalence (SCE) framework. The main idea of the SCE framework is to reduce the temporal unpredictability posed by shared resource accesses in multicore systems in order to simplify the worst-case timing analysis of tasks. The SCE framework is built on top of three pillars: 1) Memguard [Heechul Yun et al., 2013]; 2) PALLOC [Yun et al., 2014]; and 3) cache coloring and locking [Mancuso et al., 2013]. The concept of Memguard was introduced by Yun et al. [Heechul Yun et al., 2013] in which memory bandwidth is assigned to each core in the system and per-core memory accesses cannot be greater than the assigned memory bandwidth. In other words, the memory bandwidth assigned to a given core limits the maximum number of main memory requests that can be served on that core during a given time interval. The worst-case timing analysis can then be performed by considering the memory bandwidth assigned to all the cores in the system. Although initial works consider static memory bandwidth reservation for cores [Mancuso et al., 2015, Yao et al., 2016b, Mancuso et al., 2017], the idea was later extended for dynamic memory bandwidth allocation [Agrawal et al., 2018]. PALLOC [Yun et al., 2014] is a software-based solution that provides bank partitioning to cores to avoid multiple cores accessing the same bank at the same time. Finally, cache coloring and locking [Mancuso et al., 2013] is a technique that enforces a deterministic cache hit rate on frequently accessed memory pages by combining page coloring and cache lockdown in the shared cache.

2.2.2 Related Work for the Phased Execution Model

Although all the solutions presented in Section 2.2.1 provide valuable and important solutions, they are limited to the generic task model. Therefore, in this section, we will discuss existing works that are related to the phased execution models such as the PREM [Pellizzoni et al., 2011] and the 3-phase task model [Durrieu et al., 2014].

2.2.2.1 Offline Scheduling-based Approaches

Since PREM/3-phase tasks divide the execution of tasks into distinct computation and memory phase(s), many of the existing works proposed solutions by co-scheduling tasks in the system in order to avoid shared resource contention, e.g., see Figures 2.3 and 2.4. These approaches are commonly known as table-driven, static scheduling, offline scheduling, or time-triggered scheduling-based approaches since the system-level schedule is constructed at the design time with the goal of avoiding shared resource contention.

Building upon this, Alhammad et al. [Alhammad and Pellizzoni, 2014b] proposed solutions to build a static schedule using PREM tasks such that multiple tasks cannot execute their memory phases at the same time. Their work assumes that each core has a private cache or shared cache that is partitioned among cores. Specifically, the authors presented the notion of multithreaded PREM, which schedules shared resource accesses of concurrent threads of tasks such that shared resource contention can be avoided. The authors validated their approach using realistic benchmarks.

Similarly, Becker et al. [Becker et al., 2016] proposed a shared resource contention-free execution framework to execute automotive applications on many-core platforms, i.e., platforms composed of multiple clusters in which each cluster is composed of multiple cores. The authors assume bank privatization for interference-free execution and consider a set of shared banks for the purpose of communication between tasks. Specifically, the authors proposed an ILP-based formulation to generate a time-triggered schedule for 3-phase tasks to avoid shared resource contention. Furthermore, their work proposed heuristics to map tasks to cores by taking into account the memory accesses generated by the memory phases of 3-phase tasks. The authors demonstrated the applicability of their approach using synthetic tasksets and a case study deployed on a many-core platform such as MPPA-256 from Kalray [de Dinechin et al., 2014].

Paggeti et al. [Pagetti et al., 2018] proposed a framework to generate the offline schedule for the 3-phase task model considering multicore systems. Their work considers a multicore platform that uses the TDMA bus arbitration policy. Their framework also maps tasks to cores on the basis of the WCET of tasks while generating the system-level offline schedule. Specifically, their framework uses an ILP solver to generate the offline schedule of tasks mapped to each core by considering the constraints of applications and platform descriptions. The system-level schedule is generated considering partitioned non-preemptive scheduling.

Recently, Senoussaoui et al. [Senoussaoui et al., 2022b] proposed a solution to minimize the shared resource contention suffered by PREM tasks. Unlike the actual PREM [Pellizzoni et al., 2011], this work separately handles the memory phases and the computation phases. The memory phases are scheduled in a non-preemptive manner, whereas the execution phases can be preempted. The main goal of this work is to eliminate/minimize shared resource contention that can be suffered by the memory phases of PREM tasks. To achieve this goal, the authors built three approaches 1) task-level time-triggered approach; 2) job-level time-triggered approach; and 3) online scheduling approach. The task-level time-triggered schedule ensures that memory phases of multiple tasks do not overlap at run-time, i.e., multiple memory phases do not execute at the same time. Similarly, the job-level time-triggered schedule ensures that the multiple memory phases of different jobs of the same task do not execute at the same time. In the third approach, the authors do not focus on time-triggered scheduling and use the notion of online scheduling approach that assigns an intermediate deadline to the memory phases to schedule them on the memory bus.

2.2.2.2 Shared Resource Contention-based Approaches

Although offline scheduling is predictable, it has some limitations. For example, these approaches are very restrictive such that a system-level schedule is imposed with known release times of tasks

and memory access times. Consequently, when tasks are of event-triggered or sporadic nature, such approaches may not be applicable due to the uncertainties in task behavior, e.g., release times, memory access times, etc. Furthermore, scalability can be an issue in such approaches because the system-level schedule may need to be reconstructed in the presence of even minor changes, e.g., varying the length of a memory phase, adding a new task in the system, etc. Rebuilding the system-level schedule may also mean remapping tasks to cores in some cases. Consequently, when tasks are scheduled by the scheduler on the CPU based on task priorities, it can potentially lead to a scenario in which tasks running on multiple cores execute their memory phases concurrently. This can lead to the problem of *shared resource contention* in the PREM/3-phase tasks. Therefore, this section discusses the existing approaches that focus on analyzing the shared resource contention that can be suffered by PREM/3-phase tasks.

Alhammad et al. [Alhammad and Pellizzoni, 2014a] present the notion of global PREM for scheduling PREM tasks considering global fixed-priority non-preemptive scheduling. In this approach, the global scheduler maintains a system-wide queue of ready tasks with fixed priorities and schedules them with dynamic processor assignment. The highest priority task is then extracted from the top of the queue and scheduled on the first available processor. This approach only schedules a task when the main memory is available and at least one processor is idle. It ensures that the main memory/processors are never idle when there is a pending workload. Furthermore, it also ensures that only one memory phase is executed system-wide. In this approach, a task can be delayed when it is ready to execute but other tasks, e.g., higher priority tasks, are scheduled by the global scheduler. The schedulability analysis for PREM tasks is then presented using the notion of the problem window.

Maia et al. [Maia et al., 2017] improves the work in [Alhammad and Pellizzoni, 2014a] by analyzing the schedulability of the system from the memory bus perspective instead of the core's perspective. Specifically, this work focuses on analyzing the maximum bus contention that can be suffered by 3-phase tasks considering a fixed-priority bus arbitration policy and global scheduling. Their approach maintains a priority queue, i.e., sorted on the basis of task priorities, and a FIFO queue, i.e., sorted on the basis of first-in first-out policy. The pending A-phases of all tasks are inserted into the priority queue whereas the pending R-phases of tasks are inserted into the FIFO queue. If the bus is available and the priority queue is non-empty, the A-phase of the highest priority task among all tasks in the priority queue is scheduled on the bus. However, if the priority queue is empty, the R-phases from the FIFO queue are scheduled at the bus if the bus is available. The authors show that their analysis can schedule a higher number of task sets than the work in [Alhammad and Pellizzoni, 2014a] in most of the considered scenarios.

Recently, Thilakasiri and Becker [Thilakasiri and Becker, 2023a] improved the work of Maia et al. [Maia et al., 2017] by proposing an exact schedulability test for 3-phase tasks using global fixed-priority scheduling. Similarly to the previous works in this direction, the global scheduler schedules tasks and memory phases by using a set of rules, e.g., tasks are maintained in the ready queue, a memory phase is scheduled when the bus and at least one core is available, etc. Specifically, the proposed schedulability test is built on top of timed automata where the schedulability

problem is described as a reachability problem. The experimental results reveal that their approach can schedule up to 65% of more task sets than the work in [Maia et al., 2017].

Schuh et al. [Schuh et al., 2020] presented a detailed comparison between the PREM and the 3-phase task model in different scenarios for data-flow applications. Specifically, this work considers the Kalray MPPA2 processor and three applications that are coded using the industrial toolchain SCADE Suite. One of the results in this work concludes that enforcing isolation between memory phases can increase the WCRT of tasks in comparison to scenarios in which such isolation is not enforced and tasks can suffer shared resource contention.

Casini et al. [Casini et al., 2020] proposed the memory contention analysis for the 3-phase task model considering DRAM. Their work considers partitioned fixed-priority non-preemptive scheduling. For the DRAM memory controller, their work assumes FR-FCFS intra-bank scheduling and RR-based inter-bank scheduler. Their analysis considers architectures that use the write batching mechanism to prioritize read memory requests over write requests since writes do not stall the processing pipeline. Furthermore, the work considers an architecture that facilitates a point-to-point connection between each core and each bank. Specifically, this work uses an ILP-based formulation to accurately quantify the memory contention that can be suffered by tasks. The maximum memory contention that can be suffered by 3-phase tasks is then integrated into their WCET in order to obtain their inflated WCET.

2.2.2.3 Memory Centric Scheduling-based Approaches

The concept of *Memory Centric Scheduling* (MCS) was introduced so that memory phases of tasks can be scheduled by the global memory-centric scheduler. The global memory-centric scheduler ensures that at most one memory phase access the shared resources at a time at the system level. Consequently, tasks do not suffer any shared resource contention since there can be only one task accessing the shared resource at a time. Although tasks do not suffer shared resource contention, tasks can still be delayed. For example, if a core is ready to execute the memory phase of a task but the global memory-centric scheduler is busy scheduling the memory phase of a co-running task. In such a scenario, the task can be delayed and this delay is commonly referred to as *memory interference*. This memory interference suffered by a task depends on the behavior of the memory-centric scheduler and the memory phases of co-running tasks.

The concept of MCS was first introduced by Yao et al. [Yao et al., 2012] to schedule PREM tasks. The main idea of their work is to use a TDMA-based memory-centric scheduler in which static TDMA slots are assigned to each core at the design time in which memory phases can execute. Their approach considers partitioned scheduling. TDMA-based MCS uses the notion of *memory promotion* in which memory phases were prioritized over computation phases such that memory phases can preempt computation phases on the same core. This allows cores to efficiently utilize the available TDMA slots by scheduling the memory phases of tasks during the TDMA slots. This reduces the memory interference suffered by tasks in comparison to conventional TDMA-based scheduling [Schranzhofer et al., 2010].

In their subsequent paper, Yao et al. [Yao et al., 2016a] extended their work on MCS by considering global scheduling without relying on the TDMA schedule. In this work, the multicore platform is modeled by two different types of resources, i.e., virtual execution cores and virtual memory cores, to schedule the execution and memory phases of PREM tasks. Similarly to [Yao et al., 2012], the notion of memory promotion is used to prioritize memory phases over the execution phases of PREM tasks. The main goal is to reduce the memory interference suffered by tasks while avoiding saturating the memory bandwidth. Upon the availability of the main memory, the memory scheduler checks all ready tasks and schedules the memory phase of the task with the highest priority among all ready tasks. The WCRT analysis-based schedulability analysis is presented which also takes into account the memory bandwidth.

Recently, Schwäricke et al. [Schwäricke et al., 2020] presented the notion of a fixed-priority memory-centric scheduler for partitioned fixed-priority non-preemptive scheduling. The authors state that TDMA-based MCS [Yao et al., 2012] is built on top of TDMA which is a non-work-conserving policy and can potentially overestimate the memory interference suffered by tasks. To fill this gap, the authors proposed fixed Processor Priority (PP) based MCS in which a unique fixed priority is assigned to each core at design time. Specifically, PP-based MCS considers a two-level scheduling approach: 1) fixed-priority non-preemptive scheduling at the core level; and 2) fixed processor priority based scheduling to schedule the memory phases of all tasks at the system level. The memory scheduler prioritizes the memory phases of tasks executing on higher priority cores over the memory phases of tasks running on lower priority cores. The WCRT analysis is then formulated by integrating the maximum memory interference that tasks can suffer from memory phases of tasks running on higher priority cores.

2.3 Chapter Summary

In this chapter, we discuss the relevant background concepts, i.e., task characterization, task scheduling, hardware platform characterization, worst-case timing analysis, phased execution models, etc., that are important to understand the work presented in this dissertation. We also discuss existing works that are related to the problems addressed in this dissertation. Specifically, we divided the existing approaches into two categories: 1) existing approaches that focus on relevant problems considering the generic task model; and 2) existing approaches that focus on the relevant problems considering the phased execution model.

Part I

Bus Contention Analysis for the 3-Phase Task Model

Chapter 3

Bus Contention-Aware Schedulability Analysis for the 3-Phase Task Model

As discussed in Chapters 1 and 2, 3-phase tasks can suffer bus contention when tasks running on multiple cores execute their memory phases, i.e., by accessing memory bus/main memory, concurrently. As a consequence, the bound on the length of the level-i busy window computed for uniprocessor scheduling using Equation 2.2 is not valid for tasks executing on multicore platforms. Therefore, to compute the length of the level-i busy window and WCRT of task τ_i, we need to determine the maximum bus contention that can be suffered by 3-phase tasks. The bound on the maximum bus contention can then be integrated into the computation of the level-i busy window to safely derive the schedulability analysis. To achieve this goal, this chapter proposes the bus contention analysis for 3-phase tasks executing on a multicore platform.

Specifically, we propose the bus contention analysis considering partitioned fixed-priority non-preemptive scheduling and First-Come-First-Serve (FCFS) bus arbitration policy. Furthermore, we consider two memory access models built on top of the FCFS bus arbitration policy, i.e., *Dedicated Memory Access Model* (DMAM) and the *Fair Memory Access Model* (FMAM). In the DMAM, a core permitted to access the memory bus is allowed to execute more than one ready memory phase. This can improve the throughput of the system by executing multiple memory phases, e.g., an R-phase followed by an A-phase, of tasks within a single memory access allocated to the processing core. On the other hand, the FMAM facilitates the fair distribution of the memory resources, i.e., memory bus/main memory, among all the cores resulting in improved predictability. This is achieved by allowing a core to execute at most one memory phase, i.e., A- or R-phase, when it is granted an access to the bus. After bounding the maximum bus contention that can be suffered by a task τ_i, we derive the WCRT-based schedulability analysis by integrating the maximum bus contention that task τ_i can suffer.

The main **contributions** of this chapter are as follows.

1. We propose the bus contention analysis for 3-phase tasks considering partitioned fixed-priority non-preemptive scheduling. We show that the bus contention suffered by tasks can be different when considering different memory access models. As a consequence, we formulate the bus contention analysis considering both DMAM and FMAM models.

2. We derive a schedulability test for the fixed-priority 3-phase task model by integrating the impact of maximum bus contention into the WCRT analysis of each task.

3. We compare our presented analyses against the state-of-the-art by means of case study experiments, i.e., performed using Mälardalen Benchmarks suite [Gustafsson et al., 2010], as well as through empirical evaluation, i.e., using synthetic task sets. Results show that our presented analysis tightly bounds the bus contention and improves task set schedulability by up to 88 percentage points.

Chapter Organization: The rest of the chapter is organized as follows: Section 3.1 describes the system model, task model, and memory access models. The bus contention analysis for the DMAM is presented in Section 3.2, and the bus contention analysis for the FMAM in Section 3.3. The bus contention aware schedulability analysis is presented in Section 3.4. The experimental results are presented in Section 3.5. Finally, the chapter summary is presented in Section 3.6.

For the sake of convenience, we use the terms ***bus blocking*** and ***bus contention*** interchangeably in the rest of the chapter.

3.1 System Model

We consider a multicore platform with m identical cores $(\pi_1, \pi_2, \ldots, \pi_m)$ where each core has a local memory (i.e., L1/L2 cache), large enough to store the data/instructions of the task with the largest memory footprint running on that core. Tasks are partitioned to cores at design time and cannot migrate to any other core at run-time. Similarly to existing works [Rosen et al., 2007, Schranzhofer et al., 2010, Dasari et al., 2011, Dasari and Nelis, 2012, Dasari et al., 2015, Maia et al., 2017, Rashid et al., 2020], we assume a single-channel *shared memory bus* that connects all the cores to the main memory and the memory bus can only handle one memory phase[1] at a time, i.e., only one task can access the main memory at a time. A memory phase cannot be preempted once it accesses the memory bus to perform memory transactions, i.e., the memory bus remains busy until the completion of the memory phase. Furthermore, we assume that the memory bus arbitration policy is First-Come First-Served (FCFS) which is a work-conserving policy.

3.1.1 Task Model

We consider a task set Γ comprising n sporadic tasks from which the subset Γ' is assigned to each core according to a given task-to-core mapping strategy. Each task τ_i is characterized by its

[1] A memory phase, e.g., A or R, may comprise multiple memory requests.

minimum inter-arrival time T_i and its constrained deadline D_i, where $D_i \leq T_i$. Each task τ_i is executed according to the 3-phase task model in which the task execution is divided into three phases, namely: *Acquisition* (A), *Execution* (E), and *Restitution* (R) (see Section 2.1.5.2 for details). The maximum number of memory requests that can be issued during the A-phase (resp. R-phase) of task τ_i, when it executes in isolation, is denoted by MD_i^A (resp. MD_i^R). Similarly to the existing works [Dasari et al., 2011, Yao et al., 2016b, Rashid et al., 2020], we assume that each memory request will be served within t^{mem} time units, i.e., the maximum time required to serve one memory request. Similarly, the WCET of the E-phase of task τ_i in *isolation* is denoted by C_i^E. The values of MD_i^A, MD_i^R, t^{mem}, and C_i^E can be derived using any static WCET analysis tools or by using any measurement-based techniques [Wilhelm et al., 2008]. The WCET of the A-phase (resp. R-phase) of task τ_i in *isolation* is denoted by C_i^A (resp. C_i^R) and given by $C_i^A = MD_i^A \times t^{mem}$ (resp. $C_i^R = MD_i^R \times t^{mem}$). Finally, the total WCET of task τ_i in *isolation* is given by $C_i = C_i^A + C_i^E + C_i^R$.

The utilization of task τ_i is given by $U_i = \frac{C_i}{T_i}$ and the *core utilization* of a given core π_l is given by $\sum_{\tau_i \in \Gamma'_l} U_i$. The bus utilization of task τ_i is given by $\frac{C_i^A + C_i^R}{T_i}$ and the *total bus utilization* of the taskset Γ is given by $\sum_{\tau_i \in \Gamma} \frac{C_i^A + C_i^R}{T_i}$. Each task releases potentially infinite number of jobs where each job instance is denoted by k. The response time of the k^{th} job of task τ_i is denoted by $R_{i,k}$ and the Worst-Case Response Time (WCRT) of task τ_i is denoted by R_i^{max}.

For notational convenience, we define the following set of tasks: $hep_{i,l}$ denotes the set of tasks with higher or equal priority than τ_i (including τ_i) executing on core π_l; $hp_{i,l}$ (resp. $lp_{i,l}$) denotes the set of tasks with priority higher (resp. lower) than τ_i on core π_l.

For clarity, throughout the chapter, we refer to the core on which task τ_i (i.e., the task under analysis) executes as the *local core*, denoted by π_l. Similarly, any core other than the local core is referred to as a *remote core* and denoted by π_r.

3.1.2 Execution Model

Each core maintains its own *ready queue* with tasks that are ready to execute, sorted by task priorities. Whenever a task in the queue becomes ready to execute, the core requests access to the memory bus and if the memory bus is available, the core executes the A-phase of that task. However, if the memory bus is busy serving a memory phase from another core, then the core busy-waits until the bus becomes available, at which point it executes the A-phase of the task with the highest priority in its ready queue. Once the A-phase of a task is completed, the E-phase of the same task starts executing immediately on the core. Once the E-phase is completed, the task requests access to the bus to execute its R-phase. At this point, the core may have to busy-wait for the bus again if the bus is busy serving memory phases of other co-running tasks. Once the bus becomes available, the task can execute its R-phase and finalize its execution. Note that under the considered execution model, due to its non-preemptive nature, a lower priority task τ_j running on the same core can only cause blocking to a higher priority task τ_i if τ_j starts executing before τ_i.

Symbol	Description
τ_i	i^{th} task
T_i	Minimum inter-arrival time between any two consecutive jobs of τ_i
D_i	Relative deadline of τ_i
C_i	WCET of τ_i in isolation
C_i^A	WCET of the A-phase of τ_i in isolation
C_i^E	WCET of the E-phase of τ_i in isolation
C_i^R	WCET of the R-phase of τ_i in isolation
U_i	Utilization of task τ_i
π_l	Local core (i.e., the core on which τ_i is running)
π_r	Remote core (i.e., any core other than the local core)
$hep_{i,l}$	Set of tasks with priorities higher than or equal to that of τ_i running on core π_l
$hp_{i,l}$	Set of tasks with priorities higher than that of τ_i running on core π_l
$lp_{i,l}$	Set of tasks with priorities lower than that of τ_i running on core π_l
Γ'_l	Set of tasks assigned to the local core π_l
Γ'_r	Set of tasks assigned to a remote core π_r
$\eta_i^+(\Delta)$	Maximum number of jobs that task τ_i can release during any time window of length Δ.
Γ'_r	Set of tasks assigned to a remote core π_r
$N_{\pi_l}(\Delta)$	The maximum number of times that tasks executing on core π_l can suffer bus contention during any time interval of length Δ
$N_{\pi_r}(\Delta)$	The maximum number of times that tasks running on a core π_r can cause bus contention during any time interval of length Δ
$Bus_{i,r}(\Delta)$	Maximum bus contention suffered by τ_i due to tasks running on a remote core π_r during any time interval of length Δ
$Bus_{i,l}^{max}(\Delta)$	Total bus contention suffered by τ_i due to tasks running on all remote cores during any time interval of length Δ
$W_{i,l}$	Level-i busy window for task τ_i executing on core π_l
$R_{i,k}$	Response time of k^{th} job of τ_i executing on core π_l
R_i^{max}	WCRT of τ_i

Table 3.1: Table of Symbols

3.1.3 Memory Access Models

We consider two memory access models detailed as follows:

Dedicated Memory Access Model (DMAM): When a 3-phase task is scheduled using non-preemptive scheduling, after the completion of its A-phase, it will immediately start its E-phase followed by the R-phase. However, once the R-phase of a task is completed, we may have an A-phase of a subsequent task ready to execute. At this point, the bus/memory scheduler has to decide whether it will execute the A-phase of the subsequent task on the same core or it will allocate the memory access to a different core. In the DMAM, the bus scheduler ensures that if a core has a

ready A-phase after the completion of an R-phase, the A-phase must be served before allocating the bus to any other core. The main idea of the DMAM is to allow each core to execute all its pending memory phases within an access to the memory bus. However, due to the 3-phase task model, a core can execute at most one R- and one A-phase in a single bus access, as the core has to release the bus during the E-phase execution. Once the memory phase(s) of the given core is served, the bus access can be granted to other cores. This type of memory access model can be useful in systems in which cores can execute a set of pending memory phases when access to the bus is granted.

Fair Memory Access Model (FMAM): In the FMAM, each core can execute at most one memory phase (i.e., either A- or R-phase) when access to the bus is granted and if another core is waiting to access the memory bus to execute a memory phase. After the completion of all memory requests of a memory phase, the bus can be granted to other cores. Due to the work-conserving nature of the FCFS bus arbitration policy, a core can execute another memory phase after the completion of a memory phase if other cores are not waiting to access the memory bus.

Note that in Table 3.1, $\eta_i^+(\Delta)$ is the upper event arrival function [Schliecker and Ernst, 2010] that upper bounds the maximum number of events that can arrive on i^{th} event stream during any time window of length Δ. In this chapter, we use $\eta_i^+(\Delta)$ to denote the maximum number of jobs that τ_i can release during any time window of length Δ.

3.2 Bus Blocking Analysis for the Dedicated Memory Access Model (DMAM)

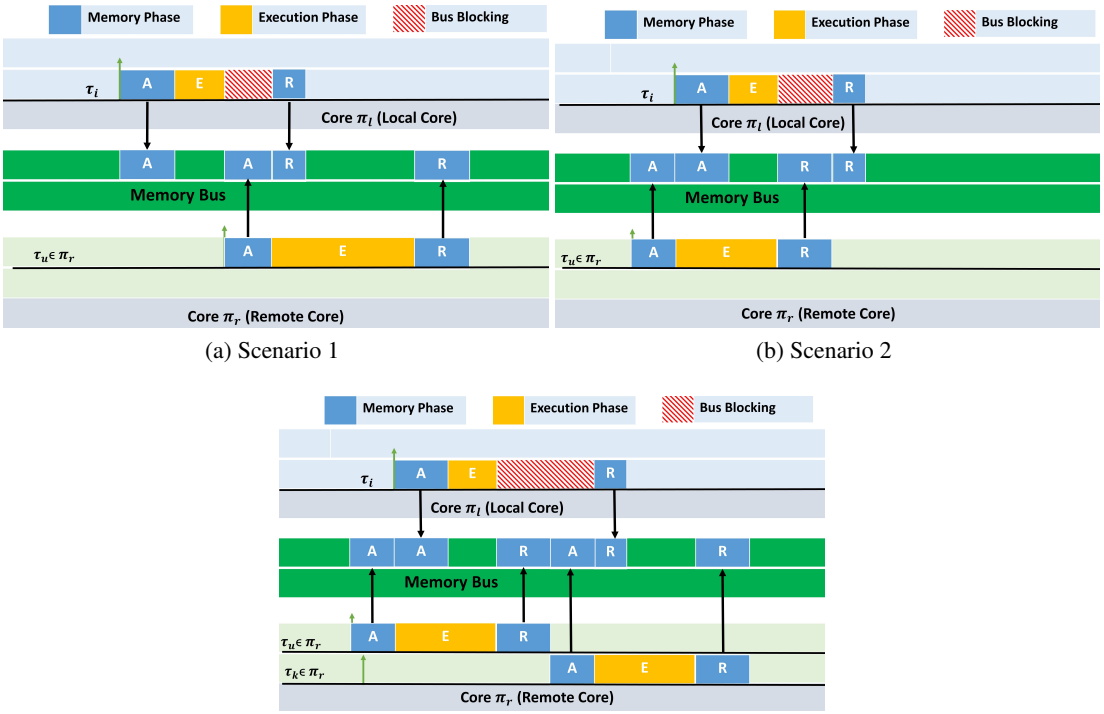
As defined in Section 3.1, the Dedicated Memory Access Model (DMAM) allows each core to execute at most one R- and one A-phase back-to-back without granting the bus access to any other waiting core. Consequently, an A-phase cannot suffer bus blocking when it executes immediately after the completion of an R-phase running on the same core. An example scenario is shown in Figure 3.1c in which task τ_k running on the remote core π_r does not suffer any bus blocking before its A-phase as it executes immediately after the R-phase of τ_u on core π_r.[2] We will explain the computation of maximum bus blocking for DMAM in this section.

Before explaining the proposed bus blocking analysis, we first present important properties on the DMAM that will be useful for deriving the maximum bus blocking in the next subsection.

3.2.1 Properties of the DMAM

Property 3.1. For each bus blocking suffered by a job on the local core, a remote core can cause at most one bus blocking, either from one memory phase (A or R-phase) of a job or from one R and one A-phase of two different jobs running on that remote core.

[2]For Figure 3.1 and for most of the schedule described in this dissertation, the local core is depicted on top, the remote core is depicted on the bottom, and the memory bus is depicted in the middle.



(a) Scenario 1

(b) Scenario 2

(c) Scenario 3

Figure 3.1: Bus blocking caused by a remote core for each bus blocking suffered at the local core

Proof. When a job of task τ_i running on the local core requests access to the bus, the following scenarios are possible.

Scenario 1: A job of task τ_u running on the remote core is already executing its A-phase. Consequently, a job of task τ_i on the local core can only access the bus after the completion of the A-phase of the job of task τ_u currently executing on the remote core. Therefore, in this scenario, the bus blocking that can be caused by the remote core to one job running of task τ_i on the local core is equivalent to the WCET of the A-phase of task τ_u executing on the remote core. This scenario is depicted in Figure 3.1a.

Scenario 2: A job of task τ_u running on the remote core is executing its R-phase and the ready queue of the remote core is empty. In this scenario, the bus blocking caused by the remote core to a job executing on the local core is equivalent to the WCET of the R-phase of task τ_u executing on a remote core as depicted in Figure 3.1b.

Scenario 3: A job of task τ_u on the remote core is executing its R-phase and the remote core's ready queue is non-empty. Once the R-phase of the currently executing job is completed, the A-phase of the next job in the remote core's ready queue will execute immediately. Thus, the bus will only be released after the execution of one R and one A-phase of two different jobs of the remote core. In this case, the bus blocking caused by the remote core is equivalent to the sum of the WCET of one R-phase and one A-phase of two different jobs running on that remote core. See Figure 3.1c for an example scenario.

Therefore, for each bus blocking suffered by a job on the local core, a remote core can cause at

most one bus blocking by either a memory phase (A or R-phase) of one job or a combination of one R and one A-phase of two different jobs running on that remote core. The property follows. □

Property 3.2. When a single job of a task on the remote core participates in one bus blocking, it can participate either by its A-phase or R-phase.

Proof. Directly follows from Property 3.1. □

3.2.2 Bounding the Number of Bus Blockings for the DMAM

As discussed earlier, in multicore systems, all the jobs that execute on the local core π_l during the level-i busy window $W_{i,l}$ can suffer bus blocking from co-running tasks executing on remote cores. This can directly impact the length of the level-i busy window (see definition 2.1.3.1) and the WCRT of the task under analysis τ_i. Therefore, to accurately compute the length of the level-i busy window the WCRT of the task under analysis τ_i, we must compute the maximum bus contention that tasks can suffer.

Without loss of generality, we start by computing the maximum bus blocking that can be suffered by the local core π_l from a remote core π_r during any time interval of length $W_{i,l}$ (i.e., the longest level-i busy window on core π_l). We later generalize our analysis to account for the bus blocking that can be suffered by the local core π_l from all remote cores in Section 3.2.4.

To bound the maximum bus blocking suffered by tasks executing on the local core π_l due to co-running tasks executing on a remote core π_r, we define the following notations:

- $N_{\pi_l}(W_{i,l})$: the maximum number of *times* that tasks executing on the *local core* π_l *can suffer bus blocking* during any time window of length $W_{i,l}$. This value is computed in Lemma 3.1.
- $N_{\pi_r}(W_{i,l})$: the maximum number of *times* that tasks running on a *remote core* π_r *can cause bus blocking* during $W_{i,l}$. This value is computed in Lemma 3.2.

Lemma 3.1. The maximum number of times that tasks executing on the local core π_l can suffer bus blocking during any time window of length $W_{i,l}$ is upper bounded by:

$$N_{\pi_l}(W_{i,l}) = \sum_{\tau_h \in hep_{i,l}} \eta_h^+(W_{i,l}) + 1 \tag{3.1}$$

where $\eta_h^+(W_{i,l})$ bounds the maximum number of jobs that task τ_h can release during any time window of length $W_{i,l}$.

Proof. By the definition of the level-i busy window, there is always a pending A-phase whenever an R-phase completes its execution; otherwise, the level-i busy window would terminate with the execution of the R-phase. Also, knowing that under the DMAM, each core can simultaneously execute the R- and A-phases of two subsequent jobs, each job that executes during the level-i busy window (except for the first job) can suffer bus blocking only once, i.e., before its R-phase. The A-phases of all such jobs will not suffer any bus blocking because they will execute immediately

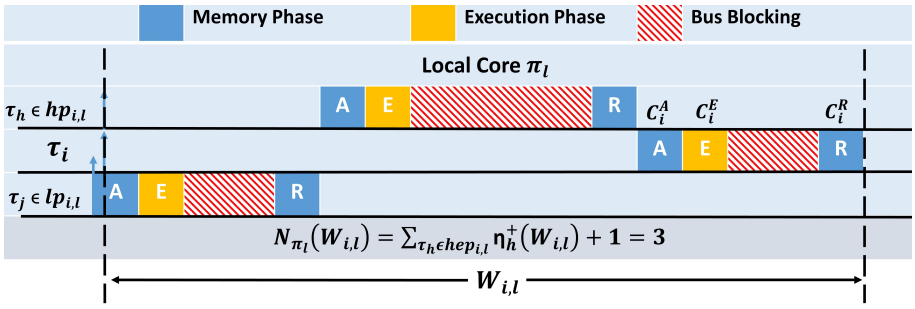


Figure 3.2: Maximum number of bus blockings when $\tau_j \in lp_{i,l}$ executes at the start of $W_{i,l}$

after the R-phases of a previous job. Therefore, the maximum number of bus blockings that can be suffered by all jobs (except the first job) of all tasks that execute on core π_l during $W_{i,l}$ is upper bounded by $\sum_{\tau_h \in hep_{i,l}} \eta_h^+(W_{i,l})$.

The additional 1 in Equation 3.1 represents two possible execution scenarios:

Scenario 1: If task τ_i is not the lowest priority task, one job of a lower priority task $\tau_j \in lp_{i,l}$ can cause blocking to tasks in $hep_{i,l}$, in the scenario when $\tau_j \in lp_{i,l}$ has started its execution before the start of the level-i busy window, i.e., starting by the execution of its A-phase. Consequently, the additional 1 in the Equation 3.1 accounts for the bus blocking that can be suffered by $\tau_j \in lp_{i,l}$ before executing its R-phase which can impact the length of the level-i busy window, e.g., see Figure 3.2.

Scenario 2: If τ_i does not suffer any blocking from a lower priority task (e.g., if τ_i is the lowest priority task) then the first job executed in the longest level-i busy window can also suffer bus blocking before its A-phase. In this scenario, the additional 1 accounts for the bus blocking suffered by the first job of task $\tau_h \in hep_{i,l}$ before starting its A-phase on core π_l, e.g., see Figure 3.3.

Hence, the maximum number of bus blockings that can be suffered by all tasks executing on the local core π_l during $W_{i,l}$ is upper bounded by $\sum_{\tau_h \in hep_{i,l}} \eta_h^+(W_{i,l}) + 1$. The Lemma follows. □

Lemma 3.2. The maximum number of times that tasks running on a remote core π_r can cause bus blocking during any time window of length $W_{i,l}$ is upper bounded by $N_{\pi_r}(W_{i,l})$, where

$$N_{\pi_r}(W_{i,l}) = \sum_{\tau_u \in \Gamma'_r} \eta_u^+(W_{i,l}) \tag{3.2}$$

Proof. According to the 3-phase task model, each job consists of two memory phases (i.e., one A- and one R-phase) that can potentially cause bus blocking. As we cannot predict the schedule of a

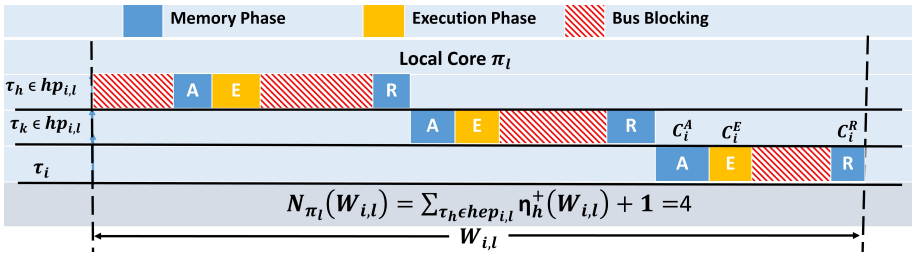


Figure 3.3: Maximum number of bus blockings when $\tau_h \in hp_{i,l}$ executes at the start of $W_{i,l}$

remote core, the bus blocking can be caused by all jobs released on a remote core π_r during any time window of length $W_{i,l}$, where each bus blocking caused by a remote core π_r can be composed of one R and one A-phase. From the upper event arrival function, we know that the maximum number of jobs that can be released by a task τ_u on core π_r during any time window of length $W_{i,l}$ is upper-bounded by $\eta_u^+(W_{i,l})$. Consequently, the maximum number of bus blockings that can be caused by a task τ_u on core π_r during any time interval of length $W_{i,l}$ is also upper-bounded by $\eta_u^+(W_{i,l})$. Since any task released on core π_r during $W_{i,l}$ can participate in the bus blocking, the maximum number of bus blockings that can be caused by a remote core π_r can be bounded by considering all jobs of all tasks released on core π_r during any time window of length $W_{i,l}$. Thus, $N_{\pi_r}(W_{i,l})$ is upper bounded by $\sum_{\tau_u \in \Gamma'_r} \eta_u^+(W_{i,l})$. The Lemma follows. □

3.2.3 Maximum Bus Blocking Computation for the DMAM

Having bounded the values of $N_{\pi_l}(W_{i,l})$ and $N_{\pi_r}(W_{i,l})$, it is possible to compute the maximum bus blocking that can be suffered by tasks running on core π_l during $W_{i,l}$ from co-running tasks executing on a remote core π_r. Before explaining how the maximum bus blocking can be computed, we first define some notations that will be used during computation.

Let M_r^A (resp. M_r^R) be an ordered set that contains the WCET of the A-phases (resp. R-phases) of all jobs released on core π_r in a time window of length $W_{i,l}$, *sorted in non-increasing order* as follows:

$$M_r^A = \{C_{r,1}^A, C_{r,2}^A, \ldots, C_{r,\hat{N}_{\pi_r}}^A \mid C_{r,x}^A \geq C_{r,x+1}^A\}$$
$$M_r^R = \{C_{r,1}^R, C_{r,2}^R, \ldots, C_{r,\hat{N}_{\pi_r}}^R \mid C_{r,y}^R \geq C_{r,y+1}^R\}$$

where $\hat{N}_{\pi_r}$ is equal to the value of $N_{\pi_r}(W_{i,l})$ computed using Equation 3.2. Note that $C_{r,x}^A$ and $C_{r,y}^R$ may belong to the same/different jobs released on core π_r during $W_{i,l}$.

We compute the maximum bus blocking for the DMAM using the following three cases.
(i) **Case 1:** $N_{\pi_l}(W_{i,l}) > N_{\pi_r}(W_{i,l})$, the maximum *number of bus blockings that can be suffered* by tasks executing on core π_l is *greater* than the maximum number of *bus blockings that can be caused* by tasks running on core π_r during any time window of length $W_{i,l}$.
(ii) **Case 2:** $N_{\pi_l}(W_{i,l}) = N_{\pi_r}(W_{i,l})$, the maximum *number of bus blockings that can be suffered* by tasks executing on core π_l is *equal* to the maximum *number of bus blockings that can be caused* by tasks running on core π_r during any time window of length $W_{i,l}$.
(iii) **Case 3:** $N_{\pi_l}(W_{i,l}) < N_{\pi_r}(W_{i,l})$, the maximum *number of bus blockings that can be suffered* by tasks executing on core π_l is *less than* the maximum *number of bus blockings that can be caused* by tasks running on core π_r during any time window of length $W_{i,l}$.

3.2.3.1 Maximum Bus Blocking Computation for Case 1

For $N_{\pi_l}(W_{i,l}) > N_{\pi_r}(W_{i,l})$, all memory phases of all jobs released on core π_r during $W_{i,l}$ can contribute to the bus blocking (e.g., see Figure 3.4). This leads to the following lemma.

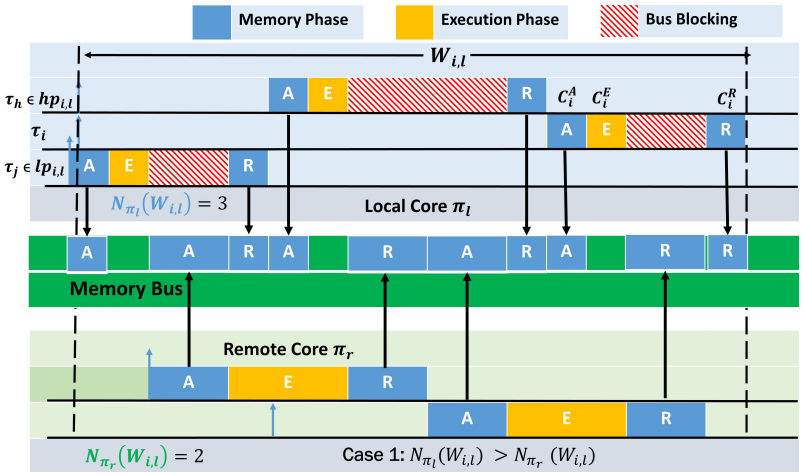


Figure 3.4: Maximum bus blocking for $N_{\pi_l}(W_{i,l}) > N_{\pi_r}(W_{i,l})$

Lemma 3.3. If $N_{\pi_l}(W_{i,l}) > N_{\pi_r}(W_{i,l})$, then the maximum bus blocking suffered by tasks executing on the local core π_l due to tasks running on a remote core π_r during any time interval of length $W_{i,l}$ is upper bounded by $Bus_{i,r}(W_{i,l})$, where

$$Bus_{i,r}(W_{i,l}) = \sum_{x=1}^{\hat{N}_{\pi_r}} C_{r,x}^A + \sum_{y=1}^{\hat{N}_{\pi_r}} C_{r,y}^R \tag{3.3}$$

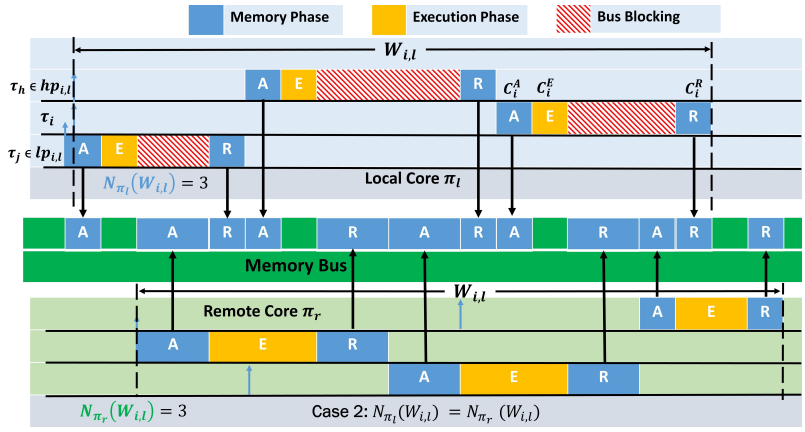
where $C_{r,x}^A$ (resp. $C_{r,y}^R$) is the WCET of an A-phase (resp. R-phase) in the set M_r^A (resp. M_r^R).

Proof. As proven in Property 3.1, under the DMAM, each bus blocking caused by a remote core π_r can be composed of either an A- or an R-phase of a job, or one R- and one A-phase of two different jobs released on core π_r during $W_{i,l}$. Since the precise bus access times of tasks running on core π_r are unknown, if $N_{\pi_l}(W_{i,l}) > N_{\pi_r}(W_{i,l})$, then in the worst-case all the memory phases of all jobs released on core π_r during $W_{i,l}$ can cause bus blocking to all tasks executing on core π_l during $W_{i,l}$. Therefore, if $N_{\pi_l}(W_{i,l}) > N_{\pi_r}(W_{i,l})$, the maximum contribution of the memory phases of $\hat{N}_{\pi_r}$ jobs, i.e., $\sum_{x=1}^{\hat{N}_{\pi_r}} C_{r,x}^A + \sum_{y=1}^{\hat{N}_{\pi_r}} C_{r,y}^R$, upper bounds the maximum bus blocking. The Lemma follows. □

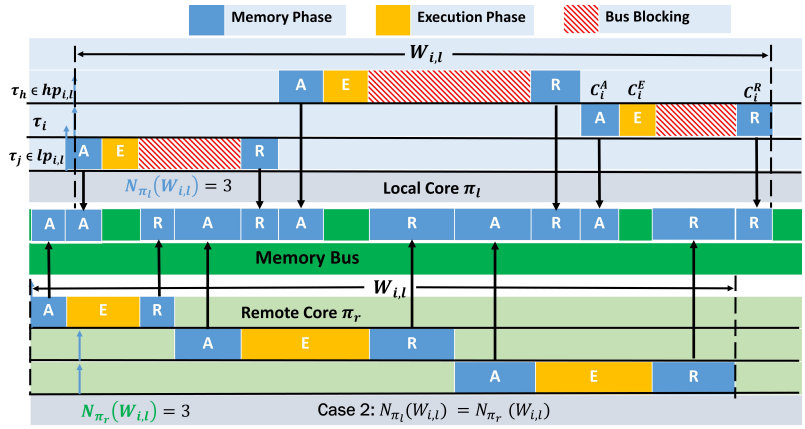
3.2.3.2 Maximum Bus Blocking Computation for Case 2

If $N_{\pi_l}(W_{i,l}) = N_{\pi_r}(W_{i,l})$, then all the memory phases *except one* from all the jobs released on core π_r during any time window $W_{i,l}$ can contribute to the bus blocking. To explain, assume that the number of bus blockings that can be suffered (resp. caused) by tasks executing on core π_l (resp. core π_r) during $W_{i,l}$ is three. In this case, there can be two possible scenarios, either the R-phase of the last job that executes on core π_r during $W_{i,l}$ (e.g., see Figure 3.5a) or the A-phase of the first job that executes on core π_r during $W_{i,l}$ (e.g., see Figure 3.5b) cannot participate in the bus blocking. This leads to the following lemma.

Lemma 3.4. If $N_{\pi_l}(W_{i,l}) = N_{\pi_r}(W_{i,l})$, then the maximum bus blocking suffered by tasks executing on the local core π_l due to tasks running on a remote core π_r during any time interval $W_{i,l}$ is upper



(a) Possible scenario 1



(b) Possible scenario 2

Figure 3.5: Possible scenarios when $N_{\pi_l}(W_{i,l}) = N_{\pi_r}(W_{i,l})$

bounded by $Bus_{i,r}(W_{i,l})$, given by:

$$Bus_{i,r}(W_{i,l}) = \sum_{x=1}^{\hat{N}_{\pi_r}} C_{r,x}^A + \sum_{y=1}^{\hat{N}_{\pi_r}} C_{r,y}^R - \min(\min_{\forall x \in M_r^A}\{C_{r,x}^A\}, \min_{\forall y \in M_r^R}\{C_{r,y}^R\}) \tag{3.4}$$

Proof. We prove the lemma using the following two observations:

Observation 1. If the A-phase of the first job on core π_r participates in the bus blocking of any job of core π_l released during $W_{i,l}$, then the first bus blocking is composed of only an A-phase (see Property 3.2) while the rest of the bus blockings can be composed of one R- and one A-phase of two different jobs running on π_r within $W_{i,l}$ (see Property 3.1). Consequently, the R-phase of the last job executing on core π_r within $W_{i,l}$ cannot participate to $Bus_{i,r}(W_{i,l})$. Since we do not know which job on core π_r will be the last to execute during $W_{i,l}$, we assume that the job with the smallest R-phase is the last job that executes on core π_r during $W_{i,l}$, given by $\min_{\forall y \in M_r^R}\{C_{r,y}^R\}$, (e.g., see Figure 3.5a).

Observation 2. If the A-phase of the first job on core π_r does not block the memory phase of any job of core π_l released during $W_{i,l}$, i.e., the first bus blocking is composed of an R-phase of the first job and an A-phase of any other job executed on π_r within $W_{i,l}$ (see Property 3.1), then

all memory phases except the A-phase of the first job executed on π_r within $W_{i,l}$ can contribute to $Bus_{i,r}(W_{i,l})$. Since we do not know which job on core π_r will execute first within $W_{i,l}$, we assume that the job with the smallest A-phase is the first job that executes on core π_r and the length of that A-phase is given by $\min_{\forall x \in M_r^A}\{C_{r,x}^A\}$. See Figure 3.5b for an example scenario.

Building on the above observations, the maximum bus blocking $Bus_{i,r}(W_{i,l})$ is given by the sum of all the memory phases (expressed as $\sum_{x=1}^{\hat{N}_{\pi_r}} C_{r,x}^A + \sum_{y=1}^{\hat{N}_{\pi_r}} C_{r,y}^R$) *except* the smallest memory phase, i.e., either A- or R-phase (expressed as $\min(\min_{\forall x \in M_r^A}\{C_{r,x}^A\}, \min_{\forall y \in M_r^R}\{C_{r,y}^R\})$) of tasks released on core π_r during $W_{i,l}$. The Lemma follows. □

3.2.3.3 Maximum Bus Blocking Computation for Case 3

If $N_{\pi_l}(W_{i,l}) < N_{\pi_r}(W_{i,l})$, then at most $N_{\pi_l}(W_{i,l})$ bus blockings can be caused by tasks running on core π_r to tasks executing on core π_l during $W_{i,l}$. To extract the $N_{\pi_l}(W_{i,l})$ A and R-phases with the largest execution times among all jobs that execute on π_r during $W_{i,l}$, we first divide the set M_r^A (resp. M_r^R) into two subsets namely M_r^{AH} and M_r^{AL} (resp. M_r^{RH} and M_r^{RL}). The subset M_r^{AH} (resp. M_r^{RH}) contains $N_{\pi_l}(W_{i,l})$ A-phases (resp. R-phases) with the *largest execution times* while the rest of the A-phases (resp. R-phases) are in the subset M_r^{AL} (resp. M_r^{RL}). Formally, these subsets are defined as follows:

$$
\begin{aligned}
M_r^{AH} &= \{C_{r,1}^A, C_{r,2}^A, \ldots, C_{r,\hat{N}_{\pi_l}}^A \mid C_{r,x}^A \geq C_{r,x+1}^A\} \\
M_r^{AL} &= \{C_{r,\hat{N}_{\pi_l}+1}^A, C_{r,\hat{N}_{\pi_l}+2}^A, \ldots, C_{r,\hat{N}_{\pi_r}}^A \mid C_{r,y}^A \geq C_{r,y+1}^A\} \\
M_r^{RH} &= \{C_{r,1}^R, C_{r,2}^R, \ldots, C_{r,\hat{N}_{\pi_l}}^R \mid C_{r,x}^R \geq C_{r,x+1}^R\} \\
M_r^{RL} &= \{C_{r,\hat{N}_{\pi_l}+1}^R, C_{r,\hat{N}_{\pi_l}+2}^R, \ldots, C_{r,\hat{N}_{\pi_r}}^R \mid C_{r,y}^R \geq C_{r,y+1}^R\}
\end{aligned}
$$

where $\hat{N}_{\pi_l} = N_{\pi_l}(W_{i,l})$ and can be computed using Equation 3.1.

We then identify two possible sub-cases:

Sub-case 3.1: All the elements of the M_r^{AH} and M_r^{RH} subsets can participate in the $\hat{N}_{\pi_l}$ number of bus blockings such that each bus blocking is composed of one R and one A-phase of tasks released on a remote core π_r during any time window of length $W_{i,l}$. The maximum bus blocking in this sub-case can be simply derived by considering the sum of all the A- and R-phases in M_r^{AH} and M_r^{RH} subsets. We discuss this sub-case in Lemma 3.5.

Sub-case 3.2: At least one element of the M_r^{AH} or M_r^{RH} subset cannot participate in the $\hat{N}_{\pi_l}$ number of bus blockings. This can only happen if all elements of M_r^{AH} and M_r^{RH} are associated to the same set of jobs. In other words, the A- and R-phases pertain to the exact same job. In this sub-case, one memory phase in M_r^{AH} or M_r^{RH} does not participate in the bus blockings. This sub-case is discussed in Lemma 3.6.

Lemma 3.5. If all the elements of the M_r^{AH} and M_r^{RH} subsets can participate in the $\hat{N}_{\pi_l}$ number of bus blockings, then the maximum bus blocking suffered by tasks executing on the local core π_l due to tasks running on a remote core π_r during any time interval $W_{i,l}$ is upper bounded by

$Bus_{i,r}(W_{i,l})$, given by

$$Bus_{i,r}(W_{i,l}) = \sum_{x=1}^{\hat{N}_{\pi_l}} C^A_{r,x} + \sum_{y=1}^{\hat{N}_{\pi_l}} C^R_{r,y} \tag{3.5}$$

where $C^A_{r,x}$ (resp. $C^R_{r,y}$) is the execution time of an A-phase (resp. R-phase) such that $C^A_{r,x} \in M^{AH}_r$ (resp. $C^R_{r,y} \in M^{RH}_r$).

Proof. If all elements of M^{AH}_r and M^{RH}_r subsets can participate in $\hat{N}_{\pi_l}$ number of bus blockings caused by π_r such that each bus blocking is composed of one R- and one A-phase of two jobs, then all the memory phases of M^{AH}_r and M^{RH}_r can participate in the bus blocking. Since M^{AH}_r and M^{RH}_r are the subsets that contain memory phases with the largest execution times, the maximum bus blocking that can be caused by tasks running on core π_r to tasks running on core π_l during any time window of length $W_{i,l}$ is upper bounded by summing all the memory phases in the M^{AH}_r and M^{RH}_r subsets. The sum of the WCET of all the A-phases (resp. R-phases) in subset M^{AH}_r (resp. M^{RH}_r) is given by $\sum_{x=1}^{\hat{N}_{\pi_l}} C^A_{r,x}$ (resp. $\sum_{y=1}^{\hat{N}_{\pi_l}} C^R_{r,y}$). Consequently, Equation 3.5 bounds the maximum bus blocking for this sub-case. The Lemma follows. □

Lemma 3.6. If at least one element of the M^{AH}_r or M^{RH}_r subset cannot participate in the $\hat{N}_{\pi_l}$ number of bus blockings, then the maximum bus blocking suffered by tasks executing on the local core π_l due to tasks running on a remote core π_r during any time interval $W_{i,l}$ is upper bounded by $Bus_{i,r}(W_{i,l})$, given by

$$Bus_{i,r}(W_{i,l}) = \sum_{x=1}^{\hat{N}_{\pi_l}} C^A_{r,x} + \sum_{y=1}^{\hat{N}_{\pi_l}} C^R_{r,y} - \min\Big(\big(\min_{\forall x \in M^{AH}_r}\{C^A_{r,x}\} - \max_{\forall y \in M^{AL}_r}\{C^A_{r,y}\}\big), \big(\min_{\forall x \in M^{RH}_r}\{C^R_{r,x}\} - \max_{\forall y \in M^{RL}_r}\{C^R_{r,y}\}\big)\Big) \tag{3.6}$$

where $\min_{\forall x \in M^{AH}_r}\{C^A_{r,x}\}$ (resp. $\min_{\forall x \in M^{RH}_r}\{C^R_{r,x}\}$) returns the smallest element of M^{AH}_r (resp. M^{RH}_r); and $\max_{\forall y \in M^{AL}_r}\{C^A_{r,y}\}$ (resp. $\max_{\forall y \in M^{RL}_r}\{C^R_{r,y}\}$) returns the largest element of M^{AL}_r (resp. M^{RL}_r).

Proof. We know that core π_r can cause at most $\hat{N}_{\pi_l}$ bus blockings in which each bus blocking can be from one R- and one A-phase of two different jobs. To derive the maximum bus blocking, it is necessary to consider all the elements of M^{AH}_r and M^{RH}_r subsets as they contain the memory phases with the largest execution times. However, if all the elements of M^{AH}_r and M^{RH}_r are associated to the exact same set of jobs of core π_r, then it is not possible to obtain $\hat{N}_{\pi_l}$ bus blockings such that each bus blocking is composed of one R- and one A-phase of two different jobs of core π_r. In such a scenario, at least one memory phase from either M^{AH}_r or M^{RH}_r cannot participate in the bus blocking. This happens because either an A-phase (i.e., an element from M^{AH}_r) or an R-phase executing on π_r (i.e., an element from M^{RH}_r) cannot participate to the bus blockings. As $N_{\pi_l}(W_{i,l}) < N_{\pi_r}(W_{i,l})$, one memory phase from M^{AL}_r or M^{RL}_r subset can participate such that $\hat{N}_{\pi_l}$ bus blockings can be obtained in which each bus blocking is composed of one R- and one A-phase of two different jobs of core π_r.

Considering the above, the bus blocking is maximized when the non-participating memory phase in M_r^{AH} or M_r^{RH} is smallest and the participating memory phase in M_r^{AL} or M_r^{RL} is largest. This is achieved by first considering the term $\sum_{x=1}^{\hat{N}_{\pi_l}} C_{r,x}^A + \sum_{y=1}^{\hat{N}_{\pi_l}} C_{r,y}^R$ which sums all the elements of M_r^{AH} and M_r^{RH} subset. Then, the next step is to remove an element from M_r^{AH} or M_r^{RH} and add an element from M_r^{AL} or M_r^{RL} such that the bus blocking is maximized. This is achieved by first computing the difference between the smallest element of M_r^{AH} (resp. M_r^{RH}) and the largest element of M_r^{AL} (resp. M_r^{RL}), expressed as $(\min_{\forall x \in M_r^{AH}}\{C_{r,x}^A\} - \max_{\forall y \in M_r^{AL}}\{C_{r,y}^A\}), (\min_{\forall x \in M_r^{RH}}\{C_{r,x}^R\} - \max_{\forall y \in M_r^{RL}}\{C_{r,y}^R\})$. Finally, we take the minimum of the difference between the smallest element of M_r^{AH} (resp. M_r^{RH}) and the largest element of M_r^{AL} (resp. M_r^{RL}) and subtract it from the sum of the WCET of all the elements of M_r^{AH} and M_r^{RH}. The Lemma follows. □

3.2.4 Bus Contention Analysis for all Remote Cores

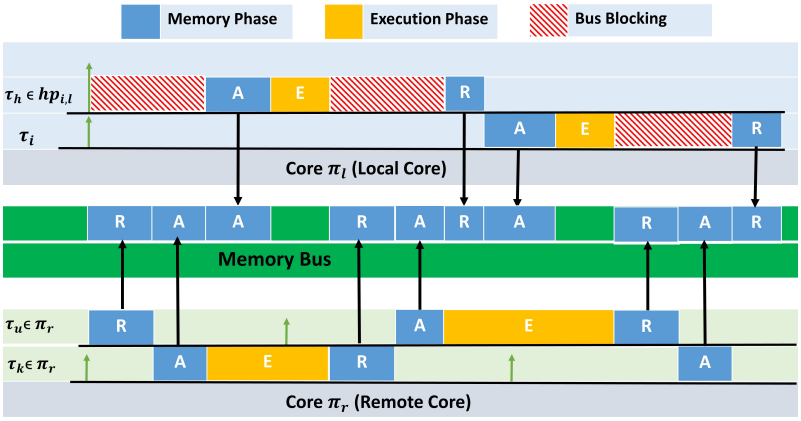
Under the FCFS bus arbitration policy, a task τ_i executing on the local core π_l will suffer the worst-case bus contention when tasks released on all other remote cores, i.e., $\forall \pi_r \in m \setminus \pi_l$, execute their memory phases before τ_i. Considering that when we only have one remote core π_r bus contention can be derived using Lemma 3.1 to Lemma 3.6. Similarly, to consider the worst case under the FCFS bus arbitration, we need to repeat the same procedure for each remote core with respect to the local core.

The total bus contention that can be suffered by tasks that execute on the local core π_l during $W_{i,l}$ due to tasks running on *all remote cores* is denoted by $Bus_{i,l}^{max}(W_{i,l})$ and is computed using Algorithm 1.

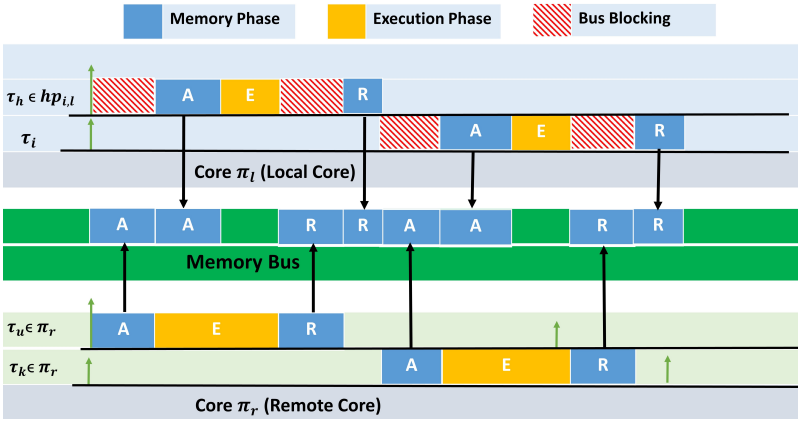
Algorithm 1 Computing the total bus contention that can be suffered by tasks that execute on the local core π_l due to tasks running on all remote cores during $W_{i,l}$

1: $Bus_{i,l}^{max}(W_{i,l}) := 0$
2: **for** $\pi_r \in [1,m]$ such that $\pi_r \neq \pi_l$ **do**
3: $\quad Bus_{i,r}(W_{i,l}) := 0$
4: $\quad$ Compute $N_{\pi_l}(W_{i,l})$ using Lemma 3.1.
5: $\quad$ Compute $N_{\pi_r}(W_{i,l})$ using Lemma 3.2.
6: $\quad$ Compute $Bus_{i,r}(W_{i,l})$ using Lemma 3.3 up to Lemma 3.6.
7: $\quad Bus_{i,l}^{max}(W_{i,l}) += Bus_{i,r}(W_{i,l})$
8: **end for**
9: Total bus contention suffered by core π_l during $W_{i,l}$ due to all remote cores is given by $Bus_{i,l}^{max}(W_{i,l})$

Algorithm 1 iterates over all remote cores by first computing the value of $N_{\pi_l}(W_{i,l})$, and $N_{\pi_r}(W_{i,l})$, (line 4 and 5) for each remote core π_r. It then computes the maximum bus blocking $Bus_{i,r}(W_{i,l})$ that can be caused by tasks running on core π_r during $W_{i,l}$ using Lemma 3.3 to Lemma 3.6 (line 6). Finally, line 7 computes the total bus contention $Bus_{i,l}^{max}(W_{i,l})$ that can be suffered by tasks that execute on the local core π_l during any time interval of length $W_{i,l}$ due to tasks



(a) Bus blocking under DMAM



(b) Bus blocking under FMAM

Figure 3.6: Maximum bus blocking for DMAM and FMAM

running on all remote cores by summing the bus blocking caused by each remote core $\pi_r \in [1,m]$ such that $\pi_r \neq \pi_l$.

3.3 Bus Blocking Analysis for the Fair Memory Access Model (FMAM)

In the DMAM, each core is allowed to execute up to two memory phases, i.e., the R-phase of a job and the A-phase of a subsequent next job, whenever it accesses the bus. However, to realize the DMAM in an actual system, a hardware/software mechanism will be required to manage the control of the bus, ensuring that each core will be able to execute an R-phase and an A-phase of any two jobs. Considering that the implementation of such hardware/software mechanism is non-trivial, a possible alternative is to use the Fair Memory Access Model (FMAM) that distributes the bus bandwidth among the cores in a fairer manner. The following example demonstrates an example scenario where the FMAM can tightly bound the bus blocking for tasks in comparison to the DMAM.

Example 1: *Let τ_i be the task under analysis which is executing on core π_l along with a higher priority task τ_h. Both τ_i and τ_h execute one job each during the level-i busy window $W_{i,l}$. During the same time interval of length $W_{i,l}$, tasks executing in parallel on core π_r release several jobs,*

e.g., greater than 3. Figure 3.6a and 3.6b show the task execution schedule under the DMAM and the FMAM, respectively.

*Under the **Dedicated Memory Access Model (DMAM)**, for each bus blocking suffered by tasks executing on the local core, there can be a combination of one R- and one A-phase of tasks released on the remote core π_r that can cause bus blocking. Knowing that two jobs are released on core π_l during the level-i busy window $W_{i,l}$ and τ_i is the lowest priority task on that core, the maximum number of bus blockings that can be suffered during $W_{i,l}$ are three (see Lemma 3.1). Consequently, considering that each bus blocking from the remote core π_r may be composed of two memory phases, i.e., an R-phase followed by an A-phase, as shown in Figure 3.6a. The worst-case bus blocking that will be suffered during $W_{i,l}$ will be equal to the sum of the WCET of six memory phases of tasks released on core π_r.*

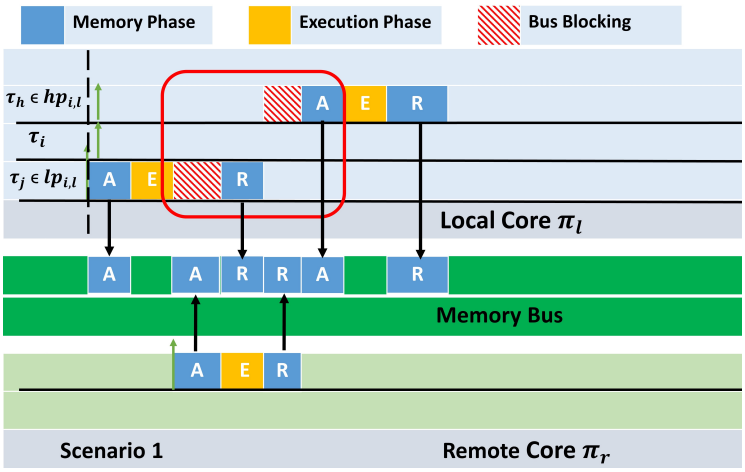
*By definition of the **Fair Memory Access Model (FMAM)**, each core can execute only one memory phase during an access to the bus. So, in the worst case, each memory phase that executes on the local core can suffer bus blocking from a memory phase executing on the remote core. Considering the scenario shown in Figure 3.6b, four memory phases are executed on core π_l during $W_{i,l}$. Therefore, the maximum bus blocking that can be suffered by all tasks executing on core π_l during $W_{i,l}$ is also upper bounded by the sum of the WCET of four memory phases that execute on core π_r during $W_{i,l}$.*

The simple example presented above shows that the FMAM can provide tighter estimates on the bus blocking suffered by the 3-phase tasks under an FCFS bus arbitration scheme. However, before we formally present the bus blocking analysis for the FMAM in Section 3.3.2 and 3.3.3, we will first introduce some properties pertaining to the model.

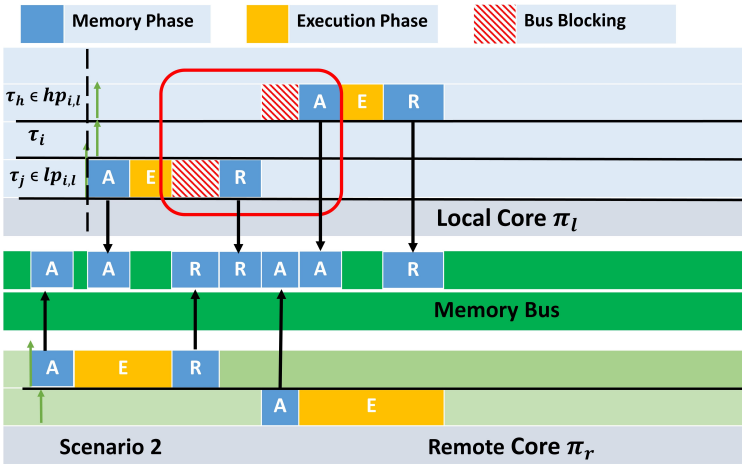
3.3.1 Useful Properties for the FMAM

Property 3.3. During the level-i busy window, the local core always executes an A-phase after the execution of an R-phase except for the A-phase of the first job and the R-phase of the last job that executes on the local core during the level-i busy window.

Proof. By the definition of the level-i busy window, the workload due to tasks in $hep_{i,l}$ taskset remains positive at all time instances within the level-i busy window except at the boundaries. So, within the level-i busy window whenever a job of tasks in $hep_{i,l}$ completes its R-phase, there is always a job that is ready to execute its A-phase; otherwise, the level-i busy window terminates with the execution of the R-phase. Therefore, within the level-i busy window, it is only the A-phase of the first job that does not execute after any R-phase on the local core because the level-i busy window begins with that A-phase. Similarly, the level-i busy window completes when the R-phase of the last job executes on the local core; thus, another A-phase does not execute. The property follows. □



(a) Scenario 1 of Property 3.4



(b) Scenario 2 of Property 3.4

Figure 3.7: Bus blocking suffered by a pair of one R and one A-phase on the local core

Property 3.4. For each pair of R and A memory phases that are to be executed sequentially on the local core π_l during the level-i busy window, the bus blocking that can be caused by tasks executing on the remote core π_r will always be composed of one A-phase and one R-phase.

Proof. As proven in Property 3.3, during the level-i busy window, the local core always executes an A-phase after the execution of the R-phase except for the A-phase of the first job and the R-phase of the last job that executes during the level-i busy window. Now, if the bus blocking is suffered by both the memory phases in a pair (i.e., an R-phase followed by an A-phase), then the bus blocking that can be caused by tasks executing on the remote core π_r to that pair of R- and A-phases will also be composed of one A-phase and one R-phase. To explain further, consider the following scenarios.

Scenario 1: If an R-phase and a subsequent A-phase executing on the local core both suffer blocking from the remote core, then, if the blocking of the first R-phase is caused by an A-phase of the remote core, the bus blocking suffered by the next A-phase of the local core will intuitively be caused by R-phase of the remote core due to the 3-phase task model (e.g., see Figure 3.7a).

Scenario 2: If an R-phase and a subsequent A-phase executing on the local core both suffer blocking from the remote core, then, if the blocking of the first R-phase is caused by an R-phase of the remote core, the blocking suffers by the next A-phase of the local core will intuitively be caused by A-phase of the remote core due to the 3-phase task model (e.g., see Figure 3.7b).

Hence, for each pair of R and A memory phases that are to be executed sequentially on the local core π_l during the level-i busy window, the bus blocking that can be caused by tasks executing on the remote core π_r will always be composed of one A-phase and one R-phase. The property follows. □

3.3.2 Bounding the Number of Bus Blockings for the FMAM

Similarly to the DMAM, we first compute the values of $N_{\pi_l}(W_{i,l})$ and $N_{\pi_r}(W_{i,l})$ for the FMAM. The computation of $N_{\pi_l}(W_{i,l})$ and $N_{\pi_r}(W_{i,l})$ for the FMAM are given by the following lemmas.

Lemma 3.7. The maximum number of times that tasks executing on the local core π_l can suffer bus blocking during any time interval of length $W_{i,l}$ is upper bounded by $N_{\pi_l}(W_{i,l})$, where $N_{\pi_l}(W_{i,l})$ is given by:

$$N_{\pi_l}(W_{i,l}) = \begin{cases} (\sum_{\tau_h \in hep_{i,l}} \eta_h^+(W_{i,l}) \times 2) + 1, & \text{if } lp_{i,l} \neq \emptyset \\ \sum_{\tau_h \in hep_{i,l}} \eta_h^+(W_{i,l}) \times 2, & \text{otherwise} \end{cases} \tag{3.7}$$

Proof. We prove this lemma using two possible scenarios by considering the priority of $\tau_i \in \pi_l$:

Scenario 1. Task τ_i is the lowest priority task of the local core: In the FMAM, each core can execute at most one memory phase during an access to the bus. This also implies that each memory phase that executes on the local core π_l can suffer bus blocking. Knowing that each task in $hep_{i,l}$ that executes on the local core during $W_{i,l}$ can release at most $\eta_h^+(W_{i,l})$ jobs and each job has 2 memory phases (i.e., A-phase and R-phase), the maximum number of bus blockings that can be suffered by all the tasks in $hep_{i,l}$ during $W_{i,l}$ is upper bounded by $\sum_{\tau_h \in hep_{i,l}} \eta_h^+(W_{i,l}) \times 2$ (e.g., see Figure 3.8).

Scenario 2. Task τ_i is not the lowest priority task of the local core: If task τ_i is not the lowest-priority task, one job of a lower-priority task, e.g., $\tau_j \in lp_{i,l}$, can cause blocking to tasks in $hep_{i,l}$ when τ_j starts executing before the start of the level-i busy window. Nevertheless, there is the need to account for the bus blocking that can be suffered by τ_j while executing its R-phase

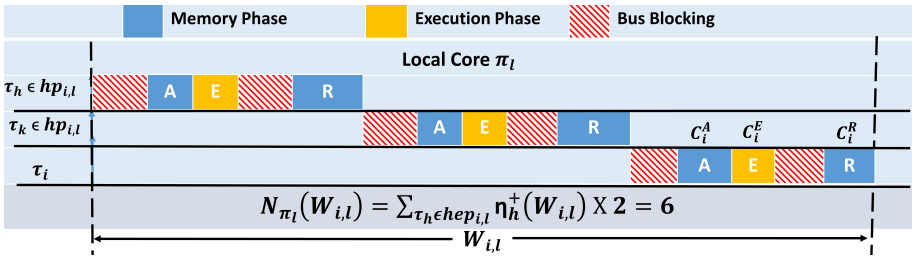


Figure 3.8: Maximum number of bus blockings suffered by the local core during $W_{i,l}$ when $lp_{i,l} = \emptyset$

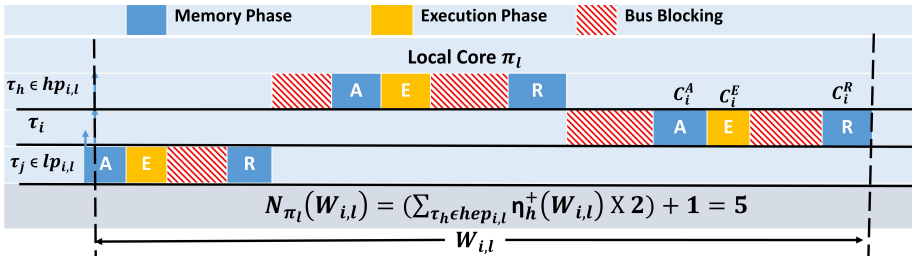


Figure 3.9: Maximum number of bus blockings suffered by the local core during $W_{i,l}$ when $lp_{i,l} \neq \emptyset$

as it can impact the length of the level-i busy window[3]. Therefore, the maximum number of bus blockings that can be suffered by tasks that execute on core π_l during any time interval of length $W_{i,l}$ is upper-bounded by $(\sum_{\tau_h \in hep_{i,l}} \eta_h^+(W_{i,l}) \times 2) + 1$ when $lp_{i,l} \neq \emptyset$ (e.g., see Figure 3.9). The Lemma follows. □

Lemma 3.8. The maximum number of times that tasks running on a remote core π_r can cause bus blocking during any time interval of length $W_{i,l}$ is upper bounded by $N_{\pi_r}(W_{i,l})$, where $N_{\pi_r}(W_{i,l})$ is given by:

$$N_{\pi_r}(W_{i,l}) = \sum_{\tau_u \in \Gamma'_r} \eta_u^+(W_{i,l}) \times 2 \tag{3.8}$$

Proof. Due to the nature of the FMAM, each time the bus blocking is suffered by the local core, a remote core can cause bus blocking using one memory phase. Furthermore, the maximum number of jobs released by a task τ_u on a remote core π_r during $W_{i,l}$ is upper bounded by $\eta_u^+(W_{i,l})$. This implies that the maximum number of bus blockings that can be caused by a task τ_u released on a remote core π_r during $W_{i,l}$ is upper bounded by $\eta_u^+(W_{i,l}) \times 2$, i.e., using its A- and R-phases. Extending this to all tasks running on a remote core π_r, i.e., Γ'_r, the maximum number of bus blockings that can be caused by all tasks released on a remote core π_r during $W_{i,l}$ is upper bounded by $\sum_{\tau_u \in \Gamma'_r} \eta_u^+(W_{i,l}) \times 2$. The Lemma follows. □

3.3.3 Maximum Bus Blocking Computation for the FMAM

Having bounded the values of $N_{\pi_l}(W_{i,l})$ and $N_{\pi_r}(W_{i,l})$, we can now derive the maximum bus blocking $Bus_{i,r}(W_{i,l})$ that can be suffered by the local core π_l from a remote core π_r during any time interval of length $W_{i,l}$ using the following cases.

- **Case 1:** $N_{\pi_l}(W_{i,l}) \geq N_{\pi_r}(W_{i,l})$, i.e., the maximum *number of bus blockings* that can be *suffered* by tasks executing on core π_l is *greater than or equal* to the maximum *number of bus blockings* that can be *caused* by tasks running on core π_r during any time window of length $W_{i,l}$. The maximum bus blocking computation for this case is given by Lemma 3.9.

- **Case 2:** $N_{\pi_l}(W_{i,l}) < N_{\pi_r}(W_{i,l})$, i.e., the maximum *number of bus blockings* that can be *suffered* by tasks executing on core π_l is *less* than the maximum *number of bus blockings* that

[3] We do not need to account for the bus blocking that can be suffered by $\tau_j \in lp_{i,l}$ while executing its A-phase as the $\tau_j \in lp_{i,l}$ has started its A-phase execution before the start of the level-i busy window.

can be *caused* by tasks running on core π_r during any time window of length $W_{i,l}$. The maximum bus blocking computation for this case is given by Lemma 3.10.

Lemma 3.9. If $N_{\pi_l}(W_{i,l}) \geq N_{\pi_r}(W_{i,l})$, then the maximum bus blocking suffered by tasks executing on the local core π_l due to tasks running on a remote core π_r during any time interval $W_{i,l}$ is upper bounded by $Bus_{i,r}(W_{i,l})$, where

$$Bus_{i,r}(W_{i,l}) = \sum_{\tau_u \in \Gamma'_r} \eta_u^+(W_{i,l}) \times (C_u^A + C_u^R) \tag{3.9}$$

Proof. By Lemma 3.7, we know that the local core π_l can suffer at most $N_{\pi_l}(W_{i,l})$ bus blockings due to tasks running on a remote core π_r during $W_{i,l}$. As the exact schedule of the tasks executing on a remote core cannot be predicted, for $N_{\pi_l}(W_{i,l}) \geq N_{\pi_r}(W_{i,l})$, all the bus blockings caused by core π_r during any time interval of length $W_{i,l}$ can impact the tasks that execute on the local core π_l. Consequently, the maximum bus blocking that can be caused by a task τ_u released on a remote core π_r during $W_{i,l}$, is upper bounded by the maximum number of jobs released during $W_{i,l}$ times the sum of the WCET of its memory phases, i.e., $\eta_u^+(W_{i,l}) \times (C_u^A + C_u^R)$. Extending this result for all tasks, the maximum bus blocking $Bus_{i,r}(W_{i,l})$ that can be suffered by the local core π_l due to the execution of all tasks released on a remote core π_r during $W_{i,l}$ is upper bounded by $\sum_{\tau_u \in \Gamma'_r} \eta_u^+(W_{i,l}) \times (C_u^A + C_u^R)$. The Lemma follows. □

In case 2, as the maximum number of bus blockings that can be suffered by the local core is less than the maximum number of bus blockings that can be caused by a remote core during $W_{i,l}$, we need to extract a set of memory phases released by all tasks on a remote core during $W_{i,l}$ that provide a safe and tighter bound on the bus blocking. To do this, we first introduce the following notations:

Let $\hat{P}$ denote the maximum number of jobs released by all the tasks in $hep_{i,l}$ during any time interval of length $W_{i,l}$, i.e., $\hat{P} = \sum_{\tau_h \in hep_{i,l}} \eta_h^+(W_{i,l})$. Let $\hat{Q}$ denote the maximum number of jobs released by all tasks on a remote core π_r during any time interval of length $W_{i,l}$ i.e., $\hat{Q} = \sum_{\tau_u \in \Gamma'_r} \eta_u^+(W_{i,l})$. These terms will be later used to extract a given number of memory phases among all the memory phases released on a remote core during $W_{i,l}$.

Now, we define M_r^{AH} and M_r^{RH} as ordered sets that contain the $\hat{P}$ largest A-phases and R-phases, respectively, released on core π_r in a time window of length $W_{i,l}$. We assume that M_r^{AH} and M_r^{RH} are sorted in a non-increasing order. Additionally, we define M_r^{AL} and M_r^{RL} as ordered sets that contain remaining A- and R-phases, respectively, released on core π_r in a time window of length $W_{i,l}$, sorted in a non-increasing order:

$$\begin{aligned}
M_r^{AH} &= \{C_{r,1}^A, C_{r,2}^A, \ldots, C_{r,\hat{P}}^A \mid C_{r,x}^A \geq C_{r,x+1}^A\} \\
M_r^{AL} &= \{C_{r,\hat{P}+1}^A, C_{r,\hat{P}+2}^A, \ldots, C_{r,\hat{Q}}^A \mid C_{r,y}^A \geq C_{r,y+1}^A\} \\
M_r^{RH} &= \{C_{r,1}^R, C_{r,2}^R, \ldots, C_{r,\hat{P}}^R \mid C_{r,x}^R \geq C_{r,x+1}^R\} \\
M_r^{RL} &= \{C_{r,\hat{P}+1}^R, C_{r,\hat{P}+2}^R, \ldots, C_{r,\hat{Q}}^R \mid C_{r,y}^R \geq C_{r,y+1}^R\}
\end{aligned}$$

where $C^A_{r,x}$ (resp. $C^R_{r,x}$) is the WCET of an A-phase (resp. R-phase) of a task released on a remote core π_r in a time window of length $W_{i,l}$.

Furthermore, we introduce the terms $\vec{V}_r$, $\vec{X}_r$, $\vec{Y}_r$, and $\vec{Z}_r$ in order to simplify case 2 as follows:

$$\begin{aligned}\vec{V}_r &= \max(C^A_{r,\hat{P}+1}, C^R_{r,\hat{P}+1})\\ \vec{X}_r &= C^A_{r,\hat{P}} + C^R_{r,\hat{P}}\\ \vec{Y}_r &= C^A_{r,\hat{P}} + C^A_{r,\hat{P}+1}\\ \vec{Z}_r &= C^R_{r,\hat{P}} + C^R_{r,\hat{P}+1}\end{aligned}$$

As M^{AH}_r, M^{RH}_r, M^{AL}_r and M^{RL}_r are ordered sets, the term $\vec{V}_r$ returns the largest memory phase among all the memory phases in M^{AL}_r and M^{RL}_r sets. The term $\vec{X}_r$ sums the smallest A-phase of M^{AH}_r and the smallest R-phase of the M^{RH}_r set. Similarly, the term $\vec{Y}_r$ sums the smallest A-phase in M^{AH}_r and the largest A-phase in M^{AL}_r. Finally, the term $\vec{Z}_r$ sums the smallest R-phase in M^{RH}_r and the largest R-phase in M^{RL}_r. Now we can compute the maximum bus blocking for case 2 using the following lemma.

Lemma 3.10. If $N_{\pi_l}(W_{i,l}) < N_{\pi_r}(W_{i,l})$, then the maximum bus blocking suffered by tasks executing on the local core π_l due to tasks running on a remote core π_r, during any time interval of length $W_{i,l}$, is upper bounded by $Bus_{i,r}(W_{i,l})$, which is given by:

$$Bus_{i,r}(W_{i,l}) = \begin{cases} \sum_{x=1}^{\hat{P}} C^A_{r,x} + \sum_{y=1}^{\hat{P}} C^R_{r,y} + \vec{V}_r & \text{, if } lp_{i,l} \neq \emptyset \\ \sum_{x=1}^{\hat{P}-1} C^A_{r,x} + \sum_{y=1}^{\hat{P}-1} C^R_{r,y} + \max\left(\vec{X}_r, \vec{Y}_r, \vec{Z}_r\right), & \text{otherwise.} \end{cases} \tag{3.10}$$

where $C^A_{r,x}$ (resp. $C^R_{r,y}$) is the WCET of A-phase (resp. R-phase) that belongs to M^{AH}_r (resp. M^{RH}_r) set.

Proof. We prove this lemma using two possible scenarios on the basis of the priority of τ_i:

Scenario 1. Task τ_i is not the lowest priority task on the local core: It is proven in Lemma 3.7 that if task τ_i is not the lowest priority task of the local core π_l, all tasks that execute on core π_l during $W_{i,l}$ can suffer at most $(\sum_{\tau_h \in hep_{i,l}} \eta^+_h(W_{i,l}) \times 2) + 1$ bus blockings. As $N_{\pi_l}(W_{i,l}) < N_{\pi_r}(W_{i,l})$, we need to extract $(\sum_{\tau_h \in hep_{i,l}} \eta^+_h(W_{i,l}) \times 2) + 1$ bus blockings that can lead to the maximum bus blocking that can be caused by a remote core π_r during $W_{i,l}$.

As proven in Property 3.3, during the level-i busy window, the local always executes an A-phase after an R-phase, except the A-phase of the first job and the R-phase of the last job that executes during the level-i busy window. Consequently, by applying Property 3.4 to all the memory phases that execute on core π_l during $W_{i,l}$, except the first A-phase and last R-phase that executes on core π_l during $W_{i,l}$, the maximum bus blocking can be bounded by taking the sum of the execution time of $\hat{P}$ largest A- and R-phases released on a remote core π_r during $W_{i,l}$, i.e., using the sets M^{AH}_r and M^{RH}_r.

Furthermore, when $lp_{i,l} \neq \emptyset$, the level-i busy window starts when τ_i is released, but a lower priority task τ_j has started executing its A-phase. Consequently, we do not need to account for

the bus blocking suffered by the A-phase of τ_j, i.e., the first A-phase that executes on the local core π_l during $W_{i,l}$. Finally, the maximum bus blocking that can be suffered by the last R-phase that executes on the core π_l during $W_{i,l}$ is computed using $\vec{V}_r$, where $\vec{V}_r$ returns the largest memory phase (i.e., A or R-phase) among M_r^{AL} and M_r^{RL} sets. Hence, Equation 3.10 upper-bounds the bus blocking when $lp_{i,l} \neq \emptyset$.

Scenario 2. Task τ_i is the lowest priority task on the local core: It is proven in Lemma 3.7 that if task τ_i is the lowest priority task executed on core π_l then it can suffer at most $(\sum_{\tau_h \in hep_{i,l}} \eta_h^+(W_{i,l}) \times 2)$ bus blockings. As $N_{\pi_l}(W_{i,l}) < N_{\pi_r}(W_{i,l})$, we need to extract the $(\sum_{\tau_h \in hep_{i,l}} \eta_h^+(W_{i,l}) \times 2)$ number of bus blockings that can lead to the maximum bus blocking that can be caused by a remote core π_r during $W_{i,l}$.

As τ_i is the lowest priority task, the bus blocking can also be suffered by the first job before its A-phase. Consequently, Property 3.4 can be applied to all the memory phases that execute on core π_l during $W_{i,l}$ *except* the first A-phase and last R-phase that execute on core π_l during $W_{i,l}$. Therefore, the maximum bus blocking that can be suffered by all memory phases except the first A-phase and last R-phase that execute on core π_l during $W_{i,l}$ can be bounded by taking the sum of the execution time of $\hat{P}-1$ largest A- and R-phases (from sets M_r^{AH} and M_r^{RH}) released on a remote core π_r during $W_{i,l}$, i.e., $\sum_{x=1}^{\hat{P}-1} C_{r,x}^A + \sum_{y=1}^{\hat{P}-1} C_{r,y}^R$.

The next step is to compute the maximum bus blocking that can be suffered by the first A-phase and last R-phase that execute on core π_l during $W_{i,l}$. To maximize the bus blocking that can be suffered by the first A-phase and last R-phase that execute on core π_l during $W_{i,l}$, we consider the two largest memory phases (i.e., two A-phases, two R-phases or a combination of one A and R-phases) that were not considered in the $\hat{P}-1$ largest A- and R-phases. This is achieved by taking the maximum among the values of $\vec{X}_r$, $\vec{Y}_r$, and $\vec{Z}_r$ where $\vec{X}_r$ returns the sum of the largest one A- and R-phase, $\vec{Y}_r$ (resp. $\vec{Z}_r$) returns the sum of the WCET of two largest A-phases (resp. R-phases) released on a remote core π_r during $W_{i,l}$ that were not previously considered in $\hat{P}-1$ A- and R-phases. The Lemma follows. □

Having bounded the maximum bus blocking $Bus_{i,r}(W_{i,l})$ that can be suffered by tasks executing on the local core π_l due to the tasks running on a remote core π_r during any time interval of length $W_{i,l}$, the next step is to compute the *total bus contention* $Bus_{i,l}^{max}(W_{i,l})$ that can be suffered by the local core due to *all remote cores*. The total bus contention $Bus_{i,l}^{max}(W_{i,l})$ that can be suffered by the local core due to all remote cores during $W_{i,l}$ can be computed using algorithm 1 by first computing $N_{\pi_l}(W_{i,l})$ (line 4) using Lemma 3.7, $N_{\pi_r}(W_{i,l})$ (line 5) using Lemma 3.8, and the maximum bus blocking $Bus_{i,r}(W_{i,l})$ caused by a remote core π_r (line 6) during $W_{i,l}$ using Lemma 3.9 to Lemma 3.10.

3.4 Schedulability Analysis

Having bounded the total bus contention $Bus_{i,l}^{max}(W_{i,l})$ that can be suffered by the local core π_l due to *all* remote cores in the level-i busy window, i.e., $W_{i,l}$, under both the dedicated memory access

model and the fair memory access model, we will now show how the response time of tasks can be computed. To compute the WCRT of a task τ_i executing on core π_l, we first need to compute the length of its longest level-i busy window (see definition 2.1.3.1) using the following equation.

Lemma 3.11. The length of the level-i busy window for a given task τ_i executing on core π_l is denoted by $W_{i,l}$, where $W_{i,l}$ is given by the first positive solution to the fixed-point iteration of the following equation

$$W_{i,l} = C^{max}_{lp_{i,l}} + \sum_{\tau_h \in hep_{i,l}} (\eta^+_h(W_{i,l}) \times C_h) + Bus^{max}_{i,l}(W_{i,l}) \tag{3.11}$$

Proof. Due to fixed priority non-preemptive scheduling, at most one job of a lower priority task can cause blocking to task τ_i. This blocking is maximized by considering a task with the largest WCET among all tasks in $lp_{i,l}$, is expressed as $C^{max}_{lp_{i,l}}$ where $C^{max}_{lp_{i,l}} = \max_{\tau_j \in lp_i}\{C_j\}$. Furthermore, we need to account for the maximum workload that can be generated by all tasks in $hep_{i,l}$ (including τ_i) during any time interval of length $W_{i,l}$. The maximum workload that can be generated by a task τ_h is upper bounded by $\eta^+_h(W_{i,l}) \times C_h$ in which $\eta^+_h(W_{i,l})$ is the upper event arrival function that gives the maximum number of jobs that τ_h can release during any time window of length $W_{i,l}$ and C_h is the WCET of τ_h. Since each task that executes on the local core π_l can suffer bus contention from all remote cores during $W_{i,l}$, we need to integrate the total bus contention $Bus^{max}_{i,l}(W_{i,l})$ that can be suffered by the local core π_l during $W_{i,l}$ due to tasks running on all remote cores and it can be computed using Algorithm 1 by first computing the maximum bus blocking for the DMAM (computed using Lemma 3.1 to Lemma 3.6) and the FMAM (computed using Lemma 3.7 to Lemma 3.10). The Lemma follows. □

Note that $W_{i,l}$ appears on both sides of Equation 3.11 which means Equation 3.11 is recursive and a fixed-point computation on $W_{i,l}$ can be used to find a solution by initiating $W_{i,l} = C^{max}_{lp_i} + \sum_{\tau_h \in hep_{i,l}} C_h$. The length of the level-i busy window $W_{i,l}$ will then be given by the smallest positive value of $W_{i,l}$ for which Equation 3.11 converges.

As proven in [Bril et al., 2007], to compute the WCRT of task τ_i, we need to determine the response time of each job of task τ_i that executes during the level-i busy window $W_{i,l}$. Therefore, we first compute the maximum number of jobs of task τ_i that can execute within $W_{i,l}$ using the following equation.

$$K_i = \eta^+_i(W_{i,l}) \tag{3.12}$$

To compute the response time of the k^{th} job of τ_i that execute on the local core π_l during $W_{i,l}$, i.e., denoted as $\tau_{i,k}$, we first compute the latest start time of the *R-phase* of $\tau_{i,k}$. This is due to the fact that each job that executes on core π_l during the response time of $\tau_{i,k}$ (including $\tau_{i,k}$) can suffer bus blocking until the start of the R-phase of $\tau_{i,k}$. The latest start time of the R-phase of $\tau_{i,k}$ on core π_l is computed using the following lemma.

Lemma 3.12. The latest start time of the R-phase of $\tau_{i,k}$ is denoted by $s^R_{i,k}$, where $s^R_{i,k}$ is given by the first positive solution to the following fixed-point iteration:

$$s^R_{i,k} = C^{max}_{lp_{i,l}} + (k-1)\times C_i + (C^A_i + C^E_i) + \sum_{\tau_h \in hep_{i,l}\setminus\tau_i} \eta^+_h(s^R_{i,k} - (C^A_i + C^E_i))\times C_h + Bus^{max}_{i,l}(s^R_{i,k}) \quad (3.13)$$

Proof. The proof is divided into two steps. In the first step, we upper bound the contribution of tasks executing on the local core to the start time of the R-phase of $\tau_{i,k}$. In step two, we upper bound the impact of tasks running on all remote cores to the start time of R-phase of $\tau_{i,k}$.

Step 1. Task τ_i can suffer blocking from at most one job from lower priority tasks in $lp_{i,l}$. This blocking is maximized by considering a task with the maximum WCET among all tasks in $lp_{i,l}$ and is expressed as $C^{max}_{lp_{i,l}}$ where $C^{max}_{lp_{i,l}} = \max_{\tau_j \in lp_{i,l}}\{C_j\}$. Knowing that $k-1$ jobs of task τ_i may have been executed before $\tau_{i,k}$, their contribution to the latest start time of the R-phase of $\tau_{i,k}$ is given by $(k-1)\times C_i$. Furthermore, to compute the start time of the R-phase of $\tau_{i,k}$, we add the WCET of the A- and E-phases of τ_i, given by $C^A_i + C^E_i$. Finally, all jobs released by the higher or equal priority tasks in $hep_{i,l}$ except τ_i can cause interference on $\tau_{i,k}$ *until the start of its A-phase* due to the fixed-priority non-preemptive scheduling. Hence, the total interference that can be caused by a task $\tau_h \in hep_{i,l}$ until the start of the A-phase of $\tau_{i,k}$ is upper bounded by $\eta^+_h(s^R_{i,k} - (C^A_i + C^E_i))\times C_h$, where C_h is the WCET of task τ_h in isolation. Effectively, the total contribution from all tasks in $hep_{i,l}$ except τ_i to the start time of the A-phase of $\tau_{i,k}$ is upper bounded by $\sum_{\tau_h \in hep_{i,l}\setminus\tau_i} \eta^+_h(s^R_{i,k} - (C^A_i + C^E_i))\times C_h$.

Step 2. It is possible that each job that executes on core π_l suffers bus blocking due to tasks running on remote cores. Thus, the total bus contention suffered by the local core until the start of the R-phase of $\tau_{i,k}$ due to tasks of all the remote cores is upper bounded by $Bus^{max}_{i,l}(s^R_{i,k})$, using Algorithm 1 by first computing the maximum bus blocking for the DMAM (computed using Lemma 3.1 to Lemma 3.6) and the FMAM (computed using Lemma 3.7 to Lemma 3.10). □

Note that $s^R_{i,k}$ appears on both sides of Equation 3.13 which means Equation 3.13 is recursive and a fixed-point computation on $s^R_{i,k}$ can be used to find a solution by initiating $s^R_{i,k} = C^{max}_{lp_{i,l}} + C^A_i + C^E_i + \sum_{\tau_h \in hep_{i,l}\setminus\tau_i} C_h$. The latest start time $s^R_{i,k}$ will then be given by the smallest positive value of $s^R_{i,k}$ for which Equation 3.13 converges.

The response time $R_{i,k}$ of $\tau_{i,k}$ can be computed by adding $s^R_{i,k}$ to the WCET of the R-phase of task τ_i, i.e., C^R_i, and subtracting the minimum inter-arrival time of previously executed jobs of task τ_i, i.e.,

$$R_{i,k} = s^R_{i,k} + C^R_i - (k-1)\times T_i \quad (3.14)$$

Finally, the WCRT of a given task τ_i can be computed by maximizing equation 3.14 over all jobs of τ_i that execute during the level-i busy window. Hence,

$$R^{max}_i = \max_{k\in[1,K_i]}\{R_{i,k}\} \quad (3.15)$$

where K_i is computed using Equation 3.12.

Note that task τ_i is deemed schedulable only if its WCRT (computed using Equation 3.15) is less than or equal to its relative deadline D_i, i.e., $R_i^{max} \leq D_i$. A task set is deemed schedulable only if all tasks in that task set are schedulable and the total bus utilization of the system is less than or equal to the capacity of the bus, i.e., 1, since the memory bus is saturated otherwise.

3.5 Experimental Evaluation

In this section, we evaluate the effectiveness of the proposed approaches. To the best of our knowledge, no work exists that focuses on bounding the bus contention for the 3-phase task model considering partitioned scheduling. A similar work [Schwäricke et al., 2020] exists that focuses on memory-centric scheduling of PREM tasks under partitioned fixed-task priority scheduling. We compare the proposed work with the work in [Schwäricke et al., 2020] because it focuses on deriving maximum memory interference for PREM tasks considering fixed priority non-preemptive partitioned scheduling[4]. The work in [Schwäricke et al., 2020] considers a fixed processor priority bus arbitration policy and allows global memory preemption, i.e., the memory phases running on higher-priority processors can preempt the memory phases running on lower priority processors. This is different from the proposed work as we assume that memory phases execute non-preemptively. To compare the performance of our proposed Dedicated Memory Access Model (DMAM) and Fair Memory Access Model (FMAM) with the work in [Schwäricke et al., 2020], we consider two variations of the analysis presented in [Schwäricke et al., 2020], i.e., with/without allowing global memory preemption. The analysis that allows global memory preemption is the exact analysis presented in [Schwäricke et al., 2020]. The analysis without global memory preemption is a slightly modified version of [Schwäricke et al., 2020] to allow the execution of non-preemptive memory phases.

To evaluate the performance of all the analyzed approaches, we performed two sets of experiments. A case study experiment performed using task parameters obtained from the Mälardalen benchmark suite [Gustafsson et al., 2010] is presented in Section 3.5.1. Experiments performed using synthetic task sets are detailed in Section 3.5.2.

3.5.1 Case Study

For the case study experiments, we use task parameters taken from Table 2 of [Davis et al., 2017]. Table 2 in [Davis et al., 2017] is generated from the Mälardalen benchmark suite using the gem5 instruction set simulator by modeling a quad-core multicore platform considering ARMv7 cores and a shared memory bus that connects the cores to the main memory.

Although Table 2 of [Davis et al., 2017] contains several task parameters for the analyzed benchmarks, we only consider the *Processing Demand* (PD) and *Memory Demand* (MD) of tasks in our experiments. Also, we consider non-preemptive task scheduling which can suffer from

[4]Note that works like [Maia et al., 2017] exists that focus on bus contention analysis for 3-phase tasks considering global scheduling. However, we do not compare with it since partitioned and global scheduling is fundamentally different. Furthermore, the work in [Maia et al., 2017] prioritizes A-phases over R-phases.

the long task problem, i.e., task sets that contain some tasks with short deadlines and others with long WCETs are trivially unschedulable due to blocking from lower priority tasks. This problem has been identified in the state-of-the-art, e.g., see Section 6 of [Davis et al., 2016]. Therefore, to circumvent this problem, we only selected benchmarks from Table 2 of [Davis et al., 2017] such that the $PD_i + MD_i$ of each task τ_i remains in the range of 2000 to 12000. Task parameters considered for the case study experiments are given in Table 3.2. For all the tasks, we assume that C_i^E is equal to the task's processor demand PD_i. Similarly, the value of $C_i^A + C_i^R$ is considered equal to the task's memory demand MD_i.[5] Finally, the total WCET of task τ_i is given by $C_i = C_i^A + C_i^E + C_i^R$.

Name	PD_i	MD_i	$PD_i + MD_i$
cnt	7765	573	8338
compressdata	3166	494	3660
compress	8793	993	9786
cover	3661	696	4357
duff	3121	553	3674
expint	8058	716	8774
fdct	5923	1088	7011
fir	6938	1207	8145
insertsort	2218	415	2633
jfdctint	7771	1086	8857
ludcmp	8278	768	9046
nsichneu	8648	1582	10230
petrinet	2272	438	2710
qurt	8663	735	9398
recursion	5564	907	6471
select	7211	986	8197

Table 3.2: Benchmark parameters used in the experiments.

By default, we consider a multicore platform with 4 cores and a task set size of 32 tasks with 8 tasks per core. For task-to-core mapping, we randomly map tasks to cores while ensuring that each core has the same number of tasks and that the core utilization for each core is the same. To assign the benchmark parameters to tasks mapped on the cores, we randomly select a benchmark from Table 3.2 and assign its C_i^A, C_i^E, C_i^R, and C_i, values to a task. We then randomly generate tasks' utilizations U_i using the UUnifast discard algorithm [Emberson et al., 2010]. Having assigned the values of C_i and U_i, we generate the task period by using the equation $T_i = C_i/U_i$. The task priorities are then assigned using rate monotonic [Liu and Layland, 1973] and task deadlines are equal to their periods.

In the case study, we performed two experiments by varying: 1) the core utilization (i.e., utilization of each core); 2) the number of cores in the system, and compared the performance of all the analyzed approaches in terms of task set schedulability. In the results for the case study

[5] Although the proposed analysis is valid for all the values of A- and R-phases, we assume that MD_i is equally divided among the A- and R-phases for the experimental evaluation.

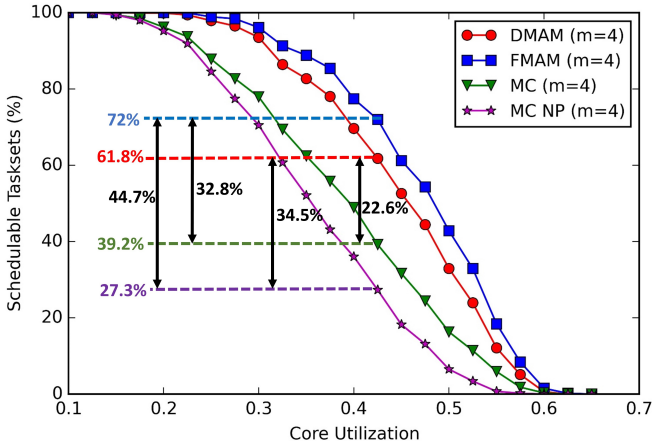
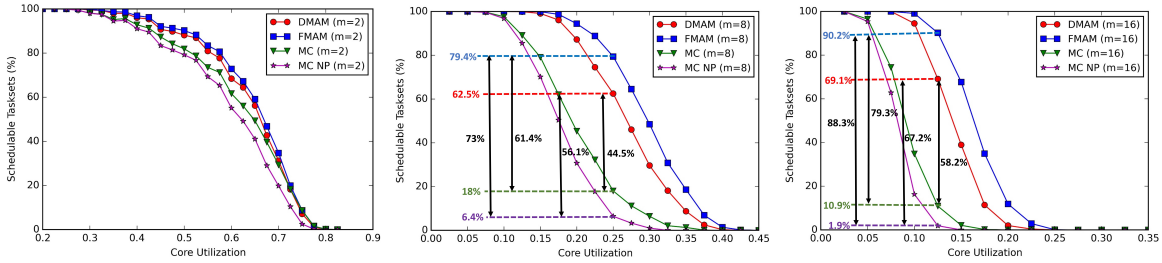


Figure 3.10: Varying core utilization

(and also for the experiments in Section 3.5.2), the analysis for the dedicated memory access model is marked as "DMAM" whereas the analysis for the fair memory access model is marked as "FMAM". Similarly, the memory-centric scheduling approach of [Schwäricke et al., 2020] is marked as "MC" and the memory-centric scheduling approach of [Schwäricke et al., 2020] without global memory preemption is marked as "MC-NP". In each experiment, 1000 task sets were generated per point.

1. Core Utilization: In this experiment, we vary the core utilization of each core in the range of 0.025 to 1 in steps of 0.025. As shown in Figure 3.10, the schedulability using all the approaches decreases with the increase in core utilization. This is intuitive as increasing core utilization increases tasks utilization, which directly impacts the task period/deadline. We observe that none of the approaches were able to schedule tasksets with core utilization higher than 0.60. This is mainly due to the higher number of tasks in the task set under the default configuration, i.e., for 4 cores we have 32 tasks in the taskset. Effectively, this results in increasing bus contention between tasks leading to reduced schedulability. However, we can see in Figure 3.10, that the FMAM and DMAM analyses outperform the MC and MC-NP analyses. For instance, at the core utilization value of 0.425, FMAM analysis was able to schedule 32.8% more tasksets as compared to MC analysis and 44.7% more taksets as compared to MC-NP analysis. This is mainly due to two reasons. The first reason is that unlike [Schwäricke et al., 2020], the proposed analysis provides a fine-grained bus contention analysis using different cases, that account for different scheduling scenarios that can be observed on the core under analysis as well as remote cores. This results in tightening the bound on bus contention suffered by the tasks. The second reason is that the proposed work shares the bus among all cores in a more fair manner (e.g. FMAM), whereas the analysis of [Schwäricke et al., 2020] assigns the bus to the higher priority cores. In such a case, there can be a scenario in which even the highest priority task running on the lowest priority core may suffer bus contention from all the tasks released on the higher priority cores. On the contrary, the proposed analysis bounds the bus contention on the basis of the number of jobs/memory phases that can suffer/cause bus contention during the response time of the task under analysis.

We also observe that the FMAM performs the best among all the analyses, whereas MC-NP



(a) Varying core utilization for $m = 2$ (b) Varying core utilization for $m = 8$ (c) Varying core utilization for $m = 16$

Figure 3.11: Varying the number of cores and core utilization

performs the worst. This is because FMAM distributes the bus among all the cores in a fair manner and due to the fine-grained bus contention analysis for FMAM. The MC-NP performs worse than MC as tasks can additionally suffer the bus contention from lower priority cores due to the non-preemptive execution of memory phases under MC-NP.

Even though we vary the core utilization from 0.025 to 1, Figure 3.10 and the rest of the figures in the experimental section mainly show the taskset schedulability for the core utilization values which is of interest,[6] i.e., Figure 3.10 only show taskset schedulability between 0.10 to 0.65 core utilization as the taskset schedulability of all approaches is 100% for all values below 0.10 core utilization and 0% for all values above 0.65 core utilization.

2. Number of Cores: In this experiment, we redo the previous experiment by varying the number of cores along with the core utilization. The number of cores (m) was varied from 2 to 16 along with core utilization that was varied from 0.025 to 1 in steps of 0.025. The percentage of task sets that were deemed schedulable by all approaches for different values of m is shown in Figure 3.11. We can see that by increasing the number of cores, the number of tasksets that were deemed schedulable by all the approaches decreases. This is mainly due to the fact that increasing the number of cores also increases the number of remote cores and the number of tasks in the taskset, which results in increasing the bus contention that can be suffered by the task under analysis from the remote cores.

We observe that the performance gain of FMAM and DMAM analysis against MC and MC-NP increased with an increase in the number of cores, i.e., $m = 8, 16$. For instance, at the core utilization value of 0.125, the FMAM analysis was able to schedule 79.3% more tasksets as compared to the MC analysis and 88.3% more taksets compared to the MC-NP analysis for $m = 16$ as shown in Figure 3.11c.

On the other hand, we observe that the performance gain of the FMAM and DMAM analysis against MC and MC-NP is reduced by decreasing the number of cores to $m = 2$. In fact, the MC analysis performed almost the same as FMAM and DMAM for some core utilization values. We explain these variations in the gain as follows:

An increase in the number of cores results in an increase in the number of remote cores. However, the bus contention suffered by the tasks using the proposed analysis depends on several

[6]This is also the case for the experimental results in all the chapters of this dissertation.

cases/sub-cases. This implies that even when the number of cores is increased, the bus contention suffered by the task under analysis may not increase significantly as there may be a few tasks from remote cores that participate in the bus blocking.

On the other hand, an increase in the number of cores results in an increase in the number of higher priority cores, which can cause bus contention to the lowest priority core. Consequently, the MC and MC-NP analyses can be significantly impacted as even the highest priority task (i.e., the task that has the smallest period/deadline) running on the lowest priority core can suffer bus contention from all the tasks released on all the higher priority cores.

Interestingly, we also observe that for higher values of m, the difference between FMAM and DMAM was significant. For instance, FMAM was able to schedule up to 67.7% tasksets whereas DMAM was able to schedule only 38.9% at 0.15 core utilization for $m = 16$ as shown in Figure 3.11c.

3.5.2 Experiments using Synthetic Tasks

In this section, we will explain the experiments that were performed using synthetic task sets to compare the performance of DMAM, FMAM, MC and MC-NP approaches. The default configuration was a multicore platform with 4 cores and a task set size of 32 tasks with 8 tasks per core. Task utilizations were generated using the Uunifast-discard algorithm [Emberson et al., 2010]. Task periods were generated using log-uniform distribution in the range of [100,1000]. In each experiment, 1000 task sets were generated per point.

The WCET C_i of each task τ_i was obtained by the product $U_i \times T_i$. The memory demands MD for each task was assigned randomly in the range of $[10\%, 50\%] \times C_i$, i.e., $MD = rand(10\%, 50\%) \times C_i$. The values of C_i^A, C_i^E and C_i^R are then chosen such that $C_i^A = C_i^R = MD/2$ [7] and $C_i^E = C_i - (C_i^A + C_i^R)$. Task deadlines were implicit with priorities assigned using Rate-Monotonic [Liu and Layland, 1973].

We performed several experiments by varying: 1) the core utilization; 2) the number of cores; 3) the task memory demands; and 4) the task periods.

1. Core Utilization: In this experiment, we varied each core utilization between 0.025 and 1 in steps of 0.025 and plotted the number of task sets that were deemed schedulable by all the analyzed approaches, i.e., DMAM, FMAM, MC, and MC-NP. The percentage of task sets that were deemed schedulable using all the approaches for each core utilization value are shown in Figure 3.12. As shown in 3.12, the schedulability of all the approaches decreases with an increase in the core utilization. This is intuitive as increasing the core utilization can increase the values of C_i, C_i^A, C_i^E, and C_i^R that can eventually increase the interference/blocking from the local core and bus contention from remote cores. We note that the overall task set schedulability for all the approaches is quite low as no tasksets were schedulable at 0.50 core utilization. This is intuitive as the MD value of tasks can be up to 50% of their WCET, which can directly contribute to the bus contention that can be suffered/caused by tasks. We observe that the FMAM analysis performed

[7] Note that for the analysis in [Schwäricke et al., 2020] we consider a single memory phase of length MD.

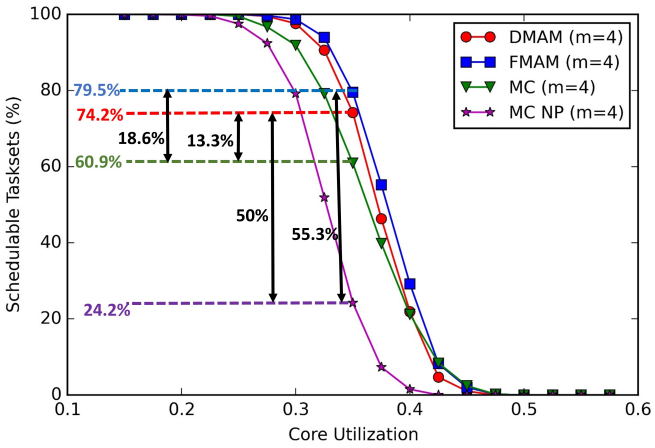


Figure 3.12: Varying core utilization

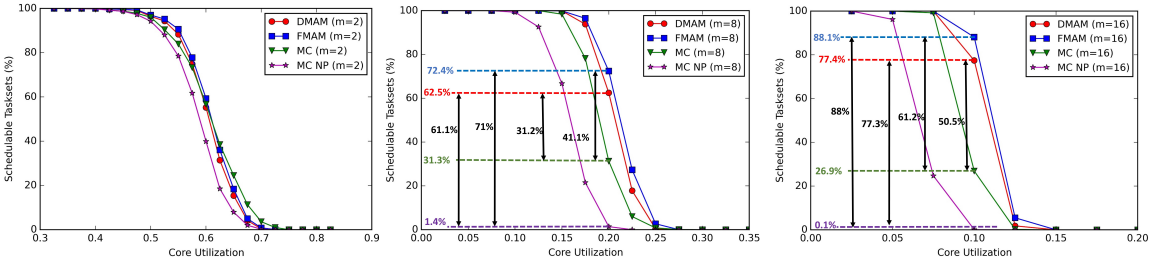
slightly better than DMAM, as expected. As shown in 3.12, the MC analysis performs better than MC-NP. This is intuitive because tasks can additionally suffer the bus contention from lower priority cores due to the non-preemptive execution of memory phases under MC-NP.

We can also see in Figure 3.12 that the proposed analysis for DMAM and FMAM outperforms the memory-centric scheduling analysis of [Schwäricke et al., 2020]. In particular, at the core utilization value of 0.35, FMAM can schedule up to 55.3% more tasksets as compared to MC-NP and up to 18.6% more tasksets as compared to MC. Similarly, at the core utilization value of 0.35, DMAM can schedule up to 50% more tasksets as compared to MC-NP and up to 13.3% more tasksets as compared to MC. As discussed earlier, the improved performance of DMAM and FMAM over MC and MC-NP is mainly due to a more fine-grained bus contention analysis used by DMAM and FMAM.

Interestingly, we observe that no taskset is schedulable after the core utilization value of 0.475 using any of the approaches as shown in Figure 3.12. On the contrary, almost all the approaches were able to schedule tasksets up to 60% core utilization under the case study, i.e., see Figure 3.10. This is because the value of MD is quite small in benchmark parameters given in Table 3.2 whereas the value of MD can go up to $50\% \times C_i$ while randomly generating the tasks.

2. Number of Cores: In this experiment, we vary the number of cores along with the core utilization, keeping default values for all other parameters. The number of cores (m) varied from 2 to 16, and for each value of (m), the core utilization varied from 0.025 to 1 in steps of 0.025. The percentage of task sets that were deemed schedulable for different values of m by all the approaches are shown in Figure 3.13. We can see in Figure 3.13 that by increasing the number of cores, the number of task sets that were deemed schedulable by all the approaches decreases. This is mainly due to the fact that by increasing the number of cores, the number of tasks in the taskset also increases, which results in increasing the bus contention that can be suffered by the task under analysis from the remote cores/higher priority cores. For example, for two cores all task sets were deemed schedulable by all the approaches at the core utilization of 0.35 but no task set was schedulable at the same core utilization when the value of m is increased to 8 or 16.

Figure 3.13b and 3.13c show that the FMAM, and DMAM analysis can outperform MC and



(a) Varying core utilization for $m = 2$ (b) Varying core utilization for $m = 8$ (c) Varying core utilization for $m = 16$

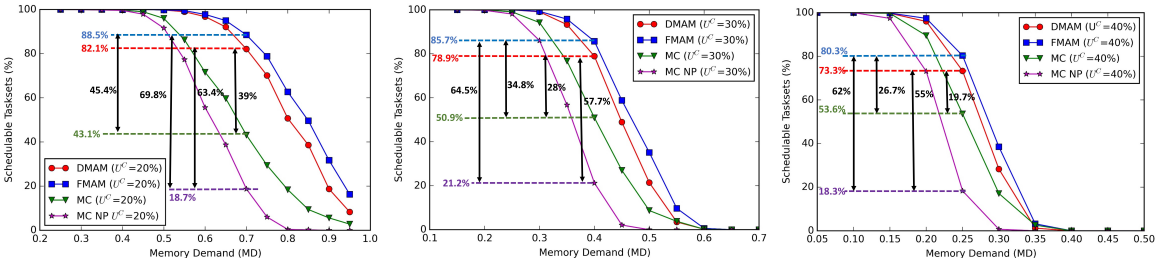
Figure 3.13: Varying the number of cores and core utilization

MC-NP analysis when the value of m is increased to 8 and 16. For instance, at the core utilization value of 0.20, the FMAM analysis can schedule up to 41.1% more tasksets as compared to MC and 71% more tasksets as compared to MC-NP for $m = 8$ (see Figure 3.13b). Similarly, at the core utilization value of 0.20, the DMAM analysis can schedule up to 31.2% more tasksets as compared to MC and 61.1% more tasksets as compared to MC-NP for $m = 8$ (see Figure 3.13b). These performance gains were further increased for $m = 16$ as shown in Figure 3.13c. However, all the approaches perform almost similarly for $m = 2$. In fact, MC analysis was able to perform better than FMAM and DMAM analysis for some of the core utilization values for $m = 2$ as shown in Figure 3.13a. We explain these performance gains as follows.

As discussed earlier, MC analysis gets significantly impacted by increasing/decreasing the number of cores as the analysis is based on processor priority whereas the FMAM/DMAM analysis is based on FCFS bus arbitration in which the bus contention depends on several cases and subcases. In particular, for $m = 2$, under the MC analysis tasks running on only one core (i.e., all except the highest priority core) can suffer bus contention whereas under the DMAM and FMAM analyses, the tasks running on both the cores can suffer bus contention, i.e., 2x more than MC analysis, due to the FCFS bus arbitration. On the contrary, for the $m = 16$, under the MC analysis tasks running on 15 cores can suffer bus contention whereas under the DMAM and FMAM analyses, the tasks running on 16 cores can suffer bus contention, i.e., 1.066x more than MC analysis, due to the FCFS bus arbitration. Therefore, the FMAM and DMAM analyses performed significantly better than the MC analysis at higher values of m, i.e., $m = 8$, 16, but performed slightly worse than MC at some core utilization values for the lower value of m, i.e., $m = 2$.

3. Task Memory Demands: In this experiment, we varied the Memory Demand (MD) of tasks w.r.t their WCET and analyzed its impact on the task set schedulability. Effectively, we used the value of MD to determine C_i^A, C_i^E, and C_i^R such that $C_i^A + C_i^R = MD$ and $C_i^E = C_i - (C_i^A + C_i^R)$. The value of MD was varied from 0.05 to 0.95 (i.e., 5% to 95%) in steps of 0.05 and the number of task sets that were deemed schedulable by all the approaches are plotted in Figure 3.14. We choose different sets of core utilizations (denoted by U^C), i.e., 20%, 30%, and 40% to show the impact of MD on task set schedulability.

We can see in Figure 3.14 that for the values of core utilization of 20%, 30%, and 40%, the percentage of tasksets that were deemed scheduled using all the approaches decreases with the in-



(a) Varying MD for 20% core utilization (b) Varying MD for 30% core utilization (c) Varying MD for 40% core utilization

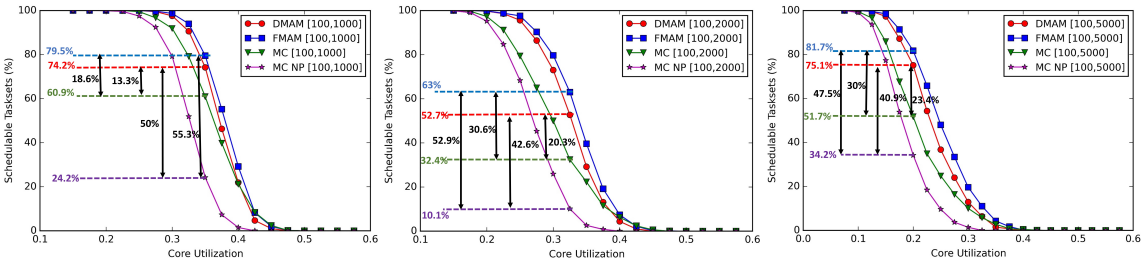
Figure 3.14: Varying the tasks' Memory Demand (MD)

crease in MD. This is intuitive, as for higher values of MD, the values of C_i^A, and C_i^R also increase which may result in increasing bus contention. Furthermore, we observe that for lower values of core utilization, the number of task sets that were deemed schedulable by all approaches was much higher even for larger values of MD. For example, at a core utilization of 20% (i.e., U^C=20%), tasks with very high memory demand, i.e., up to 80% of their WCET, were still schedulable as shown in Figure 3.14a. However, the taskset schedulability decreases rapidly for higher values of core utilization as shown in Figure 3.14b, and 3.14c. Finally, we can also observe that the FMAM and DMAM analysis outperformed the MC and MC-NP analysis. For instance, the FMAM analysis was able to schedule up to 45.4% more tasksets as compared to MC and up to 69.8% more tasksets as compared to MC-NP for MD value of 70% at 20% core utilization, as shown in Figure 3.14a.

4. Task Periods: In this experiment, we varied the period range of tasks and analyzed its impact on schedulability. As we generate the WCET C_i of tasks using the task periods T_i, i.e., $C_i = U_i \times T_i$, which is then used to generate C_i^A, C_i^E and C_i^R, therefore, the value of task periods can significantly impact schedulability.

In this experiment, the core utilization was varied for three different period ranges, i.e., [100,1000], [100,2000], [100,5000] and the percentage of task sets that were deemed schedulable using all the approaches is shown in Figure 3.15. We observe that an increase in the period range has a negative impact on the task set schedulability. We explain these variations as follows:

Increasing the task period increases the WCET of tasks due to the relation between C_i and T_i, i.e., $C_i = U_i \times T_i$. This in turn increases the blocking from one job of a lower priority task, i.e.,



(a) 100 to 1000 task period range (b) 100 to 2000 task period range (c) 100 to 5000 task period range

Figure 3.15: Varying the tasks' period Range and core utilization

a larger period leads to a larger WCET which causes a larger blocking from lower priority tasks. This implies that increasing the task period range increases the blocking caused by a lower priority task. This increase in lower priority blocking also increases the length of the level-i busy window which may result in converging the level-i busy window at a later stage due to additional jobs released by higher priority tasks. This causes a degradation in task set schedulability when the period ranges are increased. However, we can still see that even for higher values of task periods the proposed FMAM and DMAM analyses dominate the MC and MC-NP analyses.

3.6 Chapter Summary

This chapter addresses problem **P1.** by proposing the bus contention analysis for the 3-phase model considering partitioned fixed-priority non-preemptive scheduling and FCFS bus arbitration policy. We show that the bus contention suffered by the tasks executing on a multicore platform depends on the underlying memory access model. As a consequence, we present the contention analysis for two memory access models referred to as the dedicated memory access model and the fair memory access model. For each model, the maximum bus contention is derived using different cases and sub-cases to emulate different scheduling scenarios that can happen when concurrent tasks execute on a multicore platform and try to access the bus. This allows us to achieve tighter bounds on the maximum bus contention that can be suffered by the tasks as well as improves taskset schedulability.

We also show how the maximum bus contention suffered by tasks can be integrated into their WCRT analysis to perform bus contention-aware schedulability analysis. Experimental evaluation shows that the proposed analysis can improve the number of task sets that are deemed schedulable by up to 88 percentage points, in comparison to a state-of-the-art approach.

Chapter 4

Evaluating the Impact of Bus Arbitration Policy on Bus Contention

This chapter addresses problem **P2.** by investigating the impact of the bus arbitration policy on the bus contention suffered by 3-phase tasks. Specifically, this chapter considers a fairer bus arbitration scheme such as the *Round-Robin* (RR). We analyze the maximum bus contention analysis that can be suffered by 3-phase tasks considering the RR bus arbitration policy. We show that the bus contention suffered by 3-phase tasks can be significantly reduced using the RR bus policy in comparison to the FCFS bus arbitration policy considered in Chapter 3.

We show that if the blocking caused by a lower priority task in the multicore platform is computed in a manner similar to that of the uniprocessors, it can yield unsafe bounds. To address this issue, we propose an algorithm to correctly quantify the maximum blocking that can be caused by a lower priority task under the 3-phase task model executing on a multicore platform. Finally, the bus contention-aware WCRT analysis is formulated by integrating the maximum bus contention that can be suffered by tasks.

The main **contributions** of this chapter are as follows.

1. We propose the bus contention analysis for the 3-phase task model executing on multicore architectures that use partitioned fixed-priority scheduling and Round-Robin (RR) bus arbitration policy. The analysis is fine-grained such that it takes into account the number of memory phases and the maximum number of memory requests per memory phase.

2. We show that when computing the WCRT of a task τ_i under the 3-phase execution model scheduled using partitioned fixed-priority non-preemptive scheduling on a multicore system, each task τ_j having a lower priority than τ_i can block the execution of τ_i and each τ_j may contribute differently to the total bus contention. Therefore, we propose an algorithm to accurately quantify the contribution of a lower priority task τ_j in order to maximize the delay that can be suffered by a task τ_i during its response time interval. We then integrate the resulting bounds on the bus contention and blocking from lower-priority tasks into the schedulability analysis of partitioned fixed-priority non-preemptive systems.

3. We perform an extensive experimental evaluation under different settings to show the effectiveness of the proposed analysis in comparison to the FCFS-based analysis, i.e., both the DMAM and the FMAM, presented in Chapter 3. Experimental results show that the proposed RR-based bus contention analysis improves taskset schedulability by up to 70% percentage points in comparison to the FCFS-based bus contention analysis.

Chapter Organization: The rest of the chapter is organized as follows: Section 4.1 describes the system model. A motivational example is presented in Section 4.2 to show the potential benefits of using the RR-based bus arbitration policy in comparison to the FCFS-based bus arbitration policy. The bus contention analysis for the 3-phase task model considering the RR bus arbitration policy is presented in Section 4.3. In Section 4.4, we present an algorithm to accurately quantify the impact of blocking caused by lower priority tasks. The bus contention aware schedulability analysis is presented in Section 4.5. The experimental results are presented in Section 4.6. Finally, the chapter summary is presented in Section 4.7.

4.1 System Model

The system and task models are identical to that of Section 3.1 except that we consider the RR bus arbitration policy as explained in the memory bus and execution model below.

4.1.1 Memory Bus Model

We assume that the bus arbitration policy is Round-Robin (RR). In the RR bus arbitration policy, a core can access the bus only during the *bus access slot* assigned to that core. In this chapter, we use the term *bus slot* or *slot* to refer to the bus access slot. The bus slot for a given core is said to be *active* only when that core starts performing memory requests during its assigned bus slot.[1] The maximum number of memory requests that a core can perform during its assigned bus slot depends on the slot size SS. We assume that SS is an integer multiple of t^{mem} such that $SS \geq t^{mem}$, i.e., at least one memory request can be performed during a single bus access slot. It is assumed that the value of SS is the same for each core. Due to the work-conserving nature of the RR bus arbitration policy, if a core does not have any pending memory requests, it will not use its slot and the system can then grant the slot to the next core waiting for the bus.

4.1.2 Execution Model

Whenever a task is ready to execute on a given core, the core requests access to the memory bus in order to execute the A-phase of the ready task. If the bus is free, the slot assigned to the core becomes active immediately and the A-phase starts executing. However, if the bus is busy serving the memory requests of co-running tasks, the requesting core has to wait for its turn i.e., for the completion of bus slots assigned to the other cores. Once the slot for the core becomes active,

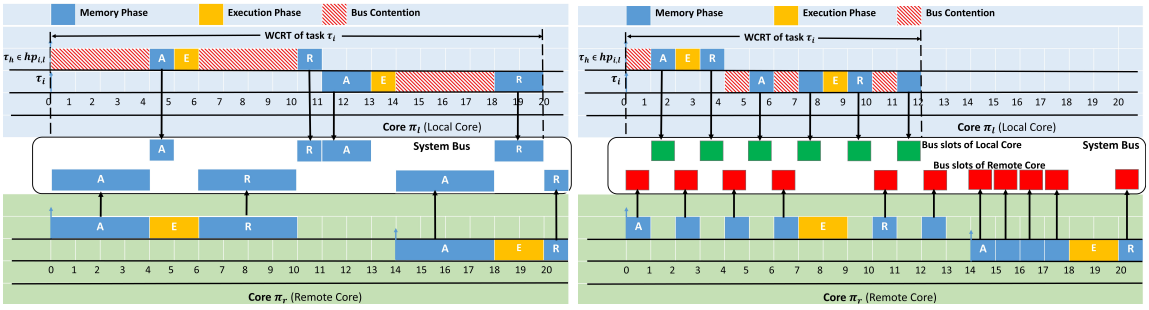
[1] Note that bus slots are not statically assigned and a core only requests a bus slot if it has any pending memory request.

the core starts executing the A-phase. If all the memory requests of an A-phase cannot be served in one bus slot, the core waits for its next active slot to execute the pending memory requests of the same A-phase. Once the *A-phase is completed, the core releases the bus, even if there is time available in the bus slot,* in order to execute the E-phase of the same task. Once the task completes the execution of its E-phase, the core may have to wait for its bus slot to execute the R-phase of the same task. Once the *R-phase is completed, the core releases the bus even if the slot is not fully utilized by the core.*

4.2 Motivational Example

The bus contention analysis presented in Chapter 3 assumes that the bus arbitration policy is FCFS. Hence, a memory phase that executes on the local core can be served only after the completion of the memory phases of tasks executing on all the remote cores. This assumption can be pessimistic in some scenarios as we show using the example scenario depicted in Figure 4.1. Figure 4.1a shows the execution of two tasks on the local core π_l, i.e., τ_h and task τ_i, along with different job releases on a remote core π_r during the same time interval. We can clearly see that the memory phases of tasks running on the local core are smaller, i.e., they issue a smaller number of memory requests. On the contrary, the memory phases of tasks running on the remote core are larger, i.e., they issue a larger number of memory requests. Under FCFS, in the worst case, a memory phase running on the local core has to wait for the completion of all the memory requests made by a memory phase running on the remote core. Consequently, we see in Figure 4.1a that tasks executing on the local core π_l suffer larger bus contention under the FCFS bus arbitration scheme.

On the other hand, RR bus arbitration makes use of bus slots in which a core can use the bus only during its assigned slot. Thus, the bus contention that can be suffered/caused by the memory phases running on the local/remote core depends on the number of memory requests that can be performed during those bus slots. As shown in Figure 4.1b, the slot size is set such that each core can execute at most one memory request during its bus slot. Consequently, we can see in Figure 4.1b that the same tasks running on the local core, i.e., τ_h and τ_i, suffer lesser bus contention under the RR-based bus arbitration policy.



(a) Example scenario for the FCFS bus arbitration policy

(b) Example scenario for the RR bus arbitration policy

Figure 4.1: WCRT of tasks under (a) FCFS and (b) RR bus arbitration policy

Simple example scenarios discussed above show that the bus contention that can be suffered by tasks executing on a multicore platform can be reduced by using an RR-bases bus arbitration scheme. In the following sections, we will explain how to upper bound the bus contention of 3-phase tasks under the RR bus arbitration scheme.

4.3 Bus Contention Analysis for RR-based Bus Arbitration Policy

To evaluate the impact of bus arbitration policies on the bus contention and WCRT of tasks, this section presents the bus contention analysis for the 3-phase task model considering a Round-Robin (RR) based bus arbitration policy.

When computing the maximum bus contention that can be suffered by tasks under the RR-based bus arbitration policy, we assume that task τ_i is the task under analysis, executing on the local core π_l of a multicore platform. Our goal is to bound the maximum bus contention that can be suffered by all jobs released by all tasks in $hep_{i,l}$ (including τ_i) on core π_l, during a level-i busy window $W_{i,l}$. For the sake of simplicity, we start by computing the bus contention that can be caused due to tasks executing on a single remote core π_r. We later generalize our analysis to account for the bus contention that can be caused by multiple remote cores.

Under an RR-based bus arbitration policy, tasks executing on the local core π_l can suffer bus contention when they have to wait for the completion of bus access slots assigned to all other remote cores. In the worst case, the bus access slot(s) required by tasks executing on the local core π_l may become active after the completion of the bus access slot(s) utilized by a remote core π_r. This means that the bus contention that can be suffered by tasks executing on the local core π_l not only depends on the number of bus slots required by tasks executing on the local core but also on the number of bus slots required by tasks executing on the remote core π_r. Building on this insight, we propose a two-step solution to bound the maximum bus contention that can be suffered by tasks executing on the local core π_l due to other co-running tasks on a remote core π_r, within a time interval of length $W_{i,l}$. The two steps are explained as follows:

- **Step 1:** Bound the maximum *number of bus slots* required by tasks executing on the local core π_l and remote core π_r within $W_{i,l}$. This step is discussed in detail in Section 4.3.1.
- **Step 2:** Compute the *maximum bus contention* that can be suffered by tasks executing on the local core π_l from a remote core π_r during $W_{i,l}$. This step is discussed in detail in Section 4.3.2.

4.3.1 Step 1: Bounding the Maximum Number of Bus Slots required by the Local/Remote Core

In this step, we compute the maximum number of bus slots required by the *local core* π_l and *remote core* π_r during any time window of length $W_{i,l}$ using the following two lemmas.

Lemma 4.1. The maximum number of bus slots required by tasks running on the local core π_l during any time interval of length $W_{i,l}$ is upper-bounded by $\beta_{\pi_l}(W_{i,l})$, where

$$\beta_{\pi_l}(W_{i,l}) = \sum_{\tau_h \in hep_{i,l}} \eta_h^+(W_{i,l}) \times \left(\left\lceil \frac{MD_h^A \times t^{mem}}{SS} \right\rceil + \left\lceil \frac{MD_h^R \times t^{mem}}{SS} \right\rceil \right) + \left\lceil \frac{MD_j^A \times t^{mem}}{SS} \right\rceil + \left\lceil \frac{MD_j^R \times t^{mem}}{SS} \right\rceil \quad (4.1)$$

In Equation 4.1, MD_h^A (resp. MD_h^R) is the maximum number of memory requests issued during the A-phase (resp. R-phase) of a task $\tau_h \in hep_{i,l}$. Similarly, MD_j^A (resp. MD_j^R) is the maximum number of memory requests issued during the A-phase (resp. R-phase) of a task $\tau_j \in lp_{i,l}$.

Proof. By definition, the maximum number of memory requests that can be generated by a task $\tau_h \in hep_{i,l}$ during its A- and R-phase is upper bounded by MD_h^A and MD_h^R, respectively. Also, we know that the maximum time needed to perform one memory request is given by t^{mem}. Consequently, the term $MD_h^A \times t^{mem}$ (resp. $MD_h^R \times t^{mem}$) gives the maximum time needed to complete the A-phase (resp. the R-phase) of task τ_h on core π_l. Knowing that under the RR-based bus arbitration policy, a core can access the bus only during its assigned bus slot for at most SS time units. Therefore, one job of task $\tau_h \in hep_{i,l}$ that execute on core π_l during $W_{i,l}$ will use $\left\lceil \frac{MD_h^A \times t^{mem}}{SS} \right\rceil + \left\lceil \frac{MD_h^R \times t^{mem}}{SS} \right\rceil$ bus slots and all the jobs of τ_h that execute during $W_{i,l}$, i.e., upper bounded by $\eta_h^+(W_{i,l})$, will require $\eta_h^+(W_{i,l}) \times \left(\left\lceil \frac{MD_h^A \times t^{mem}}{SS} \right\rceil + \left\lceil \frac{MD_h^R \times t^{mem}}{SS} \right\rceil \right)$ bus slots. Hence, $\sum_{\tau_h \in hep_{i,l}} \eta_h^+(W_{i,l}) \times \left(\left\lceil \frac{MD_h^A \times t^{mem}}{SS} \right\rceil + \left\lceil \frac{MD_h^R \times t^{mem}}{SS} \right\rceil \right)$ bounds the maximum number of bus slots required by all tasks in $hep_{i,l}$ during $W_{i,l}$.

Finally, we know that due to non-preemptive scheduling, if task τ_i is not the lowest priority task executing on core π_l, it can suffer blocking from one job of a lower priority task, e.g., $\tau_j \in lp_{i,l}$. Since, that one blocking job of τ_j will also require bus slots to complete its A- and R-phase, the maximum number of bus access slots required by that job of task τ_j is given by $\left\lceil \frac{MD_j^A \times t^{mem}}{SS} \right\rceil + \left\lceil \frac{MD_j^R \times t^{mem}}{SS} \right\rceil$. For now, it can be assumed that the task τ_j is the task that issues the maximum number of memory requests among all tasks in $lp_{i,l}$. Later in Section 4.4, we present an algorithm to correctly select a specific task from $lp_{i,l}$. The Lemma follows. □

Lemma 4.2. The maximum number of bus slots required by tasks running on a remote core π_r during any time interval of length $W_{i,l}$ is upper-bounded by $\beta_{\pi_r}(W_{i,l})$, where

$$\beta_{\pi_r}(W_{i,l}) = \sum_{\tau_u \in \Gamma_r'} \eta_u^+(W_{i,l}) \times \left(\left\lceil \frac{MD_u^A \times t^{mem}}{SS} \right\rceil + \left\lceil \frac{MD_u^R \times t^{mem}}{SS} \right\rceil \right) \quad (4.2)$$

In Equation 4.2, the term MD_u^A (resp. MD_u^R) is the maximum number of memory requests issued during the A-phase (resp. R-phase) of a task $\tau_u \in \Gamma_r'$

Proof. The proof directly follows from Lemma 4.1. However, we need to account for the bus slots required by all memory phases of all tasks released on a remote core π_r during any time interval of length $W_{i,l}$. □

4.3.2 Step 2: Bounding Maximum Bus Contention

Having bounded the maximum number of bus slots required by tasks executing on the local core π_l and a remote core π_r during $W_{i,l}$, we can now compute the maximum bus contention $Bus_{i,r}(W_{i,l})$ that can be suffered by tasks executing on the local core π_l from co-running tasks on a remote core π_r. Depending on the values of $\beta_{\pi_l}(W_{i,l})$ and $\beta_{\pi_r}(W_{i,l})$, two cases must be considered.

- **Case 1:** $\beta_{\pi_l}(W_{i,l}) \geq \beta_{\pi_r}(W_{i,l})$, i.e., the maximum number of bus slots required by tasks executing on the local core π_l during $W_{i,l}$ is greater than or equal to the maximum number of bus slots required by tasks executing on a remote core π_r during $W_{i,l}$. This case is discussed in detail in Section 4.3.2.1.
- **Case 2:** $\beta_{\pi_l}(W_{i,l}) < \beta_{\pi_r}(W_{i,l})$, i.e., the maximum number of bus slots required by tasks executing on the local core π_l during $W_{i,l}$ is less than the maximum number of bus slots required by tasks executing on a remote core π_r during $W_{i,l}$. This case is discussed in detail in Section 4.3.2.2.

4.3.2.1 Computing Maximum Bus Contention for Case 1

When the maximum number of bus slots required by tasks executing on the local core π_l is greater than or equal to the maximum number of bus slots required by a remote core π_r during any time interval of length $W_{i,l}$, then, the maximum bus contention is computed using the following lemma.

Lemma 4.3. If $\beta_{\pi_l}(W_{i,l}) \geq \beta_{\pi_r}(W_{i,l})$, then the maximum bus contention that can be suffered by tasks executing on the local core π_l from tasks running on a remote core π_r during any time interval of length $W_{i,l}$ is upper-bounded by $Bus_{i,r}(W_{i,l})$, where

$$Bus_{i,r}(W_{i,l}) = \sum_{\tau_u \in \Gamma_r'} \eta_u^+(W_{i,l}) \times \Big((MD_u^A \times t^{mem}) + (MD_u^R \times t^{mem}) \Big) \tag{4.3}$$

In Equation 4.3, the term $MD_u^A \times t^{mem}$ (resp. $MD_u^R \times t^{mem}$) gives the maximum time required to serve all the memory requests generated during an A-phase (resp. R-phase) of task τ_u executing on core π_r.

Proof. In the worst case, each bus slot required by tasks executing on the local core π_l can only be active after the completion of one bus slot used by tasks executing on the remote core π_r. We know that the number of bus slots required by the remote core, i.e., $\beta_{\pi_r}(W_{i,l})$, are less than or equal to $\beta_{\pi_l}(W_{i,l})$. This means that all the memory requests generated by all tasks executing on core π_r during $W_{i,l}$ can contribute to the bus contention suffered by tasks running on core π_l.

Knowing that each task τ_u assigned to core π_r can execute at most $\eta_u^+(W_{i,l})$ jobs during $W_{i,l}$, and the time required to complete the A-phase (resp. R-phase) of one job of task τ_u is given by $MD_u^A \times t^{mem}$ (resp. $MD_u^R \times t^{mem}$), the total time required to complete the A- and R-phases of all jobs of all tasks that execute on core π_r during $W_{i,l}$ is given by $\sum_{\tau_u \in \Gamma_r'} \eta_u^+(W_{i,l}) \times \Big((MD_u^A \times t^{mem}) + (MD_u^R \times t^{mem}) \Big)$. Equation 4.3 also upper bounds the maximum bus contention that can be

suffered by tasks executing on the local π_l due to tasks released on a remote core π_r during $W_{i,l}$, if $\beta_{\pi_l}(W_{i,l}) \geq \beta_{\pi_r}(W_{i,l})$. The Lemma follows. □

Note that it is also possible to directly compute the maximum bus contention for case 1 using the maximum number of slots required by tasks executing on the remote core π_r during $W_{i,l}$ ($\beta_{\pi_r}(W_{i,l})$) and the size of one bus slot (SS). Effectively, $\beta_{\pi_r}(W_{i,l}) \times SS$ upper bounds the bus contention for case 1. Although this bound is safe, it can be pessimistic as the remote core may not always fully utilize all its $\beta_{\pi_r}(W_{i,l})$ bus slots for SS time units as the utilization of a bus slot depends on the number of memory requests performed during that bus slot. Therefore, Equation 4.3 which is built considering the size of memory phases of tasks, provides a tighter bound on the bus contention that can be caused by a remote core π_r during a time window of length $W_{i,l}$.

4.3.2.2 Computing Maximum Bus Contention for Case 2

For any given time interval of length $W_{i,l}$, if the maximum number of bus slots required by tasks executing on the local core π_l is less than the maximum number of bus slots required by tasks executing on a remote core π_r, then, all bus slots used by the remote core can not contribute to the bus contention. This is mainly because each bus slot required by the local core π_l can only suffer bus contention from at most one slot used by the remote core π_r. Hence, $\beta_{\pi_l}(W_{i,l})$ slots requested by the local core can be served after the execution of at most $\beta_{\pi_l}(W_{i,l})$ bus slots used by the remote core. Consequently, when computing the bus contention that can be suffered by tasks executing on the local core π_l due to co-running tasks executing on core π_r, we need to consider only $\beta_{\pi_l}(W_{i,l})$ bus slots.

Knowing that at most $\beta_{\pi_l}(W_{i,l})$ bus slots used by a remote core π_r can cause bus contention to tasks executing on core π_l, we can upper bound the bus contention that can be caused by core π_r to core π_l during $W_{i,l}$ by simply multiplying $\beta_{\pi_l}(W_{i,l})$ with the size of one bus slot, i.e.,

$$Bus_{i,r}(W_{i,l}) = \beta_{\pi_l}(W_{i,l}) \times SS \tag{4.4}$$

Equation 4.4 assumes that each of $\beta_{\pi_l}(W_{i,l})$ bus slots from the remote core π_r that can contribute to the blocking of core π_l will be fully utilized, i.e., up to SS time units, by tasks that execute on π_r during $W_{i,l}$. This assumption is safe but can be pessimistic as we explain using the below example.

Example 1: *Figure 4.2 shows a schedule of tasks executing on the local core π_l and the remote core π_r, along with the utilization of bus slots by the cores. We can see in the figure that the maximum number of bus slots required by tasks running on the local core π_l during $W_{i,l}$ is seven, i.e., $\beta_{\pi_l}(W_{i,l}) = 7$. Moreover, the maximum number of bus slots required by tasks running on the remote core π_r during $W_{i,l}$ is eight, i.e., $\beta_{\pi_r}(W_{i,l}) = 8$. Since $\beta_{\pi_l}(W_{i,l}) < \beta_{\pi_r}(W_{i,l})$, the local core can suffer bus contention from at most $\beta_{\pi_l}(W_{i,l})$ bus slots from the remote core in this example. While an upper bound on the total bus contention can be computed using Equation 4.4, by looking closely at the utilization of the bus slots in Figure 4.2, we can see that this bound can be very pessimistic. This is mainly because for the scenario shown in Figure 4.2, the active time, i.e., the*

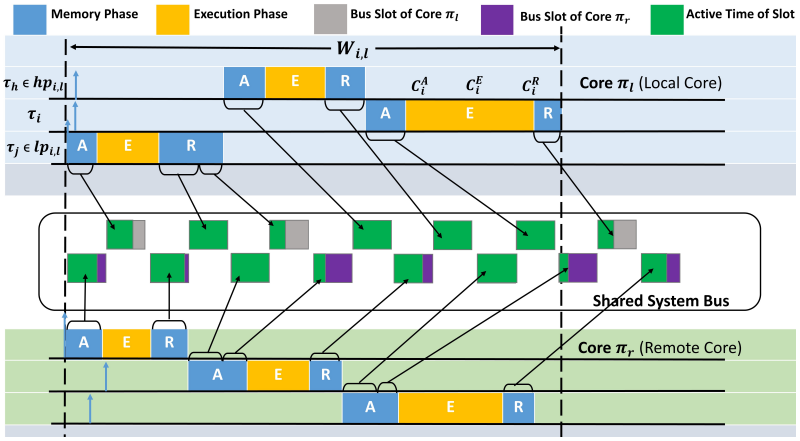


Figure 4.2: Example scenario to derive maximum bus contention for case 2

time in which memory requests are performed during a bus slot, of some of the bus slots used by the remote core π_r is much less than the size of the bus slot, i.e., SS. Hence, assuming that each bus slot of the local core suffers a delay of SS from the remote core can be very pessimistic.

The above example shows that for $\beta_{\pi_l}(W_{i,l}) < \beta_{\pi_r}(W_{i,l})$ a tighter bound on the bus contention can only be obtained by considering the active time of each bus slot accessed by tasks executing on π_r during $W_{i,l}$. Consequently, the following Lemmas are used to incorporate the active time of each bus slot used by the A- and R-phases of all tasks running on the remote core π_r in a time window of length $W_{i,l}$.

Lemma 4.4. If n represents the number of bus slots required to complete the execution of an A-phase of a task τ_u executing on a remote core π_r during $W_{i,l}$, then the active time of each bus slot in the range 1 to $n-1$, used by the A-phase of task τ_u is given by $\sigma^A_{u,r,y}$, where

$$\sigma^A_{u,r,y} = SS \qquad \forall y \in [1, n-1] \tag{4.5}$$

In Equation 4.5, u represents the task index, r the core index, and y is used to represent the bus slot index, i.e., y^{th} bus slot such that $y \in [1, n-1]$.

Proof. We prove that if a task τ_u executing on core π_r requires n bus slots to complete an A-phase, then, τ_u will fully utilize at least $n-1$ bus slots for SS time units.

The A-phase of a task τ_u executing on core π_r will only require n bus slots if it cannot complete its execution within $n-1$ bus slots. This means that for each bus slot, until the $n-1^{th}$ bus slot, the time required to serve the pending memory requests of the A-phase of task τ_u is always greater than SS. Hence, it can only be the case that the n^{th} bus slot used by the A-phase may or may not be fully utilized. The Lemma follows. □

Lemma 4.5. If n represents the number of bus slots required to complete the execution of an A-phase of a task τ_u executing on a remote core π_r during $W_{i,l}$, then the active time of the n^{th} bus slot used by the A-phase of task τ_u is given by $\sigma^A_{u,r,n}$, where

$$\sigma^A_{u,r,n} = (MD^A_u \times t^{mem}) - ((n-1) \times SS) \tag{4.6}$$

Proof. The maximum number of memory requests that can be issued during an A-phase of a task τ_u executing on a core π_r is upper bounded by MD_u^A. Moreover, in the worst case, each memory request can be served in t^{mem} time units. Consequently, the total time required to complete an A-phase of task τ_u on core π_r is given by $MD_u^A \times t^{mem}$.

From Lemma 4.4, if an A-phase of task τ_u needs n bus slots to complete its execution, then, it will fully utilize at least $n-1$ bus slots, and the maximum workload that can be done during $n-1$ bus slots is given by $(n-1) \times SS$. Consequently, the maximum time that can be used by the A-phase of task τ_u during the n^{th} bus slot, i.e., the active time of the n^{th} bus slot, is given by $(MD_u^A \times t^{mem}) - ((n-1) \times SS)$. The Lemma follows. □

Having computed the active time of bus slots used by the A-phases of tasks, we can use the same approach to compute the active time of bus slots used during the R-phases of tasks.

Lemma 4.6. If n represents the number of bus slots required to complete the execution of an R-phase of a task τ_u executing on a remote core π_r during $W_{i,l}$, then the active time of each bus slot in the range 1 to $n-1$, used by the R-phase of task τ_u is given by $\sigma^R_{u,r,y}$, where

$$\sigma^R_{u,r,y} = SS \qquad \forall y \in [1, n-1] \tag{4.7}$$

Proof. The proof directly follows from Lemma 4.4 considering the bus slots required by the R-phase of task τ_u. □

Lemma 4.7. If n represents the number of bus slots required to complete the execution of an R-phase of a task τ_u executing on a remote core π_r during $W_{i,l}$, then the active time of the n^{th} bus slot used by the R-phase of task τ_u is given by $\sigma^R_{u,r,n}$, where

$$\sigma^R_{u,r,n} = (MD_u^R \times t^{mem}) - ((n-1) \times SS) \tag{4.8}$$

Proof. The proof directly follows from Lemma 4.5 considering the bus slots required by the R-phase of task τ_u. □

Having bounded the active time of each bus access slot used by all the memory phases of all tasks released on a remote core π_r during any time interval of length $W_{i,l}$, we will now explain how to compute the maximum bus contention for case 2, i.e., $\beta_{\pi_l}(W_{i,l}) < \beta_{\pi_r}(W_{i,l})$.

Let V be an ordered set that contains the active time of each bus slot utilized by all the memory phases (i.e., both A- and R-phases) released on a remote core π_r during any time interval of length $W_{i,l}$, *sorted in non-increasing order*, i.e.,

$$V = \{\sigma_{r,1}, \sigma_{r,2}, \ldots \sigma_{r,Q} \mid \sigma_{r,x} \geq \sigma_{r,x+1}\} \tag{4.9}$$

where $\sigma_{r,x}$ denotes the active time of bus slot x used by any memory phase (i.e., A- or R-phase) of any task that may execute on core π_r during $W_{i,l}$. In Equation 4.9, the index Q is equal to the maximum number of bus slots used by all tasks that execute on π_r during $W_{i,l}$, i.e., $Q = \beta_{\pi_r}(W_{i,l})$.

Using the above-defined notations, the maximum bus contention for case 2 can be derived using the following lemma.

Lemma 4.8. If $\beta_{\pi_l}(W_{i,l}) < \beta_{\pi_r}(W_{i,l})$, then the maximum bus contention that can be suffered by tasks executing on the local core π_l from tasks running on a remote core π_r during any time interval of length $W_{i,l}$ is upper-bounded by $Bus_{i,r}(W_{i,l})$, where

$$Bus_{i,r}(W_{i,l}) = \sum_{x=1}^{x=\beta_{\pi_l}(W_{i,l})} \sigma_{r,x} \tag{4.10}$$

Proof. Since tasks executing on the local core π_l require $\beta_{\pi_l}(W_{i,l})$ bus slots during $W_{i,l}$, at most $\beta_{\pi_l}(W_{i,l})$ bus slots utilized by tasks executing on the remote core π_r can cause bus contention. However, as we cannot predict the schedule of tasks on the remote core, we do not know which memory phases of which tasks may use these bus slots. Therefore, to maximize the bus contention, we choose $\beta_{\pi_l}(W_{i,l})$ bus slots with the largest active times among all the bus slots used by tasks released on the remote core π_r during any time interval of length $W_{i,l}$. This is achieved by extracting the first $\beta_{\pi_l}(W_{i,l})$ values from set V, that contains the active times of all bus slots utilized on core π_r during $W_{i,l}$ (see Equation 4.9). The Lemma follows. □

Having bounded the maximum bus contention $Bus_{i,r}(W_{i,l})$ that can be suffered by the local core π_l from a remote core π_r during any time interval of length $W_{i,l}$, we can now compute the *total bus contention* that can be suffered by tasks executing on core π_l due to tasks running on all other cores. Knowing that under the RR-based bus arbitration policy, the worst-case scenario may happen when a bus slot required by a given task executing on the local core is served after the completion of one bus slot from each of the remote cores.

Therefore, under the RR-based bus arbitration policy, the *total bus contention* that can be suffered by the local core π_l from *all remote cores* during any time interval of length $W_{i,l}$ is given by $Bus^{max}_{i,l}(W_{i,l})$, where

$$Bus^{max}_{i,l}(W_{i,l}) = \sum_{r=1, r\neq l}^{m} Bus_{i,r}(W_{i,l}) \tag{4.11}$$

4.4 Accurately Estimating the Impact of Lower Priority Blocking

Under FPNP scheduling, the execution of a task τ_i can be blocked due to the execution of a lower priority task, e.g., task τ_j in $lp_{i,l}$, that contributes to the length of level-i busy window. The blocking that τ_j can cause during the level-i busy window is usually upper bounded by choosing τ_j such that it has the maximum execution time among all tasks in $lp_{i,l}$, i.e., $C^{max}_{lp_{i,l}} = \max_{\forall \tau_j \in lp_{i,l}} \{C_j\}$. While this assumption is sound when considering a single-core platform, it may lead to unsafe results for multicore architectures. This is mainly because when τ_j, the task that blocks the execution of τ_i, is executing on a single-core processor the memory bus and all other resources are dedicated to τ_j only. So, in the worst-case τ_j executes for its entire WCET at the beginning of the level-i

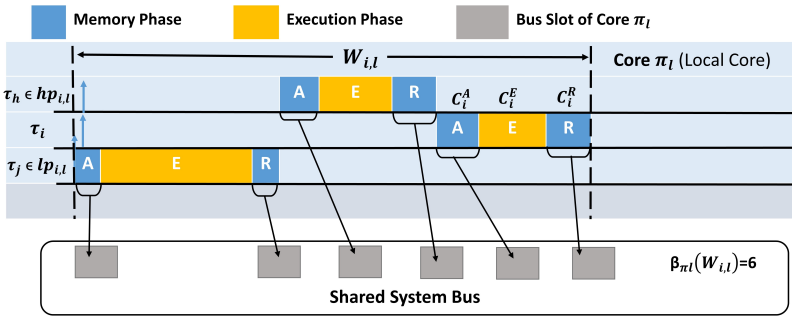


Figure 4.3: Scenario 1 when task $\tau_j \in lp_{i,l}$ cause blocking to task τ_i during $W_{i,l}$

busy window. However, on a multicore processor, task τ_j can suffer execution delays in addition to its WCET due to the bus contention it may suffer during its execution. This bus contention does not entirely depend on the WCET time of τ_j but also on its memory access demand, i.e., the total number of memory requests that can be generated by τ_j's memory phases. For instance, we can have a scenario where another task $\tau_z \in lp_{i,l}$ with $\tau_z \neq \tau_j$, having a smaller WCET than τ_j but a higher memory access demand, may suffer higher bus contention than τ_j. This will eventually result in τ_z contributing more to the blocking of τ_i than τ_j.

To illustrate, consider the two scenarios shown in Figure 4.3 and Figure 4.4.

Scenario 1, depicted in Figure 4.3, shows that a task $\tau_j \in lp_{i,l}$ is blocking the execution of tasks in $hep_{i,l}$ on core π_l at the start of the level-i busy window. τ_j has the largest WCET among all tasks in $lp_{i,l}$ but it has smaller A- and R-phases. We can see in Figure 4.3, that task τ_j requires two bus slots to complete its A- and R-phase. Similarly, all other tasks in $hep_{i,l}$ executing on core π_l require four bus slots to complete their memory phases. Eventually, in scenario 1, the maximum number of bus slots required to complete memory phases of all tasks that execute on core π_l during $W_{i,l}$ is equal to 6, i.e., $\beta_{\pi_l}(W_{i,l}) = 6$.

Figure 4.4, depicts another scenario, where a task $\tau_z \in lp_{i,l}$, with $\tau_z \neq \tau_j$, is causing blocking at the start of level-i busy window on core π_l. τ_z has a smaller overall WCET than τ_j but it has larger A- and R-phases. Consequently, τ_z needs four bus slots to complete its A- and R-phases. All other tasks in $hep_{i,l}$ executing on core π_l are the same as in Figure 4.3 and require four bus slots to complete their memory phases. Eventually, for the execution scenario shown in Figure 4.4, the maximum number of bus slots required to complete the memory phases of all tasks that execute on core π_l during $W_{i,l}$ is equal to 8, i.e., $\beta_{\pi_l}(W_{i,l}) = 8$. As shown previously, the bus contention

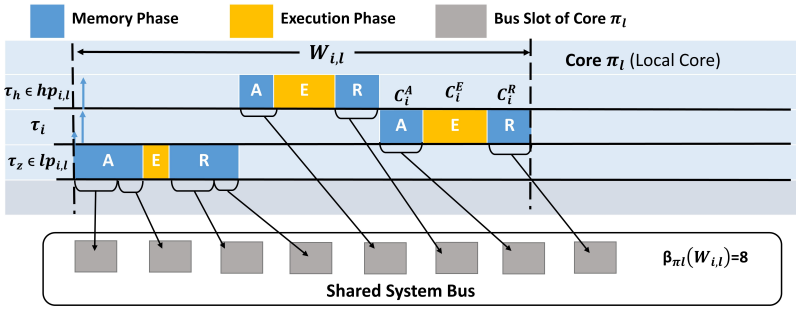


Figure 4.4: Scenario 2 when task $\tau_z \in lp_{i,l}$ cause blocking to task τ_i during $W_{i,l}$

that can be suffered by tasks executing on π_l during $W_{i,l}$ depends on the value of $\beta_{\pi_l}(W_{i,l})$ and a larger value of $\beta_{\pi_l}(W_{i,l})$ may lead to more bus contention. Therefore, when analyzing multicore systems, the scenario depicted in Figure 4.4 may lead to a higher bus contention and eventually a larger level-i busy window than the scenario depicted in Figure 4.3.

To the best of our knowledge, the only existing work in the state-of-the-art that accurately accounts for the lower priority blocking when computing bus contention is presented in [Negrean and Ernst, 2012] (See Equation 11 of [Negrean and Ernst, 2012]). However, the solution provided in [Negrean and Ernst, 2012] is developed considering the generic task model and therefore, can not be used when considering the 3-phase task model.

To accurately quantify the impact of blocking from a given task in $lp_{i,l}$, on tasks that execute on core π_l during $W_{i,l}$, we have to evaluate the impact of bus contention on each task in $lp_{i,l}$ and the resulting length of the level-i busy window. We will then select the task from $lp_{i,l}$ that maximizes the $W_{i,l}$ as the blocking task. We use Algorithm 2 to evaluate how each task in $lp_{i,l}$ can impact bus contention and also the length of the level-i busy window.

Algorithm 2 Computing the maximum delay suffered by the local core π_l during $W_{i,l}$ due to total bus contention and blocking from tasks in $lp_{i,l}$

1: $\alpha_{i,l}^{max}(W_{i,l}) := 0$
2: **for** $\forall \tau_j \in lp_{i,l}$ **do**
3: Compute $\beta_{\pi_l}(W_{i,l})$ using Lemma 4.1 assuming τ_j will cause blocking to task τ_i on core π_l.
4: Compute $\beta_{\pi_r}(W_{i,l})$ using Lemma 4.2.
5: Compute $Bus_{i,r}(W_{i,l})$ using Lemma 4.3 up to Lemma 4.8.
6: Compute the total bus contention $Bus_{i,l}^{max}(W_{i,l})$ using Equation 4.11.
7: $\alpha_{i,l}(W_{i,l}) := Bus_{i,l}^{max}(W_{i,l}) + C_j$
8: **if** $\alpha_{i,l}(W_{i,l}) > \alpha_{i,l}^{max}(W_{i,l})$ **then**
9: $\alpha_{i,l}^{max}(W_{i,l}) := \alpha_{i,l}(W_{i,l})$
10: **end if**
11: **end for**

Algorithm 2 computes the maximum delay that can be suffered by the local core π_l during $W_{i,l}$ due to the total bus contention caused by all the remote cores and blocking caused by one job from any task $\tau_j \in lp_{i,l}$. Since we need to consider all tasks in $lp_{i,l}$, Algorithm 2 iterates over all tasks in $lp_{i,l}$ (lines 2 to 10). For every task $\tau_j \in lp_{i,l}$, it first computes the maximum number of slots required by all tasks that execute on the local core π_l during $W_{i,l}$, i.e., $\beta_{\pi_l}(W_{i,l})$, using Lemma 4.1 (line 3). The maximum number of bus slots required by all tasks released on a remote core π_r during any time interval of length $W_{i,l}$, i.e., $\beta_{\pi_r}(W_{i,l})$, are computed using Lemma 4.2 (line 4).

Values of $\beta_{\pi_l}(W_{i,l})$ and $\beta_{\pi_r}(W_{i,l})$ derived in lines 3 and 4 are then used as input to Lemma 4.3 up to Lemma 4.8, to compute the maximum bus contention that can be caused by a remote core π_r to tasks running on the local core π_l during $W_{i,l}$, i.e., $Bus_{i,r}(W_{i,l})$. The total bus contention that can be suffered by the local core π_l from all remote cores during $W_{i,l}$, i.e., $Bus_{i,l}^{max}(W_{i,l})$, is then computed using Equation 4.11 (line 6). Line 7 computes $\alpha_{i,l}(W_{i,l})$, i.e., the total delay that can be suffered by the local core π_l during $W_{i,l}$ due to the total bus contention caused by all the remote cores, i.e., $Bus_{i,l}^{max}(W_{i,l})$ plus the blocking caused by one job of task τ_j. Lines 8 to 10 compare the

derived values of $\alpha_{i,l}(W_{i,l})$ for each $\tau_j \in lp_{i,l}$ to find $\alpha_{i,l}^{max}(W_{i,l})$ which is the *maximum delay* that can be suffered by the local core π_l during $W_{i,l}$.

4.5 Schedulability Analysis

In this section, we derive the schedulability analysis for the 3-phase task model while using partitioned fixed-priority scheduling, by integrating the bus contention computed in Section 4.3.2 and 4.4. To compute the WCRT of task τ_i, we must first compute the length of the longest level-i busy window $W_{i,l}$ (see definition 2.1.3.1) on the local core, i.e., π_l.

The longest level-i busy window $W_{i,l}$ for the local core π_l w.r.t task τ_i is given by the first positive fixed-point solution of the following equation:

$$W_{i,l} = \sum_{\tau_h \in hep_{i,l}} (\eta_h^+(W_{i,l}) \times C_h) + \alpha_{i,l}^{max}(W_{i,l}) \tag{4.12}$$

where $\eta_h^+(W_{i,l})$ gives the maximum number of jobs released by $\tau_h \in hep_{i,l}$ during any time interval of length $W_{i,l}$. Consequently, the term $\sum_{\tau_h \in hep_{i,l}}(\eta_h^+(W_{i,l}) \times C_h)$ captures the contribution of all the jobs from $hep_{i,l}$ task set during any time interval of length $W_{i,l}$[2]. The term $\alpha_{i,l}^{max}(W_{i,l})$ captures the maximum delay suffered by the local core π_l during any time interval $W_{i,l}$ due to the total bus contention caused by all the remote cores and blocking caused by one job from $lp_{i,l}$ task set. $\alpha_{i,l}^{max}(W_{i,l})$ is computed using Algorithm 2.

Note that $W_{i,l}$ appears on both sides of Equation 4.12 which means Equation 4.12 is recursive and a fixed-point computation on $W_{i,l}$ can be used to find a solution by initiating $W_{i,l} = \max_{\tau_j \in lp_{i,l}} \{C_j\} + \sum_{\tau_h \in hep_{i,l}} C_h$. The length of the level-i busy window $W_{i,l}$ will then be given by the smallest positive value of $W_{i,l}$ for which Equation 4.12 converges.

Having bounded the value of $W_{i,l}$, we can compute the maximum number of jobs of task τ_i that can execute on core π_l during any time interval of length $W_{i,l}$ using $K_i = \eta_i^+(W_{i,l})$.

Now we can compute the latest *finishing time* of the k^{th} job of τ_i on core π_l, i.e., $\tau_{i,k}$, using the following lemma.

Lemma 4.9. The latest finish time of $\tau_{i,k}$ is denoted by $f_{i,k}$, where $f_{i,k}$ is given by the first positive solution to the fixed-point iteration on the following equation:

$$f_{i,k} = k \times C_i + \sum_{\tau_h \in hep_{i,l} \setminus \tau_i} \eta_h^+(f_{i,k} - C_i) \times C_h + \alpha_{i,l}^{max}(f_{i,k}) \tag{4.13}$$

Proof. To compute the latest finish time of $\tau_{i,k}$, we need to consider the sum of the WCET of $k-1$ jobs of τ_i plus the WCET of $\tau_{i,k}$ itself since all jobs of τ_i that execute until $\tau_{i,k}$ can impact the finish time of $\tau_{i,k}$. This term is represented by $k \times C_i$.

Due to the fixed-priority non-preemptive scheduling, all jobs released by a higher priority task τ_h during $f_{i,k}$ can delay the execution of $\tau_{i,k}$ until the start of $\tau_{i,k}$ (remember $\tau_{i,k}$ cannot be

[2]Having accounted for the total bus contention that can be suffered by all tasks of the local core during $W_{i,l}$, we can use the value C_h to account for the time required to execute task $\tau_h \in hep_{i,l}$.

preempted after starting its execution). This implies that the maximum interference that can be generated by a higher priority task τ_h until the start of $\tau_{i,k}$ is upper bounded by $\eta_h^+(f_{i,k}-C_i)\times C_h$. Extending this to all to tasks in $hep_{i,l}$ (excluding τ_i), the maximum interference that can be suffered by $\tau_{i,k}$ until its finish time is upper bounded by $\sum_{\tau_h\in hep_{i,l}\setminus\tau_i}\eta_h^+(f_{i,k}-C_i)\times C_h$.

Due to the non-preemptive execution, one job of a task in $lp_{i,l}$ taskset can delay the execution of $\tau_{i,k}$. Furthermore, each job that executes on the local core π_l until the completion of $\tau_{i,k}$ can suffer bus contention. The term $\alpha_{i,l}^{max}(f_{i,k})$ captures the maximum delay that can be suffered by core π_l during any time interval of length $f_{i,k}$ due to the total bus contention, i.e., $Bus_{i,l}^{max}(f_{i,k})$, and the blocking caused by a task in $lp_{i,l}$ taskset[3]. The Lemma follows. □

Note that $f_{i,k}$ appears on both sides of Equation 4.13 which means Equation 4.13 is recursive and a fixed-point computation on $f_{i,k}$ can be used to find a solution by initiating $f_{i,k}=Ci+C_{lp_{i,l}}^{max}+\sum_{\tau_h\in hep_{i,l}\setminus\tau_i}C_h$. The latest finish time $f_{i,k}$ will then be given by the smallest positive value of $f_{i,k}$ for which Equation 4.13 converges.

Once Equation 4.13 converges, we can now compute the *response time* of $\tau_{i,k}$ as follows.

The response time of $\tau_{i,k}$ is denoted by $R_{i,k}$ and can be computed by subtracting the minimum inter-arrival time of previously executed jobs of task τ_i from the latest finish time $f_{i,k}$. Hence,

$$R_{i,k}=f_{i,k}-(k-1)\times T_i \tag{4.14}$$

Finally, the WCRT of task τ_i is then given by the largest response time of any job of τ_i that executes during the level-i busy window, i.e.,

$$R_i^{max}=\max_{k\in[1,K_i]}\{R_{i,k}\} \tag{4.15}$$

A task τ_i is deemed schedulable if its WCRT R_i^{max} is less than or equal to its relative deadline D_i. Similarly, a task set Γ is deemed schedulable if all tasks $\tau_i\in\Gamma$ are schedulable. Also, note that for a task set to be schedulable, the total core utilization of each individual core should not be greater than 1 and the total bus utilization of the taskset Γ should be less than or equal to 1, i.e., $\sum_{\tau_i\in\Gamma}\frac{C_i^A+C_i^R}{T_i}\leq 1$.

4.6 Experimental Evaluation

In this section, we evaluate the performance of our proposed bus contention analysis for RR bus arbitration policy and compare it with FCFS-based DMAM and FMAM analysis presented in Chapter 3. To compare our proposed RR-based analysis against the DMAM/FMAM analyses, we performed several experiments using synthetic task sets under different settings.

As a default configuration, we assume a multicore architecture with 4 cores. The total number of tasks per task set are 32, where each core is assigned 8 tasks at design time. Tasks utilizations

[3]The value of $\alpha_{i,l}^{max}(f_{i,k})$ can be computed using Algorithm 2 by simply replacing $W_{i,l}$ with $f_{i,k}$ in each line of Algorithm 2.

are generated using Uunifast-discard [Emberson et al., 2010]. Tasks' periods are generated using log-uniform distribution in the range of [1000,10000]. We assume that the slot size SS is the same for each core and its value is set such that $SS = 2 \times t^{mem}$. The WCET C_i of each task τ_i is set using its utilization and period, i.e., $C_i = U_i \times T_i$. The memory access demand MD_i of task τ_i is chosen randomly in the range $[10\%, 40\%] \times C_i$, i.e., $MD_i = rand(10\%, 40\%) \times C_i$. The length of the A-phase (resp. R-phase) of task τ_i was then assigned by $MD_i^A \times t^{mem} = MD_i/2$ (resp. $MD_i^R \times t^{mem} = MD_i/2$). Similarly, the WCET of the E-phase of task τ_i was assigned by $C_i - MD_i$. Task deadlines are implicit, i.e., $D_i = T_i$, with priorities assigned using Rate-Monotonic [Liu and Layland, 1973].

We compared the performance of our proposed RR bus policy-based analysis against the FCFS bus policy-based analyses presented in Chapter 3 by varying the core utilization, memory access demand, number of cores, tasks' periods, and slot size SS. We use task set schedulability, i.e., the number of task sets deemed schedulable as the performance metric, and evaluate 1000 randomly generated task sets per point for all the analyzed approaches.

1. Core Utilization: In this experiment, we vary the core utilization of each core from 0.025 to 1.0 in steps of 0.025 and evaluate its impact on the taskset schedulability. The percentage of task sets that were deemed schedulable using all the approaches are shown in Figure 4.5. The label marked as "RR" in Figure 4.5 represents our proposed bus contention analysis for RR bus policy-based bus arbitration policy. The FCFS bus policy-based analyses presented in Chapter 3 are marked as "FMAM" and "DMAM".

In Figure 4.5, the x-axis represents the core utilization and the y-axis represents the percentage of schedulable tasksets for all the analyzed approaches. We can see in Figure 4.5 that for all the approaches the percentage of schedulable tasksets decreases with an increase in the core utilization. This is intuitive as an increase in core utilization can increase the utilization of tasks that may result in an increase in the WCET as $C_i = U_i \times T_i$. Higher WCET of tasks eventually results in increasing the interference/blocking that tasks can suffer from the same core as well as bus contention from remote cores. However, we can also see in Figure 4.5 that the proposed RR-based bus contention analysis outperforms the FCFS-based analyses Chapter 3. In fact, the proposed RR-based analysis can schedule up to 50% more tasksets as compared to DMAM and up to 43%

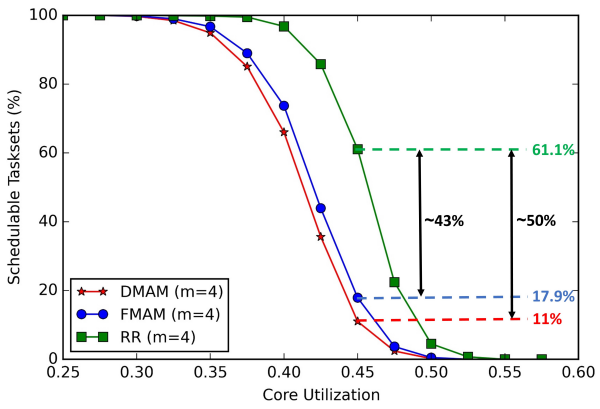
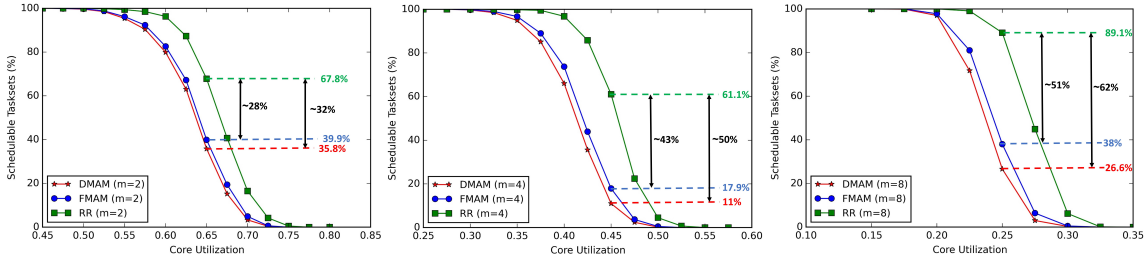


Figure 4.5: Varying core utilization



(a) Number of cores (m) = 2 (b) Number of cores (m) = 4 (c) Number of cores (m) = 8

Figure 4.6: Varying the number of cores and core utilization

more tasksets as compared to FMAM at the core utilization value of 0.45. This improvement in performance in comparison to the existing analysis can be explained as follows.

The bus contention analyses presented in Chapter 3 only focus on the number of memory phases released on the local/remote cores when bounding the bus contention and do not account for the number of memory requests issued during the memory phase or the size of bus slots assigned to each core as shown in Figure 4.1. Hence, the FCFS-based analysis can overestimate the bus contention that can be caused by the remote cores.

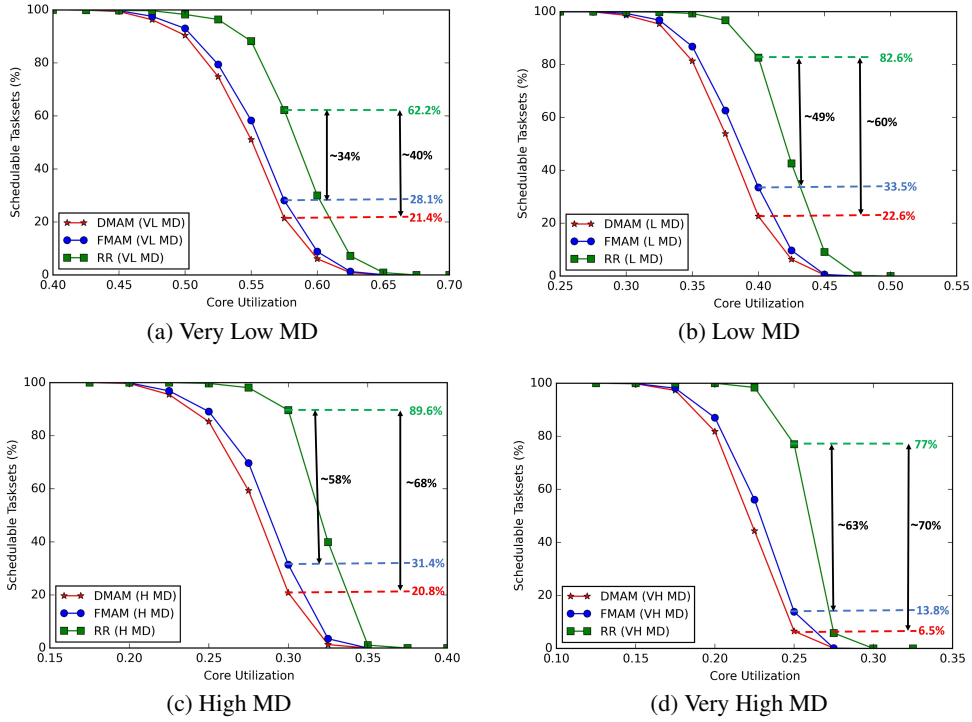
In contrast to the FCFS-based analyses, the proposed RR-based efficiently regulates the bus utilization among cores which leads to a significant reduction in the bus contention that can be suffered by tasks. Note that FMAM analysis also fairly allocates bus among all the cores but only at the memory phase level whereas the proposed RR-based analysis fairly allocates bus among all cores at the bus slot level which can be on the single memory request level.

2. Number of Cores: To evaluate the impact of the number of cores (and the number of tasks) on the performance of all analyzed approaches, we re-do experiment 1 by varying the value of m (i.e., number of cores) between 2 and 8 along with core utilizations. We observe in Figure 4.6 that by increasing the number of cores the task set schedulability for all approaches decreases. This is mainly because, by increasing the number of cores, the total number of tasks executing on the remote cores also increases. This increase in the number of tasks leads to a higher contention at the bus. Hence, we see in Figure 4.6c, a very low percentage of task sets are schedulable by all the approaches for m=8, i.e., a total of 64 tasks in the system. However, we can also note that our proposed RR-based analysis always dominates FCFS-based analyses for all the considered values of m. For example, Figure 4.6c shows that the proposed RR-based analysis improves task set schedulability by up to 62 percentage points in comparison to DMAM analysis and up to 51 percentage points in comparison to FMAM analysis.

3. Varying Memory Access Demand: In this experiment, we vary the size of memory phases of each task τ_i by varying the memory access demand MD_i. Specifically, we consider four different configurations of memory access demand that determine the number of memory requests per memory phase. These configurations are as follows.

(a) Very Low (VL) MD, i.e., $MD_i = rand(5\%, 20\%) \times C_i$;

(b) Low (L) MD, i.e., $MD_i = rand(20\%, 40\%) \times C_i$;



(a) Very Low MD (b) Low MD

(c) High MD (d) Very High MD

Figure 4.7: Varying core utilization for different MD configurations

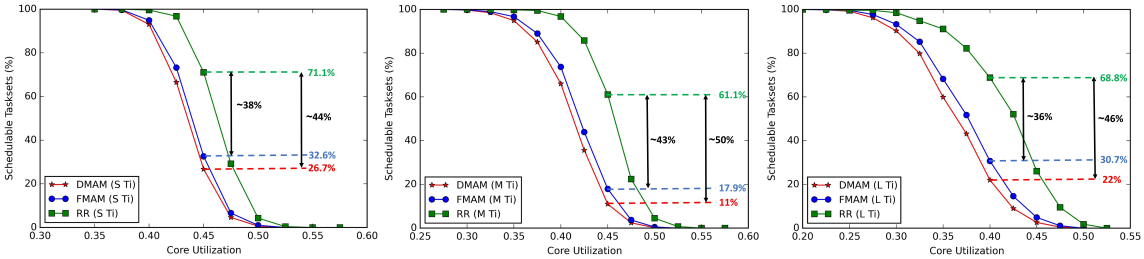
(c) High (H) MD, i.e., $MD_i = rand(40\%, 60\%) \times C_i$; and

(d) Very High (VH) MD, i.e., $MD_i = rand(60\%, 80\%) \times C_i$.

The results are shown in Figure 4.7 where the x-axis represents core utilization of tasks and the y-axis represents the percentage of schedulable tasksets for the given MD configuration.

We can see in Figure 4.7 that by increasing the memory access demand of tasks the schedulability of all the approaches decreases. This is intuitive since higher values of MD also increase the length of memory phases, which in turn increases the number of memory requests that can be generated by tasks. Consequently, tasks will suffer more bus contention for higher values of MD, i.e., all the approaches perform the best under the VL MD configuration and the worst under the VH MD configuration. Looking at the results, one can observe that the proposed RR-based analysis still outperforms the FCFS-based analyses for all values of MD. The gain of the proposed RR-based analysis over FCFS-based analyses increases with the increase in MD values. In fact, the proposed RR-based analysis can schedule up to 70% more tasksets as compared to DMAM and up to 63% more tasksets as compared to FMAM at the core utilization value of 0.275 for VH MD as shown in Figure 4.7d. This is mainly because tasks can suffer a large amount of bus contention because of the higher number of memory requests per memory phase due to the higher value of MD. Since the proposed RR-based analysis improves the bound on the bus contention, the difference between the proposed RR-based analysis and FCFS-based analyses increases significantly with the increase in the MD value.

4. Varying Task Period Range: In this experiment, we varied tasks' period ranges and analyzed the impact of tasks' period ranges on the taskset schedulability. For this, we considered



(a) Short (S) Tasks' Period Range (b) Medium (M) Tasks' Period Range (c) Long (L) Tasks' Period Range

Figure 4.8: Varying core utilization for different tasks' period range

three tasks' period ranges that are: 1) Short (S) range, i.e., task periods in the range of [1000,5000]; 2) Medium (M) range, i.e., task periods in the range of [1000,10000]; and 3) Long (L) range, i.e., task periods in the range of [1000,20000]. Figure 4.8 shows the percentage of schedulable tasksets using all the approaches by varying the core utilization for different tasks' period ranges.

We observe that taskset schedulability for all the approaches increases by decreasing the tasks' period range, i.e., see Figure 4.8a, and decreases by increasing the tasks' period range, i.e., see Figure 4.8c. This is mainly because the larger task period translates into a larger WCET of tasks since the WCET also depends on task periods due to the relation $C_i = U_i \times T_i$. The larger WCET of tasks results in larger blocking from a lower priority task. This implies that increasing the task period range increases the blocking caused by a lower priority task. This increase in lower priority blocking also increases the length of the level-i busy window which may result in converging the level-i busy window at a later stage due to additional jobs released by higher priority tasks. Moreover, the bus contention can also be suffered by those additional jobs released by higher priority tasks and further impacting the length of the level-i busy window. This causes a degradation in task set schedulability when the period ranges are increased. However, we can still see in Figure 4.8 that for all the tasks' period ranges, the proposed RR-based analysis outperforms the FCFS-based analyses, i.e., DMAM and FMAM.

5. Varying Slot Size: In this experiment, we vary the value of *SS* (i.e., slot size) along with core utilization and evaluate their impact on the taskset schedulability. Specifically, we considered

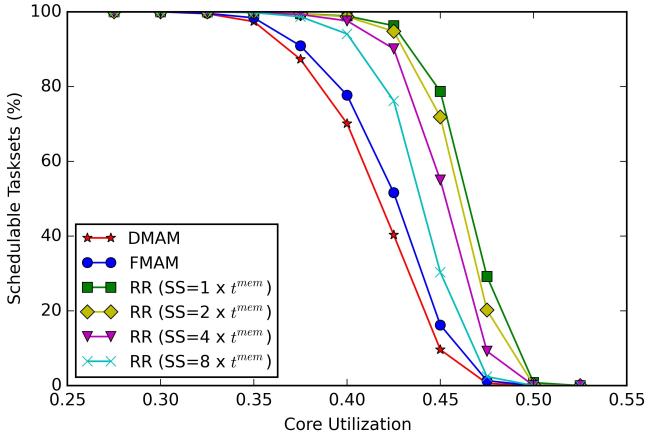


Figure 4.9: Varying slot size *SS*

different slots of length $SS = 1 \times t^{mem}, 2 \times t^{mem}, 4 \times t^{mem}, 8 \times t^{mem}$. The percentage of schedulable tasksets for different values of SS is shown in Figure 4.9. For the sake of comparison, Figure 4.9 also shows the percentage of schedulable tasksets using the FMAM and DMAM analysis. We observe that for a smaller value of slot size SS, the task set schedulability for the RR-based is higher. This is mainly because for lower values of SS, the bus contention that can be generated by remote cores is also lower and the slots can be better utilized. For higher values of SS, the task set schedulability decreases due to an increase in bus contention that can be generated due to remote cores. For example, if the $SS = 8 \times t^{mem}$ and the local core has to execute only two memory requests, then, in the worst case, the local core may have to wait for $(8 \times t^{mem}) \times m - 1$ time units to access the bus. However, as shown in Figure 4.9, the proposed RR-based analysis was able to schedule more percentage of tasksets in comparison to FCFS-based DMAM and FMAM analysis for all the considered values of SS.

4.7 Chapter Summary

In this chapter, we evaluate the impact of bus arbitration policy on the bus contention that can be suffered by 3-phase tasks and show how the bound can be improved considering a fairer bus arbitration policy such as the RR. Specifically, we propose a fine-grained bus contention analysis considering the RR bus arbitration policy by taking into account the number of bus slots required by the memory phases of 3-phase tasks. We also show if the blocking caused by lower priority tasks is computed similarly to that of uniprocessor scheduling, it can yield unsafe bounds. We then propose an algorithm that iterates over all the lower priority tasks to maximize the overall delay that can be experienced during the response time of the task under analysis. The experimental results reveal that the proposed RR-based analysis can improve the taskset schedulability by up to 70% and 63% in comparison to the FCFS bus policy-based DMAM and FMAM analysis, respectively.

Chapter 5

Cache-aware Bus Contention Analysis

Even though, Chapters 3 and 4 provide fine-grained bus contention analysis for the 3-phase task model, the analysis ignores the *interdependence* of the memory bus on cache memories. For example, the bus contention analysis discussed in Chapters 3 and 4 assume the memory phases of each job of each task that executes during a given time interval will always have the worst-case memory access demand, i.e., the maximum number of main memory accesses issued during the memory phases of one job of a task in isolation. This can be pessimistic in some scenarios because in multicore platforms when a task executing on a core needs code/data, it first checks the local cache or Last Level Cache (LLC). If the requested code/data is not available in the cache, i.e., referred to as cache miss, a memory bus request is initiated. Once the data/instructions are loaded in the local cache/LLC, it should allow for data/instructions re-use and thus reduce the overall number of accesses to main memory. Consequently, the bus contention suffered by tasks is interdependent on the cache behavior, i.e., the bus contention depends on the number of memory requests issued by tasks, which in turn depends on the number of cache misses. This highlights the importance of the *cache-aware bus contention analysis* that considers the interdependence of the memory bus on the caches.

This chapter presents the cache-aware bus contention analysis that considers the number of cache misses. Specifically, we use the notion of *cache persistence* [Rashid et al., 2016], i.e., memory blocks that once loaded into the cache by the task can be reused by its subsequent jobs without the need to access the main memory. This can tightly upper bound the maximum number of cache misses which further improves the bound on the number of bus/memory requests of the memory phases of 3-phase tasks. This tighter bound on the number of bus/memory requests is then integrated into the bus contention analysis presented in Chapters 3 and 4. A tighter bound on the maximum interference from higher priority tasks is also derived by considering the number of LLC misses. Finally, the WCRT-based schedulability analysis is derived by integrating the improved bounds on the maximum bus contention and maximum interference suffered by tasks.

The main **contributions** of this chapter are as follows.

1. Upper bounding LLC misses of 3-phase tasks that lead to bus/memory requests generated during the memory phases of tasks. Integrating the bounds on cache misses into the bus contention analysis of Chapters 3 and 4.

2. Improved schedulability analysis by integrating the maximum bus contention into the WCRT formulation.

3. Extensive empirical evaluation under different settings to compare the proposed cache-aware bus contention analysis to the cache-oblivious bus contention analysis presented in Chapters 3 and 4. Experimental results show that the proposed cache-aware bus contention analysis outperforms the cache-oblivious bus contention analysis resulting in improving the schedulability success ratio by up to 55% percentage points for the RR policy and up to 18% percentage points for the FCFS policy.

Chapter Organization: The rest of the chapter is organized as follows: Section 5.1 describes the system model and task model. The background concepts related to cache persistence are presented in Section 5.2. The proposed persistence aware cache analysis for the 3-phase task model is presented in Section 5.3. The cache-aware schedulability analysis is presented in Section 5.5 and the experimental results are presented in Section 5.6. Finally, the chapter summary is presented in Section 5.7.

5.1 System Model

The system model and task model are identical to that of Section 3.1 but we define additional assumptions as follows.

We assume a multicore system comprising m identical cores $(\pi_1, \pi_2, \ldots, \pi_m)$ that share the Last-Level Cache (LLC). The LLC is assumed to be partitioned among all the cores such that each core has an individual non-overlapping partition. Each cache partition assigned to the cores is assumed to be large enough to store all the data/instructions required by the task with the largest memory footprint that executes on that core. We assume that cache employs the write-back policy[1], is direct-mapped, and, unified, i.e., it can store data as well as instructions. Furthermore, the write-allocate write-miss policy is assumed in the case of write-miss, which means that the memory block being written to is first loaded in the cache before performing the write operation. Note that the cache analysis presented in Section 5.3 assumes a direct-mapped cache, however, it can be easily extended to set-associative caches by building on the analysis presented in [Rashid et al., 2020]. We assume that in cases where data sharing is required between tasks of different cores, the notion of cache persistence is applied only to instructions, or the specific memory requests used for inter-core data sharing can be modeled as non-persistent cache blocks.[2]

[1] Several COTS multicore platforms support a write-back cache policy, e.g., Renesas SH7750, NEC V44181, Freescale MPC740, etc.

[2] We explain the notion of cache persistence, persistent blocks, etc. in Section 5.2.

Although the proposed cache-aware analysis is applicable to all the bus arbitration policies, in this work, we focus on Round Robin (RR) and First-Come-First-Serve (FCFS) based bus arbitration schemes studied in Chapters 4 and 3, respectively.

5.2 Background

This section presents the essential background on cache-related concepts that we later use to build our analysis in Section 5.3.

When analyzing the worst-case memory access demand of tasks, the analysis presented in Chapters 3 and 4 assume that each job of a task τ_i that execute during a time window of length Δ will always issue MD_i main memory accesses, i.e., the worst-case memory access demand of τ_i in isolation. This, in other words, means that each job of task τ_i that executes during Δ will always load all its ***Evicting Cache Blocks*** (ECBs) from the main memory to the cache.

Definition 5.1. [Evicting Cache Blocks (ECBs) (from [Tomiyama and Dutt, 2000])] The set of all memory blocks that may be used by a task τ_i during its execution.

Clearly, this assumption is pessimistic, as subsequent jobs of task τ_i that execute during Δ can *re-use* some ECBs already available in the cache due to a previous job of τ_i. These re-usable ECBs are called ***Persistent Cache Blocks*** (PCBs) [Rashid et al., 2016].

Definition 5.2. [Persistent Cache Blocks (PCBs) (from [Rashid et al., 2016])] All memory blocks used by a task, that once loaded in the cache, will never be evicted or invalidated by the task itself.

For a task τ_i executing in isolation, if all its PCBs are already loaded in the cache, e.g., by a previous job of τ_i, the memory access demand for subsequent jobs of τ_i can be much lower than the worst-case memory access demand of τ_i in isolation. This memory access demand of the task τ_i is called ***residual memory access demand***.

Definition 5.3. [Residual Memory Access Demand (from [Rashid et al., 2016])] The worst-case memory access demand of any job of task τ_i considering that all its PCBs are already loaded in the cache.

Considering the PCBs and residual memory access demand of task τ_i, the total number of main memory accesses made by all the jobs of task τ_i when it executes *in isolation* during any time window of length Δ is given by (see Lemma 1 of [Rashid et al., 2016]):

$$MD_i^{tot}(\Delta) = \min\left(\left\lceil \frac{\Delta}{T_i} \right\rceil \times MD_i, \left\lceil \frac{\Delta}{T_i} \right\rceil \times \bar{MD}_i + |PCB_i|\right) \tag{5.1}$$

where $\left\lceil \frac{\Delta}{T_i} \right\rceil$ bounds the maximum number of jobs released by task τ_i during any time window of length Δ; MD_i is the worst-case memory access demand of one job of τ_i measured in isolation; $\bar{MD}_i$ is the worst-case residual memory access demand of one job of task τ_i; and $|PCB_i|$ represents the cardinality of the set of PCBs of task τ_i, i.e., the total number of PCBs of task τ_i.

Equation 5.1 upper bounds the total memory access demand of a task in isolation. However, a task τ_i will likely have to share the core on which it executes with other tasks. So, the PCBs that were loaded by one job of τ_i can be evicted by other tasks executing on the same core. This results in generating additional main memory overhead called ***Cache Persistence Reload Overhead*** (CPRO) [Rashid et al., 2016].

Definition 5.4. [Cache Persistence Reload Overhead (CPRO) (from [Rashid et al., 2016])] The number of main memory accesses that τ_i must make due to the evictions of its PCBs caused by the execution of tasks in $hep_{i,l} \setminus \tau_i$.

The maximum CPRO that can be suffered by one job of task τ_i is denoted by ρ_i, and is given by (see Theorem 1 of [Rashid et al., 2016] for proof)

$$\rho_i = PCB_i \cap \left(\bigcup_{\forall \tau_k \in hep_{i,l} \setminus \tau_i} ECB_k \right) \tag{5.2}$$

where PCB_i is the set of PCBs of task τ_i; and $\bigcup_{\forall \tau_k \in hep_{i,l} \setminus \tau_i} ECB_k$ is the set union of the ECBs of all the tasks in $hep_{i,l} \setminus \tau_i$ that can potentially evict PCBs of τ_i.

5.3 Persistence-aware Cache Analysis for 3-Phase Tasks

In this section, we present the cache analysis to bound the maximum number of main memory accesses (i.e., LLC misses) that can be generated during the A-phases and R-phases of tasks executed in a level-i busy window (remember by the definition of the 3-phase task model there are no main memory accesses during the E-phases).

The existing works [Maia et al., 2017, Thilakasiri and Becker, 2023a] (including our works proposed in Chapters 3 and 4) that analyze the bus contention for the 3-phase task model assume that the main memory accesses generated during the A-phases (i.e., memory prefetches) and the R-phase (i.e., memory write-backs) of every job of each task τ_i are equal to its worst-case memory access demand measured in isolation. This implies that the A-phase of each job of every task must load all its ECBs, given by MD_i^A, and the R-phase of each job of every task must write back all the cache lines that are dirty at the end of the E-phase, given by MD_i^R. This assumption is pessimistic, for instance, there can be some ECBs that, once loaded during the A-phase of one job of a task, will not be evicted from the cache by the task itself. Consequently, the number of ECBs to be loaded from the main memory by the A-phase of the subsequent jobs of the same task can be less than its worst-case memory access demand. In the subsections below, we will explain in detail how the notion of PCBs, residual memory access demand, and CPRO (see Section 5.2 for details) can be used to bound the memory access demand of A- and R-phases of tasks.

5.3.1 Upper Bounding Memory Access Requests by the Local Core

In this section, we will upper bound the maximum number of main memory accesses generated by all tasks that execute on the *local core* π_l during the level-i busy window $W_{i,l}$. We start by bounding the maximum number of main memory accesses during the A-phases of those tasks.

Applying the notion of PCBs and residual memory access demand to the 3-phase task model, the total number of main memory accesses that can be generated during the *A-phases* of all the jobs of task τ_i when it executes *in isolation* during the level-i busy window $W_{i,l}$ is given by the following lemma.

Lemma 5.1. The total number of main memory accesses that can be generated during the *A-phases* of all jobs of task τ_i when they execute *in isolation* within any time window of length $W_{i,l}$ is given by $MD_i^{A,tot}$, where

$$MD_i^{A,tot}(W_{i,l}) = |PCB_i| + \bar{MD}_i^A + \left(\left\lceil \frac{W_{i,l}}{T_i} \right\rceil - 1\right) \times \bar{MD}_i^A \tag{5.3}$$

where PCB_i is the set of PCBs of task τ_i and $\bar{MD}_i^A$ is the residual memory access demand of the A-phase of one job of task τ_i.

Proof. τ_i releases at most $\left\lceil \frac{W_{i,l}}{T_i} \right\rceil$ jobs in the level-i busy window of length $W_{i,l}$. We know that the A-phase of the first job of τ_i must load all its ECBs, i.e., $|PCB_i| + \bar{MD}_i^A$. Furthermore, by definition of the residual memory access demand $\bar{MD}_i^A$, the A-phases of subsequent jobs of τ_i can make at most $\left(\left\lceil \frac{W_{i,l}}{T_i} \right\rceil - 1\right) \times \bar{MD}_i^A$ memory accesses. Thus, Equation 5.3 bounds the maximum number of main memory accesses that can be generated during all the A-phases of task τ_i when it executes in isolation during $W_{i,l}$. □

As discussed earlier, other tasks that can execute on the same core as τ_i can evict the PCBs of τ_i. Thus, we also need to account for the maximum CPRO that can be suffered by task τ_i, when computing the memory accesses generated during its A-phases.

The maximum CPRO that can be suffered by an A-phase of *one job* of task τ_i is upper bounded by the following equation (from [Rashid et al., 2016])

$$\rho_i = PCB_i \cap \left(\bigcup_{\forall \tau_h \in hep_{i,l} \setminus \tau_i} ECB_h \right) \tag{5.4}$$

where PCB_i is the set of PCBs of task τ_i, and $\bigcup_{\forall \tau_h \in hep_{i,l} \setminus \tau_i} ECB_h$ is the set union of the ECBs of all tasks in $hep_{i,l} \setminus \tau_i$ that can potentially evict the PCBs of τ_i.

The key insight for Equation 5.4 is that a task $\tau_h \in hep_{i,l}$ can only evict the PCBs of task τ_i if the ECBs of τ_h shares the same cache lines as the PCBs of τ_i. Note that a lower priority task cannot evict the PCBs of τ_i within the level-i busy window as it can only execute at the start of the level-i busy window.

Using Equations 5.3 and 5.4, the maximum number of main memory accesses that can be generated during all the A-phases of τ_i within a level-i busy window is given by the following lemma.

Lemma 5.2. The maximum number of main memory accesses that can be generated during the A-phases of all the jobs of task τ_i during any time window of length $W_{i,l}$ is denoted by $\hat{MD}_i^A(W_{i,l})$, where

$$\hat{MD}_i^A(W_{i,l}) = \min\left(\left\lceil \frac{W_{i,l}}{T_i} \right\rceil \times MD_i^A, |PCB_i| + \bar{MD}_i^A + \left(\left\lceil \frac{W_{i,l}}{T_i} \right\rceil - 1\right) \times (\bar{MD}_i^A + |\rho_i|)\right) \tag{5.5}$$

Proof. By the definition of MD_i^A, the maximum number of main memory accesses that can be generated during the A-phases of $\left\lceil \frac{W_{i,l}}{T_i} \right\rceil$ jobs of τ_i cannot be greater than $\left\lceil \frac{W_{i,l}}{T_i} \right\rceil \times MD_i^A$.

From Equation 5.3, we know that $|PCB_i| + \bar{MD}_i^A + \left(\left\lceil \frac{W_{i,l}}{T_i} \right\rceil - 1\right) \times \bar{MD}_i^A$ upper bounds the maximum number of main memory accesses generated during the A-phases of τ_i during $W_{i,l}$ when it executes in isolation. Furthermore, from Equation 5.4, we know that ρ_i bounds the maximum CPRO that can be suffered by the A-phase of one job of τ_i. In the worst case, the CPRO can be suffered by the A-phases of all the jobs except the A-phase of the first job of τ_i (as it loads all its ECBs) that execute during $W_{i,l}$. Consequently, $\left(\left\lceil \frac{W_{i,l}}{T_i} \right\rceil - 1\right) \times |\rho_i|$ bounds the maximum CPRO that can be suffered by task τ_i during $W_{i,l}$. Therefore, Equation 5.5 upper bounds the maximum number of main memory accesses that can be generated during all the A-phases of all jobs of task τ_i during $W_{i,l}$. The Lemma follows. □

Lemma 5.2 can be applied to each task in $hep_{i,l}$ as a task from the set $hep_{i,l}$ can potentially execute multiple jobs during the level-i busy window. Thus, we can tightly bound its main memory accesses using cache persistence. Building upon this, the maximum number of main memory accesses that can be generated during the A-phases of all tasks in $hep_{i,l}$ that can execute on the local core π_l during any time window of length $W_{i,l}$ is bounded by $\Psi_{i,l}^A(W_{i,l})$, where

$$\Psi_{i,l}^A(W_{i,l}) = \sum_{\forall \tau_h \in hep_{i,l}} \hat{MD}_h^A(W_{i,l}) \tag{5.6}$$

Having bounded the number of main memory accesses that can be generated during the A-phases, we can now bound the maximum number of main memory accesses generated during the *R-phases* of tasks.

The R-phase is mainly responsible for writing back all the dirty cache lines (after the execution of the E-phase) to the main memory. In order to tightly bound the number of main memory accesses generated during the *R-phase*, a task should only write back a subset of dirty cache blocks that can potentially be used by other tasks to load their ECBs. However, achieving this from the implementation perspective can be extremely complex because: 1) determining the address of specific cache lines that will be dirty at the end of the E-phase of the task is complex, as it depends on the run-time state; 2) enforcing a task to write-back a subset of dirty cache blocks (based on the set of ECBs of other tasks) during the R-phase can be extremely challenging, as it may require

additional run-time monitoring/control mechanisms. Therefore, we assume that all cache lines that are dirty at the end of the E-phase of a task will be written back (and invalidated) during the R-phase. By definition, all such cache lines cannot hold PCBs. Consequently, the memory access demand of an R-phase of a task will account for write-backs due to non-persistent memory blocks and is given by MD_i^R. Building on this, the maximum number of main memory accesses that can be generated during the *R-phases* of all tasks in $hep_{i,l}$ executing on the local core during $W_{i,l}$ is upper bounded by $\Psi_{i,l}^R(W_{i,l})$, where

$$\Psi_{i,l}^R(W_{i,l}) = \sum_{\forall \tau_h \in hep_{i,l}} \left\lceil \frac{W_{i,l}}{T_h} \right\rceil \times MD_h^R \tag{5.7}$$

For the R-phases, we assume that each job of every task has to write back all its non-persistent memory blocks to the main memory. MD_h^R bounds the maximum number of main memory requests issued by the R-phase of one job of a task τ_h, thus, $\sum_{\forall \tau_h \in hep_{i,l}} \left\lceil \frac{W_{i,l}}{T_h} \right\rceil \times MD_h^R$ bounds the maximum number of main memory requests issued by the R-phases of all the tasks in $hep_{i,l}$ during $W_{i,l}$.

5.3.2 Upper Bounding Memory Access Requests by the Remote Core

As discussed in Section 5.3.1, the maximum number of main memory accesses of tasks depends on the PCBs, residual memory access demand, and CPRO. The notion of PCBs and residual memory access demand can be applied to the A-phases of tasks running on the remote core identically to tasks of the local core (using Equation 5.3). However, we cannot use Equation 5.4 to compute CPRO because the interfering task τ_u executing on π_r may not be executing within an uninterrupted level-u busy window during the whole interval $W_{i,l}$. Therefore, we cannot assume that only tasks of higher or equal priority execute between two jobs of τ_u. We must therefore consider that any task executing on core π_r may interfere with the PCBs of τ_u during $W_{i,l}$. Thus, the maximum CPRO that can be suffered by the A-phase of *one job* of task τ_u that executes on a remote core π_r is given by $\bar{\rho}_u$, where

$$\bar{\rho}_u = PCB_u \cap \left(\bigcup_{\forall \tau_k \in \Gamma^r \setminus \tau_u} ECB_k \right) \tag{5.8}$$

where Γ^r is the set of all tasks running on a remote core π_r, PCB_u is the set of PCBs of task τ_u, and $\bigcup_{\forall \tau_k \in \Gamma^r \setminus \tau_u} ECB_k$ is the set union of all ECBs of all tasks in Γ^r except τ_u.

The maximum number of main memory accesses that can be generated during all the *A-phases* of a task τ_u running on a remote core π_r during $W_{i,l}$ is then given by the following lemma.

Lemma 5.3. The maximum number of main memory accesses that can be generated during the A-phases of all the jobs of a task τ_u running on the remote core π_r during any time window of length $W_{i,l}$ is given by $\hat{MD}_u^A(W_{i,l})$, where

$$\hat{MD}_u^A(W_{i,l}) = \min\left(\left\lceil \frac{W_{i,l}}{T_u} \right\rceil \times MD_u^A, |PCB_u| + \bar{MD}_u^A + \left(\left\lceil \frac{W_{i,l}}{T_u} \right\rceil - 1 \right) \times (\bar{MD}_u^A + |\bar{\rho}_u|) \right) \tag{5.9}$$

Proof. The proof directly follows from Lemma 5.2 except that the computation of $\bar{\rho}_u$ is given by Equation 5.8. □

To compute the maximum bus contention in some scenarios, e.g., remote core bus slots computation (see Equation 4.2) or sorting out sets per remote core, i.e., M_r^{AH}, M_r^{AL}, to compute the maximum bus contention using case 2 for FCFS bus (see Equation 3.10), it is necessary to determine the maximum number of memory requests that can be generated during *each* A-phase. This can be achieved by further breaking down Equation 5.9 such that the maximum number of memory accesses made by the first A-phase of task τ_u of remote core π_r during $W_{i,l}$ is upper bounded by MD_u^A. Similarly, the maximum number of memory accesses made by each A-phase except the first A-phase of τ_u during $W_{i,l}$ is upper bounded by $\min(MD_u^A, \bar{MD}_u^A + |\bar{\rho}_u|)$.

Applying Lemma 5.3 to all tasks of the remote core, the maximum number of main memory accesses that can be generated during all the *A-phases* of *all tasks* released on the remote core π_r during any time window of length $W_{i,l}$ is upper bounded by $\Omega_{i,r}^A(W_{i,l})$, where

$$\Omega_{i,r}^A(W_{i,l}) = \sum_{\forall \tau_u \in \Gamma^r} \hat{MD}_u^A(W_{i,l}) \tag{5.10}$$

Having bounded the number of memory accesses generated during the A-phase, we can now compute the number of main memory accesses generated during the R-phases. As discussed earlier, we assume that a task can invalidate and write back all its non-persistent cache blocks during the R-phase. Considering this, the maximum number of memory requests that can be generated during the R-phase of a task τ_u is upper bounded by MD_u^R.

The maximum number of main memory accesses that can be generated during the *R-phases* of all the jobs of *all tasks* released on the remote core π_r during any time window of length $W_{i,l}$ is upper bounded by $\Omega_{i,r}^R(W_{i,l})$, where

$$\Omega_{i,r}^R(W_{i,l}) = \sum_{\forall \tau_u \in \Gamma^r} \left\lceil \frac{W_{i,l}}{T_u} \right\rceil \times MD_u^R \tag{5.11}$$

5.4 Cache-aware Bus Contention Analysis

In this section, we will discuss how the persistence aware cache analysis presented in Section 5.3 can be integrated to improve the bound on bus contention. Specifically, we improve the bus contention analysis presented in Chapters 3 and 4 by integrating the number of cache misses while computing the number of bus/memory requests. We start by improving the bus contention analysis presented in Chapter 4 that focuses on the RR bus arbitration policy in the next section.

5.4.1 Cache-aware Bus Contention Analysis for the RR Bus Arbitration Policy

Recall from Chapter 4 that in the RR bus arbitration policy, when multiple cores require access to the bus then each core can only access the bus during its *bus slot*. Considering this, the bus contention that can be suffered by tasks will also depend on the number of bus slots that the local

core as well as the remote core requires during the level-i busy window. Building on this, RR bus policy-based contention analysis presented in Chapter 4 upper bounds the bus contention by first computing the maximum number of bus slots required by the local core as well as the remote cores. The maximum number of *bus slots* required by tasks running on the ***local core*** π_l during the level-i busy window of length $W_{i,l}$ is upper-bounded by $\beta_{\pi_l}(W_{i,l})$, given by the following equation (From Lemma 4.1).

$$\beta_{\pi_l}(W_{i,l}) = \sum_{\tau_h \in hep_{i,l}} \left\lceil \frac{W_{i,l}}{T_h} \right\rceil \times \left(\left\lceil \frac{MD_h^A \times t^{mem}}{SS} \right\rceil + \left\lceil \frac{MD_h^R \times t^{mem}}{SS} \right\rceil \right) + \max_{\forall \tau_j \in lp_{i,l}} \left\{ \left\lceil \frac{MD_j^A \times t^{mem}}{SS} \right\rceil + \left\lceil \frac{MD_j^R \times t^{mem}}{SS} \right\rceil \right\} \tag{5.12}$$

In Equation 5.12, the term $\left\lceil \frac{W_{i,l}}{T_h} \right\rceil$ upper bounds the maximum number of jobs that task τ_h can release during any time window of length $W_{i,l}$. MD_h^A (resp. MD_h^R) is the maximum number of main memory accesses that can be generated by the A-phase (resp. R-phase) of one job of task τ_h; t^{mem} is the maximum time required to serve one memory request; SS is the length of the bus slot which is always greater than or equal to t^{mem}. The term $\left\lceil \frac{MD_h^A \times t^{mem}}{SS} \right\rceil$ (resp. $\left\lceil \frac{MD_h^R \times t^{mem}}{SS} \right\rceil$) represents the maximum number of bus slots required by the A-phase (resp. R-phase) of one job of task τ_h. Finally, due to fixed-priority non-preemptive scheduling, the term $\max_{\forall \tau_j \in lp_{i,l}} \left\{ \left\lceil \frac{MD_j^A \times t^{mem}}{SS} \right\rceil + \left\lceil \frac{MD_j^R \times t^{mem}}{SS} \right\rceil \right\}$ integrates the maximum number of bus slots required by lower priority tasks.[3]

Similarly, the maximum number of *bus slots* required by tasks running on a ***remote core*** π_r during the level-i busy window of length $W_{i,l}$ is upper-bounded by $\beta_{\pi_r}(W_{i,l})$, given by the following equation (From Lemma 4.2).

$$\beta_{\pi_r}(W_{i,l}) = \sum_{\tau_u \in \Gamma'_r} \left\lceil \frac{W_{i,l}}{T_u} \right\rceil \times \left(\left\lceil \frac{MD_u^A \times t^{mem}}{SS} \right\rceil + \left\lceil \frac{MD_u^R \times t^{mem}}{SS} \right\rceil \right) \tag{5.13}$$

We can see in Equations 5.12 and 5.13 that when bounding $\beta_{\pi_l}(W_{i,l})$ and $\beta_{\pi_r}(W_{i,l})$, it is assumed that the number of memory requests issued by each A- and R-phase of every job of each task τ_i is given MD_i^A and MD_i^R, respectively. As discussed earlier, this can yield a pessimistic bound on bus contention as well on the length of the level-i busy window and WCRT. To address this, we will now show how to improve the bus contention by integrating the persistence-aware cache analysis presented in Section 5.3.

We start by computing the number of bus slots for the local core during the level-i busy window using the following lemma.

Lemma 5.4. The maximum number of bus slots required by tasks executing on the local core π_l during any time window of length $W_{i,l}$ is upper-bounded by $\hat{\beta}_{\pi_l}(W_{i,l})$, where

$$\hat{\beta}_{\pi_l}(W_{i,l}) = \sum_{\tau_h \in hep_{i,l}} \min \left(\left\lceil \frac{W_{i,l}}{T_h} \right\rceil \times \left\lceil \frac{MD_h^A \times t^{mem}}{SS} \right\rceil, \left\lceil \frac{MD_h^A \times t^{mem}}{SS} \right\rceil + \left(\left\lceil \frac{W_{i,l}}{T_h} \right\rceil - 1 \right) \times \left\lceil \frac{(\bar{MD}_h^A + |\rho_h|) \times t^{mem}}{SS} \right\rceil \right) + \left\lceil \frac{W_{i,l}}{T_h} \right\rceil \times \left\lceil \frac{MD_h^R \times t^{mem}}{SS} \right\rceil + \max_{\forall \tau_j \in lp_{i,l}} \left\{ \left\lceil \frac{MD_j^A \times t^{mem}}{SS} \right\rceil + \left\lceil \frac{MD_j^R \times t^{mem}}{SS} \right\rceil \right\} \tag{5.14}$$

[3] Note that Equation 5.12 is a simplified representation of Equation 4.1.

Proof. From Lemma 5.2, we know that the $\min\Big(\Big\lceil\frac{W_{i,l}}{T_i}\Big\rceil \times MD_i^A, |PCB_i| + \bar{MD}_i^A + (\Big\lceil\frac{W_{i,l}}{T_i}\Big\rceil - 1) \times (\bar{MD}_i^A + |\rho_i|)\Big)$ upper bounds the number of main memory accesses of the A-phases of all jobs of task τ_h during $W_{i,l}$. Furthermore, from Equation 5.12, we know that it is necessary to determine the maximum number of bus slots required by the memory phases of tasks that execute on the local core π_l during $W_{i,l}$. Consequently, using Lemma 5.2, the maximum number of bus slots require by A-phases of all jobs of a task τ_h that execute on the local core π_l during $W_{i,l}$ is upper bounded by $\min\Big(\Big\lceil\frac{W_{i,l}}{T_h}\Big\rceil \times \Big\lceil\frac{MD_h^A \times t^{mem}}{SS}\Big\rceil, \Big\lceil\frac{MD_h^A \times t^{mem}}{SS}\Big\rceil + (\Big\lceil\frac{W_{i,l}}{T_h}\Big\rceil - 1) \times \Big\lceil\frac{(\bar{MD}_h^A + |\rho_h|) \times t^{mem}}{SS}\Big\rceil\Big)$. This is further extended for all tasks in $hep_{i,l}$ set. Since we do not apply cache persistence to the R-phases, the number of bus slots required by R-phases can be computed identically to that of Equation 5.12, i.e., $\Big\lceil\frac{W_{i,l}}{T_h}\Big\rceil \times \Big\lceil\frac{MD_h^R \times t^{mem}}{SS}\Big\rceil$. Finally, the maximum number of bus slots required by one job of a lower priority task is bounded by $\max_{\forall \tau_j \in lp_{i,l}} \{\Big\lceil\frac{MD_j^A \times t^{mem}}{SS}\Big\rceil + \Big\lceil\frac{MD_j^R \times t^{mem}}{SS}\Big\rceil\}$. □

Similarly, we can tightly bound the number of bus slots required by the remote core using the following lemma.

Lemma 5.5. The maximum number of bus slots required by tasks running on the remote core π_r during any time window of length $W_{i,l}$ is upper-bounded by $\hat{\beta}_{\pi_r}(W_{i,l})$, where

$$\hat{\beta}_{\pi_r}(W_{i,l}) = \sum_{\tau_u \in \Gamma_r} \min\Big(\Big\lceil\frac{W_{i,l}}{T_u}\Big\rceil \times \Big\lceil\frac{MD_u^A \times t^{mem}}{SS}\Big\rceil, \Big\lceil\frac{MD_u^A \times t^{mem}}{SS}\Big\rceil + (\Big\lceil\frac{W_{i,l}}{T_u}\Big\rceil - 1) \times \Big\lceil\frac{(\bar{MD}_u^A + |\bar{\rho}_u|) \times t^{mem}}{SS}\Big\rceil\Big) + \Big\lceil\frac{W_{i,l}}{T_u}\Big\rceil \times \Big\lceil\frac{MD_u^R \times t^{mem}}{SS}\Big\rceil \tag{5.15}$$

Proof. The proof directly follows from Lemma 5.4 except that the computation of $\bar{\rho}_u$ is given by Equation 5.8. □

In a similar manner to that of Equation 5.15, we can improve Equations 4.3 to 4.10 by integrating the number of cache misses to compute the length of each memory phase.

Since a system is likely to have multiple remote cores, we need to bound the maximum number of bus slots for tasks running on each of the remote core $\pi_r \in m$ such that $\pi_r \neq \pi_l$ using Lemma 5.5. Based on the values of $\hat{\beta}_{\pi_l}(W_{i,l})$ and $\hat{\beta}_{\pi_r}(W_{i,l})$, we can compute the maximum bus contention $\hat{Bus}_{i,r}(W_{i,l})$ that the local core π_l can suffer from a remote core π_r during $W_{i,l}$ by improving equations 4.3 to 4.10. Having bounded the maximum bus contention $\hat{Bus}_{i,r}(W_{i,l})$ w.r.t each remote core $\pi_r \in m$ such that $\pi_r \neq \pi_l$, we can compute the *total bus contention* $Bus_{i,l}^{max}(W_{i,l})$ that tasks of local core π_l can suffer due to tasks of *all remote cores* during $W_{i,l}$ using the following Equation.

$$Bus_{i,l}^{max}(W_{i,l}) = \sum_{r=1, r\neq l}^{m} \hat{Bus}_{i,r}(W_{i,l}) \tag{5.16}$$

5.4.2 Cache-aware Bus Contention Analysis for the FCFS Bus Arbitration Policy

Recall from Chapter 3 that FCFS bus policy-based contention analysis bounds the bus contention by computing the $N_{\pi_l}(W_{i,l})$, i.e., maximum number of times bus contention is suffered on the local

core during $W_{i,l}$, and $N_{\pi_r}(W_{i,l})$, i.e., maximum number of times bus contention is caused by a remote core during $W_{i,l}$. The computation of $N_{\pi_l}(W_{i,l})$ and $N_{\pi_r}(W_{i,l})$ is based on the number of jobs/memory phases of tasks. In this case, applying cache persistence may not affect the values of $N_{\pi_l}(W_{i,l})$ and $N_{\pi_r}(W_{i,l})$ because the number of jobs/memory phases of tasks remains the same even if the memory requests of memory phases are reduced. However, when computing the maximum bus contention, the analysis presented in Chapter 3 assumes that each memory phase of every job causes bus contention considering the worst-case memory access demand of memory phases. For example, for case 1 of FMAM analysis of Chapter 3, Lemma 3.9 bounds the maximum bus contention as follows.

If $N_{\pi_l}(W_{i,l}) > N_{\pi_r}(W_{i,l})$, the maximum bus contention that can be suffered by tasks executing on the local core due to tasks running on a remote core π_r during any time window of length $W_{i,l}$ is upper bounded by $Bus_{i,r}(W_{i,l})$, where

$$Bus_{i,r}(W_{i,l}) = \sum_{\tau_u \in \Gamma_r'} \left\lceil \frac{W_{i,l}}{T_u} \right\rceil \times (MD_u^A \times t^{mem} + MD_u^R \times t^{mem}) \tag{5.17}$$

We can see that the bound on bus contention given by Equation 5.17 is pessimistic since it considers the maximum number of memory requests issued during each memory phase in isolation. To address this pessimism, we can apply the cache analysis presented in Section 5.3 to Equation 5.17. Using Lemma 5.3, we can improve and reformulate Equation 5.17 as follows.

$$\begin{aligned} \hat{Bus}_{i,r}(W_{i,l}) = \sum_{\tau_u \in \Gamma_r'} \min \Bigg(& \left\lceil \frac{W_{i,l}}{T_u} \right\rceil \times MD_u^A \times t^{mem}, MD_u^A \times t^{mem} + \left(\left\lceil \frac{W_{i,l}}{T_u} \right\rceil - 1 \right) \times (\bar{MD}_u^A + |\bar{\rho}_u|) \times t^{mem} \Bigg) \\ & + \left\lceil \frac{W_{i,l}}{T_u} \right\rceil \times MD_u^R \times t^{mem} \end{aligned} \tag{5.18}$$

In a similar manner to that of Equation 5.18, we can improve all cases considered in Chapter 3. For example, the bus contention for various cases is derived by forming sets M_r^A, M_r^R that contains the length of A- and R-phases in isolation. Consequently, applying the proposed persistence-aware cache analysis allows tightly bounding the length of each of the memory phases, thus, the maximum bus contention.

After bounding the maximum bus contention $\hat{Bus}_{i,r}(W_{i,l})$ that can be caused by each remote core $\pi_r \in m$ such that $\pi_r \neq \pi_l$, we can compute the *total bus contention* $Bus_{i,l}^{max}(W_{i,l})$ that tasks of local core π_l can suffer due to tasks of *all remote cores* during $W_{i,l}$ using Algorithm 1.

5.5 Worst Case Response Time Analysis

Having bounded the maximum number of memory requests using the analysis presented in Section 5.3, the bound on the maximum bus contention can be computed by improving the bus contention analysis presented in Chapters 3, 4. Now we can incorporate the resulting bound on the bus contention into the WCRT analysis of tasks. For this, we propose *improved WCRT* formulation that accounts for *cache reuse* when computing *bus contention* as well as the *maximum interference from higher priority tasks*. By applying cache persistence to higher priority tasks, we can tightly

bound the number of memory accesses issued during their memory phases which in turn reduces the length of their memory phases. Consequently, it can reduce the overall interference that can be caused by the memory phases of higher priority tasks that execute on the local core during the level-i busy window. Building on this, the length of the level-i busy window is given by the following lemma.

Lemma 5.6. The length of the level-i busy window for a given task τ_i executing on core π_l is denoted by $W_{i,l}$, where $W_{i,l}$ is given by the first positive solution to the fixed-point iteration of the following equation

$$W_{i,l} = (\Psi^A_{i,l}(W_{i,l}) + \Psi^R_{i,l}(W_{i,l})) \times t^{mem} + \max_{\tau_j \in lp_{i,l}} \{C_j\} + \sum_{\tau_h \in hep_{i,l}} \left\lceil \frac{W_{i,l}}{T_h} \right\rceil \times C^E_h + Bus^{max}_{i,l}(W_{i,l}) \quad (5.19)$$

Proof. From Equations 5.6 and 5.7, we know that $\Psi^A_{i,l}(W_{i,l})$ and $\Psi^R_{i,l}(W_{i,l})$ upper bounds the maximum number of memory requests that can be generated during the A- and R-phases of all tasks in $hep_{i,l}$ (including task τ_i) that execute on the local core π_l during any time window of length $W_{i,l}$. Assuming that each memory request will take t^{mem} time units, $(\Psi^A_{i,l}(W_{i,l}) + \Psi^R_{i,l}(W_{i,l})) \times t^{mem}$ upper bounds the contribution of the memory phases of all tasks in $hep_{i,l}$ that execute on the local core π_l during $W_{i,l}$. Due to the fixed priority non-preemptive scheduling, at most one job of a task in $lp_{i,l}$ can cause blocking to τ_i. This blocking is maximized by considering a task with the largest WCET among all the tasks in $lp_{i,l}$, given by $\max_{\tau_j \in lp_{i,l}} \{C_j\}$[4]. Similarly, the term $\sum_{\tau_h \in hep_{i,l}} \left\lceil \frac{W_{i,l}}{T_h} \right\rceil \times C^E_h$ upper bounds the contribution of the E-phases of all tasks in $hep_{i,l}$ (including task τ_i) that execute on the local core π_l during $W_{i,l}$.

Finally, $Bus^{max}_{i,l}(W_{i,l})$ is the *total bus contention* that can be suffered by tasks that execute on the local core π_l from all remote cores during $W_{i,l}$ and can be computed for the FCFS policy using algorithm 1 and for the RR policy using Equation 5.16. The Lemma follows. □

Note that $W_{i,l}$ appears on both sides of Equation 5.19 which means Equation 5.19 is recursive and a fixed-point computation on $W_{i,l}$ can be used to find a solution by initiating $W_{i,l} = \sum_{\tau_h \in hep_{i,l}} C_h + \max_{\tau_j \in lp_{i,l}} \{C_j\}$. The length of the level-i busy window $W_{i,l}$ will then be given by the smallest positive value of $W_{i,l}$ for which Equation 5.19 converges.

Having bounded the length of the level-i busy window, we can compute the latest *finish time* of the k^{th} job of τ_i on core π_l, i.e., $\tau_{i,k}$, using the following lemma.

Lemma 5.7. The latest finish time of $\tau_{i,k}$ is denoted by $f_{i,k}$, where $f_{i,k}$ is given by the first positive solution to the fixed-point iteration on the following equation:

$$f_{i,k} = (\Psi^A_{i,l}(f_{i,k}) + \Psi^R_{i,l}(f_{i,k})) \times t^{mem} + \max_{\tau_j \in lp_{i,l}} \{C_j\} + \sum_{\tau_h \in hep_{i,l}} \left\lceil \frac{f_{i,k}}{T_h} \right\rceil \times C^E_h + Bus^{max}_{i,l}(f_{i,k}) \quad (5.20)$$

[4]Note that we have considered a lower priority task with the largest A+R-phases while computing bus contention (see Lemma 5.4) so it is safe to consider $\max_{\tau_j \in lp_{i,l}} \{C_j\}$ while deriving the maximum blocking.

Proof. The proof directly follows from Lemma 5.6 except considering any time window of length $f_{i,k}$. □

Note that $f_{i,k}$ appears on both sides of Equation 5.20 which means Equation 5.20 is recursive and a fixed-point computation on $f_{i,k}$ can be used to find a solution by initiating $f_{i,k} = \sum_{\tau_h \in hep_{i,l}} C_h + \max_{\tau_j \in lp_{i,l}} \{C_j\}$. The latest finish time $f_{i,k}$ will then be given by the smallest positive value of $f_{i,k}$ for which Equation 5.20 converges.

Once Equation 5.20 converges, we can now compute the *response time* of $\tau_{i,k}$ as follows.

The response time of $\tau_{i,k}$ is denoted by $R_{i,k}$ and can be computed by subtracting the minimum inter-arrival time of previously executed jobs of task τ_i from the latest finish time $f_{i,k}$. Hence,

$$R_{i,k} = f_{i,k} - (k-1) \times T_i \tag{5.21}$$

Finally, the WCRT of task τ_i is denoted by R_i^{max} and can be computed by maximizing Equation 5.20 over all jobs of τ_i that execute during the level-i busy window, i.e.,

$$R_i^{max} = \max_{k \in [1, K_i]} \{R_{i,k}\} \tag{5.22}$$

where $K_i = \left\lceil \frac{W_{i,l}}{T_i} \right\rceil$.

A taskset is only said to be schedulable if $R_i^{max} \leq D_i$ for each task $\tau_i \in \Gamma$ and the total bus utilization of the taskset is less than or equal to 1, i.e., $\sum_{\tau_i \in \Gamma} \frac{(MD_i^A + MD_i^R) \times t^{mem}}{T_i} \leq 1$.

5.6 Experimental Results

In this section, we evaluate how much cache-aware bus contention analyses can improve the performance of the RR-based (presented in Chapter 4) and FCFS-based bus contention analysis (presented in Chapter 3). For the RR analysis, we assume that the bus slot size is equal to t^{mem}. This is chosen due to the observation in Chapter 4 that the bus contention is least when the bus slot size is equal to t^{mem}. Similarly, for the FCFS bus policy, we only consider the Fair Memory Access Model (FMAM) based bus contention analysis since it is the best performing FCFS-based analysis presented in Chapter 3.

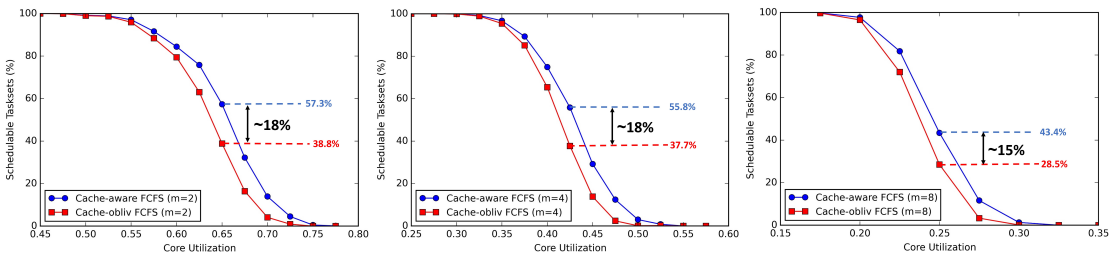
For the default configuration, we model a quad-core platform with a direct-mapped unified LLC of 32KB (1024 cache sets, 32-byte block) evenly partitioned to the cores. By default, we assume that there were 32 tasks in each taskset with 8 tasks randomly assigned to each core. Tasks utilization U_i was generated using the UUnifast-discard algorithm [Emberson et al., 2010]. Task periods T_i were randomly generated in the range of [1000-10000] using log-uniform distribution. The WCET in isolation C_i was then assigned by applying the relation $C_i = U_i \times T_i$. The total memory access demand (MD) of tasks was derived using C_i such that, $MD_i = rand(10\%, 40\%) \times C_i$. The length of the A-phase was chosen randomly in the range [60%-90%] of MD_i, i.e., $MD_i^A \times t^{mem} = rand(60\%, 90\%) \times MD_i$. The length of the R-phase was then given by $MD_i^R \times t^{mem} =$

$MD_i - (MD_i^A \times t^{mem})$. Finally, the length of the E-phase was given by $C_i^E = C_i - (MD_i^A + MD_i^R) \times t^{mem}$. We assume that tasks are mapped to the cache partition sequentially and in priority order. The number of ECBs of tasks was generated using the length of A-phase, i.e., $|ECB_i| = \frac{MD_i^A}{t^{mem}}$. Similarly, the number of PCBs for each task was generated randomly in the range [20%-80%] of its ECBs. Task priorities were assigned using rate monotonic algorithm [Liu and Layland, 1973]. Task deadlines were equal to task periods, i.e., $D_i = T_i$.

We compare the performance of the proposed cache-aware bus contention analysis for FCFS and RR bus policy with the cache-oblivious FCFS and RR policy-based bus contention analysis by varying: 1) the core utilization (i.e., utilization of each core); 2) the number of cores; 3) the memory access demand; and 4) the number of cache sets. We use taskset schedulability, i.e., the percentage of schedulable tasksets, as a metric to evaluate the performance of each approach.

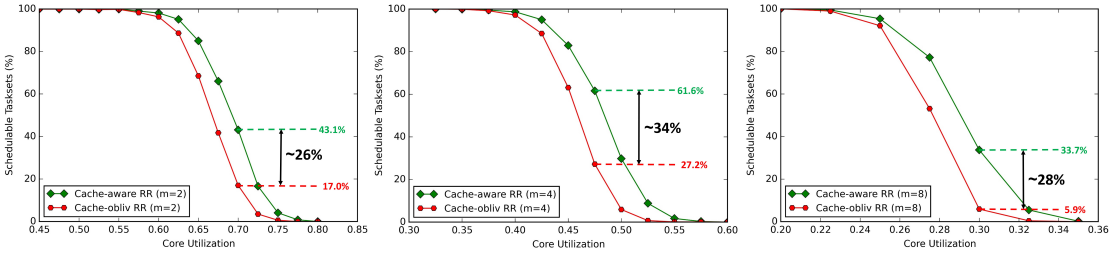
In all figures, the cache-aware bus contention analyses for the RR computed using Section 5.4.1 and FCFS bus policy computed using 5.4.2 are marked as "Cache-aware RR" and "Cache-aware FCFS", respectively. Similarly, the cache-oblivious RR policy-based analysis of Chapter 4 is marked as "Cache-obliv RR" and the cache-oblivious FCFS bus analysis of Chapter 3 is marked as "Cache-obliv FCFS". For all experiments, we randomly generated 1000 tasks per point. In all figures, the x-axis represents the core utilization and the y-axis represents the percentage of schedulable tasksets.

1) Varying Core Utilization: In this experiment, we varied the core utilization of each core under the default configuration, i.e., m=4, from 0.05 to 1 in steps of 0.025 and plotted the percentage of tasksets deemed schedulable by all the approaches. Figure 5.1b shows the percentage of tasksets deemed schedulable using the proposed cache-aware analysis and cache-oblivious analysis for the FCFS bus arbitration policy. Similarly, Figure 5.2b shows the percentage of tasksets deemed schedulable using the proposed cache-aware analysis and cache-oblivious analysis for the RR bus arbitration policy. For all the approaches, we observe that increasing the core utilization decreases the taskset schedulability. This is because an increase in the core utilization also increases tasks utilization which in turn increases the WCET of tasks as $C_i = U_i \times T_i$. This increase in WCET also increases the number of memory requests, which in turn increases the bus contention suffered by tasks, resulting in a decrease in taskset schedulability. However, we note that the proposed cache-aware analyses for FCFS and RR bus policies outperform their respective cache-oblivious counterparts. For example, we can see in Figure 5.1b that at a core utilization of 0.425, the



(a) FCFS bus policy, m=2 (b) FCFS bus policy, m=4 (c) FCFS bus policy, m=8

Figure 5.1: Varying core utilization and number of cores for the FCFS bus arbitration policy



(a) RR bus policy, m=2 (b) RR bus policy, m=4 (c) RR bus policy, m=8

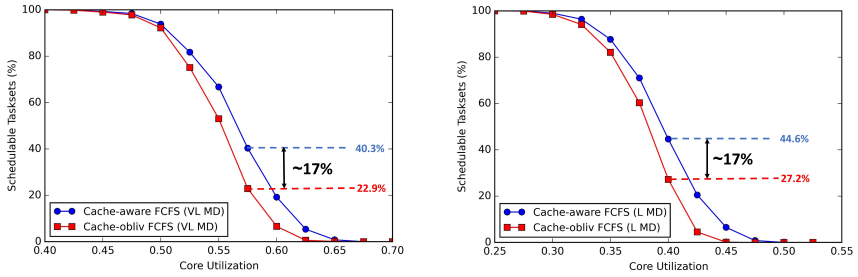
Figure 5.2: Varying core utilization and number of cores for the RR bus arbitration policy

proposed cache-aware FCFS analysis was able to schedule up to 18% more tasksets than cache-oblivious FCFS analysis. Similarly, we can see in Figure 5.2b that at a core utilization of 0.475, the proposed cache-aware RR analysis was able to schedule up to 34% more tasksets than the cache-oblivious RR analysis. These gains were observed because the proposed cache-aware bus contention analyses use a tighter bound on the number of LLC misses (computed using the analysis in Section 5.3) when computing bus requests and bus contention. On the contrary, the existing bus contention analyses are cache-oblivious policies as they always assume the worst-case number of LLC misses.

Furthermore, we observe that the gain of the proposed cache-aware analysis over the cache-oblivious analysis was significant for the RR policy (see Figure 5.2b) but was relatively smaller for the FCFS bus arbitration policy (see Figure 5.1b). This is mainly because the FCFS bus contention analyses only rely on the number of memory phases that can suffer/cause bus contention during a given time window. In such a case, applying the proposed cache-aware analysis to the FCFS bus may reduce the number of memory accesses per memory phase but it may not reduce the number of memory phases. On the contrary, the bus contention under the RR policy depends on the number of bus slots required by the local/remote core. These bus slots depend not only on the number of memory phases but also on the number of memory requests issued during each memory phase. Consequently, the proposed cache-aware analysis is more effective for the RR bus arbitration policy.

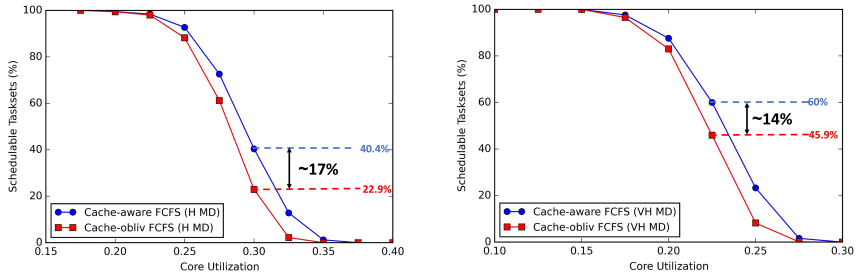
2) Varying Number of Cores: In this experiment, we varied the number of cores m from 2 to 8 and plotted the results for the FCFS bus policy in Figure 5.1 and for the RR bus policy in Figure 5.2. As shown in Figures 5.2 and 5.1, for all the approaches, increasing the number of cores decreases the taskset schedulability. For instance, when $m = 8$, the number of remote cores as well as the total number of tasks in the taskset increases, i.e., 64 tasks. This results in increasing bus contention, which decreases taskset schedulability. Due to the same reason, decreasing the number of cores improves the performance of all the approaches. Nonetheless, the cache-aware analyses outperform the cache-oblivious bus contention analyses for the FCFS and RR bus policies for all the values of m.

3) Varying Memory Access Demand (MD): In this experiment, we varied the Memory Access Demand (MD) of all the tasks in the taskset. For this, we consider the following configurations:
(a) Very Low (VL) MD, i.e., $MD_i = rand(5\%, 20\%) \times C_i$;



(a) FCFS bus policy, VL MD

(b) FCFS bus policy, L MD



(c) FCFS bus policy, H MD

(d) FCFS bus policy, VH MD

Figure 5.3: Varying Memory Demand (MD) for the FCFS bus arbitration policy

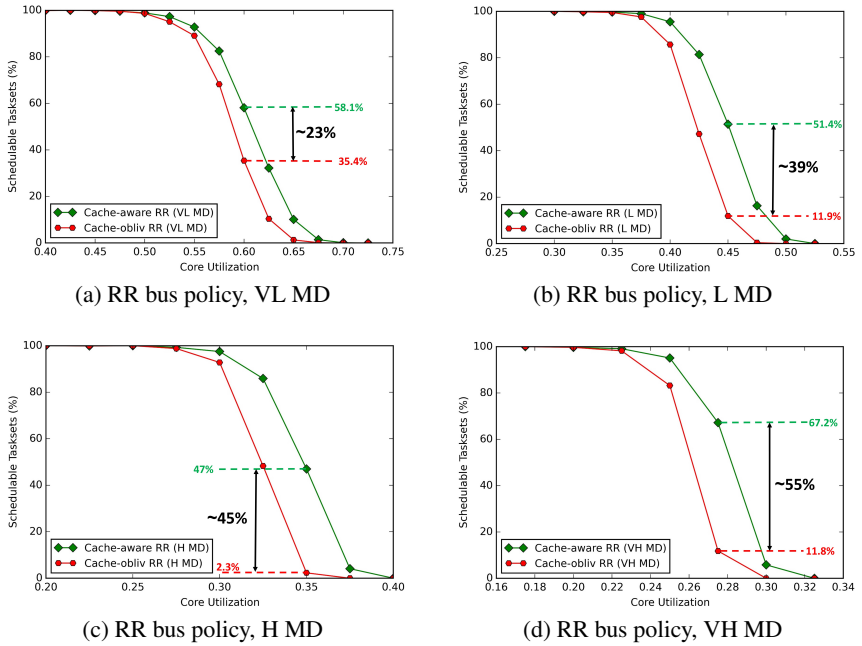
(b) Low (L) MD, i.e., $MD_i = rand(20\%, 40\%) \times C_i$;

(c) High (H) MD, i.e., $MD_i = rand(40\%, 60\%) \times C_i$; and

(d) Very High (VH) MD, i.e., $MD_i = rand(60\%, 80\%) \times C_i$.

The resulting percentage of schedulable taskset for the FCFS bus policy is plotted in Figure 5.3 and for the RR bus policy is plotted in Figure 5.4. For both the approaches and policies, we can observe in Figures 5.3 and 5.4 that the MD of tasks can significantly impact the taskset schedulability. In particular, all the approaches perform the best under the VL MD configuration and the worst under the VH MD configuration. This happens because increasing the MD value can increase the number of memory requests which in turn increases the bus contention and decreases the taskset schedulability. We observe that the gain of the proposed cache-aware FCFS analysis over the cache-oblivious FCFS analysis remains the same for almost all the values of MD. On the contrary, the gain of the proposed cache-aware RR analysis over the cache-oblivious RR analysis significantly increases with the increase in the MD value. For example, the proposed cache-aware RR analysis was able to schedule up to 55% more tasksets than the cache-oblivious RR analysis at the core utilization value of 0.275 under the VH MD configuration. However, for all the MD configurations, the proposed cache-aware bus contention analyses for both bus policies dominate the cache-oblivious analyses.

4) Varying Cache Size: In the default configuration, we assume that the total number of sets in the cache are 1024, and 256 cache sets are allocated per core. In this experiment, we consider different per-core cache set sizes, i.e., 64, 128, 256, 512, and plot the resulting taskset schedulability in Figure 5.5a for the FCFS bus policy and in Figure 5.5b for the RR bus policy. Note that increasing the number of cache sets allocated per core will effectively increase the total size of the cache. For this experiment, we do not show the taskset schedulability for the existing bus contention analyses



(a) RR bus policy, VL MD

(b) RR bus policy, L MD

(c) RR bus policy, H MD

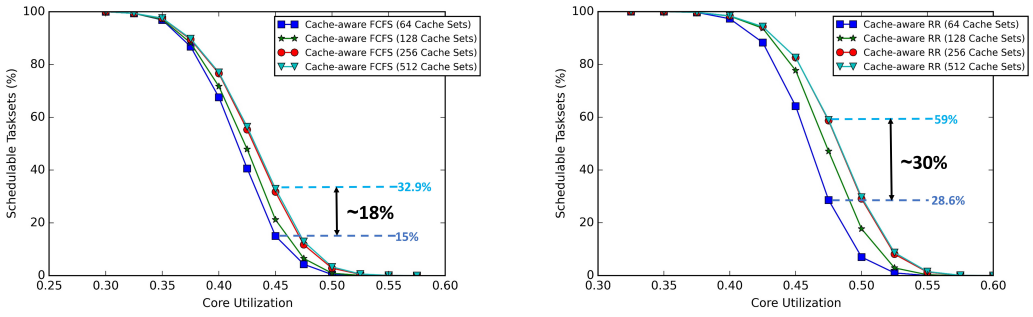
(d) RR bus policy, VH MD

Figure 5.4: Varying Memory Demand (MD) for the RR bus arbitration policy

as their schedulability does not depend on the cache size.

We can see in Figures 5.5a and 5.5b that decreasing the per-core cache sets to 64 or 128 also decreases the schedulability for both the bus arbitration policies. Intuitively, this happens because for a smaller number of cache sets per core, i.e., 64, 128, the overlap between the PCBs of a task with ECBs of other tasks increases. This leads to a higher CPRO which results in increasing the number of memory requests as well as bus contention. On the contrary, increasing the per-core cache sets to 256 or 512 also increases the schedulability for all approaches. This happens because increasing the number of cache sets allocated per-core reduces the overlap between the PCBs of tasks with ECBs of other tasks, thereby reducing CPRO. This results in improving taskset schedulability.

Interestingly, we observe that the difference between the taskset schedulability for cache set sizes of 256 and 512 is negligible. This happens because, in the default configuration, the value of



(a) Varying cache sets for the FCFS bus policy

(b) Varying cache sets for the RR bus policy

Figure 5.5: Varying number of cache sets

MD is low, i.e., $MD_i = rand(10\%, 40\%) \times C_i$, which is further divided among A- and R-phases, with the length of A-phases used to generate ECBs of tasks. This implies that a per-core cache set size of 256 is sufficient to ensure that the PCBs of tasks do not overlap with the ECBs of other tasks, i.e., tasks do not suffer CPRO. Consequently, a further increase in the per-core cache set sizes does not significantly impact taskset schedulability.

5.7 Chapter Summary

In this chapter, we discuss the importance of considering the interdependence between the memory bus and cache memories. We show how the bound on the bus contention can be improved using a cache-aware bus contention analysis that analyzes and integrates the number of cache misses while computing the bus contention. Specifically, we apply the notion of cache persistence to bound the number of memory requests issued during the memory phases of 3-phase tasks and integrate it in the computation of bus contention. The bound on the bus contention is then integrated into the WCRT analysis. Experimental results show that the taskset schedulability can be improved using the cache-aware bus contention analyses.

Part II

Memory Centric Scheduling

Chapter 6

Fixed Task Priority-based Memory Centric Scheduling

As discussed in Chapter 2, a memory-centric scheduler can be used at the system level to serialize the accesses to the main memory to reduce inter-task memory interference. Several implementations of Memory Centric Scheduling (MCS) have been proposed in the literature [Yao et al., 2012, Yao et al., 2016a, Rivas et al., 2019, Schwäricke et al., 2020]. The initial works adopted time-division multiple-access (TDMA) to implement MCS [Yao et al., 2012, Tabish et al., 2019]. However, due to its non-work-conserving nature, a TDMA-based MCS can overestimate the memory interference of tasks. In recent work, Schwäricke et al. [Schwäricke et al., 2020] have presented an analysis that implements MCS using *Processor-Priority* (PP)-based memory scheduler. In PP-based MCS, memory requests (or phases) of tasks are served depending on the priority of the processor/core on which the tasks execute. A two-level priority mechanism is used where at the core level, tasks are scheduled using partitioned fixed-priority non-preemptive scheduling and the memory arbiter employs a global fixed processor priority-based scheduling to schedule memory phases of tasks.

While the PP-based MCS approach can outperform the TDMA-based MCS, it still has limitations. For example, under the two-level priority mechanism used by the PP-based MCS, a task τ_i with the highest local priority on a core π_l may still suffer memory interference from other tasks that are executing on processors/cores with higher global priorities than π_l. This can have a significant impact on the schedulability of τ_i and consequently on the taskset schedulability. The memory interference suffered by tasks can be reduced by scheduling the memory phases at the system level on the basis of *fixed task priorities*. Thus, in this chapter, we propose a *Task-Priority* (TP) based MCS, i.e., tasks' memory requests (or phases) are served under a global priority order depending on the priority of the task that issues the requests. This leads to a significant reduction in the memory interference of tasks.

The main **contributions** of this chapter are the following:

1. We present an analysis to bound the total memory interference that can be suffered by the tasks under a TP-based MCS approach. In contrast to most recent work on MCS that considers non-preemptive tasks, our approach considers limited preemptive scheduling at the core level.

2. Existing implementations of MCS that allow task preemptions, e.g.,[Yao et al., 2012], usually assume fully preemptive computation phases of tasks. We investigate the impact of different preemption points on the memory interference suffered by tasks and show that this assumption may not always lead to a tighter bound on the memory interference of tasks.

3. We compare the performance of our proposed TP-based MCS approach to the PP-based MCS approach [Schwäricke et al., 2020] under different settings. Experimental results show that our proposed approach can provide significantly tighter bounds on the memory interference of tasks, which can lead to an improvement in the task set schedulability by up to 91 percentage points.

Chapter Organization: The rest of the chapter is organized as follows: Section 6.1 describes the system and task models. A motivational example is presented in Section 6.2. Section 6.3 discusses the proposed TP-based MCS analysis. The WCRT analysis for the proposed TP-based MCS is presented in Section 6.4. The impact of different preemption point selections on the memory interference of tasks is discussed in Section 6.5. Experimental results are detailed in Section 6.6, and followed by the chapter summary in Section 6.7.

6.1 System Model

We assume a multicore system comprising m identical cores $(\pi_1, \pi_2, \ldots, \pi_m)$ that access the *main memory* (e.g. DRAM) through a single memory arbiter that can handle only one memory request at a time. As in [Yao et al., 2012], we also assume that the local memory (i.e., L1/L2 cache) of each core can be partitioned among all the tasks running on that core such that each task has its own non-overlapping partition which is sufficiently large to store all its data/instructions. If this is not possible due to the limited size of the local memory, tasks can be divided into multiple segments using existing framework [Soliman and Pellizzoni, 2019, Soliman et al., 2019] so that the data/instructions required by any segment of a task can be stored in its own partition.

6.1.1 Task Model

We consider a task set Γ comprising n sporadic tasks from which a subset Γ' is assigned to each core at the design time according to any given task-to-core mapping strategy. Each task τ_i is characterized by C_i, that is the Worst-Case Execution Time (WCET) of τ_i *measured in isolation*, T_i, that is the minimum inter-arrival time between any two consecutive jobs of τ_i, and D_i, that is the relative deadline of τ_i. We assume $D_i \leq T_i$. Tasks assigned to a core at design time are not allowed to migrate during run-time. Task priorities are assigned at design time using a fixed-task

priority algorithm such as rate/deadline monotonic [Liu and Layland, 1973], ensuring that the index of each task is unique, which provides a global priority order. The global priority of each task translates into a local priority order on each core which is used for scheduling purposes.

We consider the 3-phase task model (see Section 2.1.5.2 for details). We assume fixed-priority limited preemptive scheduling where a lower priority task can be preempted *any time during the execution of its E-phase* by a higher priority task released on the same core. This assumption is in line with existing works, e.g., [Yao et al., 2012]. Memory phases are assumed to be non-preemptive and only one phase can execute at a time on a given core. Each task releases potentially infinite number of jobs where each job instance is denoted by k. The response time of the k^{th} job of task τ_i is denoted by $R_{i,k}$. The Worst-Case Response Time (WCRT) of task τ_i, i.e., the largest response time of any job of τ_i, is denoted by R_i^{max}.

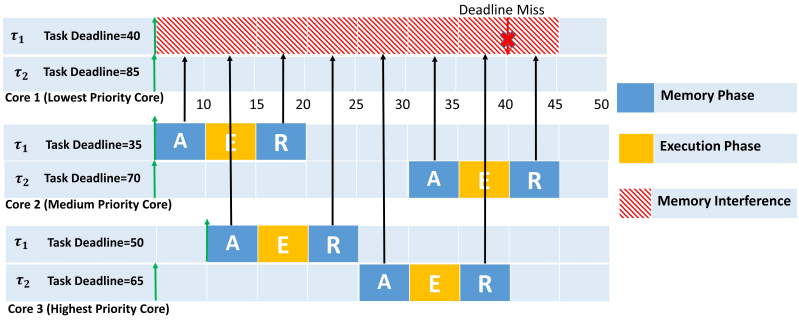
For notational convenience, we define $hp_{i,l}$, $hep_{i,l}$ and $lp_{i,l}$ to denote the set of tasks assigned to the local core π_l with priorities higher, higher or equal, and lower than that of τ_i, respectively. Similarly, $hp_{i,r}$ and $lp_{i,r}$ denotes the set of tasks assigned to a remote core π_r (i.e., $\pi_r \neq \pi_l$) with priorities higher and lower than that of τ_i, respectively.

6.1.2 Task Priority (TP) based Memory Centric Scheduler

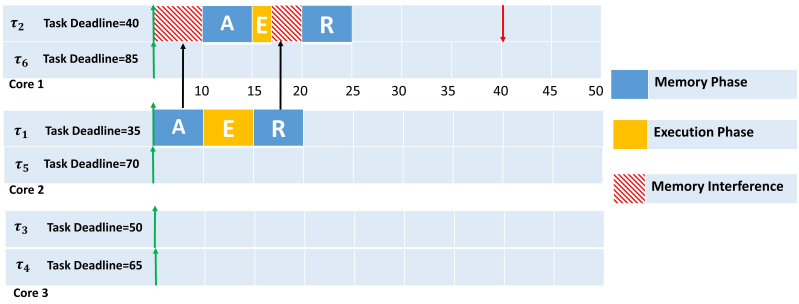
We assume that a Task Priority (TP) based memory-centric scheduler is used to control tasks' accesses to the main memory. Under the TP-based memory-centric scheduler, tasks' memory requests/phases are served in the global priority order. Each core maintains a *memory buffer* which stores at most one memory phase that is ready to execute. The state of this memory buffer can be updated by the core as per the tasks released on that core. The core's memory buffer can be empty if there is no active task or the core is executing an E-phase. If the memory buffer of at least one core is non-empty, the TP-based memory scheduler schedules a memory phase of a task that has the highest global priority among all ready tasks. Once a memory phase starts executing, the TP-based memory scheduler does not schedule any other memory phase to ensure the non-preemptive execution of the memory phase. Once the ongoing memory phase completes its execution, the TP-based memory scheduler checks the memory buffers of all the cores and schedules the memory phase of a task that has the highest global priority among all ready tasks.

6.2 Motivational Example

The first implementation of Memory Centric Scheduling (MCS) [Yao et al., 2012] considers TDMA-based static slots to schedule the memory accesses of tasks. The TDMA-based MCS allows the memory phases to preempt the execution phases at the core level to efficiently utilize the available TDMA slots. This can potentially improve the response time of tasks. However, the TDMA-based MCS is built on top of conventional TDMA which is a non-work-conserving arbitration policy and thus may overestimate memory interference of tasks. Schwäricke et al. [Schwäricke



(a) PP-based MCS [Schwäricke et al., 2020]



(b) TP-based MCS

Figure 6.1: Inter-core memory interference

et al., 2020] improved the TDMA-based MCS by considering Processor Priority (PP)-based memory scheduling. Their work considers a two-level scheduling approach: 1) fixed-priority non-preemptive scheduling to schedule tasks at the core level; 2) fixed processor priority to schedule the memory phases at the system level. Due to the two-level scheduling used by the PP-based MCS, tasks with higher local priorities that execute on lower global priority cores can suffer high memory interference, i.e., from all tasks that execute on all higher priority cores. This can potentially result in deadline misses. See Figure 6.1a, for an example scenario that shows 6 tasks are scheduled on 3 cores such that two tasks execute on each core. Task priorities are assigned at the core level using the deadline monotonic algorithm, i.e., the shorter the task deadline, the higher the task priority, and each core has a unique global priority to access the main memory. We can see in Figure 6.1a, that task τ_1 that executes on core 1 is the highest priority task on that core. However, since core 1 has the lowest global priority among all the cores, τ_1 on core 1 can suffer memory interference from all tasks executing on other higher priority cores (disregard of their local priorities). Consequently, this memory interference may lead to a deadline miss for task τ_1 on core 1.

It has been proven in the literature [Davis et al., 2017, Rashid et al., 2020] that fixed task priority-based memory scheduling can perform significantly better than fixed processor priority or TDMA-based scheduling for the generic task model (see Figure 5 of [Davis et al., 2017]). This provides a strong motivation to implement the memory-centric scheduler using a Task Priority (TP) based scheduling approach. In TP-based MCS, task priorities are assigned in a global priority order to schedule the main memory accesses. This global priority order translates into a local priority at the core, which is used to schedule the tasks at the core level. Consequently, by doing so,

TP-based MCS can improve the response time of all higher priority tasks, e.g., tasks with shorter deadlines/periods, at the system level. To illustrate, consider the same example scenario shown in Figure 6.1a applied to the TP-based MCS. The resulting schedule of tasks is shown in Figure 6.1b. Since the TP-based MCS assigns a global priority order to tasks, the task τ_1 executing on core 1 in Figure 6.1a, will be assigned a global priority of 2 according to the TP-based MCS. Effectively, task τ_1 executing on core 1 in Figure 6.1a is labeled as task τ_2 in Figure 6.1b. Consequently, we can see in Figure 6.1b that due to the global priority ordering used by TP-based MCS, task τ_2 will only suffer memory interference from one higher priority task, i.e., task τ_1 on core 2. Simple scenarios depicted in Figure 6.1 show that TP-based MCS can reduce the memory interference suffered by tasks in comparison to the PP-based MCS.

6.3 Analyzing Fixed Task Priority-based Memory Centric Scheduler

When combining phased task models, e.g., PREM or the 3-phase task model, with a memory-centric scheduler, the goal is to eliminate/minimize main memory interference suffered by the tasks. However, depending on the scheduling algorithm and the behavior of the memory scheduler, tasks may still be subjected to different types of execution delays. Under TP-based MCS, each task in the system is assigned a global priority using a fixed-priority scheduling scheme, e.g., Rate/Deadline monotonic. Effectively, any task τ_i executing on a core π_l will be served in a global priority order depending on its priority. Formally, under TP-based MCS, task τ_i can suffer four types of delays due to the tasks running on the local core and on remote cores, namely,

1. **Intra-core Interference:** The maximum interference that can be suffered by task τ_i due to all *higher priority tasks* released on the *local core* π_l.

2. **Intra-core Blocking:** The maximum blocking that can be suffered by task τ_i due to *lower priority tasks* that execute on the *local core* π_l.

3. **Inter-core Memory Interference:** The maximum memory interference that can be suffered by task τ_i due to all *higher priority tasks* executing on *all the remote cores*.

4. **Inter-core Memory Blocking:**[1] The maximum memory blocking that can be suffered by task τ_i due to all *lower priority tasks* executing on *all the remote cores*.

Now we will derive an upper bound on each of the above-mentioned terms.

6.3.1 Bounding Intra-Core Interference

The maximum intra-core interference that can be caused by all tasks in $hp_{i,l}$ during the level-i busy window $W_{i,l}$ depends on the maximum number of jobs released by all the tasks in $hp_{i,l}$ during $W_{i,l}$.

[1]Note that PP-based MCS [Schwäricke et al., 2020] use global memory preemptions to avoid inter-core memory blocking. However, global memory preemptions in TP-based MCS can lead to unbounded priority inversion as we explain in the Appendix A.

Therefore, to upper bound intra-core interference, we use the upper event arrival function $\eta_h^+(\Delta)$ that captures the maximum number of jobs released by a task τ_h in any time interval of length Δ [Schliecker and Ernst, 2010]. Consequently, the maximum intra-core interference that can be caused by all tasks in $hp_{i,l}$ during $W_{i,l}$ is given by

$$I_i(W_{i,l}) = \sum_{\tau_h \in hp_{i,l}} (\eta_h^+(W_{i,l}) \times C_h) \tag{6.1}$$

Equation 6.1 considers the WCET of all jobs released by all higher priority tasks on the local core, i.e., $\forall \tau_h \in hp_{i,l}$, during any time interval of length $W_{i,l}$.

6.3.2 Bounding Intra-Core Blocking

As explained in the system model, we assume limited preemptive scheduling where tasks can be preempted during the execution of their E-phases. Considering this, a given task τ_i can only suffer intra-core blocking due to only one memory phase of a lower priority task that starts executing before the arrival of τ_i because τ_i can preempt the lower priority task once it starts executing its E-phase. Therefore, the maximum intra-core blocking that can be suffered by task τ_i is given by the length of the largest memory phase (i.e., either A- or R-phase) among all the tasks in $lp_{i,l}$. The upper bound on the intra-core blocking of τ_i is denoted B_i and can be computed as follows:

$$B_i = \max(\max_{\forall \tau_j \in lp_{i,l}} \{C_j^A\}, \max_{\forall \tau_j \in lp_{i,l}} \{C_j^R\}) \tag{6.2}$$

6.3.3 Bounding Inter-Core Memory Interference

Under the TP-based MCS, the memory phases of a task τ_i can only be served after the completion of all the memory phases of all tasks having higher priority than τ_i. The contribution of tasks with higher priority than τ_i, executing on the local core π_l, is already accounted for in the intra-core interference $I_i(W_{i,l})$. Therefore, the maximum inter-core memory interference caused by all higher priority tasks running on all the remote cores will be computed using the following lemma.

Lemma 6.1. The maximum inter-core memory interference that can be suffered by tasks executing on the local core π_l due to higher priority tasks running on all the remote cores during $W_{i,l}$ is upper-bounded by $I_i^{Mem}(W_{i,l})$, where

$$I_i^{Mem}(W_{i,l}) = \sum_{r=1, r\neq l}^{m} \sum_{\tau_u \in hp_{i,r}} \eta_u^+(W_{i,l}) \times (C_u^A + C_u^R) \tag{6.3}$$

Proof. Under the TP-based MCS, memory phases of tasks are served in the global priority order. Thus, a task τ_i executing on a core π_l can suffer inter-core memory interference from all tasks executing on all the remote cores that have a higher priority than τ_i. A task τ_u released on a remote core π_r with a priority higher than that of τ_i, i.e., $\tau_u \in hp_{i,r}$, can only cause inter-core memory interference on τ_i when it executes its memory phases. So, the maximum inter-core

memory interference that one job of $\tau_u \in hp_{i,r}$ can cause is given by the sum of the WCET of its A- and R-phases, i.e., $C_u^A + C_u^R$. Furthermore, from the upper event arrival function, the maximum number of jobs released by task τ_u during any time interval of length $W_{i,l}$ is upper bounded by $\eta_u^+(W_{i,l})$. Hence, the maximum memory interference that can be caused by a task $\tau_u \in hp_{i,r}$ during $W_{i,l}$ is upper bounded by $\eta_u^+(W_{i,l}) \times (C_u^A + C_u^R)$. Considering that all higher priority tasks released on core π_r during $W_{i,l}$ can contribute to the inter-core memory interference, the maximum inter-core memory interference that can be caused by all tasks executing on core π_r is given by $\sum_{\tau_u \in hp_{i,r}} \eta_u^+(W_{i,l}) \times (C_u^A + C_u^R)$. Extending this result to all remote cores, the maximum inter-core memory interference that can be suffered by tasks executing on the local core during $W_{i,l}$ is upper bounded by Equation 6.3. □

6.3.4 Bounding Inter-Core Memory Blocking

Due to non-preemptive memory phases, a task τ_i can suffer inter-core memory blocking if a lower priority task on a remote core starts executing its memory phase before the release of a memory phase of task τ_i. This behavior is observed for all tasks that execute on the local core π_l during $W_{i,l}$. We use the following steps to compute the inter-core memory blocking.

- Bounding the *maximum number* of inter-core memory blockings that can be *suffered*.
- Bounding the *maximum number* of inter-core memory blockings that can be *caused*.
- Upper bounding the *maximum inter-core memory blocking* during $W_{i,l}$.

Next, we explain how each of these steps will be performed.

6.3.4.1 Bounding the maximum number of inter-core memory blockings that can be suffered

In this step, we will explain how to upper bound the maximum number of inter-core memory blockings that can be suffered by tasks executing on the local core during $W_{i,l}$. Firstly, we present the following example to illustrate the computation of this step. We then use Lemma 6.2 for formal computation.

Example 1: *Figure 6.2 shows an example schedule where 3 tasks are executing on the local core and task τ_3 is the task under analysis. Global priorities are assigned to tasks and are indexed according to their priorities, i.e., τ_1, τ_2, τ_3. We can see in Figure 6.2, each time a memory phase executes after an E-phase on the local core π_l, it may suffer inter-core memory blocking due to the execution of a memory phase (i.e., A or R-phase) of a lower priority task running on a remote core π_r. Furthermore, due to preemptive E-phases of tasks, each higher priority task can preempt a lower priority task during its E-phase in the worst-case scenario. So, when an E-phase is executed on core π_l, a lower priority task on a remote core can start executing its A/R-phase, causing inter-core memory blocking. Therefore, we see in Figure 6.2 that the inter-core memory blocking is suffered by each memory phase of tasks τ_1, τ_2, and τ_3 that executes after an E-phase on the local core.*

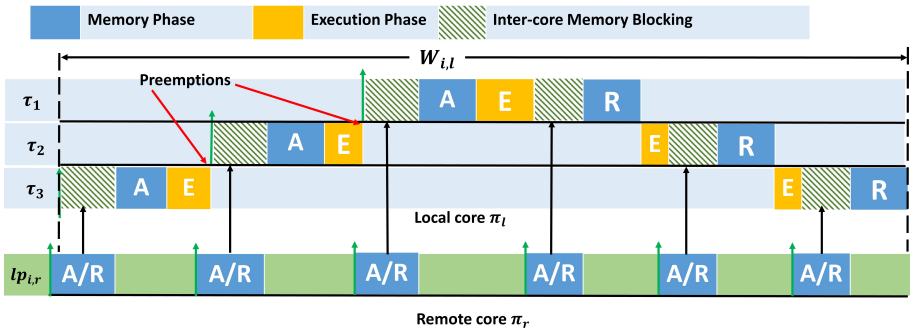


Figure 6.2: Maximum number of inter-core memory blockings suffered on the local core π_l during $W_{i,l}$

Lemma 6.2. The maximum number of times that tasks executing on the local core π_l can suffer inter-core memory blocking during $W_{i,l}$ is upper-bounded by $\Phi_i(W_{i,l})$, where

$$\Phi_i(W_{i,l}) = \sum_{\tau_h \in hep_{i,l}} \eta_h^+(W_{i,l}) \times 2 \tag{6.4}$$

Proof. It is only during the execution of E-phases that the local core can not schedule any memory phases during the level-i busy window. Consequently, in the worst case, the local core can suffer an inter-core memory blocking from a lower priority task executing on a remote core for every memory phase that executes on the local core after an E-phase. For example, if the local core is executing an E-phase at time instant t, the memory scheduler is allowed to schedule a memory phase of a lower priority task τ_l' executing on a remote core. Now, when the local core completes the execution of its E-phase and wants to execute a memory phase at time instant $t+\varepsilon$, it may suffer inter-core memory blocking as τ_l' is already executing a memory phase. This implies that the maximum number of memory blockings that the local core can suffer depends on the number of times E-phases are executed on the local core during $W_{i,l}$. However, considering that in our model the E-phases are preemptive, a task can be preempted several times during its E-phase and each preemption may lead to an inter-core memory blocking. Consequently, the local core can suffer several memory blockings during the execution of an E-phase. Although we cannot predict how many times an E-phase is preempted during $W_{i,l}$, we know that in the worst case each memory phase that executes during $W_{i,l}$ can suffer inter-core memory blocking. Therefore, knowing that $\eta_i^+(W_{i,l})$ upper bounds the number of jobs that can be released by task τ_i during $W_{i,l}$, $\eta_i^+(W_{i,l}) \times 2$ upper bounds the number of times τ_i can suffer inter-core memory blocking during $W_{i,l}$. Similarly, $\sum_{\tau_h \in hep_{i,l}} \eta_h^+(W_{i,l}) \times 2$ upper bounds the maximum number of inter-core memory blockings that can be suffered by all tasks executing on core π_l during $W_{i,l}$. □

6.3.4.2 Bounding the maximum number of inter-core memory blockings that can be caused

The maximum number of inter-core memory blockings that can be caused by lower priority tasks running on all remote cores during $W_{i,l}$ are computed using the following lemma.

Lemma 6.3. The maximum number of times that lower priority tasks running on all remote cores can cause inter-core memory blocking during $W_{i,l}$ is upper-bounded by $\mu_i(W_{i,l})$, where

$$\mu_i(W_{i,l}) = \sum_{r=1, r\neq l}^{m} \sum_{\tau_q \in lp_{i,r}} \eta_q^+(W_{i,l}) \times 2 \tag{6.5}$$

Proof. For a task τ_q running on a remote core π_r such that $\tau_q \in lp_{i,r}$, the maximum number of jobs that can be released by τ_q during $W_{i,l}$ is upper bounded by $\eta_q^+(W_{i,l})$. As memory phases are non-preemptive, each inter-core memory blocking caused by a lower priority task can be of at most one memory phase. Consequently, the maximum number of inter-core memory blockings that can be caused by one job of task τ_q during $W_{i,l}$ is 2 (i.e., by its A- and R-phases) and the maximum number of inter-core memory blockings that can be caused by all jobs of τ_q that execute during $W_{i,l}$ is upper bounded by $\eta_q^+(W_{i,l}) \times 2$. Similarly, the maximum number of inter-core memory blockings that can be caused by all lower priority tasks released on a remote core π_r, i.e., $lp_{i,r}$, during $W_{i,l}$ is upper bounded by $\sum_{\tau_q \in lp_{i,r}} \eta_q^+(W_{i,l}) \times 2$. Extending this to all remote cores, the Lemma follows. □

6.3.4.3 Upper bounding the maximum inter-core memory blocking

Having bounded the values of $\Phi_i(W_{i,l})$ and $\mu_i(W_{i,l})$, we will now compute an upper bound on the maximum inter-core memory blocking that can be suffered by tasks executing on the local core during $W_{i,l}$. To do so, we consider the following cases:

Case 1: $\Phi_i(W_{i,l}) \geq \mu_i(W_{i,l})$, the maximum number of inter-core memory blockings that can be suffered by tasks executing on core π_l is *greater than or equal* to the maximum number of inter-core memory blockings that can be caused by all lower priority tasks running on all remote cores during $W_{i,l}$.

Case 2: $\Phi_i(W_{i,l}) < \mu_i(W_{i,l})$, the maximum number of inter-core memory blockings that can be suffered by tasks executing on core π_l is *less* than the maximum number of inter-core memory blockings that can be caused by all lower priority tasks running on all remote cores during $W_{i,l}$.

Maximum Inter-Core Memory Blocking for Case 1: Under Case 1, the maximum inter-core memory blocking will be computed using the following lemma.

Lemma 6.4. If $\Phi_i(W_{i,l}) \geq \mu_i(W_{i,l})$, then the maximum inter-core memory blocking that can be suffered by tasks executing on the local core π_l due to lower priority tasks running on all remote cores during any time interval of length $W_{i,l}$ is upper-bounded by $B_i^{Mem}(W_{i,l})$, where

$$B_i^{Mem}(W_{i,l}) = \sum_{r=1, r\neq l}^{m} \sum_{\tau_q \in lp_{i,r}} \eta_q^+(W_{i,l}) \times (C_q^A + C_q^R) \tag{6.6}$$

Proof. As proven in Lemma 6.2, tasks running on the local core π_l during $W_{i,l}$ can suffer at most $\Phi_i(W_{i,l})$ inter-core memory blockings. As the precise memory access time of the lower priority tasks running on a remote core is not known at design-time, if $\Phi_i(W_{i,l}) \geq \mu_i(W_{i,l})$, there can be a scenario in which all the memory phases of all lower priority tasks released on all the remote cores

during $W_{i,l}$ can cause inter-core memory blocking to tasks executing on the local core π_l during $W_{i,l}$.

Thus, the maximum inter-core memory blocking that can be caused by one job of a lower priority task τ_q released on a remote core π_r, i.e., $\tau_q \in lp_{i,r}$, is upper-bounded by the sum of the WCET of its A- and R-phases, i.e., $C_q^A + C_q^R$. So, the maximum inter-core memory blocking that can be caused by all the jobs of task τ_q during $W_{i,l}$ is upper-bounded by $\eta_q^+(W_{i,l}) \times (C_q^A + C_q^R)$. Similarly, the maximum inter-core memory blocking that can be caused by all lower priority tasks executing on a remote core π_r during $W_{i,l}$ is upper-bounded by $\sum_{\tau_q \in lp_{i,r}} \eta_q^+(W_{i,l}) \times (C_q^A + C_q^R)$. Finally, the maximum inter-core memory blocking that can be suffered by tasks executing on the local core due to lower priority tasks running on *all the remote cores* during $W_{i,l}$ is upper-bounded by $\sum_{r=1,r\neq l}^{m} \sum_{\tau_q \in lp_{i,r}} \eta_q^+(W_{i,l}) \times (C_q^A + C_q^R)$. □

Maximum Inter-Core Memory Blocking for Case 2: We know that all tasks that execute on core π_l during $W_{i,l}$ can suffer at most $\Phi_i(W_{i,l})$ inter-core memory blockings. If $\Phi_i(W_{i,l}) < \mu_i(W_{i,l})$, we need to extract $\Phi_i(W_{i,l})$ number of memory phases released by all the lower priority tasks running on all the remote cores during $W_{i,l}$ that can lead to the maximum inter-core memory blocking. To do this computation, we introduce the following notations.

Let M be an ordered set that contains the WCET of all the memory phases (i.e., A- and R-phases) of all the *lower priority tasks released on all the remote cores* during any time interval of length $W_{i,l}$, *sorted in a non-increasing order* as follows:

$$M = \{C_1^{A/R}, C_2^{A/R}, \ldots, C_V^{A/R} \mid C_x^{A/R} \geq C_{x+1}^{A/R}\} \tag{6.7}$$

where $C_x^{A/R}$ denotes the WCET of either A- or R-phase of a lower priority task released on a remote core π_r during $W_{i,l}$. In Equation 6.7, the index V is equal to the $\mu_i(W_{i,l})$.

The maximum inter-core memory blocking for case 2 is then computed using the following lemma.

Lemma 6.5. If $\Phi_i(W_{i,l}) < \mu_i(W_{i,l})$, then the maximum inter-core memory blocking that can be suffered by tasks executing on the local core π_l due to lower priority tasks running on all remote cores during any time interval of length $W_{i,l}$ is upper-bounded by $B_i^{Mem}(W_{i,l})$, where

$$B_i^{Mem}(W_{i,l}) = \sum_{x=1}^{\Phi_i(W_{i,l})} C_x^{A/R} \text{ where } C_x^{A/R} \in M \tag{6.8}$$

Proof. As proven in Lemma 6.2, tasks running on the local core π_l during $W_{i,l}$ can suffer at most $\Phi_i(W_{i,l})$ inter-core memory blockings. As $\Phi_i(W_{i,l}) < \mu_i(W_{i,l})$, we need to extract $\Phi_i(W_{i,l})$ number of memory phases of the lower priority tasks released on all the remote cores during $W_{i,l}$ that can lead to the maximum inter-core memory blocking. As we cannot predict the actual schedule of task executions on remote cores, we do not know the specific memory phases of lower priority tasks running on remote cores that can cause inter-core memory blocking during $W_{i,l}$. Therefore, to maximize the inter-core memory blocking, we choose $\Phi_i(W_{i,l})$ number of memory phases with

the *largest execution times* among all the memory phases of lower priority tasks released on all the remote cores during $W_{i,l}$. This is achieved by summing up the first $\Phi_i(W_{i,l})$ elements of M, which contains the WCET of all memory phases of all lower priority tasks released on all remote cores during $W_{i,l}$. The Lemma follows. □

6.4 WCRT Analysis

Having bounded all the terms that can contribute to the length of the level-i busy window on core π_l, i.e., $I_i(W_{i,l})$, B_i, $I_i^{Mem}(W_{i,l})$, and $B_i^{Mem}(W_{i,l})$, the length of the level-i busy window $W_{i,l}$ is given by the first positive fixed-point solution of the following equation:

$$W_{i,l} = I_i(W_{i,l}) + B_i + \eta_i^+(W_{i,l}) \times C_i + I_i^{Mem}(W_{i,l}) + B_i^{Mem}(W_{i,l}) \tag{6.9}$$

where $\eta_i^+(W_{i,l}) \times C_i$ considers the maximum contribution of all jobs released by task τ_i during $W_{i,l}$.

Note that $W_{i,l}$ appears on both sides of Equation 6.9 which means Equation 6.9 is recursive and a fixed-point computation on $W_{i,l}$ can be used to find a solution by initiating $W_{i,l} = \sum_{\tau_h \in hep_{i,l}} C_h + \max_{\tau_j \in lp_{i,l}} \{C_j\}$. The length of the level-i busy window $W_{i,l}$ will then be given by the smallest positive value of $W_{i,l}$ for which Equation 6.9 converges.

Having bounded the length of the level-i busy window $W_{i,l}$, we compute the maximum number of jobs of task τ_i that can execute on core π_l during $W_{i,l}$ using the following equation.

$$K_i = \eta_i^+(W_{i,l}) \tag{6.10}$$

Using the values of $W_{i,l}$ and K_i, we can now compute the WCRT of task τ_i. For this, we need to analyze the response time of each job of task τ_i that executes during $W_{i,l}$. Let $\tau_{i,k}$ be the k^{th} job of task τ_i that execute during $W_{i,l}$. To compute the response time of $\tau_{i,k}$, we compute the latest start time of the R-phase of $\tau_{i,k}$ as it can be delayed by tasks running on the local core/remote cores until the start of its R-phase.

The latest start time of the R-phase of $\tau_{i,k}$ is denoted by $s_{i,k}^R$, where $s_{i,k}^R$ is given by the first positive solution to the fixed-point iteration on the following equation.

$$s_{i,k}^R = I_i(s_{i,k}^R) + B_i + ((k-1) \times C_i) + C_i^A + C_i^E + I_i^{Mem}(s_{i,k}^R) + B_i^{Mem}(s_{i,k}^R) \tag{6.11}$$

where $I_i(s_{i,k}^R)$ is the maximum intra-core interference suffered by $\tau_{i,k}$ during $s_{i,k}^R$, given by Equation 6.1. The term B_i is the maximum intra-core blocking, given by Equation 6.2. The term $(k-1) \times C_i$ considers the WCET of $k-1$ jobs of task τ_i. We consider the WCET of the A-phase and the E-phase of τ_i using $C_i^A + C_i^E$ while computing the latest start time of the R-phase of $\tau_{i,k}$. The term $I_i^{Mem}(s_{i,k}^R)$ considers the maximum inter-core memory interference suffered by $\tau_{i,k}$ during $s_{i,k}^R$, given by Lemma 6.1. The term $B_i^{Mem}(s_{i,k}^R)$ considers the maximum inter-core memory blocking that can be suffered by $\tau_{i,k}$ during $s_{i,k}^R$ and can be computed using Lemma 6.2 to Lemma 6.5.

As $s^R_{i,k}$ appears on both sides of Equation 6.11, it can be solved iteratively by initializing $s^R_{i,k} = C^A_i + C^E_i + B_i + \sum_{\tau_h \in hp_{i,l}} C_h$. The start time $s^R_{i,k}$ will then be given by the smallest positive value of $s^R_{i,k}$ for which Equation 6.11 converges.

Having computed the value of $s^R_{i,k}$, we can compute the response time $R_{i,k}$ of $\tau_{i,k}$ using the following equation.

$$R_{i,k} = s^R_{i,k} + C^R_i - (k-1) \times T_i \tag{6.12}$$

Finally, we can compute the WCRT of task τ_i by analyzing the response time of each job of τ_i that executes during $W_{i,l}$ and consider the largest response time among all the jobs, i.e.,

$$R^{max}_i = \max_{k \in [1,K_i]} \{R_{i,k}\} \tag{6.13}$$

where the computation of K_i is obtained using Equation 6.10.

A taskset Γ is said to be schedulable only if the WCRT of each task in the taskset is less than or equal to its relative deadline, the utilization of each core is less than or equal to the core's capacity, i.e., 1, and the total memory utilization of the taskset is less than or equal to 1, i.e., $\sum_{\tau_i \in \Gamma} \frac{C^A_i + C^R_i}{T_i} \leq 1$.

6.5 Analyzing the Impact of Preemption Point Selection

Most existing works in the state-of-the-art that focus on MCS of PREM/3-phase tasks assume non-preemptive scheduling at the core level [Schwäricke et al., 2020, Tabish et al., 2019]. Considering that in general, limited preemptive-based approaches tend to perform better than non-preemptive approaches in terms of schedulability, a few existing works have also considered limited preemptive scheduling-based MCS approaches [Yao et al., 2012]. The TP-based MCS approach presented in Section 6.3 also assumes limited preemptive scheduling where tasks executing on the same core can be preempted anytime during their E-phases. However, in this section, we will explore how preemption point selection can impact the TP-based MCS, by considering an alternate task scheduling approach where E-phases of tasks are also assumed to be non-preemptive, i.e., task preemptions are only allowed at the boundary of task phases. First, we will present an example that shows how this alternate preemption point selection can reduce the inter-core memory blocking of tasks. We will then discuss how the analysis presented in Section 6.3 needs to be adapted when considering this preemption scheme.

Example 2: *For the same example depicted in Figure 6.2, if the E-phases are non-preemptive, the resulting schedule is shown in Figure 6.3. Due to non-preemptive E-phases, each E-phase executes without being preempted and the local core can suffer at most one memory blocking from remote cores for each E-phase that executes on the local core. For instance, we can see in Figure 6.3 that the A-phase of τ_1, τ_2 only starts after the completion of an E-phase. Since a memory blocking can be suffered when the local core executes an E-phase, τ_1, τ_2 suffer memory blocking before their A-phases. However, this in turn leads to a scenario in which the R-phases of τ_2, τ_3 do not suffer inter-core memory blocking. This happens because the local core does not execute*

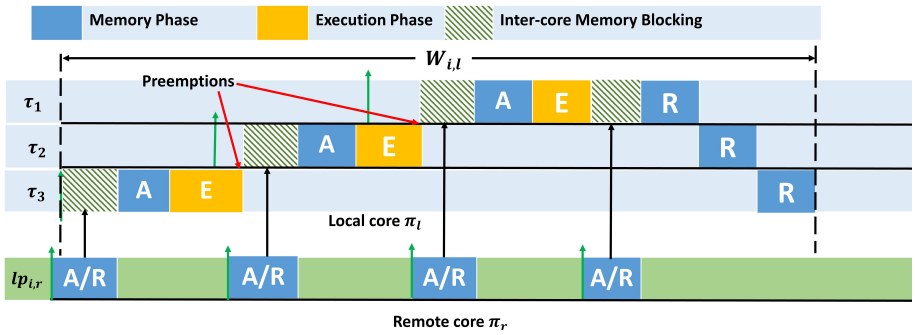


Figure 6.3: Maximum number of inter-core memory blockings for non-preemptive E-phases

any E-phase after the R-phase completion of τ_1 *and there is always a ready memory phase on the local core, thus, the memory scheduler will not schedule a memory phase of any lower priority task of a remote core. Consequently, for the same example scenario, at most* 4 *memory blockings can be suffered by tasks executing on the local core when the E-phases are non-preemptive in comparison to the* 6 *memory blockings suffered by the local core when E-phases are preemptive (see Figure 6.2).*

When analyzing the impact of non-preemptive E-phases of tasks on the TP-based MCS approach, the computation of intra-core interference and inter-core memory interference remains exactly the same as presented in Section 6.3, i.e., the intra-core interference can still be computed using Equation 6.1 and the inter-core memory interference will be upper bounded using Equation 6.3 (Lemma 6.1). However, the computation of intra-core blocking and inter-core memory blocking needs to be adapted, which is explained as follows.

6.5.1 Bounding Intra-Core Blocking

When each phase of a 3-phase task executes non-preemptively, task preemptions can only happen at the start/end of E-phases. So, if task τ_i is released when a lower priority task is already executing, τ_i suffers intra-core blocking from at most one phase (i.e., A, E, or R-phase) executing on the same core as τ_i. Therefore, the maximum intra-core blocking B_i suffered by task τ_i is given by the WCET of the largest A, E, or R-phase among all the tasks in $lp_{i,l}$, i.e.,

$$B_i = \max\Big(\max_{\forall \tau_j \in lp_{i,l}} \{C_j^A\}, \max_{\forall \tau_j \in lp_{i,l}} \{C_j^E\}, \max_{\forall \tau_j \in lp_{i,l}} \{C_j^R\}\Big) \tag{6.14}$$

6.5.2 Bounding Inter-Core Memory Blocking

As discussed in Section 6.3.4, the inter-core memory blocking of tasks depends on the value of $\Phi_i(W_{i,l})$, i.e., the maximum number of inter-core memory blockings that can be suffered by all tasks executing on the local core during $W_{i,l}$, and $\mu_i(W_{i,l})$, i.e., the maximum number of inter-core memory blockings that can be caused by all lower priority tasks executing on all the remote cores during $W_{i,l}$. When considering non-preemptive execution of E-phases of tasks, the computation of $\mu_i(W_{i,l})$ remains unchanged and it can be computed using Equation 6.5 (Lemma 6.3) as detailed

in Section 6.3.4.2. However, the computation of $\Phi_i(W_{i,l})$ needs to be adapted, which is done using the following lemma.

Lemma 6.6. If preemptions are allowed only at the start/end of E-phases of tasks, then the maximum number of times that tasks executing on core π_l can suffer inter-core memory blocking during $W_{i,l}$ is upper-bounded by $\Phi_i(W_{i,l})$, where

$$\Phi_i(W_{i,l}) = \sum_{\tau_h \in hep_{i,l}} \eta_h^+(W_{i,l}) + 1 \tag{6.15}$$

Proof. When considering non-preemptive E-phases of tasks, each E-phase that executes on the local core during $W_{i,l}$ will run until its completion. This implies that at most one inter-core memory blocking can be caused by a lower priority task of a remote core at the completion of each E-phase that executes on the local core during $W_{i,l}$. Consequently, the maximum number of inter-core memory blockings that can be suffered by all tasks that execute on the local core π_l during $W_{i,l}$ is equal to the maximum number of E-phases that execute on the local core π_l during $W_{i,l}$. As each job releases one E-phase, the maximum number of E-phases that can be released by a task τ_h can execute during $W_{i,l}$ is upper-bounded by $\eta_h^+(W_{i,l})$. Similarly, the maximum number of E-phases that can be released by all tasks in $hep_{i,l}$ during $W_{i,l}$ is upper-bounded by $\sum_{\tau_h \in hep_{i,l}} \eta_h^+(W_{i,l})$.

Additionally, we need to consider one inter-core memory blocking that can be suffered on the local core π_l at the start of the level-i busy window $W_{i,l}$. Therefore, the maximum number of inter-core memory blockings that can be suffered by tasks that can execute on the local core π_l during $W_{i,l}$ is upper bounded by $\sum_{\tau_h \in hep_{i,l}} \eta_h^+(W_{i,l}) + 1$. The Lemma follows. □

Having computed the value of $\Phi_i(W_{i,l})$ using Lemma 6.6, the maximum inter-core memory blocking can be computed using the exact same steps as detailed in Section 6.3.4. However, knowing that the maximum inter-core memory blocking that can be suffered by the tasks during $W_{i,l}$ depends both on the value of $\Phi_i(W_{i,l})$ and $\mu_i(W_{i,l})$, and the value of $\Phi_i(W_{i,l})$ computed using Lemma 6.6 can be different from the value of $\Phi_i(W_{i,l})$ when computed using Lemma 6.2. Therefore, the resulting values of the maximum inter-core memory blocking, i.e., $B_i^{Mem}(W_{i,l})$, computed considering non-preemptive E-phases can be different from the values obtained considering preemptive E-phases (i.e., analysis detailed in Section 6.3.4).

Finally, the WCRT for non-preemptive E-phase based scheduling can be computed using the exact same procedure detailed in Section 6.4, with B_i computed using Equation 6.14 and $B_i^{Mem}(W_{i,l})$ computed using Lemma 6.3 to Lemma 6.6.

6.6 Experimental Evaluation

In this section, we discuss the experiments that were performed to evaluate the effectiveness of the proposed TP-based MCS in comparison to the existing PP-based MCS [Schwäricke et al., 2020]. For the default configuration, we consider a multicore system composed of 4 cores and a taskset

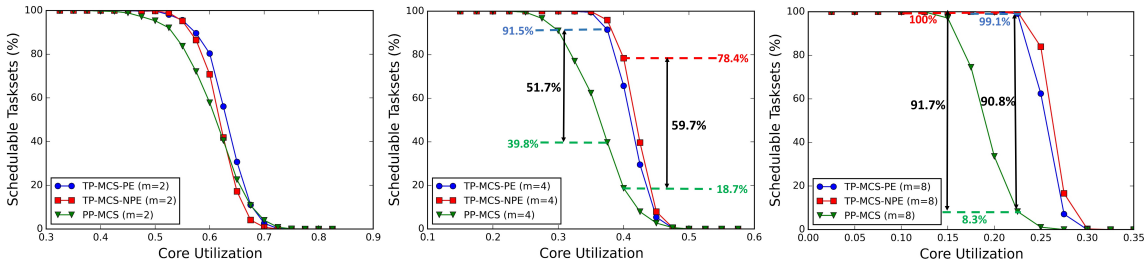
size of 32 tasks in which 8 tasks are assigned to each core. Tasks utilization U_i is randomly generated using the UUnifast-discard algorithm [Emberson et al., 2010]. Task periods T_i are randomly generated in the range of [100-1000] using log-uniform distribution. The WCET C_i is then assigned by applying the relation $C_i = U_i \times T_i$. The Memory Demand (MD) is assigned a random value in the range [10%-50%] of the WCET, i.e., $MD = rand(10\%, 50\%) \times C_i$. The WCET of the A- and R-phases[2] is given by $C_i^A = C_i^R = MD/2$ and the WCET of the E-phase is given by $C_i^E = C_i - (C_i^A + C_i^R)$. Task priorities are assigned globally using rate monotonic algorithm [Liu and Layland, 1973]. Task deadlines are implicit (i.e., $D_i = T_i$).

We evaluate the performance of the proposed TP-based MCS in comparison to the existing PP-based MCS [Schwäricke et al., 2020] by varying: 1) the core utilization (i.e., utilization of each core); 2) the number of cores; 3) task memory demands; and 4) the task period range. We use taskset schedulability, i.e., the percentage of schedulable tasksets, as a metric to evaluate the performance of each approach. For each point depicted in each plot, 1000 tasksets were randomly generated. In all the experiments, the proposed TP-based MCS that considers preemptive E-phases is marked as "TP-MCS-PE" whereas the proposed TP-based MCS that considers non-preemptive E-phases is marked as "TP-MCS-NPE". Similarly, the existing PP-based MCS [Schwäricke et al., 2020] is marked as "PP-MCS". In all plots, the x-axis represents the core utilization and the y-axis represents the percentage of schedulable tasksets for all the analyzed approaches.

1) Varying Core Utilization: In this experiment, we varied the core utilization of each core under the default configuration from 0.025 to 1 in steps of 0.025 and plotted the resulting number of schedulable tasksets in Figure 6.4b. Figure 6.4b shows that the taskset schedulability of all the approaches decreases by increasing the core utilization. This is mainly because increasing the core utilization increases the task utilizations, which, in turn, increases the WCET of tasks. This increase in C_i results in an increase in the values of C_i^A, C_i^E, and C_i^R. Consequently, the intra-core interference/blocking and inter-core memory interference/blocking increases, resulting in decreasing taskset schedulability. Nevertheless, the proposed TP-MCS-PE and TP-MCS-NPE approaches outperform the PP-based MCS. In particular, TP-MCS-PE analysis was able to schedule around 51% of more tasksets as compared to PP-based MCS at the core utilization value of 0.375. Similarly, the TP-MCS-NPE analysis was able to schedule around 59% of more tasksets as compared to PP-based MCS at the core utilization value of 0.40. This happens due to fixed task priority-based memory scheduling used by the proposed TP-based MCS that reduces the inter-core memory interference/blocking suffered by tasks in comparison to the PP-based MCS. Also, due to the use of limited preemptive scheduling, the proposed TP-based MCS approach reduces the intra-core blocking in comparison to the PP-based MCS that assumes non-preemptive task executions. Figure 6.4b also confirms that TP-MCS-NPE outperforms the TP-MCS-PE due to a tighter estimation of inter-core memory blocking.

2) Varying Number of Cores: In this experiment, we varied the number of cores, which in

[2]For PP-based MCS [Schwäricke et al., 2020], we consider one memory phase of length MD.



(a) Varying core utilization for m=2 (b) Varying core utilization for m=4 (c) Varying core utilization for m=8

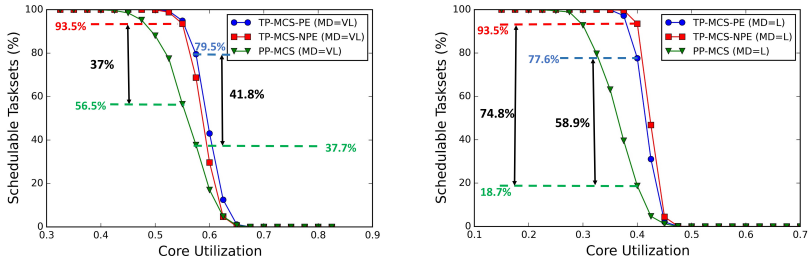
Figure 6.4: Varying core utilization and number of cores

turn, also varies the number of tasks in the taskset[3]. We varied the number of cores m between 2 to 8 along with the core utilization. As shown in Figure 6.4, increasing the value of m results in a decrease in taskset schedulability for all the considered approaches. This is because increasing the number of cores also increases the number of remote cores and thus the number of tasks running on remote cores. This increases the inter-core memory interference and memory blocking, eventually resulting in decreasing taskset schedulability.

We can see in Figure 6.4c that increasing the number of cores tends to increase the difference between our proposed approaches and the PP-based MCS. In particular, the TP-MCS-PE was able to schedule around 90% more tasksets than PP-based MCS at the core utilization value of 0.225. Similarly, the TP-MCS-NPE was able to schedule around 91% of more tasksets than PP-based MCS at the core utilization value of 0.225. On the contrary, the gain of TP-MCS-PE and TP-MCS-NPE over PP-based MCS was negligible for $m = 2$. In fact, the PP-based MCS was able to perform slightly better than TP-MCS-NPE for some values of core utilization as shown in Figure 6.4a. We explain these variations as follows: when the number of cores are smaller then the impact of inter-core memory interference and memory blocking is not that significant due to fewer tasks on remote cores. This results in producing a similar performance of all the approaches. Similarly, for $m = 2$, tasks scheduled using PP-based MCS suffer inter-core memory interference from only one remote core, thereby, resulting in a slightly better performance than the TP-MCS-NPE approach. Note that for $m = 2$, TP-MCS-PE also performs slightly better than TP-MCS-NPE. This is mainly due to the fact that TP-MCS-PE provides a slightly tighter bound on the intra-core blocking than TP-MCS-NPE whose impact is maximized when the taskset size is smaller.

3) Varying Memory Demand (MD): In this experiment, we vary the Memory Demand (MD) of all tasks in the taskset along with the core utilization. For this, we consider 4 different configurations based on the value of MD, that are, Very Low (VL) MD, i.e., MD=$(5\%, 20\%) \times C_i$, Low (L) MD, i.e., MD=$(20\%, 40\%) \times C_i$, High (H) MD, i.e., MD=$(40\%, 60\%) \times C_i$, Very High (VH) MD, i.e., MD=$(60\%, 80\%) \times C_i$. The value of MD is assigned to each task in the taskset randomly as per the chosen configuration.

[3]Per core tasks remains the same but increasing/decreasing number of cores results in increasing/decreasing the total tasks in the taskset



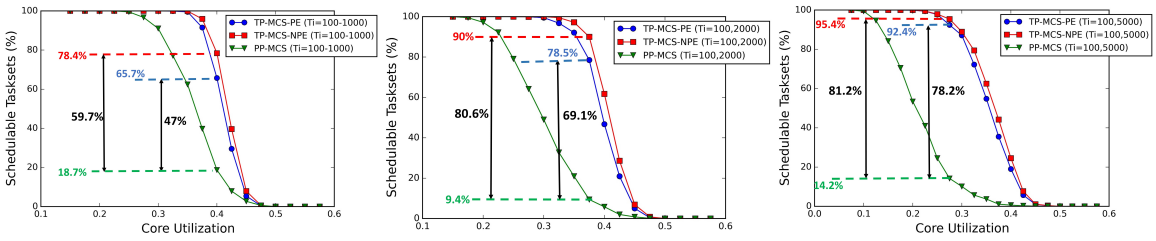
(a) VL MD (b) L MD

(c) H MD (d) VH MD

Figure 6.5: Varying core utilization for different MD configurations

As shown in Figure 6.5, all the approaches perform the best in the VL configuration and the worst in the VH configuration. This is intuitive as an increase in the value of MD also increases the WCET of memory phases which results in increasing the inter-core memory interference and memory blocking. However, we can see that for all configurations, the proposed TP-based MCS approaches outperform the PP-based MCS. In fact, for H and VH configurations, the difference between our proposed approaches and the PP-based MCS becomes more prominent. This is due to increasing the length of memory phases, which directly impacts the memory interference of tasks.

4) Varying Task Periods: In this experiment, we vary the core utilization for different task period ranges. For this, we consider three task period ranges that are [100-1000], [100-2000], and [100-5000]. As shown in Figure 6.6, the percentage of tasks deemed schedulable by all the approaches is reduced by increasing the task period ranges. This is mainly because, by increasing the task period, the WCET of execution and memory phases of tasks also increases. This has a direct impact on the inter-core memory interference/blocking of tasks. However, we can see in Figure 6.6 that the proposed TP-based MCS approaches outperform the PP-based MCS for all



(a) Task period range of 100-1000 (b) Task period range of 100-2000 (c) Task period range of 100-5000

Figure 6.6: Varying core utilization for different task period ranges

task period ranges. As the proposed TP-based MCS provides a tighter bound on the memory interference, the gain of the proposed analyses over PP-based MCS increases with the increase in task period range due to the higher impact of inter-core memory interference/blocking.

6.7 Chapter Summary

This chapter extends the notion of memory-centric scheduling to consider task priority-based memory scheduler. We showed how the memory interference of 3-phase tasks executing on a multicore platform can be bounded, assuming memory requests are served based on the priority of the generating task. Contrary to most works in the state-of-the-art, our analysis supports limited preemptive scheduling and also investigates the impact of preemption point selection on the inter-core memory interference suffered by tasks. Experimental results reveal that the proposed TP-based MCS can schedule up to 91% more tasksets than the state-of-the-art PP-based MCS.

Part III

Memory Contention Analysis

Chapter 7

Memory Contention Analysis for 3-Phase Tasks

The solutions presented in Chapters 3, 4 and 5 mainly focus on the contention that can be suffered by tasks due to the sharing of the memory bus. The common assumption in these chapters, as well as the state-of-the-art approaches that focus on the problem of memory bus contention [Schranzhofer et al., 2010, Andersson et al., 2010, Rosen et al., 2007, Chattopadhyay et al., 2010, Chattopadhyay and Roychoudhury, 2011, Kelter et al., 2011, Kelter et al., 2014, Dasari et al., 2011, Dasari and Nelis, 2012, Dasari et al., 2015, Rihani et al., 2015, Jacobs et al., 2015, Jacobs et al., 2016, Davis et al., 2017, Rashid et al., 2020, Maia et al., 2017, Thilakasiri and Becker, 2023a], is that the memory bus remains busy until the completion of a memory request. Consequently, the memory bus becomes the main source of shared resource contention since there can be at most one pending memory request at a time. However, there are architectures that use the shared memory bus with split transactions, i.e., the memory bus only remains busy during communication. Furthermore, as shown in an existing work [Casini et al., 2020], there are also some existing architectures that use multichannel interconnect that allow point-to-point communication between cores and the main memory. In such architectures, there can be a scenario in which multiple memory requests can be pending at the main memory. In such a scenario, the *main memory* can become the major source of contention.

Analyzing main memory contention is more complex due to the several low-level arbitration mechanisms employed by the memory controller which can potentially reorder requests. Considering the importance of the memory contention problem in multicore platforms, a few existing works [Kim et al., 2014, Kim et al., 2016, Yun et al., 2015, Ecco and Ernst, 2017, Hassan and Pellizzoni, 2018, Hassan and Pellizzoni, 2020, Casini et al., 2020] have focused on analyzing the maximum memory contention that can be suffered by tasks. These works consider *DRAM* (see Section 2.1.4.4 for an overview) as the global main memory and models DRAM as a *white-box*, i.e., taking into account the DRAM organization and the low-level arbitration mechanism

employed by its memory controller.

To the best of our knowledge, the only work that focuses on memory contention analysis for the 3-phase task model was proposed by Casini et al. [Casini et al., 2020]. Their memory contention analysis considered partitioned fixed-priority scheduling and assumes that the memory controller uses *write batching*[1] to prioritize read requests over write requests since writes do not stall the processing pipeline. Even though the solutions presented in their paper are important, it has some limitations. For example, instead of accurately quantifying the number of write batches that can be served by the memory controller during the execution of a task, their analysis overestimates the number of interfering write requests. Specifically, the analysis in [Casini et al., 2020] assumes that either one batch of write requests is served upon the completion of each read request or the overall delay that can be suffered by the A-phase is given by the length of the write-buffer plus all R-phases of jobs of tasks that can be released on all remote cores during the A-phase under analysis. This is a pessimistic bound because, in the 3-phase task model, an R-phase can only be issued by the processing core after the completion of an A-phase. As a result, the actual number of R-phases that can be issued by a core depends on the number of A-phases that can be completed on that core during a given time window and not necessarily on the number of jobs released by tasks running on that core.

In this chapter, we address the limitation of [Casini et al., 2020], we start by accurately quantifying the maximum number of batches that can be served by the memory controller during the execution of the task under analysis. We then propose a new memory contention analysis by leveraging the memory address mapping of tasks. The main idea is that if the memory address mapping of tasks is known, we can compute the minimum number of row-hit requests. However, achieving this can be extremely complex due to write batching. For example, Hassan et al. [Hassan and Pellizzoni, 2020] proposed the memory contention analysis for generic tasks considering various configurations of the memory controller. They also bound the minimum number of row hit requests for those configurations that do not use write batching. The main challenge is that when the memory controller uses write batching, upon serving a batch of write requests, the status of the row buffer can be changed due to write requests that target a different row than the activated row of the bank. To address this, we bound the minimum number of row hit requests by taking into account the maximum number of write batches that can be served by the memory controller during the execution of the task under analysis. Bounding the minimum number of row hits allows to tightly bound the memory contention suffered by tasks and memory access times of requests.

Formally, the main **contributions** of this chapter are the following:

1. We propose a memory contention analysis for 3-phase tasks that provides a tighter bound on memory contention that can be caused by write memory requests by showing that interfering write requests depend on interfering read requests.

[1] Please refer to Section 2.1.4.4 for detailed background on DRAM which also describes terms like write batching.

2. We show that memory access times and memory contention of tasks strongly relate to the memory address mapping of tasks. Building on this, the proposed memory contention analysis takes into account two different memory address mappings, i.e., *random memory access mapping* and *bank-level contiguous address mapping*, of tasks to accurately quantify the memory access times and memory contention suffered by tasks.

3. We show how the derived bounds on memory contention and memory access times can be incorporated in the worst-case execution time (WCET) of tasks. A WCRT-based schedulability analysis is then derived by using the WCET of tasks.

4. We perform an extensive experimental evaluation to compare the performance of the proposed memory contention analysis against the state-of-the-art. Specifically, two types of experiments were performed: (i) case study experiments, that use benchmarks from the Mälardalen Benchmark suite [Gustafsson et al., 2010], and (ii) empirical experiments, that use synthetically generated task sets. Results reveal that the proposed memory contention analysis can perform significantly better than the existing analysis [Casini et al., 2020] under different configurations by improving task set schedulability by up to 100% percentage points. Results also show that memory address mapping of tasks can have a significant impact on memory contention suffered by tasks.

7.1 System Model

We assume a multicore platform comprising m identical cores $(\pi_1, \pi_2, \ldots, \pi_m)$. The DRAM is shared among all the cores. Similarly to the existing work [Casini et al., 2020], we assume that the shared DRAM is accessed by cores via a set of crossbar switches that facilitates the point-to-point connection between each core and main memory. We assume that the shared cache is partitioned among cores such that each core has its non-overlapping partition. Furthermore, the local memory of each core is large enough to store all the data/code required by the task with the largest memory footprint that can execute on that core. The platform model considered in this chapter is shown in Figure 7.1.

7.1.1 Task Model

We consider the 3-phase task model [Durrieu et al., 2014], in which the execution of each task is divided into A, E, and R-phases (see Section 2.1.5.2 for details). Each phase as well as the complete task executes non-preemptively. We consider a task set Γ comprising n sporadic tasks $(\tau_1, \tau_2, \ldots \tau_n)$ partitioned among cores at design time. T_i denotes the minimum inter-arrival time between two consecutive jobs of task τ_i, and D_i denotes its relative deadline. We assume that tasks have constrained deadlines, i.e., $D_i \leq T_i$. We assume that the maximum number of memory requests that can be issued during the A-phase (resp. R-phase) of task τ_i *in isolation* is denoted by MD_i^A (resp. MD_i^R). Similarly, the WCET of the E-phase of task τ_i is given by C_i^E. Note that the values of MD_i^A, MD_i^R, and C_i^E can be obtained by static analysis, measurement-based analysis,

or by using the combination of both [Wilhelm et al., 2008]. Furthermore, we assume that the maximum time required to serve a single row miss (resp. row hit) memory request in *isolation* is denoted by t^{miss} (resp. t^{hit}). Finally, the WCET of task τ_i *in isolation* is denoted by C_i and can be computed by assuming every memory request a row miss request, i.e., $C_i = (MD_i^A + MD_i^R) \times t^{miss} + C_i^E$. We assume that tasks are scheduled using fixed-priority non-preemptive scheduling with priorities assigned using any fixed-priority algorithm such as Rate Monotonic or Deadline Monotonic [Liu and Layland, 1973].

For notational convenience, we define $hep_{i,l}$ to denote the set of tasks with a priority higher than or equal to that of τ_i running on a given core π_l. Similarly, $hp_{i,l}$ (and $lp_{i,l}$) denote the set of tasks with a priority higher (and lower) than that of τ_i, running on core π_l.

7.1.2 Main Memory Model

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

R1. Each bank has its *per-bank queue* in which memory requests targeting respective banks are queued. Each per-bank queue is scheduled using the *First-Ready First-Come-First-Serve* (FR-FCFS) [Kim et al., 2014, Hassan and Pellizzoni, 2018, Casini et al., 2020] policy which means 1) memory requests that result in a row-hit are prioritized over memory requests that result in a row-miss; 2) in case of a tie, older memory requests are prioritized over newer memory requests.

R2. For the A-phases of tasks, we consider that banks are partitioned to cores such that each core has its set of banks [Liu et al., 2012, Yun et al., 2015]. Specifically, A-phases of all tasks mapped on each core cannot access any bank assigned to another core. However, for the purpose of data sharing, the R-phases of all tasks in the system can access any bank[2].

[2]This method of communication is commonly referred to as explicit communication which is used in Logical Execution Time (LET).

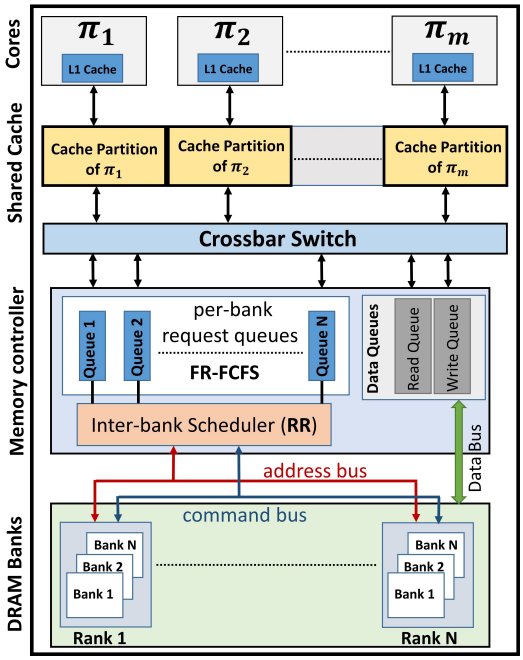


Figure 7.1: Illustration of the platform model

R3. The inter-bank scheduling policy is Round-Robin (RR) [Yun et al., 2014, Hassan and Pellizzoni, 2018, Casini et al., 2020] which serves the memory requests from each per-bank queue with the granularity of one memory request, i.e., one memory request per bank in each turn. Furthermore, to avoid unbounded delay, we assume that the inter-bank scheduler cannot reorder requests [Casini et al., 2020].

R4. Similarly to [Casini et al., 2020], we consider that cores have an *in-order pipeline* which means that there can be at most one pending memory request per core at a time. A subsequent memory request on a given core will only be issued after the previous memory request is served.

R5. We assume that reads have higher priority than writes since writes do not stall the processing pipeline. Write requests are enqueued in a write buffer of size Q_{write} and then served in batches with the watermarking mechanism [Chatterjee et al., 2012] to improve the turnaround time of data bus [Ecco and Ernst, 2017]. Specifically, if there are pending read requests, the memory controller only starts serving write requests if the number of write requests enqueued in the write buffer is greater than the *watermarking threshold* W_{th} and serves at least one *batch* of write requests where the length of the batch is denoted by N_{Wb}. Similarly to [Yun et al., 2015, Casini et al., 2020], we assume that $W_{th} > Q_{write} - N_{Wb}$.

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

R7. The memory controller serves the requested data in *burst mode* [Kim et al., 2014] in which BL successive columns of the same row are served in a single burst where BL is the burst

length. We assume that at least one memory request is served in a single burst memory access.

Symbol	Description
m	Number of cores in the system
τ_i	i^{th} task
T_i	Minimum inter-arrival time between any two consecutive jobs of τ_i
D_i	Relative deadline of τ_i
C_i	WCET of τ_i in isolation
MD_i^A	Maximum number of memory requests issued during the A-phase of τ_i in isolation
MD_i^R	Maximum number of memory requests issued during the R-phase of τ_i in isolation
C_i^E	WCET of the E-phase of τ_i in isolation
U_i	Utilization of task τ_i
π_l	Local core (i.e., the core on which τ_i is running)
π_r	Remote core (i.e., any core other than the local core)
$hep_{i,l}$	Set of tasks with priorities higher than or equal to that of τ_i running on core π_l
$hp_{i,l}$	Set of tasks with priorities higher than that of τ_i running on core π_l
$lp_{i,l}$	Set of tasks with priorities lower than that of τ_i running on core π_l
Γ'_l	Set of tasks assigned to the local core π_l
Γ'_r	Set of tasks assigned to a remote core π_r
Q_{write}	Size of the write buffer
W_{th}	Watermarking threshold
N_{Wb}	Number of write requests served in each batch of writes
BL	Burst length
t^{miss}	The maximum time required to serve a row miss request
t^{hit}	The maximum time required to serve a row hit request
$R_{i,k}$	Response time of k^{th} job of τ_i executing on core π_l
R_i^{max}	WCRT of τ_i

Table 7.1: Table of Symbols

7.2 Background

In this section, we will discuss the basic building blocks provided by the existing works to compute the memory contention suffered by tasks. We will later leverage them to build the proposed memory contention analysis.

Due to memory contention tasks can suffer *intra-bank contention*, i.e., due to interfering memory requests targeting the same bank as the task under analysis, as well as *inter-bank contention*,

i.e., due to interfering memory requests targeting a different bank than the task under analysis. Since banks are partitioned between the A-phases of different cores (see Rule **R2.**), and the cores use the in-order pipeline (see Rule **R4.**) the read requests of task τ_i can only suffer *inter-bank contention*. Now we will briefly discuss the bounds on the inter-bank contention derived in the state-of-the-art.

It has been shown in [Yun et al., 2015] that the maximum inter-bank contention that can be suffered by a memory request is given by the maximum contention it can suffer at any of the commands it comprises. Specifically, assuming that the request under analysis and interfering requests are all row miss requests, i.e., they issue *PRE*, *ACT*, and *CAS* commands in sequence, the maximum inter-bank contention that can be suffered by the request under analysis from N_{rq} interfering read requests is upper bounded by (From Theorem 1 of [Yun et al., 2015])

$$L(N_{rq}) = \max_{N_{PRE}+N_{ACT}+N_{CAS}=N_{rq}} \left(L^{PRE}(N_{PRE}) + L^{ACT}(N_{ACT}) + L^{CAS}(N_{CAS})\right) \tag{7.1}$$

Specifically, Theorem 1 of [Yun et al., 2015] upper bounds the inter-bank contention that can be suffered by the request under analysis by maximizing the contention that the request under analysis can suffer at any of its commands, i.e., *PRE*, *ACT*, and *CAS*. Each of the delay bounds on each of the commands is computed as per the JEDEC standard (see Table 2.1).

The term $L^{PRE}(N_{PRE})$ is the delay that can be caused by N_{PRE} interfering *PRE* commands to the *PRE* command of the request under analysis and can be computed using the following equation (From Lemma 2 of [Yun et al., 2015]).

$$L^{PRE}(N_{PRE}) = 2.N_{PRE} \tag{7.2}$$

The main insight behind Equation 7.2 is that there are no inter-bank timing constraints for the PRE command but a PRE command can be delayed due to the command bus contention caused by other commands.

Similarly, $L^{ACT}(N_{ACT})$ is the delay that can be caused by N_{ACT} interfering *ACT* commands to the *ACT* command of the request under analysis and can be computed using the following equation (From Equation 9 of [Hassan and Pellizzoni, 2018]).

$$L^{ACT}(N_{ACT}) = 2.N_{rq} + \max\left(N_{ACT}.tRRD, \left\lceil \frac{N_{ACT}+1}{4}.tFAW \right\rceil\right) \tag{7.3}$$

The main insight behind Equation 7.3 is that there is a tRRD inter-bank timing constraint between ACT to ACT commands. Furthermore, tFAW inter-bank timing constraint represents that no more than 4 ACT can be issued on different banks of the same rank. The term $2.N_{rq}$ represents the command bus contention that can be suffered by the ACT command from CAS and PRE commands.

Finally, $L^{CAS}(N_{CAS})$ is the delay that can be caused by N_{CAS} interfering *CAS* commands to the *CAS* command of the request under analysis and can be computed using the following equation

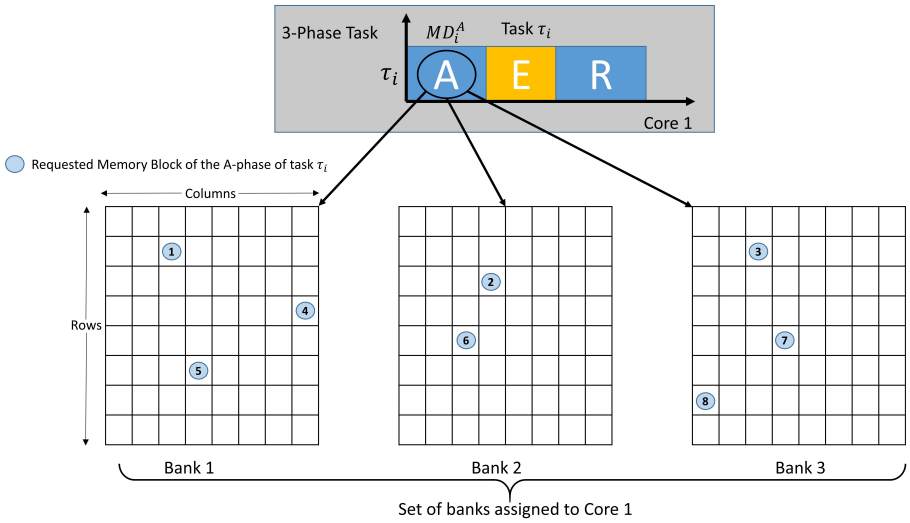


Figure 7.2: Example scenario for random mapping

(From Equations 10 and 4 of [Hassan and Pellizzoni, 2018]).

$$L^{CAS}(N_{CAS}) = (N_{CAS}+1).tCCD + 2.N_{rq} \tag{7.4}$$

Equation 7.4 is also derived by considering the CAS to CAS inter-bank timing constraints and command bus contention.

The detailed formulation of these equations can be found in [Yun et al., 2015, Hassan and Pellizzoni, 2018].

In the above equations, it was assumed that each request is a row-miss, thus, it will issue the sequence of commands, i.e., *PRE*, *ACT*, *CAS*. However, if a memory request results in a *row hit*, i.e., it only issues a *CAS* command. Such a memory request can only suffer inter-bank contention at its *CAS* command which is bounded by $L^{CAS}(N_{CAS})$. This observation has also been reflected in the state-of-the-art, (from Observation 1 in Section 4.1 of [Hassan and Pellizzoni, 2020]) "*Consider two requests targeting different bank. If both requests are close, then the first one can cause PRE, ACT and CAS delay to the second one; otherwise, it can only cause CAS delay.*"

7.3 Proposed Memory Contention Analysis for 3-phase tasks

In the proposed memory contention analysis, we consider different memory address mapping schemes that determine how the memory requests of an A-phase of tasks are mapped to the DRAM banks. Therefore, we first discuss the memory address mappings considered in this work in the following section.

7.3.1 Memory Address Mapping

In this work, we consider two different memory address mappings as described in the following subsections.

7.3.1.1 Random Mapping

In this mapping, we assume that all the memory blocks that can be requested by an A-phase of a task can be mapped to any row of any of the banks assigned to the core on which the task executes. Figure 7.2 presents an example scenario for this mapping.

We assume that the task τ_i executes on core 1 and the set of DRAM banks assigned to core 1 is bank 1, bank 2, and bank 3. Furthermore, we assume that MD_i^A, i.e., the maximum number of the memory requests of the A-phase, is equal to 8*, marked as* 1 *to* 8*. As shown in Figure 7.2, when using random mapping, all requests of the A-phase of task τ_i can be mapped to any column of any row of any of the assigned banks.*

7.3.1.2 Bank Level Contiguous Mapping

In this mapping, we assume that all the memory blocks that can be requested by an A-phase are mapped to a *single bank*. Furthermore, we assume that within the same bank, *contiguous mapping* is used which means that subsequent memory requests of the A-phase are mapped to the subsequent columns of the same row. When a memory request is mapped to the last column of a row, the subsequent memory requests are mapped to the beginning of the next row of the same bank. Contiguous address mapping is commonly used to improve the overall performance since mapping memory requests to the same row provides a better row-buffer locality. For example, some existing works have considered interleaved contiguous mapping [Kim et al., 2014, Yun et al., 2015], i.e., memory requests are spread over multiple banks and are mapped in a contiguous manner to each bank. The only variation in the bank-level contiguous mapping considered in this work is that we assume that all memory requests of an A-phase are mapped to a single bank. This can be achieved through the means of software as modern computing platforms allow programmable DRAM address mapping meaning that specific DRAM mapping can be configured on the basis of application requirement [Aldworth and Croxford, 2008, AMD, 2023].[3] Figure 7.3 presents an example scenario for this mapping.

As shown in Figure 7.3, the task τ_i executes on core 1 and the set of DRAM banks assigned to core 1 are bank 1, bank 2, and bank 3. Furthermore, we assume that MD_i^A, i.e., the maximum number of the memory requests of the A-phase, is equal to 8*, marked as* 1 *to* 8*. When using bank-level contiguous mapping, we know that all memory requests will access the same bank, e.g., bank 2 is considered in the example. Furthermore, we know that the subsequent memory requests are mapped to the subsequent columns. This is reflected in Figure 7.3 in which the first memory request in MD_i^A is mapped to the last column of a row while the rest of the memory requests are mapped to the columns of the next row.*[4]

[3] For systems that cannot comply with this mapping, random mapping can be considered as it represents the worst-case mapping scenario.

[4] For the sake of simplicity, Figure 7.3 shows that one memory request corresponds to one column but as defined in Rule **R7.**, we assume that each memory request corresponds to *BL* consecutive columns of the same row.

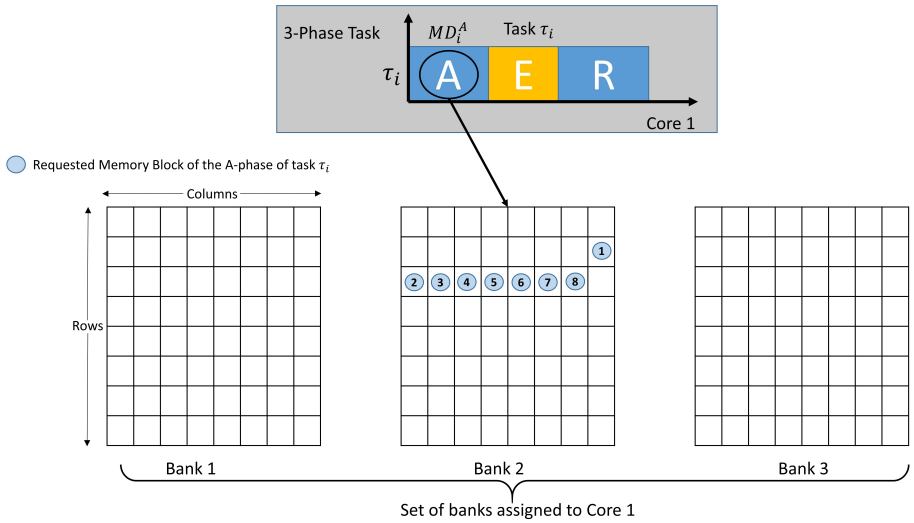


Figure 7.3: Example scenario for bank level contiguous mapping

7.3.2 Memory Contention Analysis for Random Mapping

In this section, we will discuss the proposed memory contention analysis when random memory mapping is considered. We start by computing the maximum memory contention that can be suffered by *read requests* of the A-phase of task τ_i due to *read requests* of tasks running on all *remote cores*. As banks are partitioned among cores (see Rule **R2.**), the A-phase of task τ_i can only suffer *inter-bank contention*.

Lemma 7.1. The maximum number of read memory requests of all tasks running on all remote cores that can interfere with read memory requests of the A-phase of one job of task τ_i is upper bounded by N_i^{read}, where

$$N_i^{read} = MD_i^A \times (m-1) \tag{7.5}$$

Proof. From Rule **R4.**, we know that each core uses an in-order pipeline which means each core issues one read request at a time. This implies that there can be most $m-1$ requests arrived at the memory controller before the arrival of one read request of the A-phase of task τ_i. Furthermore, from Rule **R3.**, we know that the inter-bank scheduling policy is RR which serves one request per bank and does not reorder requests. Consequently, in the worst case, a single read request issued during the A-phase of task τ_i can suffer inter-bank contention from at most $m-1$ read requests of tasks executing on all remote cores[5]. Extending this to all memory requests that can be issued by the A-phase of task τ_i, $MD_i^A \times (m-1)$ upper bounds the maximum number of read memory requests of all tasks running on all remote cores that can interfere with read memory requests of the A-phase of one job of task τ_i. The Lemma follows. □

Having bounded the number of interfering read requests, we bound the *maximum contention* that can be caused by those interfering requests to read requests of the A-phase of one job of task τ_i.

[5]Although the memory requests of the A-phase can be spread over different banks in random mapping, due to the in-order pipeline, there can be at most one pending request per core. Thus, at most one bank is active per remote core so there can be at most $m-1$ interfering read requests.

Lemma 7.2. The maximum memory contention that can be suffered by read requests of the A-phase of one job of task τ_i due to read requests of tasks running on all remote cores is upper bounded by MC_i^{read}, where

$$MC_i^{read} = MD_i^A \times \max_{N_{PRE}+N_{ACT}+N_{CAS}=m-1} \left(L^{PRE}(N_{PRE}) + L^{ACT}(N_{ACT}) + L^{CAS}(N_{CAS})\right) \tag{7.6}$$

Proof. From Lemma 7.1, we know that each read request of the task under analysis can be delayed by at most $m-1$ read requests issued by the $m-1$ remote cores. Assuming that memory requests are mapped to any columns/rows of the set of banks assigned to that core, in the worst case, each memory request can be a row-miss. As a consequence, all memory requests issue *PRE*, *ACT*, and *CAS* commands in sequence. From Equation 7.1, i.e., Theorem 1 of [Yun et al., 2015], we know that $\max_{N_{PRE}+N_{ACT}+N_{CAS}=m-1} \left(L^{PRE}(N_{PRE}) + L^{ACT}(N_{ACT}) + L^{CAS}(N_{CAS})\right)$ bounds the maximum memory contention that can be generated by $m-1$ read requests. Hence, every read request of τ_i suffers at most $\max_{N_{PRE}+N_{ACT}+N_{CAS}=m-1} \left(L^{PRE}(N_{PRE}) + L^{ACT}(N_{ACT}) + L^{CAS}(N_{CAS})\right)$ time units of interference. The MD_i^A read requests of τ_i's A-phase will thus suffer at most $MD_i^A \times \max_{N_{PRE}+N_{ACT}+N_{CAS}=m-1} \left(L^{PRE}(N_{PRE}) + L^{ACT}(N_{ACT}) + L^{CAS}(N_{CAS})\right)$ time units of interference. The Lemma follows. □

Having bounded the contention caused by read requests, the next step is to compute the maximum contention that can be caused by write requests to read requests of the A-phase of task τ_i. We start by briefly discussing how such a bound is derived in [Casini et al., 2020] and identify sources of pessimism. We then propose a new bound in Lemmas 7.3 and 7.4.

From Lemma 3 of [Casini et al., 2020] *The overall interference suffered by read requests of the A-phase of task τ_i due to write requests in any time interval of length t is bounded by*

$$MC_i^{wr}(t) = L_{WB}(\min(NR(t) \times N_{wb}, NW(t) + Q_{write})) \tag{7.7}$$

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

In Equation 7.7, $NR(t) \times N_{wb}$ captures that each read request of the task τ_i and each interfering read request from all remote cores suffer contention from at most one batch of write requests. This is a pessimistic bound since it assumes that every read request will suffer from one batch of write requests without analyzing the maximum number of batches that can be triggered during the execution of the A-phase of τ_i. Similarly, in Equation 7.7, the term $NW(t) + Q_{write}$ captures that only write requests of jobs of tasks released on remote cores during the interval of length t plus all previously enqueued write requests in the write buffer will cause memory contention. This again is a pessimistic bound because in the 3-phase task model, a core can only issue an R-phase, and thus enqueue write requests in the write buffer, after the completion of an A-phase. In such a

case, the actual number of write requests issued by a remote core depends on the number of read requests served on that remote core and not necessarily on all jobs released on that core during the interval of length t. To accurately quantify the memory contention that can be caused by write requests, we need to first determine the maximum number of write batches that can be served by the memory controller during the execution of the A-phase of τ_i.

Lemma 7.3. The maximum number of batches of write memory requests that can be served by the memory controller during the execution of the A-phase of τ_i is upper bounded by N_i^{wb}, where

$$N_i^{wb} = 1 + \left\lceil \frac{\sum_{r=1, r\neq l}^{m} \max_{\tau_u \in \Gamma_r'} \{MD_u^R\} + N_i^{read} - (W_{th} - (Q_{write} - N_{wb}))}{N_{wb}} \right\rceil \tag{7.8}$$

Proof. When the first read request of the A-phase of task τ_i arrives at the memory controller, in the worst case, the number of write requests inserted in the write buffer is equal to the length of the write buffer Q_{write}. This will trigger one batch of write requests as accounted by "+1" in Equation 7.8. After the first write batch, the maximum number of write requests still left in the write buffer is equal to $Q_{write} - N_{wb}$. Now there can be a scenario in which a remote core just completed an E-phase and starts executing an R-phase. Considering this, we need to account for write requests that can be issued by one R-phase on that remote core. In the worst case, the remote core executes the R-phase that issues the largest number of write requests among the R-phases of all tasks running on that remote core, i.e., $\max_{\tau_u \in \Gamma_r'} \{MD_u^R\}$. Extending this to all remote cores, $\sum_{r=1, r\neq l}^{m} \max_{\tau_u \in \Gamma_r'} \{MD_u^R\}$ upper bounds the number of write requests that can be issued by the R-phases of tasks that already completed their A-phases prior to the arrival of the A-phase of task τ_i. Note that to produce another R-phase on the same remote core, the core first needs to execute an A-phase.

From Lemma 7.1, we know that N_i^{read} bounds the maximum number of interfering read requests. Since the length of the R-phases is assumed to be less than or equal to their A-phases (see Rule **R6.**), in the worst case, there can at most N_i^{read} number of write requests that can be issued by all remote cores. We do not need to account for write requests issued on the local core because 1) task τ_i will only issue an R-phase after the completion of its A-phase; and 2) the R-phase of any other previously executed task on the local core must have already inserted all its write requests in the write buffer before the start of τ_i. Therefore, we have at most $Q_{write} - N_{wb} + \sum_{r=1, r\neq l}^{m} \max_{\tau_u \in \Gamma_r'} \{MD_u^R\} + N_i^{read}$ write requests enqueued in the write buffer after the first write batch. Considering all these write requests to derive the maximum number of write batches can be pessimistic because according to Rule **R5.**, to trigger a write batch, the minimum number of write requests enqueued in the write buffer should exceed W_{th}. Furthermore, according to Rule **R5.**, $W_{th} > Q_{write} - N_{wb}$ which means after serving a write batch, the number of writes left in the write buffer is less than W_{th}. In this case, after accounting for the first write batch, we should only consider the number of batches that can be triggered due to $\sum_{r=1, r\neq l}^{m} \max_{\tau_u \in \Gamma_r'} \{MD_u^R\} + N_i^{read}$

write requests which is upper bounded by $\left\lceil \frac{\sum_{r=1,r\neq l}^{m} \max_{\tau_u \in \Gamma_r'} \{MD_u^R\} + N_i^{read}}{N_{wb}} \right\rceil$. This is also slightly pessimistic because after serving the first write batch, the number of writes left in the write buffer is $Q_{write} - N_{wb}$ which is less than W_{th}. Consequently, we subtract $W_{th} - (Q_{write} - N_{wb})$ write requests because after serving the first write batch, another write batch can only be triggered after inserting $W_{th} - (Q_{write} - N_{wb})$ number of write requests in the write buffer. The Lemma follows. □

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

$$N_i^{write} = N_i^{wb} \times N_{wb} \tag{7.9}$$

Lemma 7.4. The maximum memory contention that can be suffered by the A-phase of task τ_i due to write requests is upper bounded by MC_i^{write}, where

$$MC_i^{write} = L_{WB}(N_i^{write}) \tag{7.10}$$

Proof. From Equation 2 of [Hassan and Pellizzoni, 2018] (also reused in Lemma 3 of [Casini et al., 2020]) $L_{WB}(N)$ bounds the maximum delay that can be caused by N write requests served in batches such that each write request results in a row miss and it can access any bank in the system. Furthermore, from Equation 7.9, we know that there can be at most N_i^{write} write requests that can interfere with the read requests of the A-phase of task τ_i. Hence, $L_{WB}(N_i^{write})$ upper bounds the maximum memory contention that can be suffered by the A-phase of task τ_i due to write requests. The Lemma follow. □

Lemma 7.5. The total memory contention that can be suffered by the A-phase of task τ_i is upper bounded by MC_i^{total}, where

$$MC_i^{total} = MC_i^{read} + MC_i^{write} \tag{7.11}$$

Proof. We know that Equation 7.6 upper bounds the memory contention that can be caused by interfering read requests. Similarly, Equation 7.10 upper bounds the memory contention that can be caused by write requests. Consequently, Equation 7.11 upper bounds the *total memory contention* that can be suffered by the A-phase of task τ_i by taking the sum of Equations 7.6 and 7.10. □

As proven in [Casini et al., 2020], we do not need to account for memory contention that can be suffered by the R-phase of task τ_i. This is mainly because we assume that the length of the write buffer is large enough such that all write requests of all cores can be inserted in it [Yun et al., 2015, Hassan and Pellizzoni, 2018, Casini et al., 2020]. This ensures that the R-phase of a task does not cause any additional delay to the A-phase of the subsequent task on the same core. Consequently, we only need to account for the WCET in isolation of the R-phase.

7.3.3 Memory Contention Analysis for Bank Level Contiguous Mapping

In the bank-level contiguous mapping, all memory requests of an A-phase are mapped to a single bank (e.g., see Figure 7.3). Furthermore, any two successive memory requests of an A-phase are mapped to the successive columns of the same row of the same bank. As shown in Figure 7.3, upon a row switch, i.e., accessing the last column of a row, subsequent memory requests are mapped to subsequent columns of the next row in the same bank.

We will now start bounding the memory contention that can be suffered by read requests of the A-phase of task τ_i due to the interfering read requests.

For random mapping, we know that Lemma 7.1 upper bounds the maximum number of read requests of remote cores that can interfere with the read requests of the A-phase of task τ_i. Since the set of all random mappings also includes contiguous mappings, we can also use the bound provided by Lemma 7.1 to compute the maximum number of interfering read requests from all remote cores.

However, the computation of the maximum memory contention that can be caused by all interfering requests to the A-phase of task τ_i provided by Lemma 7.2 can be pessimistic for bank-level contiguous mapping. This is mainly because, in the bank-level contiguous mapping, many of the successive memory requests may result in row hits since they are mapped on the same row of the same bank. We know that a row hit memory request can suffer inter-bank contention at its *CAS* command. Therefore, we will now compute the *minimum* number of *row hit* read requests among all read requests of the A-phase of task τ_i. For this, we first compute the maximum number of rows that can be accessed by the A-phase of task τ_i as follows.

Lemma 7.6. The maximum number of rows of a bank that can be accessed during the A-phase of task τ_i are upper bounded by $N^{row}_{i,max}$, where

$$N^{row}_{i,max} = 1 + \left\lceil \frac{(MD^A_i - 1) \times BL}{Row_{size}} \right\rceil \tag{7.12}$$

where Row_{size} denotes the length of a single row of the bank.

Proof. In the bank-level contiguous mapping, we know that all read requests of the A-phase are mapped to a single bank, and within that bank, contiguous mapping is used, i.e., subsequent memory requests are mapped to successive columns of the same row, and the request to the last column of a row is followed by a request to the first column of the next row. From Rule **R7.**, we know that memory accesses are performed in the burst mode, and at least one memory request is served in a single memory burst of length *BL*. In the worst case, the first read request is mapped to the last *BL* columns of a row. In such a case, one row is accessed by a single read request of the A-phase of task τ_i. It is captured with the "+1" in Equation 7.12. Due to contiguous mapping, we know that all the subsequent memory requests are mapped to successive row(s) starting from its first column such that two successive memory requests can be mapped to at most successive $2 \times BL$ columns of the same row. As we have already accounted for the row accessed by the first

request, there are $(MD_i^A - 1)$ requests of length BL to satisfy. Those can be mapped on at most $\left\lceil \frac{(MD_i^A - 1) \times BL}{Row_{size}} \right\rceil$ different rows. The Lemma follows. □

Having bounded the maximum number of rows of a bank that can be accessed by the A-phase of task τ_i, we can bound the minimum number of memory requests that will result in a row hit in *isolation* as follows.

Lemma 7.7. The minimum number of read requests that will result in a row hit among all read requests of the A-phase of task τ_i when task τ_i *execute in isolation* is upper bounded by $R^{hit}_{i,min,iso}$, where

$$R^{hit}_{i,min,iso} = MD_i^A - N^{row}_{i,max} \tag{7.13}$$

Proof. From Lemma 7.6, we know that $N^{row}_{i,max}$ upper bounds the maximum number of rows that can be accessed by the A-phase of task τ_i. From Rule **R4.**, we know that cores use in-order-pipeline so the memory requests will be issued by the core in sequence. This implies that among all read requests of the A-phase of task τ_i, at most $N^{row}_{i,max}$ read requests may result in row miss as each such request may first need to close the activated row and then activate the required row before performing the read operation. Since we assume task τ_i *execute in isolation*, all remaining read requests will result in row hits. Hence, Equation 7.13 bounds the minimum number of read requests that will result in a row hit among all read requests of the A-phase of task τ_i when task τ_i execute in isolation. The Lemma follows. □

Thanks to the bank partitioning among the A-phases of tasks of all the cores, we know that A-phases of remote cores cannot access the bank assigned to the local core, thus, cannot affect the status of the row buffer of banks assigned to the local core. However, R-phases of all tasks in the system can access any bank. Furthermore, we know that write requests are served using the write batching mechanism. As a consequence, batches of write requests can affect the status of the row buffer of the bank that is being accessed by the A-phase of task τ_i. We explain this using the following example.

Assume that all read requests of the A-phase of task τ_i are mapped to a single row of a bank. In such a scenario, ideally, there should be at most one row miss and the remaining memory requests should result in row hits. However, when the system uses the write batching and the worst-case is derived by assuming that one batch of write requests can be triggered upon serving each read request (as assumed in [Casini et al., 2020]), we cannot guarantee the minimum number of row hits. This is mainly because each time a batch of write requests is triggered, some or all write requests can target the same bank but a different row than the A-phase of task τ_i. Consequently, despite mapping all read requests to the same row, all memory requests may result in row-miss.

This problem has also been highlighted by the state-of-the-art, see Section 3 of [Hassan and Pellizzoni, 2020].

Thanks to Lemma 7.3, we know the maximum number of write batches that can be served during the A-phase of task τ_i. Using the Lemmas 7.3 and 7.7, we will now compute the minimum number of row hits among all read requests of the A-phase of task τ_i.

Lemma 7.8. The minimum number of read requests that will result in a row hit among all read requests of the A-phase of task τ_i is upper bounded by $R^{hit}_{i,min}$, where

$$R^{hit}_{i,min} = \max(0, R^{hit}_{i,min,iso} - N^{wb}_i) \tag{7.14}$$

Proof. From Lemma 7.7, we know that $R^{hit}_{i,min,iso}$ bounds the minimum number of read requests that will result in a row hit among all read requests of the A-phase of task τ_i in isolation. Furthermore, from Lemma 7.3, we know that N^{wb}_i upper bounds the maximum number of write batches that can be served by the memory controller during the execution of the A-phase of task τ_i. In the worst case, when each batch of write requests is served, some write requests may target the same bank as the A-phase of task τ_i but a different row. As a consequence, a batch of writes can turn a row hit in isolation to a row miss. As we cannot precisely estimate the time on which each batch of write requests will be served, in the worst case each batch of write request turn a row hit request into a row miss. Consequently, $R^{hit}_{i,min,iso} - N^{wb}_i$ bounds the minimum number of read requests that will result in a row hit among all read requests of the A-phase of task τ_i. In any case, the minimum number of row hits cannot be less than 0. The Lemma follows. □

Having bounded the minimum number of read requests that will result in a row hit among all read requests of the A-phase of task τ_i, we can compute an upper bound on the *maximum number* of read requests that will result in a *row miss* among all read requests of the A-phase of task τ_i as follows.

$$R^{miss}_{i,max} = MD^A_i - R^{hit}_{i,min} \tag{7.15}$$

Having bounded the minimum number of row hits and the maximum number of row miss requests, we can now compute the maximum *memory contention* that can be suffered by those requests.

Lemma 7.9. The maximum memory contention that can be suffered by all row miss read requests of the A-phase of task τ_i due to interfering read requests from all remote cores is upper bounded by $MC^{read}_{i,miss}$, where

$$MC^{read}_{i,miss} = R^{miss}_{i,max} \times \max_{N_{PRE}+N_{ACT}+N_{CAS}=m-1} \left(L^{PRE}(N_{PRE}) + L^{ACT}(N_{ACT}) + L^{CAS}(N_{CAS})\right) \tag{7.16}$$

Proof. We know that a row miss request issue sequence of commands, i.e., *PRE*, *ACT*, *CAS*. As we cannot precisely estimate the status of interfering requests, in the worst case, each interfering read request can also be a row miss. From Equation 7.1, we know that $\max_{N_{PRE}+N_{ACT}+N_{CAS}=m-1} \left(L^{PRE}(N_{PRE}) + L^{ACT}(N_{ACT}) + L^{CAS}(N_{CAS})\right)$ bounds the memory contention that can be suffered by a *single row miss request*. Furthermore, from Lemma 7.1, we know that each read request of the A-phase of task τ_i can suffer interference from one read request per remote core. As $R^{miss}_{i,max}$ bounds the maximum number of row miss requests of the A-phase of task τ_i, the maximum memory contention that can be suffered by $R^{miss}_{i,max}$ requests is upper bounded by Equation 7.16. The Lemma follows. □

Lemma 7.10. The maximum memory contention that can be suffered by all row hit read requests of the A-phase of task τ_i due to interfering read requests from all remote cores is upper bounded by $MC^{read}_{i,hit}$, where

$$MC^{read}_{i,hit} = R^{hit}_{i,min} \times L^{CAS}(N_{CAS}) \tag{7.17}$$

where $N_{CAS} = m - 1$.

Proof. From Lemma 7.1, we know that each read request of the A-phase of task τ_i can suffer interference from one read request per remote core. Furthermore, we know that a row hit request only issues the *CAS* command and can only suffer inter-bank contention from interfering CAS command disregarding if the interfering read request is a row hit or row miss. From Equation 7.4, we know that $L^{CAS}(N_{CAS})$ upper bounds the maximum contention that can be suffered by a *single row hit read request* from N_{CAS} interfering read requests, where $N_{CAS} = m - 1$. Extending this to all row hit read requests of the A-phase of task τ_i, i.e., $R^{hit}_{i,min}$, Equation 7.17 upper bounds the maximum memory contention that can be suffered by all row hit read requests of the A-phase of task τ_i due to interfering read requests from all remote cores. The Lemma follows. □

Having bounded the contention that can be suffered by read requests of the A-phase of task τ_i from interfering read requests from all remote cores, we can now compute the memory contention that can be suffered by read requests of the A-phase of task τ_i from write requests. As the write requests of the R-phases of all tasks in the system can access any row of any bank and write requests are served using the write batching mechanism, we can compute the contention from write requests using the same steps described in Section 7.3.2. Specifically, the maximum number of batches of write requests can be computed using Lemma 7.3, the maximum number of interfering write requests using Equation 7.9, and the maximum memory contention from write requests using Lemma 7.4. Finally, we can compute the *total memory contention* that can be suffered by the A-phase of task τ_i as follows.

Lemma 7.11. The total memory contention that can be suffered by the A-phase of task τ_i is upper bounded by $\hat{MC}^{total}_i$, where

$$\hat{MC}^{total}_i = MC^{read}_{i,miss} + MC^{read}_{i,hit} + MC^{write}_i \tag{7.18}$$

Proof. The proof directly follows from Lemma 7.5 except that the computation of $MC^{read}_{i,miss}$ and $MC^{read}_{i,hit}$ is given by Lemma 7.9 and 7.10, respectively. □

Having bounded the total memory contention that can be suffered by the task τ_i under both the memory address mappings, we will show how the bounds on the memory contention can be integrated to derive the WCRT of tasks in the next section.

7.4 WCRT Analysis

In this section, we will derive a WCRT analysis that considers the memory contention suffered by tasks. This section is divided into two subsections in which the WCRT analysis for the random mapping is presented in Section 7.4.1 and the WCRT analysis for the bank level contiguous mapping is presented in Section 7.4.2.

7.4.1 WCRT Analysis for Random Mapping

In the random mapping, the worst-case can be derived by considering that each memory request is a row miss. Building on this, the WCET of task τ_i in *isolation* can be computed using the following equation.

$$C_i = (MD_i^A + MD_i^R) \times t^{miss} + C_i^E \tag{7.19}$$

where t^{miss} is the maximum time to serve a row-miss request and can be computed by considering all the intra-bank timing constraints of the JEDEC standard.

We can now integrate the total memory contention MC_i^{total} (computed using Equation 7.11) that can be suffered by the task τ_i into its WCET in isolation. Similar to [Casini et al., 2020], integrating memory contention into the WCET of tasks will allow computing an *inflated WCET*.

Let $\hat{C}_i$ denote the inflated WCET of task τ_i under random mapping which also takes into account the memory contention that can be suffered by task τ_i, where $\hat{C}_i$ is given by:

$$\hat{C}_i = C_i + MC_i^{total} \tag{7.20}$$

where C_i is given by Equation 7.19 and MC_i^{total} is given by Equation 7.11.

After inflating the WCET of all tasks in the taskset, we can compute the WCRT analysis of tasks. The WCRT analysis can be computed similarly to that of single-core processors because we have already accounted for the memory contention that can be caused by all remote cores into the inflated WCET of tasks.[6]

It has been proven that for a task τ_i scheduled using fixed-priority non-preemptive scheduling, the WCRT is observed during the longest level-i busy window [Bril et al., 2007] (see definition 2.1.3.1). Therefore, we can compute the length of the level-i busy window as follows.

Lemma 7.12. The length of the longest level-i busy window for a task τ_i executing on a core π_l is upper bounded by $W_{i,l}$, where $W_{i,l}$ is given by the first positive solution to the fixed-point iteration of the following equation

$$W_{i,l} = \sum_{\tau_h \in hep_{i,l}} \left\lceil \frac{W_{i,l}}{T_h} \right\rceil \times \hat{C}_h + \max_{\tau_j \in lp_{i,l}} \{\hat{C}_j\} \tag{7.21}$$

[6]Note that any contention on the memory controller results in busy waiting on the core. Therefore, there are no anomalies caused by any kind of self-suspension.

Proof. Due to fixed-priority scheduling, all tasks in $hep_{i,l}$ can execute during the level-i busy window. Furthermore, the memory contention can be suffered by each job of all tasks in $hep_{i,l}$ that execute within the level-i busy window $W_{i,l}$. Therefore, we first need to inflate the WCET of each task in $hep_{i,l}$ using Equation 7.20. Then we can derive the worst-case by assuming that each job of task $\tau_h \in hep_{i,l}$ that executes within the level-i busy window $W_{i,l}$ will require $\hat{C}_h$ time units to execute. Therefore, the maximum contribution of a task $\tau_h \in hep_{i,l}$ within the level-i busy window is upper bounded by $\left\lceil \frac{W_{i,l}}{T_h} \right\rceil \times \hat{C}_h$. Extending this to all tasks in $hep_{i,l}$, the term $\sum_{\tau_h \in hep_{i,l}} \left\lceil \frac{W_{i,l}}{T_h} \right\rceil \times \hat{C}_h$ upper bounds the maximum contribution of all tasks in $hep_{i,l}$ within the level-i busy window of length $W_{i,l}$. Furthermore, due to non-preemptive scheduling, we also need to account for the potential execution of one job of a lower priority task at the start of the level-i busy window which can also suffer memory contention. The maximum blocking that can be caused by a lower priority task within the level-i busy window is upper bounded by $\max_{\tau_j \in lp_{i,l}} \{\hat{C}_j\}$. The Lemma follows. □

Note that $W_{i,l}$ appears on both sides of Equation 7.21 so it needs to be solved iteratively using $W_{i,l} = \sum_{\tau_h \in hep_{i,l}} \hat{C}_h + \max_{\tau_j \in lp_{i,l}} \{\hat{C}_j\}$ as the starting point.

Having bounded the level-i busy window, we can compute the latest *finish time* of task τ_i. Let $\tau_{i,k}$ denotes the k^{th} job of task τ_i. The latest finish time of $\tau_{i,k}$ can be computed as follows.

Lemma 7.13. The latest finish time of $\tau_{i,k}$ is denoted by $f_{i,k}$, where $f_{i,k}$ is given by the first positive solution to the following fixed-point iteration:

$$f_{i,k} = k \times \hat{C}_i + \sum_{\tau_h \in hep_{i,l} \setminus \tau_i} \left\lceil \frac{f_{i,k} - C_i}{T_h} \right\rceil \times \hat{C}_h + \max_{\tau_j \in lp_{i,l}} \{\hat{C}_j\} \tag{7.22}$$

Proof. While computing the latest finish time of k^{th} job of task τ_i, we need to consider the contributions of the previous jobs of task τ_i, i.e., $k-1$, and the WCET of k^{th} job of task τ_i itself. This is upper bounded using $k \times \hat{C}_i$. Due to fixed priority non-preemptive scheduling, $\tau_{i,k}$ can only suffer interference from higher priority tasks until the start of its A-phase. As a consequence, the maximum interference that can be caused by a task τ_h to task τ_i during $f_{i,k}$ is upper bounded by $\left\lceil \frac{f_{i,k} - C_i}{T_h} \right\rceil \times \hat{C}_h$. Extending this to all tasks in $hep_{i,l}$ except task τ_i (remember we have already accounted for the contribution of task τ_i), the term $\sum_{\tau_h \in hep_{i,l} \setminus \tau_i} \left\lceil \frac{f_{i,k} - C_i}{T_h} \right\rceil \times \hat{C}_h$ upper bounds the maximum contribution of all tasks in $hep_{i,l}$ except task τ_i during $f_{i,k}$. Finally, as proven in Lemma 7.12, the maximum blocking that can be caused by a lower priority is bounded by $\max_{\tau_j \in lp_{i,l}} \{\hat{C}_j\}$. □

Note that $f_{i,k}$ appears on both sides of Equation 7.22 so it needs to be solved iteratively using $f_{i,k} = \sum_{\tau_h \in hep_{i,l}} \hat{C}_h + \max_{\tau_j \in lp_{i,l}} \{\hat{C}_j\}$ as the starting point.

Once Equation 7.22 converges, we can now compute the *response time* of $\tau_{i,k}$ as follows.

The response time of $\tau_{i,k}$ is denoted by $R_{i,k}$ and can be computed by subtracting the minimum inter-arrival time of previously executed jobs of task τ_i from the latest finish time $f_{i,k}$. Hence,

$$R_{i,k} = f_{i,k} - (k-1) \times T_i \tag{7.23}$$

Finally, the WCRT of task τ_i is then given by the largest response time of any job of τ_i that executes during the level-i busy window, i.e.,

$$R_i^{max} = \max_{k \in [1,K_i]} \{R_{i,k}\} \tag{7.24}$$

where the computation of $K_i = \left\lceil \frac{W_{i,l}}{T_i} \right\rceil$.

A taskset Γ is said to be schedulable only if the WCRT R_i^{max} of each task τ_i in the taskset is less than or equal to its relative deadline D_i, the utilization of each core is less than or equal to the core's capacity, i.e., 1, and the total memory utilization of the taskset is less than or equal to 1, i.e., $\sum_{\tau_i \in \Gamma} \frac{(MD_i^A + MD_i^R) \times t^{miss}}{T_i} \leq 1$.

7.4.2 WCRT Analysis for Bank Level Contiguous Mapping

In the memory contention analysis for the bank-level contigous mapping, we know that some memory requests may result in row hits. Building on this, the WCET of task τ_i in *isolation* can be computed using the following equation.

$$C_i = R_{i,min}^{hit} \times t^{hit} + (R_{i,max}^{miss} + MD_i^R) \times t^{miss} + C_i^E \tag{7.25}$$

where t^{hit} and (resp. t^{miss}) is the maximum time to serve a row-hit (resp. row-miss) request and can be computed by considering all the intra-bank timing constraints of the JEDEC standard; and $R_{i,min}^{hit}$ (resp. $R_{i,max}^{miss}$) is given by Equation 7.14 (resp. Equation 7.15).

The next step is to inflate the WCET by integrating the maximum memory contention that can be suffered by tasks. We show the computation of inflated WCET as follows.

Let $\hat{C}_i$ denote the inflated WCET of task τ_i under bank level contiguous mapping which also takes into account the memory contention that can be suffered by task τ_i, where $\hat{C}_i$ is given by:

$$\hat{C}_i = C_i + \hat{MC}_i^{total} \tag{7.26}$$

where C_i is given by Equation 7.25 and $\hat{MC}_i^{total}$ is given by Equation 7.18.

Finally, we can compute the length of the level-i busy window using Lemma 7.12, the latest finish time using Lemma 7.13, the response time using Equation 7.23 and WCRT using Equation 7.24 by first inflating the WCET of each task τ_i using Equation 7.26.

7.5 Experimental Evaluation

In this section, we will evaluate the performance of the proposed memory contention analysis in comparison to the state-of-the-art. As we improve on the work of [Casini et al., 2020], we need to directly compare the proposed analysis with [Casini et al., 2020]. However, comparing the proposed analysis with the exact analysis of [Casini et al., 2020] can be biased. This is mainly because the analysis of [Casini et al., 2020] does not consider bank partitioning. This implies that

tasks suffer both intra-bank and inter-bank contention. As a consequence, the analysis of [Casini et al., 2020] may tend to perform poorly in comparison to the proposed analysis not necessarily due to the pessimism in the existing analysis but due to fundamental differences in the assumptions. Therefore, we assume that banks are partitioned when computing memory contention bounds for the analysis in [Casini et al., 2020]. By doing so, the contention caused by read requests can then be modeled exactly as in the proposed analysis, i.e., each read request can be interfered by at most one read request per remote core (From Property 1 of [Casini et al., 2020]). Similarly to the proposed work, the analysis in [Casini et al., 2020] also leverages Theorem 1 of [Yun et al., 2015] to compute the inter-bank contention that can be caused by read requests. For the existing work, we compute the maximum contention that can be caused by all interfering write requests using Lemma 3 of [Casini et al., 2020]. For the proposed work, we compute the maximum number of interfering write requests using Equation 7.9 and the maximum contention caused by all those interfering write requests using Equation 7.10. Note that similarly to the proposed work, the analysis in [Casini et al., 2020] also leverages Equation 2 of [Hassan and Pellizzoni, 2018] to compute contention that can be caused by N write requests. Using the maximum contention from read and write requests, we compute the total memory contention suffered by tasks and then inflate their respective WCET by integrating the total memory contention as described in Section 7.4.

For the experimental evaluation, we performed two sets of experiments. A case study experiment performed using task parameters obtained from the Mälardalen benchmark suite [Gustafsson et al., 2010] is presented in Section 7.5.1. Experiments performed using synthetic tasksets are detailed in Section 7.5.2.

7.5.1 Case Study

For the case study experiments, we use task parameters given in Table 3.2, i.e., task parameters generated using the Mälardalen benchmark suite [Gustafsson et al., 2010]. In Table 3.2, PD_i represents the processor demand and MD_i represents the memory demand of task τ_i. We assume that the WCET of the E-phase C_i^E is equal to the processor demand PD_i, i.e., $C_i^E = PD_i$. Using the total memory access demand MD_i, the memory demand of the A-phase, denoted by A_i, was chosen randomly in the range [50%-90%] of MD_i, i.e., $A_i = rand(50\%, 90\%) \times MD_i$. Similarly, the memory demand of the R-phase was then chosen by $MD_i - A_i$. Assuming that every memory request is a row miss request, the maximum number of memory requests that can be issued during the A-phase (resp. R-phase) was then generated by $MD_i^A = \left\lceil \frac{A_i}{t^{miss}} \right\rceil$ (resp. $MD_i^R = \left\lceil \frac{MD_i - A_i}{t^{miss}} \right\rceil$). Finally, the WCET of τ_i in isolation was then computed using Equation 7.19 for random mapping and using Equation 7.25 for bank-level contiguous mapping.

By default, we consider a quad-core platform and a task set size of 32 tasks with 8 tasks randomly assigned to each core. To assign benchmark parameters to tasks, we randomly select a benchmark from Table 3.2 and assign C_i as discussed earlier. We then generate tasks' utilizations U_i using UUnifast discard [Emberson et al., 2010] algorithm. Having assigned the values of C_i and U_i, we generate the task period by using the equation $T_i = C_i/U_i$. The task priorities were then

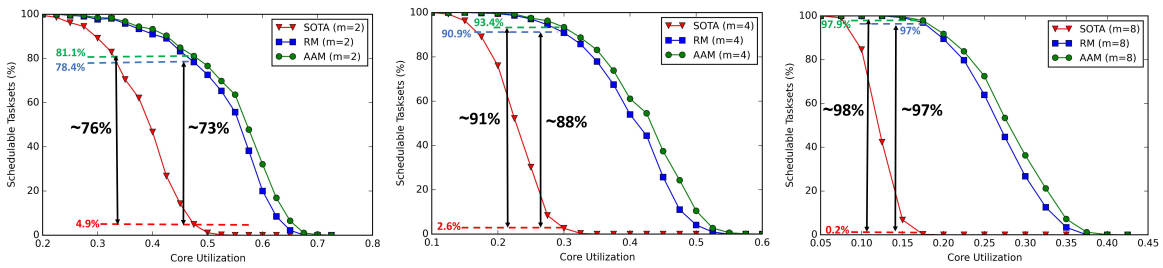
assigned using rate monotonic algorithm [Liu and Layland, 1973]. Task deadlines were equal to task periods, i.e., $D_i = T_i$.

For the memory controller, in all the experiments in this section and all experiments in Section 7.5.2, we consider the following parameters.

$Row_{size} = 1024$ as in the DDR3 SDRAM chip [JEDEC, 2008]; $BL = 8$ as assumed in [Kim et al., 2014]; $Q_{write} = 64$ as assumed in [Yun et al., 2015, Casini et al., 2020]; $W_{th} = 54$ as assumed in [Yun et al., 2015]; and $N_{wb} = 18$ as assumed in [Yun et al., 2015, Casini et al., 2020].

To compare the performance of the proposed analysis with the state-of-the-art, we performed two experiments in this case study by varying: 1) the core utilization (i.e., utilization of each core); 2) the number of cores in the system. We use taskset schedulability, i.e., the percentage of schedulable tasksets, as a metric to evaluate the performance of each approach. In this case study (and also for the experiments in Section 7.5.2), the proposed random mapping-aware analysis presented in Section 7.3.2 is marked "**RM**". Similarly, the proposed address-aware mapping analysis, i.e., bank-level contiguous address mapping, presented in Section 7.3.3 is marked as "**AAM**". Finally, the state-of-the-art analysis of [Casini et al., 2020] is marked as "**SOTA**". In each experiment, 1000 task sets were generated per point.

1. Core Utilization: In this experiment, we varied the core utilization of each core in the range of 0.05 to 1 in steps of 0.025 considering the default configuration, i.e., $m = 4$. The resulting taskset schedulability using all the approaches is plotted in Figure 7.4b. We can observe in Figure 7.4b that increasing the core utilization negatively impacts the taskset schedulability for all the approaches. This is mainly because an increase in core utilization also increases tasks utilizations which directly impacts the task period/deadline since $D_i = T_i = C_i/U_i$. Nonetheless, we can observe in Figure 7.4b that the proposed analyses performed significantly better than the existing analysis. For example, the proposed RM analysis was able to schedule around 88% more tasksets than the existing analysis at the core utilization value of 0.30. Similarly, the proposed AAM analysis was able to schedule around 91% more tasksets than the existing analysis at the core utilization value of 0.30. The gain of RM analysis over SOTA is mainly observed due to the tighter bound on the number of interfering write requests. The tighter bound on the number of interfering write requests tightly bounds the total memory contention that tasks can suffer. Consequently, a tighter bound on total memory contention tightly bounds the WCET and WCRT of tasks, resulting in better taskset schedulability. Similarly, the gain of AAM analysis over SOTA is mainly observed



(a) Number of cores $m = 2$ (b) Number of cores $m = 4$ (c) Number of cores $m = 8$

Figure 7.4: Varying the core utilization and number of cores

due to tightly bounding the contention from write requests. Additionally, it improves the bound on contention caused by read requests and WCET in isolation by computing the minimum number of row hits. This is also the reason that AAM analysis performs better than RM as shown in Figure 7.4b.

2. Number of Cores: In this experiment, we vary the number of cores along with the core utilization. The number of cores (m) varied from 2 to 8 along with core utilization. The taskset schedulability using all the approaches on different values of m is plotted in Figure 7.4. We observe that increasing (resp. decreasing) the number of cores negatively (resp. positively) impacted the taskset schedulability. This is mainly because increasing the number of cores also increases the number of remote cores, thus, the number of interfering memory requests. This in turn increases the memory contention suffered by tasks and decreases taskset schedulability. Nonetheless, the proposed analyses outperformed the existing analysis for all the considered values of m.

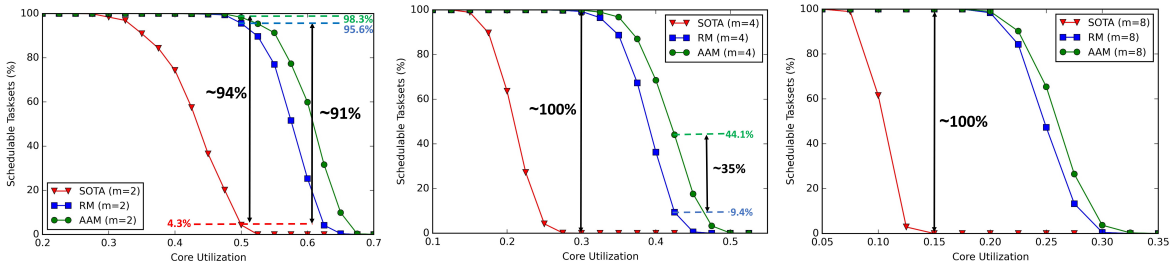
In Figure 7.4, we observe that the gains of proposed analyses over the existing analysis increased (resp. decreased) with the increase (resp. decrease) in the value of m. For example, the gain of RM over SOTA was around 73% for $m = 2$ whereas for $m = 8$ the gain of RM over SOTA increased to 97%. This is mainly because increasing the number of cores also increases the number of remote cores which in turn increases the interfering memory requests and total memory contention. This results in a decrease in the taskset schedulability. Since the proposed analyses provide a tighter bound on total memory contention, the gains of the proposed analyses over the state-of-the-art increase with the increase in the value of m.

For all the values of m, we observe that AAM analysis performs better than RM analysis. This is mainly observed because the proposed AAM analysis also bounds the minimum number of row hit requests. Bounding row hit requests allow tightly bounding memory contention and memory access times of requests. This in turn tightly bounds the inflated WCET and improves taskset schedulability.

7.5.2 Experiments using Synthetic Tasks

In this section, we will explain the experiments that were performed using synthetic task sets to compare the performance of the proposed approaches with the existing analysis.

For the default configuration, we model a quad-core platform with the taskset size of 32 tasks in which 8 tasks were randomly mapped to each core. Tasks utilization U_i was generated using the UUnifast-discard algorithm [Emberson et al., 2010]. Task periods T_i were randomly generated in the range of [100000-1000000] using log-uniform distribution. The WCET in isolation C_i was then assigned by $C_i = U_i \times T_i$. The total memory access demand (MD) of tasks was derived using C_i such that, $MD_i = rand(10\%, 30\%) \times C_i$. The memory demand of the A-phase A_i was chosen randomly in the range [50%-90%] of MD_i, i.e., $A_i = rand(50\%, 90\%) \times MD_i$. Similarly, the memory demand of the R-phase was then chosen by $MD_i - A_i$. Assuming that every memory request will be a row miss request, the maximum number of memory requests that can be generated during the A-phase (resp. R-phase) was then generated by $MD_i^A = \left\lceil \frac{A_i}{t^{miss}} \right\rceil$ (resp. $MD_i^R = \left\lceil \frac{MD_i - A_i}{t^{miss}} \right\rceil$). The value of the WCET was then updated such that the WCET of τ_i in isolation was then computed



(a) Number of cores $m = 2$ (b) Number of cores $m = 4$ (c) Number of cores $m = 8$

Figure 7.5: Varying the core utilization and number of cores

using Equation 7.19 for random mapping and using Equation 7.25 for bank-level contiguous mapping. Task priorities were assigned using rate monotonic algorithm [Liu and Layland, 1973]. Task deadlines were equal to task periods, i.e., $D_i = T_i$.

We perform various experiments to compare the performance of the proposed memory contention analysis with the existing memory contention analysis [Casini et al., 2020] by varying: 1) the core utilization; 2) the number of cores; 3) the memory access demand; and 4) the task periods. We use taskset schedulability, i.e., the percentage of schedulable tasksets, as a metric to evaluate the performance of each approach. To plot each point in every experiment, 1000 task sets were randomly generated.

1. Core Utilization: In this experiment, we varied each core utilization between 0.05 and 1 in steps of 0.025 for the default configuration, i.e., $m = 4$. We then plotted the percentage of tasksets that were deemed schedulable by all the considered approaches, i.e., AAM, RM, and SOTA, for each core utilization value in Figure 7.5b. We can see in Figure 7.5b that increasing the core utilization decreases taskset schedulability for all the approaches. This happens because the WCET C_i was generated using U_i and T_i, i.e., $C_i = U_i \times T_i$. This implies that an increase in core utilization increases tasks utilizations which further increases the WCET in isolation and memory demand of tasks. This further results in an increase in the interference and blocking that can be caused by tasks on the same core and the memory contention that can be caused by remote cores. Consequently, the taskset schedulability decreases with the increase in core utilization. Nonetheless, the proposed RM and AAM analysis outperformed the existing analysis by improving the taskset schedulability up to 100% points. As discussed earlier, this mainly happens due to the tighter bound on the memory contention proposed in this work.

Interestingly, we observe in Figure 7.5b that the overall schedulability using all the approaches is reduced in comparison to the benchmark experiments, i.e., see Figure 7.4b. This mainly happens because the value of MD_i is small in benchmark parameters which in turn produces a smaller number of memory requests of the A- and the R-phases. Consequently, the memory contention suffered by tasks is moderate and taskset schedulability for all approaches is better for benchmark experiments as shown in Figure 7.4b. On the contrary, in the synthetic experiments, we generated task periods in the range of [100000-1000000] and then generated C_i using the task periods and utilizations. As a consequence, it produces large values of C_i which also translates into larger MD_i and memory requests. Thus, tasks suffer more memory contention resulting in reduced taskset

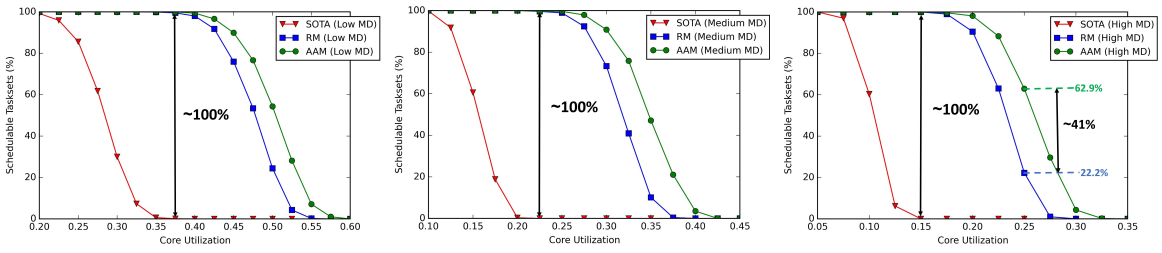
schedulability for all the approaches. Since the proposed analyses provided a tighter bound on the memory contention, the gain of the proposed analyses over the existing analysis increased in the synthetic experiments in comparison to the benchmark experiments.

Furthermore, as expected, AAM analysis outperformed RM analysis by bounding and integrating the number of row hit requests. This in turn reduces the total memory contention and improves taskset schedulability. Specifically, the proposed AAM analysis was able to schedule around 35% more tasksets than the RM analysis at the core utilization value of 0.425 as shown in Figure 7.5b.

2. Number of Cores: In this experiment, we vary the number of cores along with the core utilization. The number of cores (m) varied from 2 to 8 along with core utilization. The taskset schedulability using all the approaches on different values of m is plotted in Figure 7.5. We observe that increasing (resp. decreasing) the number of cores negatively (resp. positively) impacted the taskset schedulability. As discussed earlier, this is mainly because increasing the number of cores also increases the number of remote cores, thus, the number of interfering memory requests. This in turn increases the memory contention suffered by tasks and decreases taskset schedulability. Nonetheless, for all the values of m, the proposed AAM and RM analyses outperformed the existing analysis.

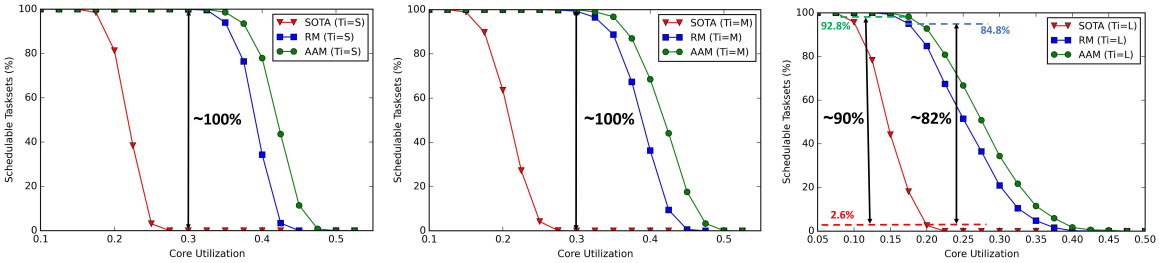
Interestingly, we observe that the gain of the AAM analysis over RM analysis reduced for $m = 8$. Intuitively, this happens because for $m = 8$, there are a significant number of remote cores which produce a significant number of memory requests. It means that a large number of interfering write requests results in a large number of write batches. As each batch of write requests can turn a row hit request into a row miss, the minimum number of row hit requests can reduce with the increase in batches of write requests. As a consequence, the gain of the AAM analysis over the RM analysis was reduced for $m = 8$.

3. Memory Access Demands: In this experiment, we vary the Memory Access Demand (MD) of tasks by considering three different configurations that are i) Low MD, i.e., MD=$(5\%, 20\%) \times C_i$, ii) Medium MD, i.e., MD=$(20\%, 40\%) \times C_i$, and iii) High MD, i.e., MD= $(40\%, 60\%) \times C_i$. The value of MD was assigned to each task in the taskset randomly as per the chosen configuration. The percentage of tasksets that were deemed schedulable by all the approaches, i.e., AAM, RM, and SOTA, for each MD configuration is plotted in Figure 7.6. We observe that the increase (resp. decrease) in the MD value negatively (resp. positively) impacts the taskset schedulability. This



(a) Low MD (b) Medium MD (c) High MD

Figure 7.6: Varying the Memory Demand (MD)



(a) Short (S) task period range (b) Medium (M) task period range (c) Long (L) task period range

Figure 7.7: Varying the task period range

happens because an increase in the MD value translates to an increase in the number of memory requests which also increases the number of interfering requests and total memory contention that tasks can suffer. As a consequence, all the approaches performed the best under the Low MD configuration and the worst under the High MD configuration. However, for all the MD configurations, the proposed RM and AAM performed significantly better than the SOTA analysis.

Furthermore, we can clearly observe in Figure 7.6 that the gain of AAM over RM increases with the increase in MD value. This mainly happens because an increase in MD value translates to an increase in the number of memory requests, thus, the total memory contention. Since the AAM analysis tightly bound the memory contention and memory access times, the gain of AAM over RM increases with the increase in MD value. In fact, the AAM analysis was able to schedule around 41% more tasksets than RM for High MD at the core utilization value of 0.25 as shown in Figure 7.6c.

4. Task Periods: In this experiment, we varied the period range of tasks and analyzed its impact on taskset schedulability. As the WCET C_i of tasks were generated using the task periods, i.e., $C_i = U_i \times T_i$, the task periods can impact the WCET C_i of tasks as well as the length of memory phases. In this experiment, we consider three different task period ranges that are i) S (Short) Range, i.e., $T_i = [100000 - 500000]$, ii) M (Medium) Range, i.e., $T_i = [100000 - 1000000]$, and iii) L (Long) Range, i.e., $T_i = [100000 - 5000000]$. For each range, the task periods were generated using log-uniform distribution. The percentage of schedulable tasksets for all the approaches and all the task period ranges is plotted in Figure 7.7.

In Figure 7.7, we observe that an increase in the period range has a negative impact on taskset schedulability. This mainly happens because increasing the task period increases the WCET of tasks due to the relation between C_i and T_i, i.e., $C_i = U_i \times T_i$. This in turn increases the blocking from one job of a lower priority task, i.e., a larger period leads to a larger WCET which causes a larger blocking from lower priority tasks. This implies that increasing the task period range increases the blocking caused by a lower priority task. This increase in lower priority blocking also increases the length of the level-i busy window and WCRT of tasks. This causes a degradation in taskset schedulability when the period ranges increase. However, for all the task period ranges, the proposed RM and AAM outperformed the SOTA analysis.

7.6 Chapter Summary

In this chapter, we present a white-box approach to the memory contention problem for 3-phase tasks. First, we identify the sources of pessimism in an existing memory contention that focuses on the 3-phase task model. We then show how this pessimism can be reduced by providing a more precise estimate of memory contention that can be caused by write requests. Additionally, we evaluate what impact memory address mapping can have on the memory contention of tasks. Experimental evaluation performed using case study experiments as well as synthetic tasksets show that our proposed memory contention analysis was able to derive tighter bounds on the memory contention of tasks which can improve task set schedulability by up to 100% percentage points. In future works, we aim to improve the proposed analysis by directly computing and integrating the total memory contention of the task under analysis into the WCRT analysis. This, for example, can be done by considering the specific set of R-phases that can be released on remote cores during the WCRT instead of always assuming the largest R-phase that can be released on each remote core.

Chapter 8

Conclusion and Future Work

For hard real-time systems deployed on top of the COTS multicore platform, a sound bound of shared resource contention is a prerequisite for accurate schedulability analysis. However, the bounds on shared resource contention need to be precise and should not be pessimistic since pessimistic bounds on shared resource contention may result in under-utilizing the computing platform. In this dissertation, we propose different solutions to accurately quantify the shared resource contention that can be suffered by tasks. We consider the phased execution model such as the 3-phase task execution model that divides tasks into distinct computation and memory phases, thus, enabling a fine-grained analysis for the shared resource contention. Specifically, we focus on the shared resource contention that can be suffered by tasks due to the shared *memory bus* and the *main memory*. We also provide a holistic view by considering the interdependence of the memory bus on cache memories. The main contributions of this dissertation are summarised in the following section.

8.1 Summary of Contributions

The main contributions of this dissertation are divided into three parts.

In the first part, we accurately quantify the shared resource contention that can be suffered by 3-phase tasks due to sharing of the *memory bus*. Specifically, the first part addresses problems **P1.**, **P2.**, and **P3.** and consists of Chapters 3, 4 and 5. In Chapter 3, we present the bus contention analysis for the 3-phase task model considering partitioned fixed-priority scheduling and the FCFS bus arbitration policy. In particular, we analyze the bus contention considering two different memory access models, i.e., dedicated and fair memory access models, that can be suited for different applications. The bus contention-aware schedulability analysis is then formulated by integrating the maximum bus contention that can be suffered by tasks into their WCRT analysis. In Chapter 4, we evaluate the impact of the bus arbitration policy on the bus contention suffered by tasks by varying the bus arbitration policy. Specifically, we show how the bounds on

the bus contention can be improved by considering the Round-Robin (RR) bus arbitration scheme. We highlight the importance of accurately quantifying the maximum blocking caused by lower priority tasks under the 3-phase task model executing on a multicore platform. We then propose an algorithm that accurately quantifies the blocking caused by lower priority tasks. The bus contention-aware WCRT analysis is then formulated by integrating the maximum bus contention and maximum blocking that can be suffered by 3-phase tasks. Finally, Chapter 5 presents a holistic view that highlights the relationship between the memory bus and cache memories. Specifically, we show that the bus contention strongly relates to the number of bus requests which in turn depends on the number of cache misses. This implies that ignoring the number of cache misses while computing a bound on bus contention can be pessimistic. Therefore, we propose the *cache-aware bus contention analysis* by first upper bounding the number of cache misses of the memory phases and then integrating them into the bus contention analysis. Evaluations show the cache-aware bus contention analyses can significantly improve the bound on bus contention in comparison to their cache-oblivious counterparts.

The second part of this dissertation focuses on the *memory-centric scheduler* that is responsible for scheduling memory phases of all tasks in the system. Specifically, the second part addresses problem **P4.**. In Chapter 6, we first identify that the pessimism in the existing Processor Priority (PP)-based memory-centric scheduler stems from the fact that it does not take into account task priorities while scheduling the memory phases of tasks and only considers the priority of cores. It can lead to a scenario in which higher priority tasks that execute on lower priority cores can suffer a large amount of memory interference. To address this, we propose the Task Priority (TP) based memory-centric scheduler that schedules the memory phases of tasks on the basis of task priorities. Furthermore, in contrast to the fixed-priority non-preemptive scheduling used by the PP-based memory-centric scheduler, we consider fixed-priority limited preemptive scheduling. We investigate the impact of preemption point selection, i.e., whether to allow preemptions anytime during the E-phases or only at the start/end of the E-phases, on the memory interference suffered by tasks. We observe that the preemption point selection can impact the memory interference suffered by tasks. The evaluation shows that the proposed TP-based memory-centric scheduler can significantly improve the bound on memory interference in comparison to the PP-based memory-centric scheduler.

The third part of this dissertation focuses on accurately quantifying the shared resource contention that can be suffered by 3-phase tasks due to sharing of the *main memory*. Specifically, the third part addresses problems **P5.** and **P6.** and consists of Chapter 7. In this part, we first identify the pessimism in the existing analysis that lies in the overestimation of the memory contention that can be caused by write memory requests. We then provide interesting insights to address the pessimism. Furthermore, we show that memory access times and memory contention suffered by tasks strongly relate to the memory address mapping, i.e., how the data required by memory phases of tasks are mapped to the address space of the main memory. Building on this, we propose a fine-grained memory contention analysis that takes into account different memory address mapping strategies. Evaluations show that the proposed analysis can significantly improve the existing

memory contention analysis that focuses on the 3-phase task model.

8.2 Thesis Validation

All the contributions carried out in the scope of this dissertation, summarized in Section 8.1, fulfill the main objective of this dissertation which is to build solutions to accurately quantify the shared resource contention between 3-phase tasks due to the sharing of two resources, i.e., the memory bus, and the main memory. All the contributions presented in this dissertation were peer-reviewed by the researchers of the real-time systems community and were published in several important journals, conferences, and workshops [Arora et al., 2020, Arora et al., 2021, Arora et al., 2022a, Arora et al., 2022b, Arora et al., 2022c, Arora et al., 2022d, Arora et al., 2023a, Arora et al., 2023b].

8.3 Future Work

Even though this dissertation provides important solutions concerning the shared resource contention in multicore platforms, there can be several different future directions that can either improve the work of this dissertation or focus on unaddressed issues. These future research directions are as follows.

8.3.1 Improving Preciseness of Bound on Shared Resource Contention

In this dissertation, we derive the maximum shared resource contention by considering the worst-case scenario without actually considering all the possible execution scenarios. There is a possibility that such a worst-case scenario may not occur in practice. For example, when a remote core releases a certain number of memory phases, they may not interfere with the task under analysis depending on the specific time on which the memory phases and computation phase actually execute on remote cores. This implies that the bound on shared resource contention can be improved by constructing all the execution scenarios that can happen on the local core and each of the remote cores. A common and well-known solution in this direction is using model checking-based approaches such as timed automata. However, modeling all execution scenarios on each core and their interaction with each other in order to derive the actual maximum shared resource contention can be extremely complicated. Furthermore, scalability is an obvious issue in such approaches as they suffer from the state-space explosion problem.

Another promising solution in this direction is the notion of the *Schedule Abstraction Graph* (SAG) [Nasri and Brandenburg, 2017, Nasri et al., 2018, Nasri et al., 2019, Nogd et al., 2020, Ranjha et al., 2022, Ranjha et al., 2023] that models all the *possible* execution scenarios of the set of tasks running on the system. The main idea is to take into account the task characteristics, i.e., task period, WCET, BCET[1], release jitter, task priorities, and build the SAG considering all possible

[1]BCET is the Best Case Execution Time which is defined as the minimum time required to execute any job of the task.

execution scenarios. Specifically, the SAG-based solutions use the novel path-merging technique that merges a set of common scheduling scenarios in order to reduce the potential number of states, hence, improving the scalability. In this direction, the first work [Nasri and Brandenburg, 2017] presents the schedulability analysis for non-preemptive scheduling on single-core processors. The authors show that for periodic tasks, the worst-case scenario provided by conventional WCRT analysis, i.e., that considers maximum blocking from lower priority tasks and maximum interference from the set of higher priority tasks, can yield pessimistic bounds. The main idea is that blocking and interference caused by a set of tasks running on the system depends on their release times so a given task may not necessarily suffer both the maximum interference and maximum blocking. For example, there can be a scenario in which not all jobs of all higher-priority tasks interfere with the task under analysis. Building on this, the schedulability analysis was proposed in [Nasri and Brandenburg, 2017] by modeling the worst-case timing behavior of tasks using the SAG. The same idea was then extended to multicore processors considering global scheduling [Nasri et al., 2018, Nasri et al., 2019, Nogd et al., 2020]. For example, in [Nogd et al., 2020], the WCRT analysis for global non-preemptive scheduling was proposed that implicitly explores all possible orders of job start times as well as their access to shared resources using the SAG.

Using the notion of SAG, a potential future direction is to analyze the maximum shared resource contention for 3-phase tasks considering partitioned fixed-priority scheduling. This can be achieved by modeling all possible execution scheduling scenarios and merging a set of similar execution scenarios for each core. Considering all possible per-core execution scenarios, the system-level worst-case scenario can be derived by taking into account the interactions between the memory phases of the task under analysis with the memory phases of tasks of remote cores. In other words, we can extract the specific scheduling scenario on the local and remote cores that can lead to the maximum shared resource contention. Having said that, achieving this can be extremely challenging. For example, using SAG is relatively easier for periodic tasks. However, sporadic tasks can add another layer of uncertainty as we cannot determine the specific release times of tasks. Due to such uncertainties, the number of possible execution scenarios may significantly increase which can further increase the complexity of the problem. Consequently, while SAG can be a promising solution to compute a more precise bound on the shared resource contention, a significant amount of effort is still required in this direction.

8.3.2 Task to Core Mapping Strategies

In this dissertation, we mainly focus on shared resource contention and assume that tasks are randomly mapped to cores. Therefore, a natural extension can be to evaluate the impact that different task-to-core mapping strategies can have on the shared resource contention that can be suffered by tasks. Furthermore, specific shared resource contention analysis can be built by taking into account the mapping of tasks to cores. Apart from the conventional task-to-core mapping strategies, e.g., first-fit, best-fit, worst-fit, next-fit, etc., different task-to-core mapping strategies can be explored, e.g., considering the per-core memory utilization, the length of memory phases of tasks, etc.

8.3.3 Holistic Frameworks

Another future direction is to build holistic solutions that take into account the interdependence of all the shared resources. For example, a holistic framework can be proposed that incorporate the contention suffered by each of the shared resource, e.g., caches, memory bus, main memory, etc. As an example, if we model memory bus transactions as split transactions, tasks can suffer both bus contention and memory contention in such cases. Therefore, there is a need for a holistic solution that analyses shared resource contention that can be suffered by tasks due to the sharing of the memory bus and main memory. By taking into account the interdependence among shared resources, the solution should first bound the number of cache misses to compute the number of bus requests and bus contention. Similarly, the memory contention analysis will then additionally integrate the number of per-core bus requests that can be served by the memory bus which further depends on the bus arbitration policy and the number of cache misses on each core. Although such approaches can provide promising solutions, the complexity of analyzing the shared resource contention can be exacerbated by considering the interdependence between shared resources. For example, the memory contention analysis itself is complicated due to several low-level arbitration mechanisms employed by the memory controller. Considering the interdependence of main memory on the memory bus can significantly increase the complexity of the problem. Another possible solution is to quantify the maximum shared resource contention in a holistic framework using Integer Linear Programming (ILP) based formulation. Each constraint can be modeled carefully by considering interdependence among shared resources with the objective function that maximizes the shared resource contention that can be suffered by the task under analysis considering all the shared resources and their interdependence.

References

[Agrawal et al., 2018] Agrawal, A., Mancuso, R., Pellizzoni, R., and Fohler, G. (2018). Analysis of dynamic memory bandwidth regulation in multi-core real-time systems. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 230–241.

[Aldworth and Croxford, 2008] Aldworth, P. J. and Croxford, D. (US12/309,762, Feb. 2008). Memory controller address mapping scheme.

[Alhammad and Pellizzoni, 2014a] Alhammad, A. and Pellizzoni, R. (2014a). Schedulability analysis of global memory-predictable scheduling. In *Proceedings of the 14th International Conference on Embedded Software*, EMSOFT '14, New York, NY, USA. Association for Computing Machinery.

[Alhammad and Pellizzoni, 2014b] Alhammad, A. and Pellizzoni, R. (2014b). Time-predictable execution of multithreaded applications on multicore systems. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6.

[AMC-20, 2022] AMC-20 (2022). General acceptable means of compliance for airworthiness of products, parts and appliances (amc-20, amendment 23), easa. https://www.easa.europa.eu/en/downloads/134971/en. Accessed: 2023-02-13.

[AMD, 2023] AMD (2023). Versal adaptive soc programmable network on chip and integrated memory controller 1.0 logicore ip product guide (pg313). Accessed: 2023-05-31.

[Andersson et al., 2010] Andersson, B., Easwaran, A., and Lee, J. (2010). Finding an upper bound on the increase in execution time due to contention on the memory bus in COTS-based multicore systems. *ACM SIGBED Review*, 7(1):1.

[Arora et al., 2021] Arora, J., Maia, C., Aftab Rashid, S., Nelissen, G., and Tovar, E. (2021). Bus-contention aware schedulability analysis for the 3-phase task model with partitioned scheduling. In *29th International Conference on Real-Time Networks and Systems*, RTNS'2021, page 123–133, New York, NY, USA. Association for Computing Machinery.

[Arora et al., 2020] Arora, J., Maia, C., Rashid, S. A., Nelissen, G., and Tovar, E. (2020). Work-in-progress: Wcrt analysis for the 3-phase task model in partitioned scheduling. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 407–410.

[Arora et al., 2022a] Arora, J., Maia, C., Rashid, S. A., Nelissen, G., and Tovar, E. (2022a). Bus-contention aware wcrt analysis for the 3-phase task model considering a work-conserving bus arbitration scheme. *Journal of Systems Architecture*, 122:102345.

[Arora et al., 2022b] Arora, J., Maia, C., Rashid, S. A., Nelissen, G., and Tovar, E. (2022b). Schedulability analysis for 3-phase tasks with partitioned fixed-priority scheduling. *Journal of Systems Architecture*, 131:102706.

[Arora et al., 2022c] Arora, J., Rashid, S. A., Maia, C., Nelissen, G., and Tovar, E. (2022c). Work-in-progress: A holistic approach to wcrt analysis for multicore systems. In *2022 IEEE Real-Time Systems Symposium (RTSS)*, pages 511–514.

[Arora et al., 2022d] Arora, J., Rashid, S. A., Maia, C., and Tovar, E. (2022d). Analyzing fixed task priority based memory centric scheduler for the 3-phase task model. In *2022 IEEE 28th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 51–60.

[Arora et al., 2023a] Arora, J., Rashid, S. A., Nelissen, G., Maia, C., and Tovar, E. (2023a). Improved bus contention analysis for 3-phase tasks. In *2023 IEEE 29th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)* ***(To appear)***.

[Arora et al., 2023b] Arora, J., Rashid, S. A., Nelissen, G., Maia, C., and Tovar, E. (2023b). Memory contention analysis for 3-phase tasks. In *Junior Researcher Workshop on Real-Time Computing, co-located with RTNS 2023 (JRWRTC 2023)*. JRWRTC.

[Audsley, 1990] Audsley, N. C. (1990). Deadline monotonic scheduling. Technical Report YCS-146, Department of Computer Science, University of York.

[Becker et al., 2016] Becker, M., Dasari, D., Nicolic, B., Akesson, B., Nélis, V., and Nolte, T. (2016). Contention-free execution of automotive applications on a clustered many-core platform. In *ECRTS 2016*, pages 14–24.

[Boppana et al., 2015] Boppana, V., Ahmad, S., Ganusov, I., Kathail, V., Rajagopalan, V., and Wittig, R. (2015). Ultrascale+ mpsoc and fpga families. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–37.

[Bril et al., 2007] Bril, R. J., Lukkien, J. J., and Verhaegh, W. F. J. (2007). Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited. In *19th Euromicro Conference on Real-Time Systems (ECRTS'07)*, pages 269–279.

[Buttazzo, 2011] Buttazzo, G. C. (2011). *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer Publishing Company, Incorporated, 3rd edition.

[Casini et al., 2020] Casini, D., Biondi, A., Nelissen, G., and Buttazzo, G. (2020). A holistic memory contention analysis for parallel real-time tasks under partitioned scheduling. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 239–252.

[CAST-32A, 2016] CAST-32A (2016). Certification authorities software team (cast), federal aviation administration. 1 november 2016. Accessed: 2023-02-13.

[Cecere et al., 2016] Cecere, N., Tipaldi, M., Wenker, R., and Villano, U. (2016). Measurement and analysis of schedulability of spacecraft on-board software. In *2016 IEEE Metrology for Aerospace (MetroAeroSpace)*, pages 545–550.

[Chatterjee et al., 2012] Chatterjee, N., Muralimanohar, N., Balasubramonian, R., Davis, A., and Jouppi, N. P. (2012). Staged reads: Mitigating the impact of dram writes on dram reads. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12.

[Chattopadhyay and Roychoudhury, 2011] Chattopadhyay, S. and Roychoudhury, A. (2011). Static bus schedule aware scratchpad allocation in multiprocessors. In *Proceedings of the 2011 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '11, page 11–20, New York, NY, USA. Association for Computing Machinery.

[Chattopadhyay et al., 2010] Chattopadhyay, S., Roychoudhury, A., and Mitra, T. (2010). Modeling shared cache and bus in multi-cores for timing analysis. In *Proceedings of the 13th International Workshop on Software & Compilers for Embedded Systems*, SCOPES '10, New York, NY, USA. Association for Computing Machinery.

[Dasari, 2014] Dasari, D. (2014). Timing analysis of real-time systems considering the contention on the shared interconnection network in multicores. Accessed: 2023-02-13.

[Dasari et al., 2013] Dasari, D., Akesson, B., Nelis, V., Awan, M. A., and Petters, S. M. (2013). Identifying the sources of unpredictability in COTS-based multicore systems. In *2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 39–48, Porto. IEEE.

[Dasari et al., 2011] Dasari, D., Andersson, B., Nelis, V., Petters, S. M., Easwaran, A., and Lee, J. (2011). Response time analysis of cots-based multicores considering the contention on the shared memory bus. In *2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 1068–1075.

[Dasari and Nelis, 2012] Dasari, D. and Nelis, V. (2012). An analysis of the impact of bus contention on the wcet in multicores. In *2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems*, pages 1450–1457.

[Dasari et al., 2015] Dasari, D., Nelis, V., and Akesson, B. (2015). A framework for memory contention analysis in multi-core platforms. *Real-Time Systems*, 52.

[Davis et al., 2017] Davis, R. I., Altmeyer, S., Indrusiak, L. S., and·Vincent Nelis, C. M., and Reineke, J. (2017). An extensible framework for multicore response time analysis. *Real-Time Systems*.

[Davis et al., 2016] Davis, R. I., Altmeyer, S., and Reineke, J. (2016). Analysis of write-back caches under fixed-priority preemptive and non-preemptive scheduling. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, RTNS '16, page 309–318, New York, NY, USA. Association for Computing Machinery.

[Davis and Burns, 2011] Davis, R. I. and Burns, A. (2011). A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4).

[de Dinechin et al., 2014] de Dinechin, B. D., van Amstel, D., Poulhiès, M., and Lager, G. (2014). Time-critical computing on a single-chip massively parallel processor. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6.

[Durrieu et al., 2014] Durrieu, G., Faugère, M., Girbal, S., Gracia Pérez, D., Pagetti, C., and Puffitsch, W. (2014). Predictable Flight Management System Implementation on a Multicore Processor. In *Embedded Real Time Software (ERTS'14)*, TOULOUSE, France.

[Ecco and Ernst, 2017] Ecco, L. and Ernst, R. (2017). Tackling the bus turnaround overhead in real-time sdram controllers. *IEEE Transactions on Computers*, 66(11):1961–1974.

[Emberson et al., 2010] Emberson, P., Stafford, R., and Davis, R. (2010). Techniques for the synthesis of multiprocessor tasksets. *WATERS'10*.

[Forsberg et al., 2018] Forsberg, B., Benini, L., and Marongiu, A. (2018). Heprem: Enabling predictable gpu execution on heterogeneous soc. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 539–544.

[Forsberg et al., 2021] Forsberg, B., Solieri, M., Bertogna, M., Benini, L., and Marongiu, A. (2021). The predictable execution model in practice: Compiling real applications for cots hardware. *ACM Trans. Embed. Comput. Syst.*, 20(5).

[Fort and Forget, 2019] Fort, F. and Forget, J. (2019). Code generation for multi-phase tasks on a multi-core distributed memory platform. In *2019 IEEE 25th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–6.

[Gracioli et al., 2019] Gracioli, G., Tabish, R., Mancuso, R., Mirosanlou, R., Pellizzoni, R., and Caccamo, M. (2019). Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms. In Quinton, S., editor, *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:25, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Gustafsson et al., 2010] Gustafsson, J., Betts, A., Ermedahl, A., and Lisper, B. (2010). The Mälardalen WCET Benchmarks: Past, Present And Future. In Lisper, B., editor, *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 136–146, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. The printed version of the WCET'10 proceedings are published by OCG (www.ocg.at) - ISBN 978-3-85403-268-7.

[Hassan and Patel, 2016] Hassan, M. and Patel, H. (2016). Mcxplore: An automated framework for validating memory controller designs. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1357–1362.

[Hassan and Patel, 2018] Hassan, M. and Patel, H. (2018). Mcxplore: Automating the validation process of dram memory controller designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(5):1050–1063.

[Hassan and Pellizzoni, 2018] Hassan, M. and Pellizzoni, R. (2018). Bounding dram interference in cots heterogeneous mpsocs for mixed criticality systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2323–2336.

[Hassan and Pellizzoni, 2020] Hassan, M. and Pellizzoni, R. (2020). Analysis of Memory-Contention in Heterogeneous COTS MPSoCs. In *ECRTS 2020*, volume 165 of *LIPIcs*, pages 23:1–23:24, Dagstuhl, Germany.

[Heechul Yun et al., 2013] Heechul Yun, Gang Yao, Pellizzoni, R., Caccamo, M., and Lui Sha (2013). MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, Philadelphia, PA. IEEE.

[Jacobs et al., 2015] Jacobs, M., Hahn, S., and Hack, S. (2015). Wcet analysis for multi-core processors with shared buses and event-driven bus arbitration. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, RTNS '15, page 193–202, New York, NY, USA. Association for Computing Machinery.

[Jacobs et al., 2016] Jacobs, M., Hahn, S., and Hack, S. (2016). A framework for the derivation of wcet analyses for multi-core processors. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 141–151.

[JEDEC, 2008] JEDEC (2008). Ddr3 sdram standard jesd79-3b, 2008. Accessed: 2023-03-03.

[Joseph and Pandya, 1986] Joseph, M. and Pandya, P. (1986). Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390–395.

[Kelter et al., 2011] Kelter, T., Falk, H., Marwedel, P., Chattopadhyay, S., and Roychoudhury, A. (2011). Bus-aware multicore wcet analysis through tdma offset bounds. In *2011 23rd Euromicro Conference on Real-Time Systems*, pages 3–12.

[Kelter et al., 2014] Kelter, T., Falk, H., Marwedel, P., Chattopadhyay, S., and Roychoudhury, A. (2014). Static analysis of multi-core tdma resource arbitration delays. *Real-Time Systems*, 50.

[Kessler and Hill, 1992] Kessler, R. E. and Hill, M. D. (1992). Page placement algorithms for large real-indexed caches. *ACM Trans. Comput. Syst.*, 10(4):338–359.

[Kim et al., 2014] Kim, H., de Niz, D., Andersson, B., Klein, M., Mutlu, O., and Rajkumar, R. (2014). Bounding memory interference delay in cots-based multi-core systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 145–154.

[Kim et al., 2016] Kim, H., de Niz, D., Andersson, B., Klein, M., Mutlu, O., and Rajkumar, R. (2016). Bounding and reducing memory interference in cots-based multi-core systems. *Real-Time Systems*, 52.

[Kloda et al., 2023] Kloda, T., Gracioli, G., Tabish, R., Mirosanlou, R., Mancuso, R., Pellizzoni, R., and Caccamo, M. (2023). Lazy load scheduling for mixed-criticality applications in heterogeneous mpsocs. *ACM Trans. Embed. Comput. Syst.*, 22(3).

[Koike et al., 2020] Koike, R., Fukunaga, T., Igarashi, S., and Azumi, T. (2020). Contention-free scheduling for clustered many-core platform. In *2020 IEEE International Conference on Embedded Software and Systems (ICESS)*, pages 1–8.

[Koo and Kim, 2018] Koo, C. H. and Kim, H. (2018). Measurement of cache-related preemption delay for spacecraft computers. In *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 234–235.

[Lattner and Adve, 2004] Lattner, C. and Adve, V. (2004). Llvm: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86.

[Lehoczky, 1990] Lehoczky, J. (1990). Fixed priority scheduling of periodic task sets with arbitrary deadlines. *[1990] Proceedings 11th Real-Time Systems Symposium*, pages 201–209.

[Liedtke et al., 1997] Liedtke, J., Haertig, H., and Hohmuth, M. (1997). Os-controlled cache predictability for real-time systems. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, RTAS '97, page 213, USA. IEEE Computer Society.

[Liu and Layland, 1973] Liu, C. L. and Layland, J. W. (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61.

[Liu et al., 2012] Liu, L., Cui, Z., Xing, M., Bao, Y., Chen, M., and Wu, C. (2012). A software memory partition approach for eliminating bank-level interference in multicore systems. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 367–375.

[Lugo et al., 2022] Lugo, T., Lozano, S., Fernández, J., and Carretero, J. (2022). A survey of techniques for reducing interference in real-time applications on multicore platforms. *IEEE Access*, 10:21853–21882.

[Maia et al., 2017] Maia, C., Nelissen, G., Nogueira, L., Pinho, L. M., and Perez, D. G. (2017). Schedulability analysis for global fixed-priority scheduling of the 3-phase task model. In *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10, Hsinchu, Taiwan. IEEE.

[Maia et al., 2016] Maia, C., Nogueira, L., Pinho, L. M., and Perez, D. G. (2016). A closer look into the AER Model. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, Berlin, Germany. IEEE.

[Maiza et al., 2019] Maiza, C., Rihani, H., Rivas, J. M., Goossens, J., Altmeyer, S., and Davis, R. I. (2019). A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems. *ACM Computing Surveys*, 52(3):1–38.

[Mancuso et al., 2013] Mancuso, R., Dudko, R., Betti, E., Cesati, M., Caccamo, M., and Pellizzoni, R. (2013). Real-time cache management framework for multi-core architectures. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 45–54.

[Mancuso et al., 2014] Mancuso, R., Dudko, R., and Caccamo, M. (2014). Light-prem: Automated software refactoring for predictable execution on cots embedded systems. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10.

[Mancuso et al., 2015] Mancuso, R., Pellizzoni, R., Caccamo, M., Sha, L., and Yun, H. (2015). Wcet(m) estimation in multi-core systems using single core equivalence. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 174–183.

[Mancuso et al., 2017] Mancuso, R., Pellizzoni, R., Tokcan, N., and Caccamo, M. (2017). WCET Derivation under Single Core Equivalence with Explicit Memory Budget Assignment. In Bertogna, M., editor, *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:23, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Matějka et al., 2018] Matějka, J., Forsberg, B., Sojka, M., Hanzálek, Z., Benini, L., and Marongiu, A. (2018). Combining prem compilation and ilp scheduling for high-performance and predictable mpsoc execution. In *Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM'18, page 11–20, New York, NY, USA. Association for Computing Machinery.

[Melani et al., 2015] Melani, A., Bertogna, M., Bonifaci, V., Marchetti-Spaccamela, A., and Buttazzo, G. (2015). Memory-processor co-scheduling in fixed priority systems. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, RTNS '15, page 87–96, New York, NY, USA. Association for Computing Machinery.

[Meunier et al., 2022] Meunier, R., Carle, T., and Monteil, T. (2022). Correctness and Efficiency Criteria for the Multi-Phase Task Model. In Maggio, M., editor, *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*, volume 231 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:21, Dagstuhl, Germany. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[Nasri and Brandenburg, 2017] Nasri, M. and Brandenburg, B. B. (2017). An exact and sustainable analysis of non-preemptive scheduling. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 12–23.

[Nasri et al., 2018] Nasri, M., Nelissen, G., and Brandenburg, B. B. (2018). A Response-Time Analysis for Non-Preemptive Job Sets under Global Scheduling. In Altmeyer, S., editor, *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, volume 106 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:23, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Nasri et al., 2019] Nasri, M., Nelissen, G., and Brandenburg, B. B. (2019). Response-Time Analysis of Limited-Preemptive Parallel DAG Tasks Under Global Scheduling. In Quinton, S., editor, *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:23, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Negrean and Ernst, 2012] Negrean, M. and Ernst, R. (2012). Response-time analysis for non-preemptive scheduling in multi-core systems with shared resources. In *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*, pages 191–200.

[Nélis et al., 2016] Nélis, V., Yomsi, P. M., and Pinho, L. M. (2016). The Variability of Application Execution Times on a Multi-Core Platform. In Schoeberl, M., editor, *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, volume 55 of *OpenAccess Series in Informatics (OASIcs)*, pages 6:1–6:11, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Nesbit et al., 2006] Nesbit, K. J., Aggarwal, N., Laudon, J., and Smith, J. E. (2006). Fair queuing memory systems. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 208–222.

[Nogd et al., 2020] Nogd, S., Nelissen, G., Nasri, M., and Brandenburg, B. B. (2020). Response-time analysis for non-preemptive global scheduling with fifo spin locks. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 115–127.

[Nowotsch and Paulitsch, 2012] Nowotsch, J. and Paulitsch, M. (2012). Leveraging multi-core computing architectures in avionics. In *2012 Ninth European Dependable Computing Conference*, pages 132–143.

[P4080, 2011] P4080 (2011). P4080 development system user's guide - nxp. Accessed: 2023-02-26.

[Pagetti et al., 2018] Pagetti, C., Forget, J., Falk, H., Oehlert, D., and Luppold, A. (2018). Automated generation of time-predictable executables on multicore. In *Proceedings of the 26th International Conference on Real-Time Networks and Systems*, RTNS '18, page 104–113, New York, NY, USA. Association for Computing Machinery.

[Pellizzoni et al., 2011] Pellizzoni, R., Betti, E., Bak, S., Yao, G., Criswell, J., Caccamo, M., and Kegley, R. (2011). A Predictable Execution Model for COTS-Based Embedded Systems. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279, Chicago, IL, USA. IEEE.

[PL310, 2008] PL310 (2008). Arm. primecell level 2 cache controller (pl310) - technical reference manual, revision:r2p0. Accessed: 2022-07-28.

[Radojković et al., 2012] Radojković, P., Girbal, S., Grasset, A., Quiñones, E., Yehia, S., and Cazorla, F. J. (2012). On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments. *ACM Trans. Archit. Code Optim.*, 8(4).

[Ranjha et al., 2023] Ranjha, S., Gohari, P., Nelissen, G., and Nasri, M. (2023). Partial-order reduction in reachability-based response-time analyses of limited-preemptive dag tasks. *Real-Time Systems*, 59:1–55.

[Ranjha et al., 2022] Ranjha, S., Nelissen, G., and Nasri, M. (2022). Partial-order reduction for schedule-abstraction-based response-time analyses of non-preemptive tasks. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 121–132.

[Rashid et al., 2022] Rashid, S. A., Awan, M. A., Souto, P. F., Bletsas, K., and Tovar, E. (2022). Cache-aware schedulability analysis of prem compliant tasks. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1269–1274.

[Rashid et al., 2016] Rashid, S. A., Nelissen, G., Hardy, D., Akesson, B., Puaut, I., and Tovar, E. (2016). Cache-persistence-aware response-time analysis for fixed-priority preemptive systems. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 262–272.

[Rashid et al., 2020] Rashid, S. A., Nelissen, G., and Tovar, E. (2020). Bounding cache persistence reload overheads for set-associative caches. In *2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10.

[Rashid et al., 2020] Rashid et al., S. A. (2020). Cache persistence-aware memory bus contention analysis for multicore systems. In *DATE*, pages 442–447.

[Rihani et al., 2015] Rihani, H., Moy, M., Maiza, C., and Altmeyer, S. (2015). Wcet analysis in shared resources real-time systems with tdma buses. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, RTNS '15, page 183–192, New York, NY, USA. Association for Computing Machinery.

[Rivas et al., 2019] Rivas, J. M., Goossens, J., Poczekajlo, X., and Paolillo, A. (2019). Implementation of Memory Centric Scheduling for COTS Multi-Core Real-Time Systems. In Quinton, S., editor, *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:23, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Rixner et al., 2000] Rixner, S., Dally, W., Kapasi, U., Mattson, P., and Owens, J. (2000). Memory access scheduling. In *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*, pages 128–138.

[Rosen et al., 2007] Rosen, J., Andrei, A., Eles, P., and Peng, Z. (2007). Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pages 49–60.

[Rouxel et al., 2017] Rouxel, B., Derrien, S., and Puaut, I. (2017). Tightening contention delays while scheduling parallel applications on multi-core architectures. *ACM Trans. Embed. Comput. Syst.*, 16(5s).

[Rouxel et al., 2019] Rouxel, B., Skalistis, S., Derrien, S., and Puaut, I. (2019). Hiding Communication Delays in Contention-Free Execution for SPM-Based Multi-Core Architectures. In Quinton, S., editor, *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:24, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Schliecker and Ernst, 2010] Schliecker, S. and Ernst, R. (2010). Real-time performance analysis of multiprocessor systems with shared memory. *ACM Transactions on Embedded Computing Systems*, 10(2):1–27.

[Schranzhofer et al., 2010] Schranzhofer, A., Chen, J.-J., and Thiele, L. (2010). Timing analysis for tdma arbitration in resource sharing systems. In *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 215–224.

[Schuh et al., 2020] Schuh, M., Maiza, C., Goossens, J., Raymond, P., and de Dinechin, B. D. (2020). A study of predictable execution models implementation for industrial data-flow applications on a multi-core platform with shared banked memory. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 283–295.

[Schwäricke et al., 2020] Schwäricke, G., Kloda, T., Gracioli, G., Bertogna, M., and Caccamo, M. (2020). Fixed-Priority Memory-Centric Scheduler for COTS-Based Multiprocessors. In Völp, M., editor, *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, volume 165 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:24, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[Senoussaoui et al., 2022a] Senoussaoui, I., Benhaoua, M. K., Zahaf, H.-E., and Lipari, G. (2022a). Toward memory-centric scheduling for prem task on multicore platforms, when processor assignments are specified. In *2022 3rd International Conference on Embedded & Distributed Systems (EDiS)*, pages 11–15.

[Senoussaoui et al., 2022b] Senoussaoui, I., Zahaf, H.-E., Lipari, G., and Benhaoua, K. M. (2022b). Contention-free scheduling of prem tasks on partitioned multicore platforms. In *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8.

[Shivappa., 2014] Shivappa., V. (2014). V. Shivappa. x86: Intel cache allocation technology support. Accessed: 2022-07-28.

[Soliman et al., 2019] Soliman, M. R., Gracioli, G., Tabish, R., Pellizzoni, R., and Caccamo, M. (2019). Segment streaming for the three-phase execution model: Design and implementation. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 260–273.

[Soliman and Pellizzoni, 2019] Soliman, M. R. and Pellizzoni, R. (2019). PREM-Based Optimal Task Segmentation Under Fixed Priority Scheduling. In Quinton, S., editor, *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:23, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Stankovic, 1988] Stankovic, J. (1988). Misconceptions about real-time computing: a serious problem for next-generation systems. *Computer*, 21(10):10–19.

[Tabish et al., 2016] Tabish, R., Mancuso, R., Wasly, S., Alhammad, A., Phatak, S. S., Pellizzoni, R., and Caccamo, M. (2016). A real-time scratchpad-centric os for multi-core embedded systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11.

[Tabish et al., 2019] Tabish, R., Mancuso, R., Wasly, S., Pellizzoni, R., and Caccamo, M. (2019). A real-time scratchpad-centric os with predictable inter/intra-core communication for multi-core embedded systems. *Real-Time Systems*, 55.

[Tabish et al., 2023] Tabish, R., Pellizzoni, R., Mancuso, R., Gracioli, G., Mirosanlou, R., and Caccamo, M. (2023). X-stream: Accelerating streaming segments on mpsocs for real-time applications. *Journal of Systems Architecture*, 138:102857.

[Thilakasiri and Becker, 2023a] Thilakasiri, T. and Becker, M. (2023a). An exact schedulability analysis for global fixed-priority scheduling of the aer task model. In *Proceedings of the 28th Asia and South Pacific Design Automation Conference*, ASPDAC '23, page 326–332, New York, NY, USA. Association for Computing Machinery.

[Thilakasiri and Becker, 2023b] Thilakasiri, T. and Becker, M. (2023b). Methods to realize preemption in phased execution models. *ACM Trans. Embed. Comput. Syst.*, 22(5s).

[Tomiyama and Dutt, 2000] Tomiyama, H. and Dutt, N. D. (2000). Program path analysis to bound cache-related preemption delay in preemptive real-time systems. In *Proceedings of the Eighth International Workshop on Hardware/Software Codesign. CODES 2000 (IEEE Cat. No. 00TH8518)*, pages 67–71. IEEE.

[Wasly and Pellizzoni, 2014] Wasly, S. and Pellizzoni, R. (2014). Hiding memory latency using fixed priority scheduling. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 75–86.

[Wilhelm et al., 2008] Wilhelm, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., and Heckmann, R. (2008). The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):1–53.

[Wu et al., 2016] Wu, Z., Pellizzoni, R., and Guo, D. (2016). A composable worst case latency analysis for multi-rank dram devices under open row policy. *Real-Time Systems*, 52.

[Yao et al., 2012] Yao, G., Pellizzoni, R., Bak, S., Betti, E., and Caccamo, M. (2012). Memory-centric scheduling for multicore hard real-time systems. *Real-Time Systems*, 48.

[Yao et al., 2016a] Yao, G., Pellizzoni, R., Bak, S., Yun, H., and Caccamo, M. (2016a). Global real-time memory-centric scheduling for multicore systems. *IEEE Transactions on Computers*, 65(9):2739–2751.

[Yao et al., 2016b] Yao, G., Yun, H., Wu, Z. P., Pellizzoni, R., Caccamo, M., and Sha, L. (2016b). Schedulability analysis for memory bandwidth regulated multicore real-time systems. *IEEE Transactions on Computers*, 65(2):601–614.

[Yun et al., 2014] Yun, H., Mancuso, R., Wu, Z.-P., and Pellizzoni, R. (2014). PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166, Berlin, Germany. IEEE.

[Yun et al., 2015] Yun, H., Pellizzon, R., and Valsan, P. K. (2015). Parallelism-aware memory interference delay analysis for cots multicore systems. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 184–195.

[Zhuravlev et al., 2010] Zhuravlev, S., Blagodurov, S., and Fedorova, A. (2010). Addressing shared resource contention in multicore processors via scheduling. *SIGARCH Comput. Archit. News*, 38(1):129–142.

Appendix A

Unbounded Priority Inversion Problem

In this appendix, we will discuss the potential problems that may occur in the TP-based MCS (discussed in Chapter 6) when using *global memory preemptions*. Global memory preemption is a phenomenon in which the memory phase of a task can be preempted by the memory phases of higher-priority tasks executing on remote cores and it has been used in the existing PP-based MCS [Schwäricke et al., 2020]. However, considering global memory preemptions in the TP-based MCS can lead to the problem of *unbounded priority inversion*. The unbounded priority inversion is a phenomenon in which the higher priority task is being delayed due to the execution/memory accesses of lower priority tasks that can execute on the system. We will now discuss how this problem can occur in the proposed TP-based MCS using the following example.

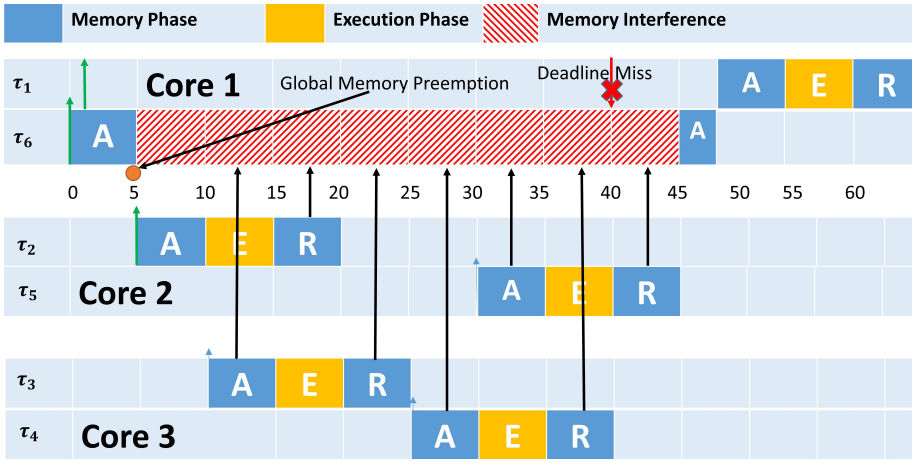


Figure A.1: Unbounded priority inversion problem in the TP-based MCS using global memory preemptions

As shown in Figure A.1, there are 3 cores in the system in which 2 tasks are assigned to each core. Furthermore, the global fixed-task priorities are assigned such that tasks are indexed as per their priorities $\tau_1, \tau_2, \ldots, \tau_6$. *As shown in Figure A.1, task* τ_1, *i.e., the highest priority task in the system, and task* τ_6, *i.e., the lowest priority task in the system, are assigned to core 1. Now assume that the* τ_1 *is the task under analysis. If* τ_6 *starts executing before the release of task* τ_1

then it can cause blocking to τ_1. This is due to the fact that MCS typically consider non-preemptive memory phases at the core level, i.e., a memory phase cannot be preempted due to a higher priority task of the same core. Now if we consider global memory preemptions, i.e., memory phases can be preempted by higher priority tasks of remote cores, it can lead to the problem of unbounded priority inversion. As shown in Figure A.1, due to global memory preemptions, the A-phase of τ_6 is preempted by the memory phases of all remote cores tasks that have priorities higher than τ_6, i.e., τ_2, τ_3, τ_4, and τ_5. As a consequence, τ_6 is being delayed due to the memory phases of tasks running on all remote cores. While this delay is acceptable for τ_6 as it is the lowest priority task in the system, this delay can also impact τ_1 which is the highest priority task in the system. This happens because τ_1, i.e., the highest priority task, is blocked by task τ_6 on core 1, and τ_6 is further delayed due to tasks of core 2 and core 3. Consequently, task τ_1 is being delayed due to all the lower-priority tasks in the system. This is problematic and can eventually result in a deadline miss by the highest priority task τ_1 as shown in Figure A.1.

To avoid this problem, the proposed TP-based MCS considers non-preemptive memory phases at the system level and fixed-priority limited preemptive scheduling at the core level.