CISTER
Research Centre in
Real-Time & Embedded
Computing Systems

PhD Thesis

Towards Timing Analysis of Multi-core Platforms for Hard Real-Time Systems

Syed Aftab Rashid

CISTER-TR-210403

2021/04/09

Towards Timing Analysis of Multi-core Platforms for Hard Real-Time Systems

Syed Aftab Rashid

CISTER Research Centre

Polytechnic Institute of Porto (ISEP P.Porto)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail: syara@isep.ipp.pt

https://www.cister-labs.pt

Abstract

Modern processors provide enhanced performance with reduced power, size and cost in average case and are becoming mainstream in almost all application domains including real-time embedded systems. However, the use of modern computing platforms in hard real-time systems, i.e., systems with stringent timing requirements, is still under scrutiny of the real-time systems community due to their unpredictable nature. This is mainly due to resources such as, caches and the memory bus that are shared among several tasks executing on the processor. As tasks can run concurrently on the processor, consequently, simultaneous use of any of these shared resources can result in inter-task resource contention which can significantly affect the timing behavior of the executing tasks. To safely conclude that any task executing on the platform may or may not fulfill its timing requirements, it is essential to first compute accurate bounds on the shared resource contention that may be experienced by that task.

The main objective of this dissertation is to provide software-based solutions that can be used to accurately quantify the shared resource contention between tasks due to two main resources, i.e., caches and the memory bus.

We start by identifying the pessimism in the existing analysis that focus on bounding intertask cache contention for direct-mapped caches. We show that this pessimism mainly comes from a unidirectional focus on the negative perspective of caches, i.e., derived from a preempting task invalidating cache lines useful to the preempted task, thereby extending a preempted task's execution time. In contrast, we identify a different positive perspective of caches, i.e., cache persistence, which refers to the re-use of cache content between different job executions of a task, leading to a tighter bound on the total memory access demand of the task. We propose a new preciser analysis that accounts for both the negative and the positive perspective of caches when computing inter-task cache contention, and results in significantly improving task's schedulability.

We then extend our analysis to set-associative caches and show that the previously developed analysis for direct-mapped caches cannot be used as is for set-associative. We present several different approaches to bound inter-task contention considering set-associative caches. Our analysis accurately determines cache blocks that may suffer additional cache reloads due to inter-task cache conflicts even in the presence of cache persistence and eliminates substantial pessimism with respect to former analyses.

We highlight additional challenges that stem from analyzing inter-task cache conflicts in the presence of a cache hierarchy and propose an analysis to bound inter-task cache contention considering multilevel caches. We identify the sources of overestimation in a preceding analysis that focus on bounding inter-task contention for multilevel caches and propose solutions to minimize that overestimation.

Finally, we present a holistic analysis that considers the interdependence between cache contention and memory bus contention and evaluate their cumulative impact on the timing requirements of tasks. We show that the analysis that tightly bounds the inter-task cache contention may also result in significantly reducing the memory bus contention suffered by the tasks, thereby, improving schedulability.

© 2021 CISTER Research Center

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Towards Timing Analysis of Multi-core Platforms for Hard Real-Time Systems

Syed Aftab Rashid

Supervisor: Prof. Eduardo Manuel Medicis Tovar

Co-Supervisor: Prof. Geoffrey Nelissen

Co-Supervisor: Prof. Luis Miguel Pinho de Almeida

Programa Doutoral em Engenharia Electrotécnica e de Computadores

April, 2021

© Syed Aftab Rashid: April, 2021

Faculdade de Engenharia da Universidade do Porto

Towards Timing Analysis of Multi-core Platforms for Hard Real-Time Systems

Syed Aftab Rashid

Dissertation submitted to Faculdade de Engenharia da Universidade do Porto
to obtain the degree of

Doctor Philosophiae in Electrical & Computer Engineering

President: Dr. José Alfredo Ribeiro da Silva Matos
External Referee: Dr. Sebastian Altmeyer
External Referee: Dr. Claire Maiza
Internal Referee: Dr. João Paulo de Castro Canas Ferreira
Internal Referee: Dr. Mário Jorge Rodrigues de Sousa
Supervisor: Dr. Eduardo Manuel Medicis Tovar

April, 2021

To my mother,
And for whom do we achieve extraordinary feats in our lives if not for our mothers?

Abstract

Modern processors provide enhanced performance with reduced power, size and cost in average case and are becoming mainstream in almost all application domains including real-time embedded systems. However, the use of modern computing platforms in hard real-time systems, i.e., systems with stringent timing requirements, is still under scrutiny of the real-time systems community due to their unpredictable nature. This is mainly due to resources such as, caches and the memory bus that are shared among several tasks executing on the processor. As tasks can run concurrently on the processor, consequently, simultaneous use of any of these shared resources can result in *inter-task resource contention* which can significantly affect the timing behavior of the executing tasks. To safely conclude that any task executing on the platform may or may not fulfill its timing requirements, it is essential to first compute accurate bounds on the shared resource contention that may be experienced by that task.

The main objective of this dissertation is to provide software based solutions that can be used to accurately quantify the shared resource contention between tasks due to two main resources, i.e., caches and the memory bus.

We start by identifying the pessimism in the existing analysis that focus on bounding inter-task cache contention for *direct-mapped* caches. We show that this pessimism mainly comes from a unidirectional focus on the *negative* perspective of caches, i.e., derived from a preempting task invalidating cache lines useful to the preempted task, thereby extending a preempted task's execution time. In contrast, we identify a different *positive* perspective of caches, i.e., *cache persistence*, which refers to the re-use of cache content between different job executions of a task, leading to a tighter bound on the total memory access demand of the task. We propose a new preciser analysis that accounts for both the negative and the positive perspective of caches when computing inter-task cache contention, and results in significantly improving task's schedulability.

We then extend our analysis to *set-associative* caches and show that the previously developed analysis for direct-mapped caches can not be used as is for set-associative. We present several different approaches to bound inter-task contention considering set-associative caches. Our analysis accurately determines cache blocks that may suffer additional cache reloads due to inter-task cache conflicts even in the presence of cache persistence and eliminates substantial pessimism with respect to former analyses.

We highlight additional challenges that stem from analyzing inter-task cache conflicts in the presence of a cache hierarchy and propose an analysis to bound inter-task cache contention considering *multilevel* caches. We identify the sources of overestimation in a preceding analysis that focus on bounding inter-task contention for multilevel caches and propose solutions to minimize that overestimation.

Finally, we present a holistic analysis that considers the *interdependence* between *cache contention* and memory *bus contention* and evaluate their cumulative impact on the timing require-

ments of tasks. We show that the analysis that tightly bounds the inter-task cache contention may also result in significantly reducing the memory bus contention suffered by the tasks, thereby, improving schedulability.

Keywords: Hard real-time systems, Shared resources, Cache Contention, Bus contention, Timing analysis.

Resumo

Processadores modernos, em geral, fornecem desempenho aprimorado com energia, tamanho, e custo reduzidos, e estão a se tornar comuns em quase todos os domínios de aplicação, incluindo sistemas embarcados em tempo real. No entanto, o uso de plataformas de computação modernas em sistemas de tempo real rígidos, ou seja, sistemas com requisitos temporais rigorosos, ainda está sob escrutínio da comunidade de sistemas de tempo real devido à sua natureza imprevisível. Isso se deve principalmente a recursos como memórias caches e o barramento de memória que são compartilhados entre várias tarefas em execução no processador. Como as tarefas podem ser executadas simultaneamente no processador, consequentemente, o uso simultâneo de qualquer um desses recursos compartilhados pode resultar em *contenção de recursos entre tarefas*, o que pode afetar significativamente o comportamento de temporização das tarefas em execução. Para concluir com segurança que qualquer tarefa em execução na plataforma pode ou não cumprir seus requisitos de tempo, é essencial primeiro calcular limites precisos na contenção de recursos compartilhados que pode ser experimentada por essa tarefa.

O principal objetivo desta dissertação é fornecer soluções baseadas em programas que possam ser usadas para quantificar com precisão a contenção de recursos compartilhados entre tarefas devido a dois recursos principais, caches e o barramento de memória.

Começamos por identificar o pessimismo nas análises existente que se concentram na limitação da contenção de cache entre tarefas para *caches mapeados diretamente*. Mostramos que esse pessimismo vem principalmente de um foco unidirecional na perspectiva negativa de caches, ou seja, derivado de uma tarefa preemptiva invalidando linhas de cache úteis para a tarefa interrompida, estendendo assim o tempo de execução de uma tarefa interrompida. Em contraste, identificamos uma outra perspectiva positiva de caches, chamada de persistência de cache, que se refere à reutilização do conteúdo do cache entre diferentes execuções de trabalho de uma tarefa, levando a um limite mais rígido na demanda total de acesso à memória da tarefa. Propomos uma nova análise mais precisa que leva em conta as perspectivas negativa e positiva dos caches ao calcular a contenção de cache entre as tarefas e resulta em uma melhora significativa na capacidade de escalonamento da tarefa.

Em seguida, estendemos nossa análise para *memórias-caches de conjuntos associativos* e mostramos que a análise desenvolvida anteriormente para memórias cache diretamente mapeadas não pode ser usada da mesma maneira para conjuntos associativos. Apresentamos várias abordagens diferentes para limitar a contenção entre tarefas, considerando memórias cache de conjunto associativo. Nossa análise determina, com precisão, os blocos de cache que podem sofrer recarregamentos de cache adicionais devido a conflitos de cache entre tarefas, mesmo na presença de persistência de cache, e elimina o pessimismo substancial em relação às análises anteriores.

Destacamos desafios adicionais que resultam da análise de conflitos de cache entre tarefas na presença de uma *hierarquia de cache* e propomos uma análise para limitar a contenção de cache

entre tarefas considerando caches multinível. Identificamos as fontes de superestimação em uma análise anterior que enfoca a contenção de limites entre tarefas para caches multinível e propomos soluções para minimizar essa superestimação.

Finalmente, apresentamos uma análise holística que considera a *interdependência* entre a *contenção do cache* e a *contenção do barramento de memória* e avaliamos seu impacto cumulativo nos requisitos de temporização das tarefas. Mostramos que a análise que limita fortemente a contenção do cache entre tarefas também pode resultar na redução significativa da contenção do barramento de memória sofrida pelas tarefas, resultando assim em uma melhora na escalonabilidade.

Acknowledgments

PhD is a roller coaster ride and no worthwhile roller coaster provides a smooth ride. But this does not mean it cannot be enjoyed, especially, when you are surrounded by a bunch of exceptional people to help, motivate and encourage you. I would start by offering my heartfelt gratitude to my supervisors, Prof. Eduardo Tovar and Dr. Geoffrey Nelissen, for their guidance and support at every step during my PhD. Prof. Eduardo is the person who selected me for the PhD position at CISTER and although it took me almost six months to join CISTER and start my PhD, he persisted with me and allowed me to join. Without his understanding, things might have been very different. He has been an amazingly supportive supervisor during the course of my PhD.

Dr. Geoffrey Nelissen my co-supervisor, is the main force in the transformation of my PhD progress into a growth function. His dedication, sharp insights, attention to detail, and comprehensive assistance has really helped me throughout my research journey. He always strives for perfection and expects the same from his students, which is very inspiring. I have learned a lot from him. I would also like to thank Prof. Luis Almeida for helping with FEUP's Administration.

I would also like to acknowledge the help of Damien Hardy, Benny Akesson, Isabelle Puaut, Sebastian Altmeyer and Robert I. Davis, with whom I have had the privilege of collaborating at the early phase of my PhD.

My fellow students, researchers and administrative staff at CISTER have been supportive in many ways. I would like to thank Giann Nandi for translating the abstract of the thesis in Portuguese. I would also like to mention Harrison Kurunathan, my best lab mate and a very good friend. His company is never boring and we always have great discussions on academic and non-academic issues. During all these years, we have shared some very memorable moments. I would also like to thank Muhammad Ali Awan for his valuable advices on professional and personal matters. I would like to add that I feel fortunate to have known Hazem Ismail Ali, Patrick Meumeu Yomsi, Claudio Maia, Humberto Carvalho, João Loureiro, Shashank Gaur, Mubarak Ojewale and Ishfaq Hussain during these years. I would also like to extend my sincere gratitude to all the administrative staff at CISTER.

I would also like to thank Niaz, Mushtaq, Zahid, Ajmal, Asif, Saad, Alam, and Saqlain for creating an excellent social environment with great parties and delicious food.

Last and the most important, none of this would have been possible without the love, support and patience of my family. My parents have always been an invariable support during my entire educational career. Especially, my mother, Munazza Parveen, she is the motivation behind all my achievements and I owe her what I am right now. I would also like to express my heartfelt gratitude to my sisters for their continuous encouragement during my long research journey. I would also like to thank my uncle Syed Asim Hussain and my aunt Nighat Firdous, who always supported and encouraged me. Finally, I would like to thank my wife and my daughter, who bare with me during the ups and downs of my PhD.

This work was partially supported by FCT (Fundação para a Ciência e Tecnologia) under the individual doctoral grant SFRH/BD/119150/2016.

Syed Aftab Rashid

List of Author's Publications

The following list of publications reflects the results achieved during the development of this dissertation. A significant part of this thesis is compiled from these publications.

Conference Publications

- **Syed Aftab Rashid**, Geoffrey Nelissen, Damien Hardy, Benny Akesson, Isabelle Puaut, and Eduardo Tovar, "*Cache-persistence-aware response-time analysis for fixed-priority preemptive systems*" **(Outstanding Paper Award)** in **ECRTS**, 2016, pp. 262–272. https://ieeexplore.ieee.org/document/7557886

- **Syed Aftab Rashid**, Geoffrey Nelissen, Sebastian Altmeyer, Robert I. Davis, and Eduardo Tovar, "*Integrated analysis of cache related preemption delays and cache persistence reload overheads*" in **RTSS**, 2017, pp. 188–198. https://ieeexplore.ieee.org/document/8277292

- **Syed Aftab Rashid**, Geoffrey Nelissen, and Eduardo Tovar, "*Trading Between Intra- and Inter-Task Cache Interference to Improve Schedulability*" in **RTNS**, 2018, pp. 125–136. https://doi.org/10.1145/3273905.3273924

- **Syed Aftab Rashid** "*Server Based Task Allocation to Reduce Inter-Task Memory Interference in Multicore Systems*" in **FIT**, 2019, pp. 322–327. https://doi.org/10.1109/FIT47737.2019.00067

- **Syed Aftab Rashid**, Geoffrey Nelissen, and Eduardo Tovar, "*Cache Persistence-Aware Memory Bus Contention Analysis for Multicore Systems*" in **DATE**, 2020, pp. 442–447. https://ieeexplore.ieee.org/document/9116265

- **Syed Aftab Rashid**, Geoffrey Nelissen, and Eduardo Tovar, "*Bounding Cache Persistence Reload Overheads for Set-Associative Caches*" **(Outstanding Paper Award)** in **RTCSA**, 2020, pp. 1–10. https://ieeexplore.ieee.org/document/9203583

- 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 **RTNS**, 2021. https://easychair.org/publications/preprint/gdNJ

Journal Publications

- **Syed Aftab Rashid**, Geoffrey Nelissen, and Eduardo Tovar, "*Tightening the CRPD Bound for Multilevel non-Inclusive Caches*" in **IEEE Access** (Under Submission).

- **Syed Aftab Rashid**, Zeeshan Haider, S.M. Chapal Hossain, Kashan Memon, Fazil Panhwar, Momoh Karmah Mbogba, Peng Hud, Gang Zhao, "*Retrofitting low-cost heating ventilation and air-conditioning systems for energy management in buildings*" in **Applied Energy**, 2019, volume. 236, pp. 648-661.

Work-in-Progress and Posters

- **Syed Aftab Rashid**, Geoffrey Nelissen, and Eduardo Tovar, "*Poster Abstract: Cache Persistence Aware Response Time Analysis for Fixed Priority Preemptive Systems*" in **RTAS**, 2016. https://ieeexplore.ieee.org/document/7461347
- **Syed Aftab Rashid**, Geoffrey Nelissen, and Eduardo Tovar, "*Integrating the calculation of preemption and persistence related cache overhead*" in **RTSS**, 2016. https://ieeexplore.ieee.org/document/7809873
- **Syed Aftab Rashid**, Geoffrey Nelissen, and Eduardo Tovar, "*ResilienceP Analysis: Bounding Cache Persistence Reload Overhead for Set-Associative Caches*" in **DCE**, 2019. https://cister.isep.ipp.pt/docs/resiliencep_analysis__bounding_cache_persistence_reload_overhead_for_set_associative_caches/1528/view.pdf
- **Syed Aftab Rashid**, Geoffrey Nelissen, and Eduardo Tovar, "*Towards Timing Analysis of Multi-core Platforms for Hard Real-Time Systems*" in **CPS Week**, 2018. https://cister.isep.ipp.pt/docs/towards_timing_analysis_of_multi_core_platforms_for_hard_real_time_systems/1362/view.pdf
- **Syed Aftab Rashid**, Geoffrey Nelissen, and Eduardo Tovar, "*ResilienceP Analysis: Bounding Cache Persistence Reload Overhead for Set-Associative Caches*" in **ECRTS**, 2019. https://cister.isep.ipp.pt/docs/resiliencep_analysis__bounding_cache_persistence_reload_overhead_for_set_associative_caches/1520/view.pdf
- 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 **RTSS**, 2016. https://ieeexplore.ieee.org/document/9355505

Contents

List of Figures xvi

List of Tables xvii

List of Algorithms xviii

List of Abbreviations xx

1 Introduction 1
- 1.1 Contributions of this Thesis . . . 3
- 1.2 Thesis Structure . . . 4

2 Theoretical Background 6
- 2.1 Real-Time Systems . . . 6
- 2.2 Basic Organization of a Real-Time System . . . 7
  - 2.2.1 Applications . . . 7
  - 2.2.2 Real-Time Operating System (RTOS) . . . 9
  - 2.2.3 Hardware Platform . . . 10
- 2.3 Ensuring Temporal Correctness of a RTS . . . 18
  - 2.3.1 Timing Analysis . . . 18
  - 2.3.2 Schedulability Analysis . . . 20
  - 2.3.3 Caches and Timing Analysis . . . 21
  - 2.3.4 System Bus and Timing Analysis . . . 22
- 2.4 Chapter Summary . . . 23

3 Related Work 24
- 3.1 Intra-task Cache Interference Analysis . . . 24
  - 3.1.1 Must Analysis . . . 26
  - 3.1.2 May Analysis . . . 27
  - 3.1.3 Persistence Analysis . . . 28
  - 3.1.4 Intra-task Cache Analysis for Multilevel Caches . . . 28
- 3.2 Inter-task Cache Interference Analysis . . . 30
  - 3.2.1 CRPD Computation for Single-level Direct-mapped Caches . . . 32
  - 3.2.2 CRPD Computation for Single-level Set-associative LRU Caches . . . 35
  - 3.2.3 CRPD Computation for Multi-level Caches . . . 36
  - 3.2.4 From CRPD to Timing Analysis . . . 39
- 3.3 Other Approaches to Handle Intra- and Inter-task Cache Interference . . . 40
  - 3.3.1 Cache Partitioning and Locking . . . 40
  - 3.3.2 Task Layout Optimization . . . 42

3.3.3 Enhanced Scheduling Models . . . 43
3.4 Memory Bus Contention Analysis . . . 44
3.5 Different Perspective of Caches . . . 46

I Analysis of Single-level Direct-mapped Caches 48

4 Using Cache Persistence to Improve the Bounds on Inter-task Cache Interference 50
4.1 Assumptions on the System Model . . . 51
4.2 Problem Definition . . . 53
4.2.1 Motivational Example . . . 53
4.2.2 Problem Formalization . . . 55
4.3 CPRO-union Approach . . . 57
4.3.1 Computation of Cache Persistence Reload Overhead . . . 57
4.3.2 WCRT Analysis . . . 58
4.4 CPRO Multi-Set Approach . . . 60
4.4.1 Computation of $\rho^{mul}_{j,i}(t)$ . . . 60
4.4.2 Improving the Accuracy of $M^{ecb}_{j,i}$ . . . 63
4.4.3 WCRT Analysis . . . 63
4.5 Static Analysis . . . 64
4.6 Experimental Evaluation . . . 64
4.6.1 Total Utilization . . . 66
4.6.2 Number of Tasks . . . 67
4.6.3 Cache Size . . . 68
4.7 Chapter Summary . . . 70

5 Integrated Analysis of Cache Related Preemption Delays and Cache Persistence Reload Overheads 71
5.1 Problem Formalization . . . 73
5.2 Integrated CRPD-CPRO Analysis . . . 76
5.3 Multi-set Approach to Integrated CRPD-CPRO analysis . . . 79
5.4 Experimental Evaluation . . . 83
5.4.1 Core Utilization. . . . 84
5.4.2 Cache size . . . 87
5.4.3 Block Reload Time (d_{mem}) . . . 87
5.4.4 Task Priority and Memory footprint . . . 89
5.5 Chapter Summary . . . 90

6 Evaluating the Impact of Memory Layout of Tasks on Schedulability 92
6.1 Cache Coloring . . . 93
6.2 Assumptions on the System Model . . . 94
6.3 Cache Interference Aware WCRT Analysis . . . 97
6.4 Bounding Intra-Task Cache Interference . . . 98
6.5 Bounding Inter-task Cache Interference . . . 99
6.5.1 Inter-Task Cache Interference due to CRPDs . . . 100
6.5.2 Inter-Task Cache Interference due to CPROs . . . 104
6.6 Optimizing Cache Color Assignment . . . 107
6.6.1 Working Example . . . 110
6.7 Experimental Evaluation . . . 112

6.8 Chapter Summary 116

II Analysis of Single- and Multi-level Set-associative Caches 117

7 CPRO Analysis for Set-associative Caches 119
7.1 Assumptions on the System Model 120
7.2 Finding PCBs for set-associative caches 122
7.3 CPRO Analysis for Set-Associative Caches 124
7.3.1 PCB-ECB Approach 124
7.3.2 ResilienceP Analysis 126
7.4 Multi-path ResilienceP Analysis 127
7.4.1 Building the CPRO-table 129
7.4.2 Bounding the CPRO 131
7.5 WCRT Analysis 133
7.6 Experimental Evaluation 133
7.7 Chapter Summary 137

8 Tightening the Bound on Inter-task Cache Interference for Multilevel Caches 138
8.1 Assumptions on the System Model 139
8.2 State-of-the-Art CRPD Analysis for Multilevel Caches (Chattopadhyay and Roychoudhury, 2014) 143
8.2.1 Calculating the Indirect Effect of Preemption 145
8.2.2 CRPD Computation 147
8.3 Multilevel Useful Cache Blocks 148
8.3.1 Finding L1/L2-UCBs 149
8.4 Tightening the Bound on the Indirect Effect of Preemption 150
8.4.1 Handling Nested/Multiple Preemptions 153
8.5 Improved CRPD Analysis for Multilevel caches 156
8.5.1 CRPD due to Eviction of L1-UCBs 156
8.5.2 CRPD due to Eviction of L2-UCBs 157
8.5.3 Computation of total CRPD and WCRT Analysis 163
8.6 Experimental Evaluation 164
8.6.1 Deriving Parameters for the Analyses 164
8.6.2 Experiments 165
8.7 Chapter Summary 173

III Extension to Multicore Platforms 174

9 Evaluating the Impact of Inter-task Cache Interference on Memory Bus Contention in Multicore Systems 176
9.1 Assumptions on the System Model 177
9.2 CRPD-aware Memory Bus Contention Analysis 179
9.3 Cache Persistence-aware Memory Bus Contention Analysis 182
9.4 Bus Contention-Aware Worst-case Response Time (WCRT) Analyses 185
9.5 Experimental Evaluation 186
9.5.1 Multicore Platforms with Single-level Caches 187
9.5.2 Multicore Platforms with Multilevel Caches 190

9.6 Chapter Summary 194

10 Thesis Summary, Limitations and Future Directions **195**
10.1 Summary of Contributions 195
10.2 Limitations of Current Work and Future Directions 196
10.2.1 Cache Persistence Analysis for Multilevel Caches 196
10.2.2 Inter-task Cache Interference Analysis for Last-level Shared Caches . . . 197
10.2.3 Holistic Memory Contention Analysis for Preemptive Systems 198
10.2.4 Cache Persistence-aware Inter-task Cache Interference Analysis considering Dynamic Priority Scheduling 198
10.3 Conclusions 199

Bibliography **200**

List of Figures

2.1 Different components of a Real-time system 7
2.2 Common Memory Architecture 12
2.3 Different types of cache associativity 13
2.4 Example access sequence of memory blocks in a 4-way set-associative cache using a LRU replacement policy 14
2.5 Basic abstraction of the system Bus 15
2.6 Work flow between different components of a timing-analysis tool (Wilhelm et al., 2008a) 19
2.7 Basic interface between timing and schedulability analysis 21

3.1 Intra-task ache analysis is one of the main components in the timing analysis (Phavorin and Richard) 25
3.2 Join and Update functions for the Must, May and Persistence analysis 27
3.3 Update function to handle U accesses for multilevel caches (Hardy and Puaut, 2008) 30
3.4 Visual representation of cache related preemption delay (CRPD) 31
3.5 Illustration of the maximum LRU-age of a UCB m_i. The dashes (from left to right) denote the sequence of memory accesses during the execution of task τ_i. 36
3.6 Illustration of the indirect effect of preemption suffered by a memory block m due to eviction of another memory block A by preemption. Both L1 and L2 caches are assumed to be two-way set-associative having only one cache set and the cache replacement policy is LRU. 38
3.7 Example schedule to highlight re-usable cache blocks between different jobs of task τ_i 46

4.1 Schedule and cache contents for a taskset $\{\tau_1,\tau_2\}$ with $C_1 = 100$, $C_2 = 400$, $MD_1 = 60$, $MD_2 = 80$, $ECB_1 = \{5,6,7,8,9,10\}$, $ECB_2 = \{1,2,3,4,5,6\}$, $UCB_1 = \{6,7\}$, $UCB_2 = \{5,6\}$, $PCB_1 = \{5,6,7,8,10\}$ and $PCB_2 = \{1,2\}$. The schedule assumes that τ_1 releases its first job with an offset of 100 time units. 54
4.2 Illustration of the pessimism associated with Equation (4.6) using the task set $\{\tau_1, \tau_2\, \tau_3\}$ when τ_1 and τ_2 releasing their first jobs with an offset. 60
4.3 Illustration of the maximum number of times the tasks in $\text{aff}(i,j)$ and $\text{hep}(j) \setminus \tau_j$ can execute between two successive jobs of τ_j. When calculating $\rho_{2,3}$, $\tau_1 \in \text{hep}(2) \setminus \tau_2$ can release maximally 3 jobs (with each job loading all its ECBs in the worst case). In contrast, the one job released by $\tau_3 \in \text{aff}(3,2)$ can execute and load its ECBs maximum 4 times. 61
4.4 Number of tasksets that are deemed schedulable for a for a varying total utilizations. 67
4.5 Weighted schedulability measure by varying the number of tasks from 5 to 25. . . 68
4.6 Weighted schedulability measure by varying the number of cache sets 69

5.1 Schedules maximizing τ_3's response time when $C_1 = 1$, $C_2 = 2$, $C_3 = 9$, $T_1 = 6$, $T_2 = 6$, $T_3 = 25$, $ECB_1 = \{7,8,9,10\}$, $ECB_2 = \{7,8,9,10\}$, $ECB_3 = \{1,2,3,4,5\}$, $UCB_2 = \{7,8,9,10\}$, $PCB_2 = \{7,8,9,10\}$ and $UCB_1 = UCB_3 = PCB_1 = PCB_3 = \emptyset$ . . . 75
5.2 Illustrating the pessimism associated with the separate UCB-union multi-set and CPRO multi-set analysis using the task set $\{\tau_1, \tau_2, \tau_3\}$ with $C_1 = 1, C_2 = 2, C_3 = 5$, $T_1 = 3$, $T_2 = 5$ and $T_3 = 20$. . . . 80
5.3 Schedulability ratio with respect to total core utilization . . . 86
5.4 Weighted schedulability measure by varying cache utilization, block reload time d_{mem} and cache size . . . 88

6.1 A visual representation of cache coloring (Kim et al., 2013) . . . 94
6.2 Increase in execution demand and memory access demand of task τ_i due to reduction in number of cache colors assigned to τ_i. . . . 98
6.3 Worst-case memory access demand $MD_i[k_i]$ of task τ_i w.r.t the number of cache colors assigned to τ_i. . . . 102
6.4 Variation in the worst-case and residual memory access demand of task τ_j w.r.t the number of cache colors assigned. . . . 106
6.5 Different cache color assignments of task set in Table 6.2. . . . 112
6.6 Schedulability w.r.t core utilization and cache size . . . 114
6.7 Schedulability w.r.t number of cache sets per color and number of tasks . . . 115

7.1 Example execution of a task τ_i (from left to right) considering (a) a direct-mapped cache with 4 cache sets, i.e., $\{S_0, S_1, S_2, S_3\}$ and (b) a 4-way set-associative cache having one cache set S_0 using a Least-Recently-Used (LRU) cache replacement policy. The LRU age of a block b refers to how many accesses were performed to the cache set in which b is saved since the last access to b. . . . 120
7.2 Maximum LRU-age of memory blocks of task τ_i (a) over the execution of two jobs of τ_i, and (b) under the assumption that τ_i is cyclic . . . 124
7.3 Example scenario to highlight the pessimism in the PCB-ECB approach . . . 126
7.4 Highlighting the pessimism in the ResilienceP analysis . . . 128
7.5 Task sets schedulability by varying (a) total task set utilization and (b) the total number of tasks in a task set . . . 132
7.6 Weighted schedulability results by varying (a) number of cache ways W and (b) memory reload time d_{mem} . . . 134
7.7 Performance of ResilienceP and multi-path ResilienceP analysis w.r.t the number of execution paths . . . 136

8.1 Highlighting the pessimism in the calculation of indirect effect of preemption by (Chattopadhyay and Roychoudhury, 2014). . . . 150
8.2 Multiple preemption scenarios with collaborating and isolated preemptions. The indirect effect of preemption suffered by memory block m due to consecutive preemptions, i.e., at P_1 and P_2, is higher than the indirect effect caused by individual preemptions. . . . 155
8.3 Example scenario to demonstrate the pessimism of (Chattopadhyay and Roychoudhury, 2014) when calculating the CRPD due to L2 cache misses resulting from preemption. . . . 158
8.4 Number of task set deemed schedulable by varying total task set utilization . . . 167
8.5 Wighted schedulability measure by varying the total number of tasks in a task set 168

8.6 Weighted schedulability measure by varying number of ways in the L1 cache. The number of ways in the L2 cache were set to 32, i.e., $W_2 = 32$ 169
8.7 Weighted schedulability measure by varying number of ways in the L2 cache . . 170
8.8 Weighted schedulability measure by varying number of sets in the L1 cache. The number of sets in the L2 cache were fixed to 512, i.e., $|\mathbb{S}_2| = 512$ 171
8.9 Weighted Schedulability measure by varying number of sets in the L2 cache. The number of sets in the L1 cache were set to their default value, i.e., $|\mathbb{S}_1| = 32$ . . . 171
8.10 Weighted schedulability results by varying d_{L1} and d_{L2} 172

9.1 Execution of task τ_1 and τ_2 on core π_x and task τ_3 on core π_y. Task parameters of interest are: PD_1=PD_3 = 4, PD_2= 32, MD_1=MD_3 = 6, MD_2 = 8, MD^r_1=MD^r_3 = 1, ECB_1=ECB_3 = $\{5,6,7,8,9,10\}$, ECB_2 = $\{1,2,3,4,5,6\}$, PCB_1=PCB_3 = $\{5,6,7,8,10\}$ and UCB_2 = $\{5,6\}$. 182
9.2 Schedulability ratio of different bus arbitration policies by varying total core utilizations 186
9.3 Wighted schedulability measure by varying the total number of cores 187
9.4 Wighted schedulability measure by varying the value of memory reload time d_{mem} 188
9.5 Wighted schedulability measure by increasing cache size between 2kB to 32kB . 189
9.6 Wighted schedulability measure by varying the RR/TDMA slot size (sl) 190
9.7 Schedulability ratio of different bus arbitration policies by varying total core utilizations for multicore architectures with two-level caches. 191
9.8 Wighted schedulability measure by varying the total number of cores in multicore platforms with two-level caches 192
9.9 Wighted schedulability measure by varying the RR/TDMA slot size (sl) for multicore platforms with two-level caches. 193

10.1 Cache persistence-aware analysis of multiple cache levels may lead more tighter WCRT bounds. 197
10.2 Under preemptive scheduling, simultaneous analysis of intra- and inter-core cache interference is a challenge. 198

List of Tables

3.1 Categorization of memory references . . . 26
3.2 Computation of CAC of a memory reference r at cache level L (Hardy and Puaut, 2008) . . . 29

4.1 List of important symbols used in Chapter 4 . . . 52
4.2 Task parameters for a selection of benchmarks from the Mälardalen Benchmark Suite (Gustafsson et al., 2010) . . . 66

5.1 List of important symbols used in Chapter 5 . . . 72
5.2 Task parameters for the benchmarks used during the experiments . . . 85
5.3 Relative gain μ_4^{gain} for the CRPD-CPRO union and multi-set approaches by increasing the number of ECBs of τ_1 . . . 90

6.1 List of important symbols used in Chapter 6 . . . 95
6.2 Task set parameters used in the working example . . . 110

7.1 List of important symbols used in Chapter 7 . . . 121
7.2 CPRO-table for every PCB m_j of task τ_j . . . 129

8.1 List of important symbols used in Chapter 8 . . . 140
8.2 Benchmarks parameters from the Mälardalen Benchmark Suite (Gustafsson et al., 2010) used during the experimental evaluation . . . 166

9.1 List of important symbols used in Chapter 9 . . . 178

List of Algorithms

6.1 Simulated annealing based algorithm to optimize cache color assignment of tasks 108

7.1 Building the CPRO-table for PCB m_j of task τ_j 130

7.2 Computing the total CPRO of task τ_j in a time interval of length t 131

8.1 Calculating the indirect effect of preemption caused due to preemption of task τ_i by τ_j at a program point P 151

8.2 Calculating the indirect effect of preemption that can be suffered by all memory blocks used by task τ_i when considering multiple preemptions by higher priority tasks in $\in$ hp(i) w.r.t preemption point P 156

8.3 Algorithm to calculate the total CRPD cost due to eviction of L2-UCBs of task τ_i w.r.t a preemption point P 163

List of Abbreviations

AI	Abstract Interpretation
ACS	Abstract Cache State
ACU	Air-bag Control Unit
AH	Always-Hit
AM	Always-Miss
CAST	Certifications Authorities Software Team
CAC	Cache Access Classification
CFG	Control Flow Graph
CHCM	Cache Hit/Miss Classification
COTS	Commercially available Off-The-Shelf
CPRO	Cache Persistence Reload Overhead
CPU	Central Processing Unit
CRPD	Cache Related Preemption Delay
CSTG	Cache State Transition Graph
DC-UCB	Definitely-Cache Useful Cache Block
DJP	Dynamic Job Priority
DM	Deadline Monotonic
DRAM	Dynamic Random Access Memory
ECB	Evicting Cache Block
EDDP	Earliest Deadline Deferrable Portion
EDF	Earliest Deadline First
FCFS	First-Come First-Serve
FEUP	Faculdade de Engenharia da Universidade do Porto
FIFO	First-In First-Out
FJP	Fixed Job Priority
FM	First-Miss
FP	Fixed Priority
FPPS	Fixed Priority Preemptive System
FSB	Front-side Bus
FTP	Fixed Task Priority
HRTS	Hard Real-Time System
HVAC	Hard Ventilation and Air-conditioning

ILP	Integer Linear Programming
IMA	Integrated Modular Avionics
IPET	Implicit Path Enumeration Technique
LLF	Least Laxity First
LLC	Last Level Cache
LRU	Least Recently Used
MCP	Multi-Core Processor
MIPS	Microprocessor without Interlocked Pipelined Stages
MMU	Memory Management Unit
MRTA	Multicore Response Time Analysis
NC	Not-Classified
nPCB	non-Persistent Cache Block
OS	Operating System
PCB	Persistent Cache Block
PLRU	Pseudo Least Recently Used
PS	Persistent
RAM	Random Access Memory
RM	Rate Monotonic
ROM	Read Only Memory
RR	Round Robin
RT	Real Time
RTES	Real-Time Embedded System
RTOS	Real-Time Operating System
RTS	Real-Time System
SA	Simulated Annealing
SMART	Strategic Memory Allocation for Real-Time
SRTS	Soft Real-Time System
TDMA	Time Division Multiple Access
UCB	Useful Cache block
WCET	Worst-Case Execution Time
WCRT	Worst-Case Response Time
WSS	Working Set Size

Chapter 1

Introduction

In recent years, embedded systems have become an integral part of our everyday lives. These systems interact with the environment and perform a set of dedicated operations. An embedded system can be formally defined as a system composed of hardware, software and/or mechanical components to perform a dedicated function or a range of functions (Kamal, 2011). These dedicated functions vary from a simple task of toasting a slice of bread to an air traffic control system that involves numerous workstations, networks and radar sites. Nowadays, the use of embedded systems span across several domains including consumer electronics, medical equipment, avionics, automotive industry, banking, and defense industry etc.

An embedded system is different from a general purpose computer system that is designed to satisfy a variety of end-user requirements. A general purpose computer is usually designed to make the average case faster and is end-user configurable, whereas, the set of operations to be performed by an embedded system are usually known a priori at design time. In this sense, an embedded system is custom-made for a specific application and is subjected to concerns regarding functional and non-functional requirements of that application.

The primary requirement of an embedded system is to correctly perform a desired functionality. However, there are embedded systems that have an additional constraint of *temporal* correctness to be met on top of the functional requirements of the system. In the scientific literature, this kind of systems are refereed to as *Real-Time Embedded Systems* (RTES) or simply *Real-Time Systems* (RTS). Real-time systems are defined as systems in which the correctness of the system behavior depends not only on the logical result of the computation, but also on the *time* at which the results are produced (Stankovic, 1988). Applications of RTS can be found in many industrial domains where timeliness is important. For example, an airbag controller system in a car is not only responsible to decide whether or not to inflate the airbags, but also to ensure that the airbags will be inflated in a timely manner, i.e., before causing an injury to the driver. Similarly, in airplanes the flight control system is responsible for a timely compensation of all external disturbances that may affect a stable flight operation (Davis et al., 2018a; Baufreton et al., 2020). Many other examples of RTS can be found in space, transportation and control industry (Cecere et al., 2016; Koo and Kim, 2018).

Functionalities of a RTS are managed by a set of entities called processes or *tasks*. Each task has a timing constraint associated to it which represents the time before which that task must perform its assigned operations. As timing behavior is one of the most important property of a RTS, ensuring timing correctness for every task in the system is of utmost importance in order to prove the timing correctness of the complete system. However, due to the massive technological advancements, tasks are often executed on complex performance oriented software and hardware architectures. These modern software/hardware platforms often produce significant performance improvements but at the cost of an increased complexity of tracing and analyzing the system. For example, the use of cache memories significantly improve the performance of modern processors however, in systems with cache memories, the completion time of tasks may vary depending on the availability of content in the cache. Therefore, to ensure that a task satisfies its specified timing requirements it is important to analyze the behavior of all software/hardware components, e.g., caches, pipelines, interconnects, main memory and I/O devices, that can impact the execution of that task.

Indeed, analyzing the temporal behavior of all these performance oriented software/hardware components is every challenging due to very brief design documentation provided by the hardware vendors. The analysis complexity is even more amplified in multi-tasking systems where different tasks executing on the platform may share resources such as cache, main memory, I/O devices and interconnects. As a result, the temporal behavior of tasks is significantly affected due to *contention* in accessing the shared resources. It has been identified in (Authority, 2016) that sharing of resources, such as caches and interconnect (i.e., usually a bus) among tasks executing on a modern processing unit makes the temporal and functional behavior of the system highly complex and interdependent. This highlights the need of an analysis framework that provides a holistic solution by considering the impact of shared resource contention on the timing behavior of applications comprising a RTS. The analysis should also provide both safe and precise bounds on the shared resource contention that may be experienced by all tasks in the system in order to accurately conclude that the overall system may or may not fulfill its timing requirements.

A safe bound on the shared resource contention means that the values returned by the analysis will always be larger than or equal to the shared resource contention that may occur at run-time under any possible scenario. While a precise bound is the one whose values are as close as possible to the actual shared resource contention that may be experienced at run-time. Unfortunately, most of the existing works in literature that focus on bounding the shared resource contention prioritize the safety aspect and often lead to the situation where the conclusion of the analysis is that the system does not comply with its timing requirements, while in reality it indeed does. For example, in systems where tasks are scheduled *preemptively*, i.e., task's execution can be temporarily interrupted, shared resources such as cache memory is viewed from an exclusively *negative* perspective, i.e., derived from a preempting task invalidating cache lines useful to the preempted task, thereby extending a preempted task's execution time. This increase in the execution time of the preempted task is due to the *inter-task cache interference* suffered by the preempted task due to cache conflicts with the preempting task and is often referred to as *cache related preemption delay*

(CPRD). Several works have been proposed in literature (Lv et al., 2015; Maiza et al., 2019) that account for CRPDs when analyzing systems that use preemptive task scheduling. However, a different *positive* perspective of cache memory is often overlooked in the existing literature, which refers to the cache re-use between different job executions of a task. For example, considering multiple jobs of a particular task; the next job of the task can benefit from the presence in cache of memory blocks that were loaded by a previous job of the same task and that have remained in the cache until the next job executes and can make use of those blocks. Analysis of cache re-use can be used to significantly reduce pessimism in the computation of inter-task cache interference from multiple jobs of a preempting task that can execute during the response time of the preempted task.

Another important problem that has not been fully addressed in the existing analysis on shared resource contention is to consider the dependency between the behavior of different shared resources. For example, most of the existing works in the state-of-the-art that focus on bounding contention due to shared interconnects (or memory bus) are based on the assumption of non-shared caches (Dasari et al., 2011a, 2016) or consider a fix number of memory bus requests (Schliecker and Ernst, 2011; Kim et al., 2014, 2016) that can be generated by a task during its execution. Both these approaches can lead to pessimistic/optimistic bounds considering the fact that the actual number of main memory (or bus) requests of tasks depend on the cache misses suffered by the tasks which, in turn, depends on the inter-task cache interference experienced by tasks during their executions. Therefore, to effectively bound memory bus contention, it is important to develop holistic timing analysis techniques that consider the interference caused by both caches and memory bus and evaluate their cumulative impact on the timing properties of tasks.

Building on the above observations, the high-level goal of this thesis is to provide solutions that can be used to accurately quantify the shared resource contention between tasks due to two main resources, i.e., caches and the memory bus.

1.1 Contributions of this Thesis

In support of this thesis, the following contributions are made.

- **Accurately quantify inter-task cache interference for direct-mapped caches**
  We identify substantial pessimism in the existing analysis that focus on bounding inter-task cache interference for direct-mapped caches. This pessimism mainly comes from a unidirectional focus on the negative perspective of caches, i.e., CRPD. We propose a new preciser model that accounts for both the negative perspective, i.e., CRPDs, and the positive perspective of caches, i.e., *cache persistence*, when computing the inter-task cache interference of tasks. Cache persistence refers to the re-use of cache content between different jobs of the same task. This allows to capture re-usable cache blocks between different job executions and neutralizes the negative impact of CRPDs in systems that allow preemptive task executions. We prove the correctness of this new model and propose a static program analysis to derive the parameters required by the analysis. Furthermore, we also show how to in-

corporate bounds on the total inter-task cache interference in the schedulability analysis for fixed-priority preemptive systems (FPPS).

- **Accurately quantify inter-task cache interference considering set-associative caches**
  We improve the bounds on inter-task cache interference for set-associative caches by adapting the notion of cache persistence to set-associative caches. First, we show that the previously developed analysis for direct-mapped caches can not be used as is for set-associative caches and may lead to optimistic results. We then present a new analysis that accurately determines cache blocks that may suffer additional cache reloads due to preemptions even in the presence of cache persistence. We also provide an overview of the static program analysis techniques that are used to derive the parameters needed to adapt persistence-aware cache contention analysis to set-associative caches.

- **Bounding inter-task cache interference for multilevel caches**
  We show that the literature on the computation of inter-task cache contention for multilevel caches is relatively scarce due to the additional challenges that stem from analyzing inter-task cache conflicts at different cache levels. Few existing analysis that focus on bounding inter-task contention for multilevel caches are very pessimistic as they overestimate the number of times cache blocks can be evicted from a particular cache level and therefore needed to be reloaded from the main memory. We improve on the existing analysis by accurately determining which cache blocks can be impacted due to inter-task conflicts at a particular cache level and how many times these cache blocks can be evicted and reloaded from the main memory. We also prove the correctness of the new analysis and provide a static analysis approach to obtain parameters needed by the analysis.

- **Cache interference-aware memory bus contention analysis**
  We present a holistic overview of the relationship between inter-task cache contention and the memory bus contention suffered by the tasks. We show that the memory bus contention that can be suffered by a task during its execution strongly depends on the number of cache misses suffered by that task, which, in turn, depends on the inter-task cache interference experienced by the task. Evaluations show that the analysis that tightly bound the inter-task cache contention also results in a more accurate bound on the memory bus contention suffered by tasks, which results in improving schedulability.

1.2 Thesis Structure

The thesis is organized as follows: Chapter 2 provides the necessary background needed for the understanding of this thesis. The related work presented in Chapter 3 briefly explains the existing approaches in the state-of-the-art that are in-line with the problems addressed in this dissertation. The main contribution of the thesis are then divided into three parts.

Part I focus on the analysis of inter-task cache contention for single-level direct-mapped caches and comprises Chapter 4, 5 and 6. In Chapter 4, we formally introduce the notion of cache per-

sistence and use it to compute a tighter bound on the inter-task cache interference for single-level direct mapped caches. The key focus of Chapter 5 is to integrate the computation of CRPD with the computation of persistence related cache overheads. The integrated analysis provides a tighter bound on the total inter-task cache interference in comparison to the analysis in Chapter 4. In Chapter 6, we evaluate the impact of memory layout of tasks on inter-task cache interference and on schedulability.

Part II comprises Chapter 7 and 8 and focus on the analysis of inter-task cache interference considering single-level and multilevel set-associative caches. Chapter 7 provides solutions that analyze the impact of cache persistence on the schedulability of tasks considering set-associative caches and presents different approaches to compute persistence related cache overheads for set-associative caches. In Chapter 8, we present the CRPD analysis for multilevel caches that provides a tighter bound than the existing analysis in the state-of-the-art.

In Part III we present a holistic overview of the shared resource contention in modern computing platforms by focusing on the relationship between cache contention and memory bus contention. It comprises Chapter 9 that evaluates how inter-task cache interference can impact the contention due to sharing of memory bus in modern processors and what is their cumulative affect on schedulability. Finally, Chapter 10 concludes this thesis by providing some future research directions.

Chapter 2

Theoretical Background

2.1 Real-Time Systems

Real-time systems fall into the category of embedded systems in which ensuring timing correctness of the system is of utter importance. A real-time system runs several real-time processes that are triggered in a sporadic/aperiodic fashion. A real-time *process* is a software entity that is executed by the processing unit in a parallel/sequential fashion and has a timing constraint associated to it (Buttazzo, 2011). This constraint on the timing is commonly known as the *deadline*, which represents the time before which a process should complete its execution to not cause any damage to the system (Buttazzo, 2011). It is important to note that in this thesis, the term *task* is used as synonym of process.

Depending on the consequences of a missed deadline, real-time systems are broadly categorized in the following two categories:

- **Hard Real-Time systems (HRTS) :** Hard Real-Time systems are the class of RTS in which missing the deadline may cause catastrophic consequences on the system under control, surrounding environment or people. In HRT systems, the results obtained after a given time interval (or deadline) are considered useless. One such example can be the air-bag control unit (ACU) in modern cars. In case of an accident, the ACU must be able to inflate the air bags within 60-80 milliseconds, otherwise the persons inside the car are at a risk of an injury which can be severe.

- **Soft Real-Time systems (SRTS) :** In contrast to HRT systems, in soft real-time system, missing a deadline might still have some utility for the system, although causing a performance degradation. In SRT systems, missing a deadline does not have dire consequences. For example, a degradation in the quality of the on-line audio/video streaming applications is annoying but not life threatening.

This work focuses on HRT systems.

2.2 Basic Organization of a Real-Time System

A basic RTS is composed of three main components, i.e., applications, software module to run the applications (i.e., a real-time operating system (RTOS)) and the underlying hardware platform. In the context of this work, we briefly discuss the basic functionality of the components involved in the design of RTSs.

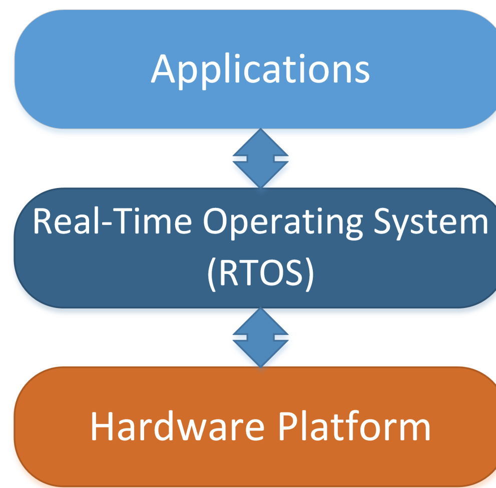


Figure 2.1: Different components of a Real-time system

2.2.1 Applications

Real-Time applications are an abstract representation of the workload used while analyzing a real-time system. The functionality of a RT application is usually modeled as a collections of finite, simple and repetitive abstract entities called real-time tasks (Baruah and Goossens, 2004). These real-time tasks are recurrent in nature where each instance of the task is called a *job*. All jobs related to a particular task are semantically related and represents a basic unit of work that executes on the physical hardware platform (Liu). In the context of this thesis, the functionality of a RT application is represented as a set Γ of n tasks called a *task set*, i.e., $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$. Depending on the frequency with which a task releases its jobs, it can be categorized into the following three types:

- **Periodic Task:** A task is *periodic* if it releases its jobs periodically, i.e., the time interval between different jobs of the task is constant. The fixed time between two consecutive job releases of the task is called the *period* of the task.
- **Sporadic Task:** A task that releases its jobs at some arbitrary time instant however, consecutive jobs of the task are separated by a minimum inter-arrival time is called a sporadic task.
- **Aperiodic Task:** An aperiodic task can release its jobs at any arbitrary time instant and their activations are not regularly interleaved.

In this work, we focus on sporadic tasks.

Any real-time task $\tau_i \in \Gamma$ can be formally defined by several parameters that can be *static*, i.e., set before executing the task and do not change during run-time of the system, or *dynamic*, i.e., task parameters that may change during the execution of the task. In the context of this work, general parameters used to define a task τ_i are:

- C_i: *Worst-case execution time (WCET)* – The maximum amount of time required by any instance or job of task τ_i to complete its execution without any interruptions.
- T_i: *Minimum inter-arrival time or period* – The minimum inter-arrival time between two consecutive instances or jobs of task τ_i.
- D_i: *Deadline* – The time before which task τ_i should be completed in order to avoid any damage to the system.
- U_i: *Utilization* – Utilization of a task is defined as the fraction of the processor time required by the task. Utilization U_i of a task τ_i is given by $U_i = C_i/T_i$.
- R_i: *Worst-case response time (WCRT)* – The maximum value of the difference between the arrival time and completion time amongst all instances or jobs released by task τ_i.

It is important to note that deadlines are relative to the nature of the application. For example, the air-bag control application installed in a car might have a relative deadline of 60-80 milliseconds to inflate the air-bags, whereas a room temperature monitoring application can have a relative deadline of a few seconds to change the temperature on the HVAC thermostat. Depending on the relation between the deadline D_i and period of a task τ_i, τ_i can be categorized into following three classes:

- **Implicit deadline task:** The deadline of a task τ_i is equal to the minimum inter-arrival time between two jobs of τ_i, i.e., $D_i = T_i$.
- **Constrained deadline task:** The deadline of a task τ_i is less than or equal to the minimum inter-arrival time between two jobs of τ_i, i.e., $D_i \leq T_i$.
- **Arbitrary deadline task:** The deadline of a task τ_i can be less than, equal to, or greater than the minimum inter-arrival time between two jobs of task τ_i.

In this work, we focus on tasks with constrained deadlines.

A RT application usually consists of one or more RT tasks working together to achieve a certain functionality. However, these tasks can have precedence constraints and data dependencies between them. For example, in the air-bag control application, a *sensor data acquisition* task must always be executed before the *inflate air bags* task in order to have the most recent value of the intensity of the impact in case of an accident. Similarly, depending on the nature of the application

different tasks can have different precedence constraints or data dependencies. However, in this work, we only focus on *independent* tasks, i.e., tasks that can be executed without ensuring any precedent constraints. Also, these tasks do not depend on the outcome of any other task in order to initiate their execution.

In this work, we only focus on independent tasks.

2.2.2 Real-Time Operating System (RTOS)

In general, an operating system (OS) performs basic operations such as memory management, process scheduling, inter-process communication and Input/Output management and a RT operating system (RTOS) is no different. However, the most important functionality of a RTOS is to provide reliability and predictability in the system. Reliability refers to the ability of the system to perform its required functions under stated conditions for a specified period of time (Deck, 1998). Whereas, predictability means the ability of the system to guarantee the timing properties at design time. Few examples of RTOS include RTEMS, VxWorks and PikeOS.

As discussed earlier a RTOS can perform many functionalities, however, in this section we limit our discussion to task management or scheduling function of the RTOS.

Every RT task needs to use the hardware platform at some point in time to achieve its desired functionality, which is mainly performed by requesting some execution resources from the processing element (i.e., processor) present at the hardware level. When a single processor has to execute a set of tasks that can overlap in time, the RTOS has to allocate the processor to each task based on a predefined criteria. This functionality is achieved by a specialized service of the operating system kernel called the *scheduler*. Scheduler is responsible for deciding which task (or job of the task) should be executing at any particular time and the set of rules that determine the order in which tasks are executed on the processor is called a *scheduling algorithm*. Scheduling algorithms can be categorized into many classes based on different factors, i.e., off-line or on-line, preemptive or non-preemptive, static or dynamics etc. However, in this thesis we primarily focus on the *priority driven scheduling algorithms*.

When using priority driven scheduling algorithms tasks/jobs are executed based on their priorities. These priorities can be assigned to tasks or jobs based on different criterion such as deadlines, arrival rate, execution time and laxity etc. Priority driven scheduling algorithms can be further divided into following categories:

- **Fixed Task Priority (FTP):** As the name suggests, in fixed task priority scheduling algorithms, priorities are assigned to tasks and the priority of all instances of a task (i.e., all its jobs) is the same and remains fixed throughout the execution of the task. Prominent examples of FTP based scheduling algorithms are Rate-Monotonic (RR) (Liu and Layland, 1973) and Deadline-Monotonic(DM) (Leung and Whitehead, 1982) algorithms.

- **Fixed Job Priority (FJP):** In this category, priorities are assigned to jobs rather than tasks, meaning that different jobs of the same task may execute on the processor with different

priorities. However, the priority of a job does not change between its release time and deadline. Examples of such algorithms include Earliest-Deadline First (EDF) (Liu and Layland, 1973), Earliest Deadline Deferrable Portion (EDDP) (Kato and Yamasaki, 2008) and EDF with C = D (Burns et al., 2012).

- **Dynamic Job Priority (DJP):** In dynamic job priority based scheduling algorithms, priority of a job can change dynamically at any instant during its execution. Least-laxity first (LLF) (Mok, 1983) is an example of DJP based scheduling algorithms. In LLF, priority of a job depends on the job's laxity (its deadline minus its remaining execution time). A job with the minimum laxity is allocated the highest priority and vice versa.

This work focuses on priority based scheduling algorithms and in particular use fixed task priority based algorithms such as RM and DM.

Independent of the priority assignment used, a scheduler can either be preemptive or non-preemptive. In preemptive schedulers, a preemption occurs when the execution of a job on a processor is suspended in order to execute another higher priority job. Whereas, non-preemptive schedulers allow a job to complete its execution once started without any interruption. In preemptive scheduling, the process of preempting the job of one task and activating the other involves a switch of the job execution context potentially inducing an extra overhead as the preempted job has to save its status to resume its execution later in time. In literature, this overhead is usually referred to as the *preemption overhead*. Several techniques have been introduced in literature to effectively bound preemption overheads and consider their affect when analyzing the system. Some prominent approaches presented in this context will be discussed later in Chapter 3.

This work focuses only on preemptive schedulers.

2.2.3 Hardware Platform

A hardware platform is a set of physical components on which a RT application executes to achieve its desired functionality. In RTS, a basic hardware platform typically consists of a processing unit or processor (to perform computations), memories (main memory and caches to load/store instructions/data), I/O devices (to perform input/output operations) and an interconnect or system bus (to transfer instructions/data between processor and the main memory). Below we briefly explain the functionality of hardware components that are most relevant in the context of this thesis.

2.2.3.1 Processors

The Central Processing Unit (CPU) or processor is the main electronic circuitry within a system that executes instructions that make up a RT task. It performs basic arithmetic, logic, controlling, and input/output (I/O) operations specified by tasks executing on the processor. A *single-core* processor has only one on-chip CPU or processing core and is capable of executing only a single

task at a time. A *multi-core* processor (MCP) is an integrated circuit with a set of independent processors (or cores) fabricated on a single chip. Typically, a MCP can have two to eight cores on a chip and is capable of executing multiple tasks in parallel.

On a MCP, the scheduler can schedule tasks on any of the available processors. Efficiently scheduling tasks on a MCP is complex in comparison to scheduling them on a single processor since different jobs of a task can be scheduled to execute on any one of the available processors. This phenomenon of suspending the execution of a job from one processor and later resuming it on another processor is called *migration*. Based on whether migration is allowed or not multi-processor scheduling algorithms can be broadly categorized into three main classes.

- **Global Scheduling Algorithms:** In global scheduling, tasks/jobs are allowed to migrate from one processor to another. All tasks within the system are maintained in a single global ready queue and m high priority tasks in the ready queue are allocated to the m available processors. Task assignment to processors is not static and hence, a task may start its execution on one processor but as a result of preemptions may later resume its execution on another processor.

- **Partitioned Scheduling Algorithms:** In partitioned scheduling migrations are not allowed. A given task-set is distributed among the processors based on some criterion, e.g., first-fit, best-fit, next-fit etc. This task to processor assignment is static and tasks/jobs can only execute on the assigned processors. In partition scheduling, the most important phase is task to processor mapping and once the mapping is done any uniprocessor scheduling algorithm can be used to schedule tasks on individual processors.

- **Semi-partitioned Scheduling Algorithms:** Semi-partitioned scheduling algorithms are a combination of global and partitioned scheduling approaches. At first, some tasks from a given task-set are assigned to specific processors and are not allowed to migrate. Remaining tasks (those that can not be mapped to a specific processor) are split between processors effectively allowing them to migrate from one processor to another. Detailed survey on multi-processor scheduling algorithms can be found in (Davis and Burns, 2011a).

In the context of this work, we use partitioned task-level fixed priority scheduling algorithms to schedule tasks on a MCP.

2.2.3.2 Cache Memory

Memories are an essential components of an embedded RTS. Where processors are used to make the computations fast and efficient, memories, e.g., off-chip Random Access Memories (RAM) or non-volatile Read Only Memory (ROM), are required to effectively manipulate instructions and data. However, the techniques in designing memory systems did not catch up with the processor speeds and hence, memory access latencies were non-negligibly high leading to large processor stalls. To bridge this gap between processor and main memory operating speeds, hardware platforms used in modern RT embedded systems employ on-chip cache memories or caches.

Caches are high speed memories that reside between the processor and the main memory and hold data/instructions that can be used by the processor in a speedy manner. Depending on the kind of resources they store, caches can be categorized as, e.g., *instruction* caches, *data* caches or *unified* caches. As the name suggests, instruction caches are used only to hold instructions, similarly, data caches are used for data only. A unified cache can hold both data and instructions. The rationale behind the need for caches is that frequently accessed data/instructions must be kept closer to the processing source or in other words can be “cached", to reduce processor stall cycles. Assuming an empty cache, the first access to a particular address results in a *cache miss* (when the required data/instruction is not in the cache). Therefore, the required data or instruction is fetched from the off-chip main memory and a copy is also stored on the local caches. On subsequent accesses to the same address, the cache is checked and if the required data/instruction is found (called a *cache hit*), it is retrieved from the cache itself without incurring the (high) latency to fetch that data/instruction all the way from the main memory.

Cache Organization

In a basic RTS architecture, cache are organized in a stacked hierarchy. Figure 2.2 shows a basic memory architecture that depicts a trade-off between size and speed. The processor is at the top of the hierarchy with very high operating speeds followed by two layers of caches and then the main memory. The cache that is closest to the processor is the fastest and it is called level one or L1 cache. L1 caches can be further divided into *independent* instruction and data caches as shown in Figure 2.2. L1 caches can have a typical capacity of upto 32 KB with an access latency of 1-2 cycles. Level two or L2 cache is only queried if the required data/instructions are not available in L1 caches. L2 caches are usually *unified* caches, i.e., capable of holding both instructions and data, with a storage capacity ranging from hundreds of KB to several MB. Access latency of an L2 cache is typically around 10 cycles. Some modern high performance processors may also have a

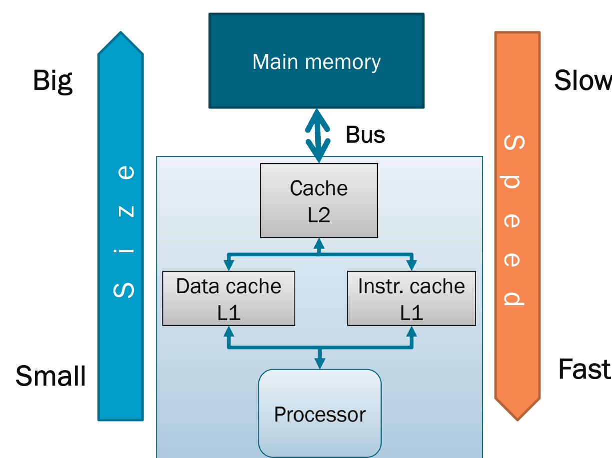


Figure 2.2: Common Memory Architecture

level three, i.e., L3 cache, to further increase the storage capacity. If the required data/instructions

are not available in the last level cache, it results in accesses to the main memory using the off-chip bus as shown in Figure 2.2. These accesses to main memory can cause a delay in the order of hundreds of cycles.

In most processors, the unit for cache access is called a *cache line*, i.e., the smallest unit of data/instructions that can be transferred to or from a cache. Cache line size signify the minimum amount of data the cache must read or write from the main memory or from the cache-level below it. Accessing one element within a cache line causes the whole cache line to be loaded into the cache. As a result, a following access to another element of the same cache line might also results in a cache hit.

Caches are usually partitioned into different sets of equal sizes, i.e., *cache sets*, where each cache set may contain one or more cache lines. A *memory block*, i.e., the smallest amount in bytes which can be loaded at a time from the main memory, is first mapped onto a cache set and 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 number of cache *ways* or the *associativity* of the cache and such a cache is called a *set-associative* cache. There are two special cases of set-associative caches:

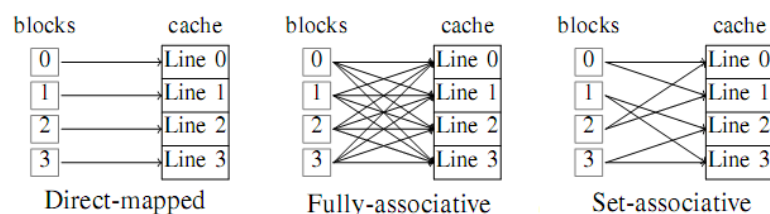


Figure 2.3: Different types of cache associativity

- **Direct-mapped caches:** In direct-mapped caches, the number of cache ways or associativity is 1, i.e., each cache set consists of a single cache line, this means that a memory block can reside in exactly one cache line.

- **fully-associative caches:** In fully-associative caches, the number of cache ways or associativity is equal to the number of sets in the cache, i.e., the cache consist of a single set, this means that a memory block can reside in any cache line.

Figure 2.3 shows the different type of caches based on the mapping of cache lines to main memory blocks.

Contents in the cache should be consistent with the main memory. This is usually done based on the write policy used by the cache. Write policy determines at what time the modified cache line will be written back into the main memory. Based on the write policy, caches can be categorized as *write-through* or *write-back*. For a cache using the write-through policy, main memory is made consistent with the cache immediately after a cache line is modified. Alternatively, in a write-back cache the process of updating the main memory is differed to a later time, until the given cache line is evicted.

Note that in this work we only focus on instruction references therefore, we do not make any assumption on the write policy used by the caches.

Cache Replacement Policy

When loading a memory block from the main memory to cache, processor first determines the cache set the block maps to. A lookup is performed to find the target cache set. If all the cache ways of the targeted cache set are occupied, then the *cache replacement policy* determines which old block can be evicted from that cache set to make room for the new block. Common examples of cache replacement policies used in modern processors are Least-Recently-Used (LRU), First-In-First-Out (FIFO) and Pseudo-Least Recently Used (PLRU). Note that direct-mapped caches do not need any replacement policy as each cache set has only one way so each memory block maps to a specific position in the cache. In the context of this thesis, we will only explain the working of a LRU cache replacement policy.

- **LRU replacement policy:** LRU policy maintains a queue of memory blocks sorted in an ascending order based on their *age*. Age of a memory block refers to its position in the cache and is given by the number of accesses to different memory blocks from the last use of that memory block. The most-recently used memory block is assigned an age 0 whereas the least-recently used memory block has an age given by $cacheassociativity - 1$. In case of a cache miss, new element is added at the front of the queue (and assigned an age 0). However, if the cache is full, the last element of the queue, i.e., the element with age $cacheassociativity - 1$, is removed to accommodate the new memory block. Similarly, at a cache hit, the corresponding element is moved from its position in the queue to the front and all younger elements are aged by one. Figure 2.4 shows a sequence of references in a 4-way set-associative cache using a LRU replacement policy. Majority of the state-of-the-art on cache analysis has focused on caches with LRU replacement strategy. This is mainly because LRU replacement policy is predictable and easier to analyze in comparison to non-LRU policies such as FIFO and PLRU (Guan et al., 2013, 2014).

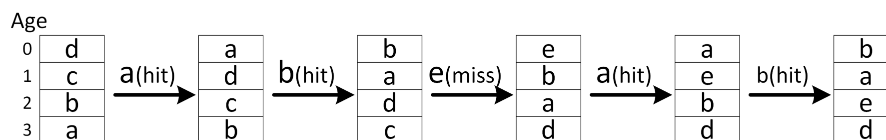


Figure 2.4: Example access sequence of memory blocks in a 4-way set-associative cache using a LRU replacement policy

In this work, we will focus on the analysis of direct-mapped caches and set-associative caches that use a LRU replacement policy.

Caches with multiple levels are categorized into *inclusive*, *exclusive* and *non-inclusive* caches. Inclusive caches require that the content in the higher cache levels should also be present in the

lower level cache, i.e., if a memory block is available in the L1 cache it should also be loaded in the L2 cache. In exclusive caches, the content in the higher cache levels must not be duplicated in the lower cache levels, i.e., a memory block can be available only in L1 or L2. Non-inclusive caches allow duplicated content at any cache level, however they do not strictly enforce the inclusion property, i.e., a memory block can be available in only L1/L2 or in both.

In this thesis, when required we will assume a non-inclusive cache hierarchy.

2.2.3.3 System Bus

Off-chip memories like the random access memories (RAM) or non-volatile read only memory (ROM) are very slow in comparison to the caches and are only accessed when the data/instructions are not found in the caches. The processor is connected to the off-chip main memory (or a memory controller) over a shared interconnection network usually called as the Front-Side Bus (FSB). The FSB is also referred to as the processor system bus or simply the system bus.

In a MCP, system bus is used to communicate between processing cores and the main memory. Bus handles different types of communication traffic including interrupt messages, memory requests, I/O traffic and coherency messages. Basic positioning of the system bus w.r.t the MCP and the memory is shown in Figure 2.5.

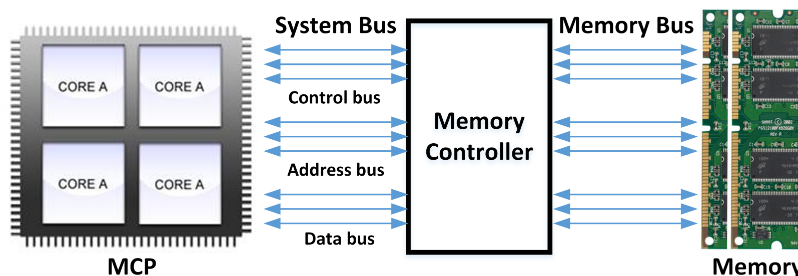


Figure 2.5: Basic abstraction of the system Bus

As shown in Figure 2.5, system bus is composed of three components, i.e., an address bus, data bus and control bus. These are separate channels used to transmit data, memory addresses from where data is to be fetched from or written to and some control signals that are used to control the overall functionality of the bus. Bus *width* defines the number of bits that can be transferred by the data bus, e.g., 32 or 64 bits. Similarly width of the address bus represents the maximum amount of addressable memory. Bus *speed* is an also important property which indicates the speed at which the bus can transfer data. Bus speed is expressed as number of cycles per second or Hertz (Hz).

Another important characteristic of a bus is the *bandwidth* or the maximum amount of data it can transfer per unit time. Bandwidth of the system bus is given by the product of the data path width, bus clock frequency and the number of data transfers the bus can perform per clock cycle (bus, 2017a), i.e.,

$$Bus\ Bandwidth = width \times clock\ frequency \times data\ transfers\ per\ cycle$$

For example, a 8-byte (64 bits) wide bus with an operating frequency of 100 MHz with a capacity to perform 4 transfers per cycle has a bandwidth of 3200 megabytes per second (MB/s) (bus, 2017a), i.e.,

$$8B \times 100\,MHz \times 4\,transfers/cycle = 3200\,MB/s$$

Bus perform communication using *messages* and *transactions*. A message is a logical unit of information that holds the memory address at which the data must be written to or read from, control signals and the data to be written (in case of a write operation). A transaction on the other hand is a sequence of messages. For example, a read transaction contains a read message with the memory address to read and a corresponding reply with the requested data. Bus transactions can be performed in several ways.

- **Atomic transactions:** The word atomic implies to indivisibility or irreducibility, so an atomic operation must be performed entirely or not performed at all. Similarly, an atomic bus transaction is modeled as an indivisible request-reply pair. This means that no new request can be entertained unless the bus transmits the response to the prior request. Atomic transactions are simpler to implement however, when using atomic transactions bus is underutilized since only one request can be fulfilled at a given time.

- **Pipelining:** In a pipelined bus, transactions are divided into different stages, e.g., arbitration, bus request, reply, data, error reporting etc. The basic idea is to combine any two phases of a transaction that use different physical lines on the bus. For example, the data bus is only responsible for transmitting the data written to or read from the memory and therefore only use some physical lines on the bus. Similarly, the control bus only handles operations like the arbitration, request, error reporting using independent bus lines. Therefore, multiple transactions that do not use the bus components can be pipelined together to increase the overall utilization of the bus. One example can be to overlap the address cycle of each transaction with data cycles of the previous transaction since the data bus is not used during address cycle and vice versa.

- **Split transaction:** In a bus that uses split transactions, a transaction is split into two components a request transaction and a reply transaction. Both transactions are handled independently of each other, where each transaction has to compete for an access to the bus. With a split transaction bus once a memory request is made by a core, it immediately releases the bus. In this way, other cores can also place there requests on the bus, increasing the overall utilization of the bus. When the response to a memory request is ready, the memory controller acquires the bus for the reply transaction and places the result on the bus. This response is then delivered to the corresponding core by the bus controller. Bus controller uses tags to identify the destination cores.

  In split transaction buses, certain memory requests may be served out off order, i.e., the responses may arrive in an order which does not match the order of requests issued. A split

transaction bus is an example of an out-of-order bus whereas, both the atomic transaction bus and the pipelined bus are an example of in-order buses.

In this work, we considering a multi-core platform, we will assume that the bus is shared between cores and it uses an atomic transaction protocol.

Another important mechanism that is used to minimize the main memory overhead latency is by using the hardware prefetching. Hardware prefetchers predict the next memory addresses to be accessed and pro-actively fetch this data to the last-level caches from the main memory based on the observed memory access patterns. However, in case of real-time tasks hardware prefetching may results in non-deterministic delays in task executions time. For example, the prefetchers also use the bus to perform transactions and hence may delay the requests issued by real-time tasks or the prefetched cache lines might evict cache lines that were used by the real-time tasks. Hardware prefetching is available in many commercial MCPs, with programmer having the facility to enable or disable this feature (Hegde, 2008).

In this work, we assume that the hardware prefetching is disabled.

Bus arbitration protocols

Bus arbitration protocols define the order in which the devices attached to the bus can access the bus. In a MCP, bus arbitration protocols control the access of multiple cores to the shared memory. Simultaneous requests by different cores to access the main memory may result in conflicts at the bus. These conflicts are resolved by the bus arbiter based on an arbitration policy.

Arbitration policies used in MCP can be mainly categorized as static or dynamic arbitration policies. In a static arbitration policy bus access patterns are defined at design time and does not change at run time. Whereas, in a dynamic arbitration policy the access patterns may change dynamically depending on the conflicts between cores or based on any other criteria, e.g., task priority or the arrival time of the requests etc. Time division multiple access (TDMA) is a prominent example of static bus arbitration policy whereas, First-in First-out (FIFO) and Round-robin (RR) arbiters are examples of the dynamic arbitration policies. Below we briefly describe the functionality of some prominent arbitration policies used in modern MCPs.

- **TDMA:** TDMA bus arbitration uses a fixed schedule to provide different cores of a MCP, access to the bus. This schedule or frame is periodic and is of a fixed size. At design time, each core is assigned one or more fixed slots within the frame to access the bus. Requests from a core are only entertained during the slots allocated to that core. Each core either uses its allocated slots or these slots are not utilized. Therefore, TDMA bus arbiter can under utilize the available bus capacity. However, TDMA arbitration is predictable and composable. It is predictable because the maximum time required by a task running on one core of a MCP to access the bus can be bounded at design time. Similarly, TDMA is also composable since the access time of one task is independent of the requests issues by other tasks running on other cores.

- **FIFO:** FIFO arbitration scheme works on the principle of first-in first-out. It maintains an in-order list of requests issued by different cores. The core whose request is the earliest is placed at the front of queue while later requests are subsequently added at the end of the queue. FIFO arbitration may sometime result in starvation, since the core that has the control of the bus may never complete hence, not allowing other cores to access the bus.
- **Fixed priority (FP) arbitration:** In FP arbitration policy, each request is assigned a priority based on a certain criteria. The request with the highest priority is then granted access to the bus. The drawback of FP arbitration is similar to that of FIFO arbitration, i.e., in case where highly memory intensive tasks are assigned higher priorities by the arbiter, then requests from lower priority tasks may need to wait indefinitely before receiving a response. Fixed priority arbiters are neither composable nor predictable as the time for access to the bus cannot be upper bounded without the knowledge about the access patterns of the higher priority requesters.
- **Round-robin (RR) arbitration:** Round-robin (RR) arbiter is a fair arbiter that allows in-order access of the bus to every requester. Fixed time slots of equal length are allocated to each requester. RR arbiter follows a rotating policy, i.e., the requester who is most recently granted the bus in one frame will be the last to receive the access in next frame. RR arbiter is predictable but however not composable. It is predictable since the maximum time to access the bus can be bounded but, as the access time to the bus of one requester depends on the number of other active requesters RR arbiter is not composable.

2.3 Ensuring Temporal Correctness of a RTS

Proving timing correctness of a RTS is traditionally a two-step process:

1. **Timing analysis:** The process of computing the WCET of tasks in *isolation*, i.e., an upper bound on the time that a given task can take to complete its execution under all feasible system states.
2. **Schedulability analysis:** The process used to ensure the schedulability of tasks, i.e., all tasks will meet their deadlines when deployed on the target hardware.

In the subsections below, we will provide a brief overview of the timing analysis and schedulability analysis relevant for this thesis.

2.3.1 Timing Analysis

As stated earlier, timing analysis is the process of estimating the worst-case timing requirement, e.g., WCET, of an isolated task. When computing the WCET of a task, activities other than the ones related to the considered task e.g., interrupts, blocking, preemptions or any kind of interference from other tasks in the system, are ignored. Different approach have been presented in

literature to bound the WCET of tasks. However, without disrespecting the variety of individual approaches, three approaches are commonly applied nowadays, i.e., *static* analysis, *measurement-based* analysis and *hybrid* analysis (Wilhelm et al., 2008a). In the context of this work, we will only discuss the working of static timing analysis.

2.3.1.1 Static Analysis

In static analysis, a task is analyzed by constructing the control flow of the program (or task) rather than executing the task on the real hardware or a simulator. An abstract model is used for the target hardware and for the inputs to the program and an upper bound on the WCET of task is obtained using this combination. Core components used by any static timing analysis approach are explained as follows:

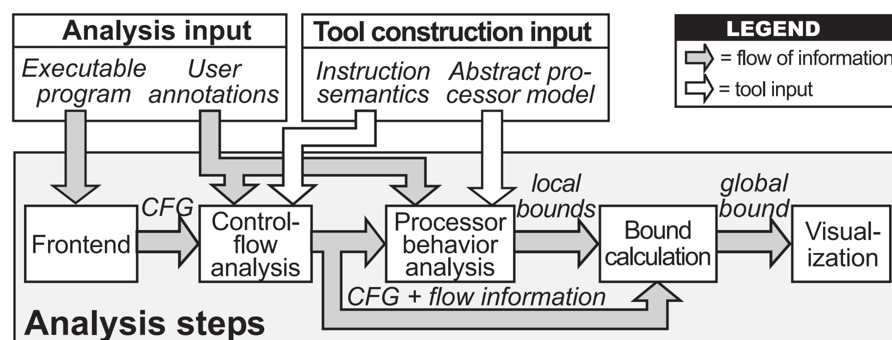


Figure 2.6: Work flow between different components of a timing-analysis tool (Wilhelm et al., 2008a)

Value Analysis

Value analysis computes the effective address where a memory accesses goes. In case the exact address of the referenced data cannot be determined, a range of addresses is conservatively provided. The analysis determines these addresses statically from a disciplined code (Thesing et al., 2003) by computing ranges for the values in processor registers at every possible program point. Value analysis can also compute the number of loop iterations and recursions (Martin et al., 1998).

Control-Flow Analysis

Control flow analysis use the parameters computed by the values analysis, e.g., ranges for the input data and iteration bounds of some loops, along with the call graph or the control-flow graph (CFG) of the task to gather information about possible execution paths. The result of the control flow analysis are usually constraints on the dynamic behavior of the task, e.g., which functions may be called, dependencies between different conditional operations and information relating to feasibility or in-feasibility of paths, etc.

Processor-Behavior Analysis

Processor-behavior analysis is the most important phase when determining the WCET of tasks under the static analysis based approaches. It uses a conservative timing model of the targeted hardware architecture in particular, of the components that influence the execution times, such as memory, caches, pipelines, and branch prediction along with the information provided by the value and control-flow analysis to determine upper bounds on the execution times of instructions or basic blocks. Most approaches for processor-behavior analysis use techniques based on the theory of Abstract Interpretation (Cousot and Cousot, 1977) to compute invariants about the processor's execution states at each program point. These invariants provide information about the contents of caches, the occupancy of functional units and processor queues, and of states of branch-prediction units. This information is then used to, e.g., exclude pipeline stalls and to classify memory accesses as cache hits/misses (detailed overview of the intra-task cache analysis is presented in Section 3.1).

Bound Calculation

This phase computes an upper bound on the execution times of the whole task using the flow and timing information derived in the previous phases. Different methods can be used to combine the timing estimates determined in the previous phases into an end-to-end estimate. For example, in approaches that use *Implicit Path Enumeration Technique* (IPET) (Li and Malik, 1997; Puschner and Schedl, 1995; Theiling, 2002) techniques program flow and basic-block execution time bounds are combined into set of arithmetic constraints. Each program flow edge in the task and basic-block is assigned a time coefficient t_{entity}, i.e., an upper bound on the contribution of that entity to the total execution time every time it is executed, and a count variable x_{entity}, i.e., an upper bound on the number of times that entity is executed. An upper bound on the task's WCET is then obtained by maximizing the objective function $\sum_{i \in entities} x_i \times t_i$, where the value of x_i is subject to constraints reflecting the structure of the task and possible execution flows.

Many commercial and research prototype based static analysis tools are available today such as Bound-T (Holsti and Saarinen, 2002), aiT (Ferdinand et al., 2007), Heptane (Colin and Puaut, 2001; Hardy et al., 2017), Chronos (Li et al., 2007) and SWEET (Ermedahl, 2003). For a detailed survey on the computation of WCET of tasks readers are directed to (Wilhelm et al., 2008a).

2.3.2 Schedulability Analysis

The output of the timing analysis, i.e., WCET of tasks, along with other timing constraints, e.g., task's period and deadline, are used by the schedulability analysis to determine if all tasks in the system comply with their timing requirements. Several different approaches can be used to perform the schedulability analysis depending on the scheduling algorithm and the priority assignment of tasks. However, since in this work we focus on preemptive fixed-task priority based scheduling algorithms such as RM and DM, we will use the traditional *response time* based schedulability analysis (Liu and Layland, 1973; Joseph and Pandya, 1986). Under the response

time based schedulability analysis, any task τ_i is referred to as *schedulable*, i.e., the task meets its timing constraints, if each of its instances or jobs complete their execution before the deadline, i.e., $R_i \leq D_i$. The response time R_i of a task τ_i is computed as follows:

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

where C_i is an upper bound on the WCET of task τ_i, $\mathrm{hp}(i)$ denotes the set of tasks with higher priority than τ_i and C_j is an upper bound on the WCET of any task $\tau_j \in \mathrm{hp}(i)$. Note that under priority-driven preemptive scheduling task τ_i can be preempted several times by higher priority tasks in $\mathrm{hp}(i)$. However, within the response time R_i of task τ_i, any higher task $\tau_j \in \mathrm{hp}(i)$ can execute at most $\left\lceil \frac{R_i}{T_j} \right\rceil$ times. Hence, the interference, i.e., the execution delay, task τ_i may suffer due to the executions of task τ_j during its response time is upper bounded by $\left\lceil \frac{R_i}{T_j} \right\rceil \times C_j$. As R_i appears on both sides in Equation (2.1), i.e., Equation (2.1) is recursive, a fixed-point computation on R_i can be used to find a solution by initiating R_i to C_i. For each task, the computation is stopped if R_i does not evolve anymore, i.e., the task is schedulable, or the value of R_i exceeds the deadline, i.e., $R_i > D_i$, in which case the task is deemed unschedulable. Note that a task set Γ is only said to be schedulable according to any schedulability analysis if each task $\tau_i \in \Gamma$ is schedulable.

2.3.3 Caches and Timing Analysis

The WCET of a task in isolation is an inherent property of the task which acts as an interface between the timing analysis and schedulability analysis. However, the WCRT of a task when co-executing with other tasks largely depends on the interference due to contention for resource accesses such as the processor, caches, bus and the main memory.

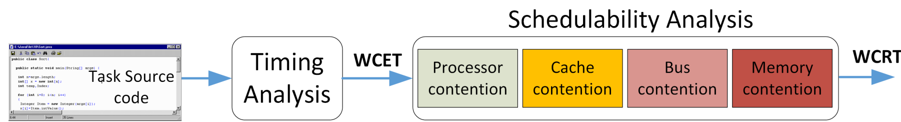


Figure 2.7: Basic interface between timing and schedulability analysis

Figure 2.7 shows the basic interface between timing and schedulability analysis. We can see that the WCRT of tasks depends on the timing behavior of resource such as caches, bus and the main memory. Therefore, in order to have a sound estimate of task's WCRT it is very important to first compute upper bounds on the interference due to contention for these resources and integrate these bounds into the schedulability analysis.

As discussed above, interference due to contention for resources that are shared between tasks can have a significant affect on the timing behavior of tasks. Indeed, caches are considered as one of the most important resource that can impact the execution of tasks. This is mainly because, the time spent by the tasks to perform memory related operations largely depends on the availability of data/instructions in the cache. In order to upper bound this time, a cache interference analysis

is performed that quantifies the number main memory accesses that may be generated during the execution of tasks. The output of the cache interference analysis is then used as an input for analyzing contention at the bus and main memory. This interdependence between the timing behavior of caches and other resources makes the cache interference analysis very crucial in providing deterministic bounds on the WCRT of tasks. Typically, tasks can be subjected to two types of cache interferences described as follows:

- **Intra-task cache interference:** Intra-task cache interference points to a situation where a task can evict its own cache lines. This can happen in two situations; (i) when two memory entries in the working set size of a task are mapped to same cache sets or (ii) when the working set size of a task is larger than the cache size. Intra-task cache interference analysis is usually considered as a part of the WCET analysis and several approaches have been presented in literature to bound this type of cache interference (Wilhelm et al., 2008b) (intra-task cache analysis will be discussed in detail in Section 3.1).

- **Inter-task cache interference:** Inter-task cache interference can be generated between different tasks running on the processor that are sharing the cache. Bounding inter-task cache interference is a challenging task since it depends not only on the cache footprint of the task under analysis but also on the cache usage of other tasks. Inter-task cache interference is mainly observed in priority-driven preemptive systems were a higher priority task can evict the preempted task's cached content. This results in extra delays during the execution of the preempted task. It has been shown by several works in literature (Stärner and Asplund, 2004; Bui et al., 2008; Bastoni et al., 2010; Bertogna et al., 2011) that inter-task cache interference can have a significant affect on task's execution times and ignoring the impact of inter-task cache interference on task schedulability may lead to optimistic results. Therefore, in order to have a sound estimate of the WCRT of tasks, a precise bound on the inter-task cache interference must be computed and integrated into the schedulability analysis. Several different approaches (Lv et al., 2015) have been presented in this regard that will be discussed in detail in Section 3.2.

2.3.4 System Bus and Timing Analysis

In modern processors, multiple concurrently executing tasks can access the main memory in parallel which leads to contention on the shared bus. This contention results in an increased response time of tasks running on the platform. Bounding this increase in response time or in other words the interference due to shared bus is one of the main challenge due to following reasons:

1. Due to the lack of documentation provided by the vendors, system bus is usually considered as a black box, with very little information available about the bus arbitration and bus controller implementation.

2. It is very difficult to predict at what instant the tasks will access the bus. Moreover, these accesses are not explicitly controlled by the operating system scheduler since they are mostly initiated as a result of cache misses, coherency traffic etc.

3. In FPPS higher priority tasks are executed prior to the lower priority tasks as they might be performing some critical operations. However, once the memory requests are issued by different tasks running on the processor the bus controller might reorder these requests based on some different criterion. Consequently, this might result in an unexpected situation where requests issued by the higher priority tasks may be served later than those from the lower priority tasks.

4. Also, most modern processors use an out-of-order bus that employ different performance enhancement mechanisms such as pipelining and split transactions which further complicates the bus interference analysis.

In this thesis, we will focus on the computation of inter-task cache interference and its integration into the schedulability analysis. Due to strong interdependence between caches and the system bus, we will also evaluate the impact of inter-task cache interference on bus contention.

2.4 Chapter Summary

In this chapter, we presented basic concepts that are necessary for the understanding of this thesis, i.e., the basics of real-time systems, functionality of its core components and methods used to ensure timing correctness of a real-time system. We also highlighted the importance of cache and bus contention analysis to ensure timing correctness of tasks. In next chapter, we will describe the related work focusing on cache and bus related contention in multi-tasking real-time systems.

Chapter 3

Related Work

In this chapter, we will provide a survey of the state-of-the-art in the computation of intra- and inter-task cache interference. It also serves as a starting point for understanding different analysis presented in subsequent chapters. This chapter is organized as follows: Section 3.1 presents prominent state-of-the-art approaches used to quantify intra-task cache interference. Section 3.2 presents the background on inter-task cache interference analysis focusing on single-level direct-mapped (Section 3.2.1), set-associative (Section 3.2.2) and multi-level caches (Section 3.2.3). State-of-the-art approaches that manage intra- and inter-task cache behavior by using methods such as cache partitioning, cache locking, task layout optimization and enhanced scheduling models are discussed in Section 3.3. Section 3.4 discusses most relevant approaches in the state-of-the-art that quantify memory bus contention and evaluate its impact on schedulability. Finally, in Section 3.5, we introduce a different positive prospective of cache memories due to intra-task cache re-use and highlight its potential in improving schedulability.

3.1 Intra-task Cache Interference Analysis

Intra-task cache interference analysis or simply *intra-task cache analysis* is often considered as a part of the WCET analysis. The basic purpose of an intra-task cache analysis is to determine the cache behavior of a task in isolation. This results in the classification of individual memory requests of a task as cache hits or misses. It also bounds the number of cache loads in different segments of a program. Different approaches have been presented in literature for intra-task cache analysis such as the static cache simulation presented by Muller et al. (Mueller, 2000) and the abstract interpretation based approach presented by Ferdinand and Wilhelm (Ferdinand and Wilhelm, 1999). Abstract interpretation (AI) (Cousot and Cousot, 1977) is a method for static program analysis based on the semantic of the considered programming language. It uses an abstract version of the program with an abstraction of the underline hardware components instead of executing the program on the actual hardware. Abstract interpretation based approaches for cache analysis are widely used in industry, e.g., in the aiT tool of AbsInt (abs).

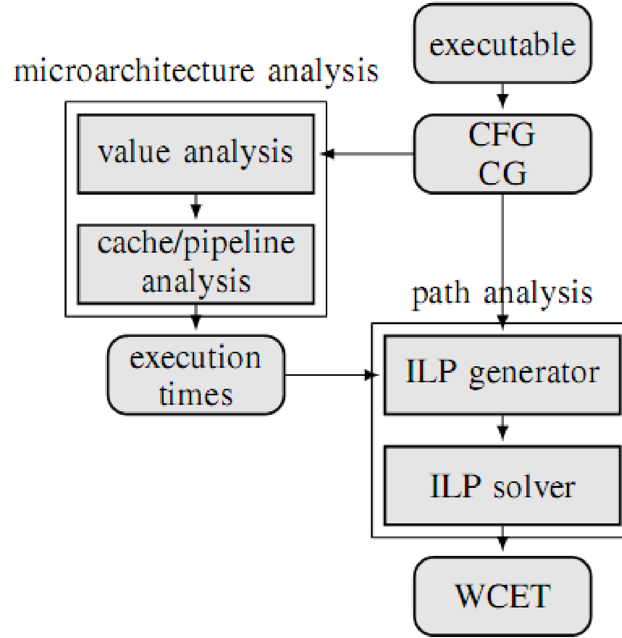


Figure 3.1: Intra-task ache analysis is one of the main components in the timing analysis (Phavorin and Richard)

In the context of this work, we will explain the working of Ferdinand's (Ferdinand and Wilhelm, 1999; Theiling, 2002) intra-task cache analysis which is based on AI. For simplicity, we assume a fully-associative LRU cache comprising of a set of cache lines $\hat{l} = \{l_0, l_1, ..., l_{S-1}\}$, where $S = CacheSize/LineSize$. The cache stores a set of memory blocks $M = \{m_a, m_b, .., m_z\}$. Note that for set-associative caches, the analysis can be performed independently for each cache set.

The work in (Ferdinand and Wilhelm, 1999; Theiling, 2002) uses the concept of abstract cache states (ACS) to estimate the cache contents at different points during the execution of a task.

Definition 3.1. *Abstract Cache State (ACS): An abstract cache state $ACS : \hat{l} \rightarrow 2^M$ represents all possible mappings between the set of memory blocks and the set of cache lines. For example, an abstract cache state $ACS(l_x) = m_y$, represent that in the abstract cache state ACS memory block m_y is mapped to cache line l_x, where x denote the age of the memory block according to the LRU replacement strategy.*

The analysis in (Ferdinand and Wilhelm, 1999; Theiling, 2002) uses three fixed point analyses, i.e., *must*, *may* and *persistence*, to categorize memory references into *Always-hit*(AH), *Always-miss*(AM), *Persistent or First-miss*(PS or FM) and *Not-classified*(NC). This classification is sometimes referred to as *Cache Hit/Miss Classification* (CHMC) and is described in Table 3.1. For each analysis, Abstract cache sates are computed at every program point using two functions, named *Update* and *Join*.

Definition 3.2 (Update Function)**.** *The Update function computes the abstract cache state after a memory reference, .i.e., ACS_{out}, using as inputs the abstract cache sate before the memory ref-*

Table 3.1: Categorization of memory references

Category	Description
Always-hit (AH)	The memory reference will always result in a cache hit
Always-miss (AM)	The memory reference will always result in a cache miss
Persistent or First-miss (PS or FM)	The first execution of the reference may result in a cache miss however, all further executions of the memory reference will always result in a cache hit
Not-classified (NC)	The memory reference can not be classified as AH, AM or PS.

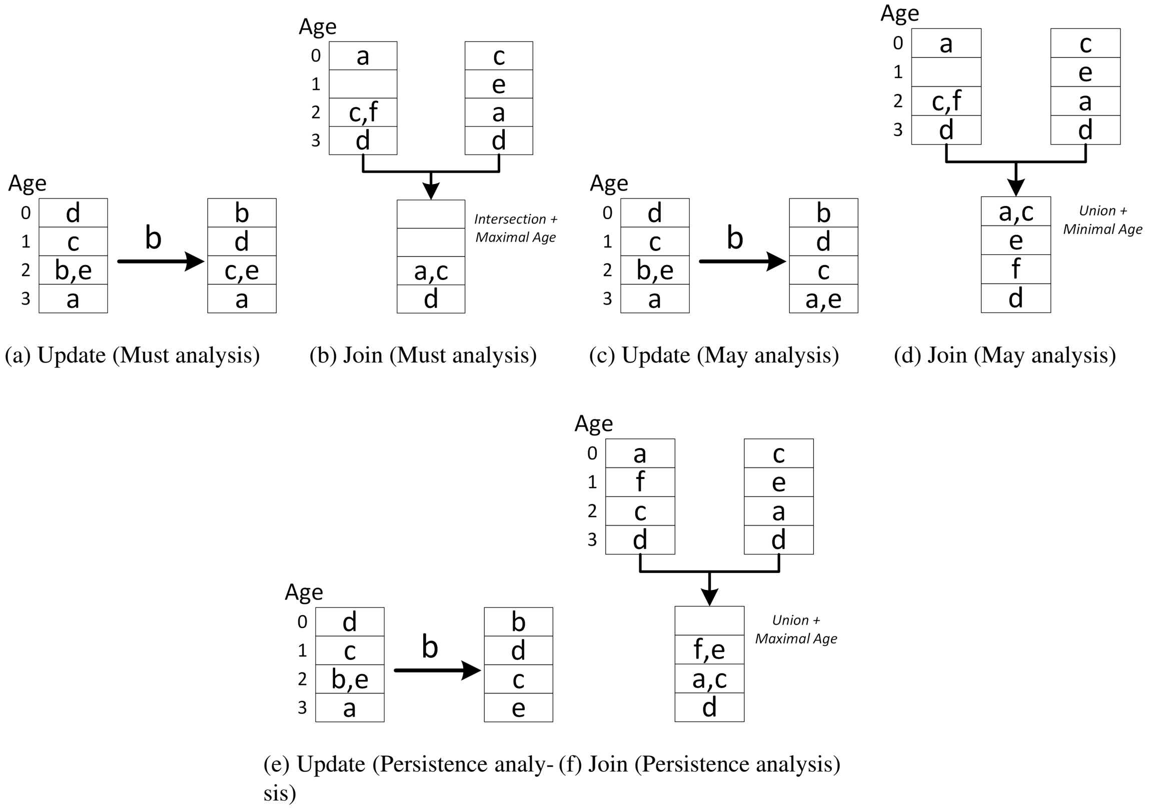
erence, i.e., ACS_{in}, and the referenced memory block. This function considers both the cache replacement policy and the semantics of the analysis.

Definition 3.3 (Join Function)**.** *A Join function is used to combine abstract cache states at control flow nodes with two or more predecessors, e.g., at the end of a conditional construct.*

3.1.1 Must Analysis

The must cache analysis is used to find Always-hit (AH) cache references, i.e., the cache blocks that are guaranteed to be in the cache at a specific program point. AH references represents the common cache contents for all possible execution paths in the CFG leading to a program point. In must cache analysis, the positions of the memory blocks in the abstract cache state are upper bounds on the ages of the memory blocks. For example, if ACS^{must} represents a must abstract cache state at any control flow node that references a memory block m_a and $m_a \in ACS^{must}(l_x)$ for a cache line l_x, then m_a is definitely in the cache and has an LRU age of x. A reference to m_a at this program point will always result in a cache hit. Moreover, m_a will stay in the cache for the next $S-x$ references to memory blocks that are not in the cache or are older than m_a. The update function for must analysis performs an access to a memory reference, e.g., to memory block m_b, using the abstract cache state before the memory access as an input, i.e., ACS^{must}_{in}, and produces the output abstract cache state, i.e., ACS^{must}_{out} after the memory access. In ACS^{must}_{out} the age of m_b will be 0 as it is now the most-recently used element. If m_b was not in ACS^{must}_{in}, then the age of all elements in ACS^{must}_{in} will be increased by 1 to produce ACS^{must}_{out}. If m_b was already in ACS^{must}_{in}, then the age of all element that were younger than m_b in ACS^{must}_{in} will be increased by 1 in ACS^{must}_{out}. Figure 3.2a shows an example of the update function for the must analysis.

The join function for the must analysis is similar to set intersection. A memory block m_b only stays in the output abstract cache state ACS^{must}_{out} if it is in both operand abstract caches states, e.g., ACS^{must}_{in1} and ACS^{must}_{in2}. In ACS^{must}_{out} m_b will get the maximal age if it has two different ages in ACS^{must}_{in1} and ACS^{must}_{in2}. Figure 3.2b shows an example of the join function for the must analysis.



(a) Update (Must analysis) (b) Join (Must analysis) (c) Update (May analysis) (d) Join (May analysis)

(e) Update (Persistence analysis) (f) Join (Persistence analysis)

Figure 3.2: Join and Update functions for the Must, May and Persistence analysis

3.1.2 May Analysis

May cache analysis is used to determine the Always-Miss(AM) cache references, i.e., the cache blocks that are guaranteed not to be in the cache at a certain program point. May analysis gives the content that may be cached in all possible executions leading to a program point. In may cache analysis, the positions of the memory blocks in the abstract cache state are lower bounds on the ages of the memory blocks. For example, let ACS^{may} represent the may abstract cache state at any control flow node that references a memory block m_a. If $m_a \notin ACS^{may}(l_x)$ for any arbitrary cache line l_x, then m_a is definitely not in the cache. A reference to m_a at this program point will be categorized as Always-miss(AM). The update function of the may cache analysis is similar to the update function of the must cache analysis however, the only difference is in the treatment of elements with the same age as the accessed element, e.g., m_b. In the update function of the may cache analysis, if m_b was already in ACS_{in}^{may}, then the age of all element that had the same age as m_b or were younger than m_b in ACS_{in}^{may} will be increased by 1 in ACS_{out}^{may}. An example of the update function for the may cache analysis is shown in Figure 3.2c.

The join function for the may analysis is similar to set union. If a memory block m_b has two different ages in operand abstract caches states, e.g., ACS_{in1}^{may} and ACS_{in2}^{may}. Then, m_b will have the minimal age in ACS_{out}^{may}. Figure 3.2d shows an example of the join function for the may analysis.

3.1.3 Persistence Analysis

The persistence analysis is used to classify memory references as Persistent or First-Miss (PS or FM), i.e., the memory reference to whom first access might be a miss but all subsequent accesses are AH. If ACS^{persis} represents the abstract cache state under the persistence analysis at any control flow node that references a memory block m_a. Then, m_a will be categorized as PS if $m_a \in ACS^{persis}(l_y)$ for $y \in \{0,...,S-1\}$ and m_a cannot be categorized as AH. The join function for the persistence analysis is similar to set union. If a memory block m_b has two different ages in operand abstract caches states, e.g., ACS^{persis}_{in1} and ACS^{persis}_{in2}. Then, m_b will have the maximal age in ACS^{may}_{out}. Figure 3.2f shows an example of the join function for the Persistence analysis.

The update function for the persistence analysis presented in (Theiling et al., 2000) is the same as the update function of their Must analysis, i.e., upon an access to memory block m_b only the ages of younger blocks are incremented. However, it has been recently found by (Cullmann, 2013; Huynh et al., 2011) that this update function is unsound. Cullmann (Cullmann, 2013) resolved this problem by proposing a modification, i.e., upon an access to block m_b, the age bounds of all blocks other than m_b, that have potentially been accessed before m_b will be increased. Figure 3.2e shows an example of the update function for the Persistence analysis. Note that an exact persistence analysis has been recently presented in (Stock et al., 2019; Reineke, 2018).

With the exception of AI-based approaches, different methods have been presented to bound the intra-task cache interference. Li et al. (Li et al., 1996) presented an approach that uses the concept of *Cache State Transition Graphs (CSTGs)* to model cache behavior. A CSTG models the cache-state transitions for a given cache set and is built using a CFG. In (Li et al., 1996) the authors try to bound the number of cache hits for each memory block. These bounds are then modeled as linear constraints and combined into an integer linear program (ILP) to obtain the WCET for the tasks. The approach in (Li et al., 1996) can provide a good analysis precision however, it does not scale with program size due to the complexity of the ILP. In other works model checking (Clarke et al., 1999) and timed automata (Alur and Dill, 1994) based approaches have been used to model the cache behavior of programs. Prominent works in this regard include the METAMOC approach (Dalsgaard et al., 2010), McAiT tool (Lv et al., 2011) and Gustavsson et al.'s analysis (Gustavsson et al., 2010) etc.

3.1.4 Intra-task Cache Analysis for Multilevel Caches

Most modern processors are equipped with multilevel caches. Therefore, in order to have a precise estimate of the number of main memory requests generated by a task intra-task cache analysis should be conducted considering all cache levels. In state-of-the-art two main analysis frameworks are used to analyze multilevel caches, i.e., *separate* analysis and *integrated* analysis. In separate analysis (Mueller, 1997; Hardy and Puaut, 2008), cache levels are analyzed independently, e.g., L1 cache is analyzed first. Then, the result of the L1 cache analysis is used as an input for the analysis of L2 cache, and so on. On the contrary, integrated analysis (Sondag and Rajan, 2010) analyze all cache level at the same time by building a holistic abstract domain. Separate analysis

have several advantages over the integrated approaches such as the flexibility to apply a different analysis method for each cache level and scalability, i.e., the overall analysis is scalable as long as the adopted single-level analysis is scalable. On the other hand, the integrated analysis can be more precise than the separate analysis due to imprecise transfer of cache access information across cache levels in the separate analysis. However, the integrated analysis are usually subjected to scalability issues. As in this work, we only focus on non-inclusive multilevel caches, we will briefly describe the functionality of the multilevel cache analysis of (Hardy and Puaut, 2008). The multilevel cache analysis proposed in (Hardy and Puaut, 2008) uses Ferdinand's cache analysis (Ferdinand and Wilhelm, 1999; Theiling, 2002) at each cache level (see Section 3.1). The main difficulty in analyzing multiple cache levels is to handle the interaction between cache levels, i.e., to predict which memory reference will be accessed at which cache level. For example, level-one, i.e., L1, cache is always accessed for each memory reference so, if a memory access is predicted as AH at L1, then that memory access should not be considered during the analysis of level-two, i.e., L2, cache. An interface called *Cache Access Classification*(CAC) is proposed to describe this information (Hardy and Puaut, 2008). For any memory reference r and cache level L, the CAC is defined as follows:

- N (Never): the access to r will never be performed at cache level L
- A (Always): the access to r will always be performed at cache level L
- U (Uncertain): it can not be guaranteed if the access to r will be/not be performed at cache level L.

CAC information relating to a memory reference is used as an input by the cache analysis at each level to decide if that reference is to be considered during the analysis of that cache level or not. The CAC for a reference r at a cache level L depends on its CAC and CHMC at cache level $L-1$ as shown in Table 3.2. Combination of $CAC_{r,L-1}$ and $CHMC_{r,L-1}$ values in Table 3.2

Table 3.2: Computation of CAC of a memory reference r at cache level L (Hardy and Puaut, 2008)

$CAC_{r,L-1}$ \ $CHMC_{r,L-1}$	AM	AH	FM	NC
Always (A)	A	N	U	U
Uncertain (U)	U	N	U	U
Never (N)	N	N	N	N

can be used to compute the CAC of r at level L, i.e., $CAC_{r,L}$. For example, if CHMC and CAC of r at level $L-1$ is AH and A respectively, i.e., $CHMC_{r,L-1} = AH$ and $CAC_{r,L-1} = A$, then the reference to r is never considered in the analysis of level L, i.e., $CAC_{r,L-1} = N$. In an earlier work, Mueller (Mueller, 1997) proposed that memory accesses with a U CAC at cache level $L-1$ should always be considered in the analysis of level L, i.e., should be assigned a CAC of A at level L. However, this assumption can be unsafe due to underestimation of memory block ages as demonstrated in (Hardy and Puaut, 2008). The work in (Hardy and Puaut, 2008) solves this

problem in Mueller's analysis (Mueller, 1997) by considering both N and A possibilities for U accesses in the update function. Figure 3.3 shows the update function for U accesses proposed in (Hardy and Puaut, 2008), which guarantees that the worst-case scenario is never missed. The

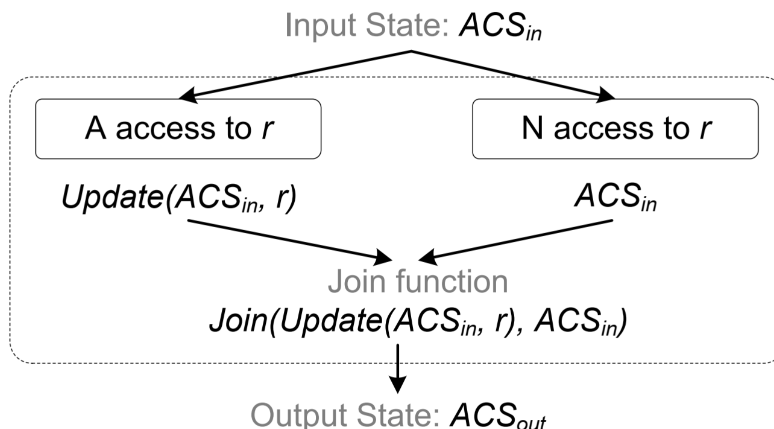


Figure 3.3: Update function to handle U accesses for multilevel caches (Hardy and Puaut, 2008)

analysis in (Hardy and Puaut, 2008) has also been extended to handle inclusive and exclusive cache hierarchy (Hardy and Puaut, 2011).

3.2 Inter-task Cache Interference Analysis

Intra-task cache interference analysis presented in the previous section can be used to precisely upper bound the number of cache hits/misses generated by a task while executing in isolation. However, when the task is co-executed with other tasks it can be subjected to inter-task cache interference due to sharing of caches. Specifically, in priority-driven preemptive scheduling, the execution of a lower priority task can be interrupted several times due to preemptions by higher priority task(s) and for every preemption, the preempting task(s) may evict cache entries of the preempted task that may be required later on. This inter-task cache interference leads to additional cache misses (other than the ones computed using the intra-task cache analysis) during the execution of the preempted task. Formally, in the state-of-the-art the increase in the execution time of task due inter-task cache interference is refereed to as *cache related preemption delay* (CRPD).

Definition 3.4 (Cache related preemption delay (CRPD))**.** *When a lower priority task τ_i is preempted by a higher priority task τ_j, the preempting task τ_j may evict cache blocks of the preempted task τ_i that has to be reloaded after τ_i resumes its execution. The additional execution time needed by τ_i to perform these cache reloads is termed as cache related preemption delay (CRPD). The CRPD a task τ_i may suffer due to a preemption by a higher priority task τ_i is usually denoted by $\gamma_{i,j}$.*

Figure 3.4 shows a visual representation of CRPD suffered by a task τ_i due to preemptions by a higher priority task τ_j. We can observe that task τ_i and τ_j both are using cache sets $\{4,5\}$. Hence,

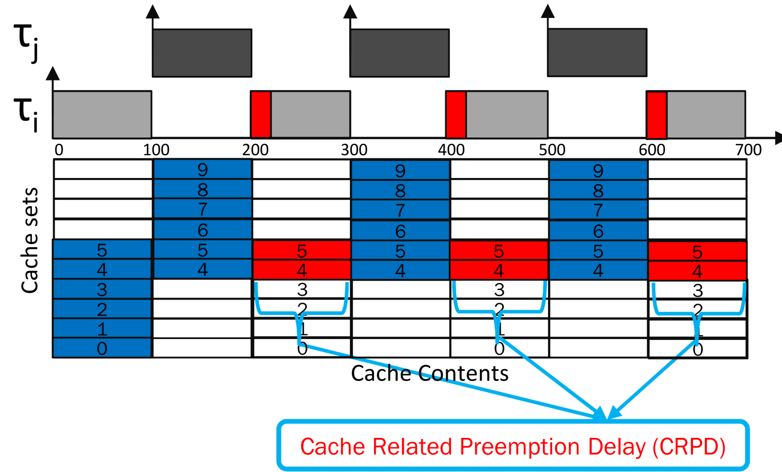


Figure 3.4: Visual representation of cache related preemption delay (CRPD)

for every preemption of τ_i by τ_j, the content of τ_i in cache sets $\{4,5\}$ is evicted and replaced by the content of τ_j. Assuming cache sets $\{4,5\}$ hold useful data/instructions that are used several times during the execution of task τ_i, τ_i will be required to reload the evicted content after every preemption by task τ_j. This leads to an increase in the WCRT of τ_i due to CRPD.

It have been shown in the state-of-the-art (Liu and Solihin, 2010; Bui et al., 2008) that CRPDs can significantly affect the WCRT of tasks and hence should be bounded accurately. In literature different approaches have been used to calculate CRPDs considering single and multilevel caches (Lv et al., 2015). These approaches use the notion of *useful cache blocks (UCBs)* and *evicting cache blocks (ECBs)* to compute CRPDs. The concept of UCBs was introduced by Lee et al. (Lee et al., 1998) and is defined as follows:

Definition 3.5 (Useful cache block (UCB))**.** *A memory block m is called a* ***Useful Cache Block (UCB)*** *at program point P, if (i) m may be cached at P and (ii) m may be reused at program point Q that may be reached from P without eviction of m on this path. The set of all UCBs of a task* τ_i *denoted by* UCB_i*.*

This definition of UCBs was later improved by Altmeyer et al (Altmeyer and Burguière, 2011), by introducing the notion of *definitely-cached useful cache blocks (DC-UCBs).*

Definition 3.6 (Definitely-cached useful cache block (DC-UCB))**.** *A memory block m is called a* ***Definitely-cached Useful Cache Block (DC-UCB)*** *at program point P, if (i) m must be cached at P and (ii) m may be reused at program point Q that may be reached from P without eviction of m on this path.*

The DC-UCB analysis (Altmeyer and Burguière, 2011) optimize the number of UCBs of tasks by only accounting the cache misses the have not already been considered during the WCET analysis. However, this method is only safe when used in combination with a WCET bound.

The notion of ECBs was introduced by Busquets-Mataix et al. (Busquets-Mataix et al., 1996) and defined it as

Definition 3.7 (Evicting cache blocks (ECBs))**.** *A memory block accessed during the execution of a preempting task is referred to as an* ***Evicting Cache Block (ECB)****. The set of all ECBs of a task* τ_i *is denoted by* ECB_i*.*

Deriving the Set of UCBs/ECBs

The set of UCBs/ECBs of a task can be derived using the intra-task cache analysis methods detailed in Section 3.2.

The set of UCBs of a task τ_i is determined at every program point P during the execution of τ_i. By definition (i.e., the original definition by Lee et al. (Lee et al., 1998)), a memory block m is a UCB at a program point P if it may be cached at P and may be reused at a later program point Q that is reachable from P without eviction of m along the path from P to Q. So to determine if m is a UCB at P, the analysis has to compute the number of accesses (to different memory block other than m) from the last use of m to program point P and the number of access (to different memory block other than m) from P to the next access to memory block m after P. Effectively, these number of accesses to/from program point P w.r.t. memory block m can be computed by using the May cache analysis (see Section 3.1.2) in forward and backward direction (Altmeyer, 2013). Consequently, if the age of m at P in both forward and backward May analysis is less than cache associativity then, m will be considered a UCB at program point P. Note that different approaches can be used to compute the set of UCBs of tasks, e.g., see (Lee et al., 1998), (Negi et al., 2003) and (Staschulat and Ernst, 2007).

Computation of the set of ECBs of a task is comparatively simple than the computation of UCBs. By definition, all memory blocks that may be used by a task during its execution are its ECBs. Therefore, to compute the set of ECBs of a task, May cache analysis (see Section 3.1.2) can be used. As the May analysis over-approximate the cache content at every program point during the execution of tasks, it sufficient to computer the set of ECBs of tasks at their exist/end point, e.g., e. Note that for set-associative caches, the number of cache ways can be used as an upper bound on the number of ECBs per cache set.

3.2.1 CRPD Computation for Single-level Direct-mapped Caches

For direct-mapped caches, CRPDs can be computed by only using the set of UCBs and ECBs of tasks.

In one of the earlier works, Busquets-Mataix et al. (Busquets-Mataix et al., 1996) and later Tomiyama and Dutt (Tomiyama and Dutt, 2000), proposed the ECB-only approach to calculate the CRPD cost. They used ECB's of the preempting task in order to bound the CRPD. Using ECB-only approach, If τ_j is the higher priority task preempting a lower priority task τ_i, the resulting CRPD cost $\gamma_{i,j}^{ecb}$ is given as follows:

$$\gamma_{i,j}^{ecb} = d_{mem} \times |ECB_j| \tag{3.1}$$

Where d_{mem} corresponds to the time required to reload one memory block to cache from the main memory. The approach of Busquets-Mataix et al. (Busquets-Mataix et al., 1996) results in pessimistic CRPD bounds since it always assumes the worst-case where each block accessed by a higher priority task can evict cache lines of the lower priority tasks. In contrast to (Busquets-Mataix et al., 1996), Lee et al. (Lee et al., 1998) proposed the UCB-only approach, which uses the UCBs of the lower priority task τ_i preempted by the higher priority task τ_j and the UCBs of all the intermediate priority tasks (i.e., tasks with priority higher or to τ_i and strictly lower than that of τ_j) to bound the CRPD cost. They used the intermediate priority tasks to account for nested preemptions the resulting value for the CRPD cost is given as

$$\gamma_{i,j}^{ucb} = d_{mem} \times \max_{\forall k \in \mathrm{aff}(i,j)} \{|UCB_k|\} \tag{3.2}$$

Where the set $\mathrm{aff}(i,j)$ contain the set of tasks with priorities higher than or equal to the priority of τ_i(including τ_i), but strictly lower than that of τ_j. UCB-only approach assumes that the maximum number of UCBs among all tasks in $\mathrm{aff}(i,j)$ will be evicted for every preemption by the preempting task. However, this is a pessimistic assumption since, in reality the number of UCBs that can be evicted depends on the memory access patterns of both the preempting and the preempted task.

The UCB-union approach presented by Tan and Mooney (Tan and Mooney, 2007) uses both the preempted and the preempting task in order to calculate the CRPD cost. It uses the UCB's of all tasks $\in \mathrm{aff}(i,j)$ and the ECBs of the preempting task in order to calculate the preemption cost. The resulting CRPD in this case is denoted by γ_{ucb-u} and is given as follows

$$\gamma^{ucb-u} = d_{mem} \times \left| \left(\bigcup_{\forall k \in \mathrm{aff}(i,j)} UCB_k \right) \cap ECB_j \right| \tag{3.3}$$

The UCB-union approach dominates the ECB-only approach (Busquets-Mataix et al., 1996) but can be pessimistic in some cases as described in (Altmeyer et al., 2012).

On a similar note to the work done by Tan and Mooney (Tan and Mooney, 2007), Altmeyer et al. (Altmeyer et al., 2011) presented the ECB-union approach that uses the ECB's of all tasks in $\mathrm{hep}(j)$ (i.e. all tasks having priority higher than or equal to τ_j) maximized over the UCB's of all tasks in $\mathrm{aff}(i,j)$. The resulting preemption cost γ_{ecb-u} is given as

$$\gamma^{ecb-u} = d_{mem} \times \max_{\forall k \in \mathrm{aff}(i,j)} \left(\left| UCB_k \cap \left(\bigcup_{\forall h \in \mathrm{hep}(j)} ECB_h \right) \right| \right) \tag{3.4}$$

ECB-union approach dominates the UCB-only approach (Lee et al., 1998) and provides a reasonably precise bound on preemption cost especially when we have nested preemptions. But both the UCB-union and the ECB-union approach are incomparable and can lead to overestimation in different situations as shown in (Altmeyer et al., 2012). To reduce this overestimation Altmeyer et al. (Altmeyer et al., 2012) proposed two multi-set variants of these approaches i.e., UCB-union

multi-set and the ECB-union multi-set approach. These multi-set versions of the UCB-union and ECB-union approaches additionally take into account the maximum number of jobs $E_j(R_i) \stackrel{\text{def}}{=} \left\lceil \frac{R_i}{T_j} \right\rceil$ that each higher priority task τ_j can release during the response time of τ_i and the number of preemptions of each low and intermediate priority task by τ_j, i.e., $E_j(R_k)E_k(R_i) \stackrel{\text{def}}{=} \left\lceil \frac{R_k}{T_j} \right\rceil \times \left\lceil \frac{R_i}{T_k} \right\rceil$. The ECB-Union Multi-set approach uses $\gamma_{i,j}^{ecb-m}$ to represent the CRPD cost due to all jobs of task τ_j executing during the response time of task τ_i. Where $\gamma_{i,j}^{ecb-m}$ is given as

$$\gamma_{i,j}^{ecb-m} = d_{mem} \times \sum_{l=1}^{E_j(R_i)} \left| M^l \right| \tag{3.5}$$

Where M^l is the lth-largest value in M, Where M is a multi-set composed of multiple sets of ECBs and UCBs of the corresponding tasks defined as follows:

$$M = \bigcup_{\forall k \in \text{aff}(i,j)} \left(\bigcup_{E_j(R_k)E_k(R_i)} \left| UCB_k \cap \left(\bigcup_{\forall h \in hep(j)} ECB_h \right) \right| \right) \tag{3.6}$$

For the UCB-Union multi-set approach the CRPD cost is upper bounded by $\gamma_{i,j}^{ucb-m}$ defined as follows:

$$\gamma_{i,j}^{ucb-m} = d_{mem} \times \left| M_{i,j}^{ucb} \cap M_{i,j}^{ecb} \right| \tag{3.7}$$

where $M_{i,j}^{ucb}$ and $M_{i,j}^{ecb}$ are multi-sets defined as

$$M_{i,j}^{ucb} = \bigcup_{\forall k \in \text{aff}(i,j)} \left(\bigcup_{E_j(R_k)E_k(R_i)} UCB_k \right) \tag{3.8}$$

and

$$M_{i,j}^{ecb} = \bigcup_{E_j(R_i)} ECB_j \tag{3.9}$$

Here, $M_{i,j}^{ucb}$ is a multi-set comprising sets of UCBs of all low and intermediate priority tasks $\in$ aff(i, j) added $E_j(R_k)E_k(R_i)$ times, i.e., the maximum number of times τ_j can preempt each τ_k during the response time of τ_i. Similarly, $M_{i,j}^{ecb}$ is a multi-set comprising the set of ECBs of all jobs of τ_j executing within the response time of τ_i. The final value of the preemption cost $\gamma_{i,j}^{ucb-m}$ comes from the intersection of both these multi-sets.

The multi-set approaches, i.e., UCB-Union multi-set and ECB-union multi-set, dominate their union counterparts, i.e., UCB-union and ECB-union respectively. However, it is demonstrated in (Altmeyer et al., 2012) that the UCB-union and ECB-union multi-set approaches are incomparable. Consequently, a combined approach is proposed in (Altmeyer et al., 2012) that uses $\min(\gamma_{i,j}^{ecb-m}, \gamma_{i,j}^{ucb-m})$ as an upper bound on the CRPD.

In a recent work, Markovic et al. (Marković et al., 2020a) has shown that the combined approach, i.e., the combination of UCB-union multi-set and ECB-union multi-set, may result in over-approximating the CRPD cost by accounting for multiple preemption combinations which cannot occur simultaneously during runtime. Markovic et al. (Marković et al., 2020a) instead pro-

poses an approach based on preemption partitioning, i.e., to divide all possible preemptions that can occur in a time interval of length t into partitions of single-job preemptions. Consequently, CRPD bound for each individual preemption is then computed using the most precise method. This leads to significant improvements in taskset schedulability.

3.2.2 CRPD Computation for Single-level Set-associative LRU Caches

The derivation of the set of UCBs and ECBs of tasks is similar for both direct-mapped and set-associative caches. However, the main challenge in the computation of CRPD for set-associative LRU caches is to safely compute the intersection between the UCBs and ECBs. This mainly because with set-associative LRU caches a single ECB of the preempting task can lead to a chain of misses of multiple UCBs of the preempted task, which is not the case for direct-mapped caches. One solution to this problem was proposed by Burguière et al. (Burguière et al., 2009) by assuming that all UCBs of the preempted task that map to a cache set will be evicted by any ECB of the preempting task that map to the same cache set. A similar approach is proposed in (Altmeyer et al., 2012; Marković et al., 2020a) to handle set-associative LRU caches. Indeed, the approaches proposed in (Burguière et al., 2009; Altmeyer et al., 2012; Marković et al., 2020a) to compute CRPD for set-associative cache are safe but overly pessimistic. The only existing approach that computes a precise bound on the CRPD for set-associative LRU caches is proposed in (Altmeyer et al., 2010). Instead of only using the set of UCBs and ECBs of tasks to compute CRPD, Altmeyer et al. (Altmeyer et al., 2010) introduced the notion of *resilience* defined as follows.

Definition 3.8 (Resilience (Altmeyer et al., 2010))**.** *The resilience of a memory block m at program point P is the maximum* disturbance *that m can endure before being evicted from the cache. This disturbance represents the number of ECBs of preempting task(s) that may be mapped to the same cache set as m.*

The Resilience of a cache block m at a program point P is given by

$$res_P(m) = (CacheAssociativity - 1) - max\text{-}age_P(m) \tag{3.10}$$

where $max\text{-}age_P(m)$ is the *maximum* LRU-age of m at program point P, i.e., the maximum number of accesses to the same cache set as m from the last use of m (before or at program point P) to the next access to m after P (Altmeyer et al., 2010). For example, assuming memory blocks m_i, m_a, m_b, m_c and m_d in Figure 3.5 are all accessed by task τ_i and that they are all mapped to the same cache set, the maximum LRU-age of UCB m_i at program point P, i.e., $max\text{-}age_P(m_i)$, is 4 and therefore for a set-assocative cache with 8-ways, i.e., $CacheAssociativity = 8$, its resilience according to Eq. (3.10) is $(8-1)-4=3$.

For every program point P, the maximum LRU-age of a UCB m can be calculated by using a forward analysis to find the maximal number of accesses from the last use of m to program point P and a backward analysis to find the maximum number of accesses from program point P to the

next access to m. The maximum LRU-age of m at program point P is then bounded by the sum of the bounds returned by both analyses (see (Altmeyer et al., 2010) for a detailed description).

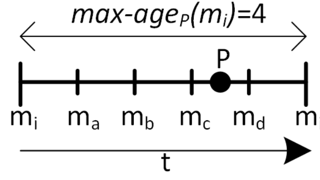


Figure 3.5: Illustration of the maximum LRU-age of a UCB m_i. The dashes (from left to right) denote the sequence of memory accesses during the execution of task τ_i.

In set-associative caches, each cache set s can be analyzed independently. Consequently, the set of ECBs and UCBs of tasks can be computed per cache set. For any cache set s, if UCB_i^s denote the set of UCBs of the preempted task τ_i in s and ECB_j^s denote the set of ECBs of the preempting task in s, then, under the resilience-analysis, the CPRD suffered by a task τ_i due to a single preemption by a higher priority task τ_j in a cache set s is given by $\gamma_{i,j}^{res,s}$, where $\gamma_{i,j}^{res,s}$ is computed as follows:

$$\gamma_{i,j}^{res,s} = d_{mem} \times \left| UCB_i^s \setminus \{m_i | res(m_i) \geq |ECB_j^s|\} \right| \tag{3.11}$$

where $res(m_i)$ is the resilience of a memory block m_i of task τ_i. Note that the CRPD cost computed using Equation (3.11), does not include the UCBs of task τ_i that may remain cached even after a preemption by task τ_j (i.e., those for which $res(m_i) \geq |ECB_j^s|$). Finally, the total CRPD task τ_i can suffer due to a single preemption by task τ_j is computed using Equation (3.11) for every set in the cache, i.e.,

$$\gamma_{i,j}^{res} = \sum_{\forall s} \gamma_{i,j}^{res,s} \tag{3.12}$$

Note that resilience analysis can be incorporated into ECB-union and ECB-union multi-set based approaches as demonstrated in Altmeyer's dissertation (Altmeyer, 2013).

3.2.3 CRPD Computation for Multi-level Caches

Computation of CRPD poses additional challenges when considering caches with multiple levels. These challenges stem from cache sharing between tasks at different cache levels with the execution of one task potentially evicting memory blocks previously loaded into one or more cache levels by other tasks. CRPD analysis for single-level caches has been extensively studied (Lee et al., 1998; Tomiyama and Dutt, 2000; Altmeyer et al., 2010, 2012; Lv et al., 2015). However, due to added complexity of analyzing cache conflicts at multiple cache levels only few approaches (Chattopadhyay and Roychoudhury, 2014; Zhang and Koutsoukos, 2016) have been presented in literature that focus on CPRD analysis for multi-level caches. Chattopadhyay et al. (Chattopadhyay and Roychoudhury, 2014) has proposed a CPRD analysis for multi-level non-

inclusive caches whereas Zhang et al. (Zhang and Koutsoukos, 2016) presented a CRPD analysis considering inclusive multi-level caches.

In both works (Chattopadhyay and Roychoudhury, 2014; Zhang and Koutsoukos, 2016), authors show that the existing analysis methods used for the computation of CRPD for single-level caches cannot be directly used for multi-level caches mainly due to the *indirect effect of preemption* that exists in multi-level caches.

Definition 3.9 (Indirect Effect of Preemption)**.** *Indirect effect of preemption refers to an increase in the intra-task cache interference, i.e., cache contention between different code segments within a task, at a lower cache level (i.e., L2 cache) due to preemptions that evict content from a higher cache level (i.e., L1 cache).*

To illustrate, consider the example scenario shown in Figure 3.6 that shows a sequence of memory references during the execution of a task (from left to right). The top half of Figure 3.6 shows the contents of L1 and L2 caches in case of a non-preempted execution whereas the bottom half of Figure 3.6 shows the contents of L1 and L2 caches in case of a preempted execution. All memory blocks used by task τ_i, i.e., memory blocks A, B and m, are mapped to same L1/L2 cache set. We can see that in case of non-preempted execution, the second reference to memory block m is a L2 cache hit. However, due to preemption at program point P (which only evicts memory block A from L1 cache) the same reference to memory block m results in a L2 cache miss. For the scenario shown in Figure 3.6, memory block m is evicted indirectly from the L2 cache due to an increase in cache conflicts at L2 caused by the eviction of memory block A from the L1 cache. This phenomenon is termed as the indirect effect of preemption.

The indirect effect of preemption happens due to memory blocks that are accessed from the higher level cache, e.g., L1, during the normal (i.e., non-preempted) execution of a task but are accessed from the lower level caches, e.g., L2, after the preemption. Since, lower level caches are only accessed upon a cache miss from a higher level cache, i.e., L2 is only accessed when there is a L1 miss, the indirect effect of preemption results in increasing the number of accesses to lower cache levels, e.g., L2, which may result in increasing intra-task cache interference in those cache levels. It has been identified in (Chattopadhyay and Roychoudhury, 2014; Zhang and Koutsoukos, 2016) that the traditional UCB concept used to analyze CRPD in single-level caches is hard to use in case of multi-level caches due to indirect effect of preemption. This is mainly because in single-level caches, when computing CRPD due to preemption at a program point P, the analysis only checks the first access of a memory block m (i.e., a UCB at P) after preemption. If the first access to m after preemption is a cache miss, the cost of reloading m from the main memory is added to the CRPD cost. However, in multi-level caches, the first access to m after preemption may result in a cache hit but one or more next accessed to m after preemption may result in a cache miss due to indirect effect of preemption. Therefore, for the computation of CRPD for multi-level caches, Chattopadhyay et al. (Chattopadhyay and Roychoudhury, 2014) introduced the notion of UCBs in the context of two-level caches and defined it as

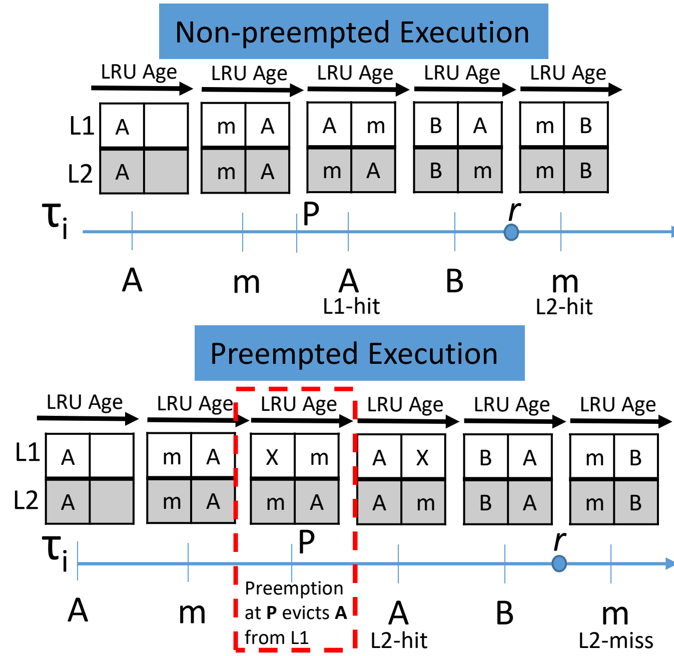


Figure 3.6: Illustration of the indirect effect of preemption suffered by a memory block *m* due to eviction of another memory block *A* by preemption. Both L1 and L2 caches are assumed to be two-way set-associative having only one cache set and the cache replacement policy is LRU.

Definition 3.10 (Useful Cache Blocks (UCBs) in a two-level cache (Chattopadhyay and Roychoudhury, 2014))**.** *In a two-level cache, a memory block $m_{x,i}$ of task τ_i is considered a UCB at program point P if (i) $m_{x,i}$ is cached at P in either L1, L2 or both and (ii) $m_{x,i}$ is reused at program point Q that must be reached from P without eviction of $m_{x,i}$ from both L1 and L2 caches.*

Based on the above definition authors in (Chattopadhyay and Roychoudhury, 2014) presented an analysis to determine the set of UCBs of tasks for two-level non-inclusive caches. This set of UCBs along with the set of memory references classified as L2-hits by the intra-task cache analysis is then used to compute the CRPD. Based on the work in (Chattopadhyay and Roychoudhury, 2014), Zhang et al. (Zhang and Koutsoukos, 2016) presented an analysis to compute CRPD for multi-level inclusive caches. The analysis in (Zhang and Koutsoukos, 2016) identifies additional challenges in the computation of CRPD due to cache inclusion policy and use the notion of *useful positive reference* (UPR) to compute CRPD.

Definition 3.11 (Useful Positive References (UPRs))**.** *The set of UPRs of a task at any program point P is given by the set of memory blocks whose references are positively classified, i.e., AH or PS, by the intra-task cache analysis w.r.t program point P.*

The analysis in (Zhang and Koutsoukos, 2016) derives the set of positive references that can be considered as UPRs at a program point P, i.e., the references that can lead to additional cache misses after a preemption at P, and bound the number of times each of them may act as a UPR. This information is then used to upper bound the CRPD any task may suffer due to preemption at

any program point P. The main idea of the CRPD analysis presented in (Zhang and Koutsoukos, 2016) is similar to the UCB-only approach (Lee et al., 1998) as it also only use the UPRs of the preempted task to bound the CRPD. Due to our focus on multi-level non-inclusive caches, we will present a detailed CRPD analysis of (Chattopadhyay and Roychoudhury, 2014) in Chapter 8.

3.2.4 From CRPD to Timing Analysis

As discussed in Section 3.1, intra-task cache interference analysis is considered part of the WCET analysis and the resulting WCET bounds of tasks also account for the intra-task cache interference. However, the inter-task cache interference or CRPDs also needs to be considered when performing the timing analysis of tasks. Different approaches have been proposed in literature to account for CRPDs in WCET or WCRT of tasks. In (Ward et al., 2014), ward et al. discussed the task-centric and preemption centric approaches. In task-centric approaches (Schneider, 2000; Altmeyer and Burguière, 2011), the CRPD cost of one preemption multiplied with the possible number of preemptions a task may suffer is added to the WCET of the preempted task. In contrast, in preemption centric approach (Basumallick and Nilsen, 1994) an upper bound on the CRPD is added to the WCET of the preempting task. Ward et al. (Ward et al., 2014) showed that task-centric approaches can be very pessimistic when the number of tasks and effectively the number of possible preemptions are high. Whereas, preemption-centric approaches can results in overestimations when tasks have highly variant working set sizes (WSSs). To remove the pessimism in task-centric and preemption-centric approaches Ward et al. (Ward et al., 2014) proposed a mixed-approach where CRPD is accounted for in the WCET of both the preempting and the preempted task. However, accounting for CRPDs in the WCET of a task can result in pessimistic WCET bounds which in turn will result in low processor utilization. Busquets-Mataix et al. (Busquets-Mataix et al., 1996) instead proposed an alternate approach to disintegrate the CRPD and the WCET of tasks. In (Busquets-Mataix et al., 1996) authors compute an upper-bound on the CRPD due to one preemption from a higher priority task τ_j of a low priority task τ_i using Equation. (3.1). The resulting value of the CRPD is then added into the classical WCRT analysis given by Joseph and Pandya (Joseph and Pandya, 1986). The resulting WCRT based schedulability analysis is given by the following equation:

$$R_i^{k+1} = C_i + \sum_{\forall j \in \mathrm{hp}(i)} \left\lceil \frac{R_i^k}{T_j} \right\rceil \times (C_j + \gamma_{i,j}^{one}) \tag{3.13}$$

where $\gamma_{i,j}^{one}$ accounts for the CRPD one job of task τ_i may suffer due to preemption by a higher priority task τ_j. Altmeyer et al. (Altmeyer et al., 2011) also used the same WCRT analysis but bounded the value of CRPD using Equation. (3.4). Essentially, the WCRT formulation given by Equation (3.13) can be used by any analysis that explicitly considers the CRPD each job of task τ_j may cause on task τ_i. However, a disadvantage of analyses by Equation (3.13) is that the worst-case CRPD cost $\gamma_{i,j}^{one}$ is always assumed each time task τ_i is preempted by task τ_j. As a result, some cache evictions can be included multiple times.

Staschulat et al. (Staschulat et al., 2005) used a slightly different formulation of the schedulability analysis as they already accounted for the number of possible preemptions of each task. Instead of computing the CRPD cost due to one preemption of task τ_i by task τ_j, the work in (Staschulat et al., 2005) computes an upper bound on the total CRPD task τ_i may suffer due to all jobs of task τ_j that may execute in a time interval of length t. The resulting CRPD bound is then incorporated into the schedulability analysis. Similar to (Staschulat et al., 2005), Altmeyer et al. (Altmeyer et al., 2012) and Markovic et al. (Marković et al., 2020a) also used the same formulation of WCRT analysis when using the multi-set based approaches that compute CRPD of tasks over a given time interval. The schedulability analysis proposed by (Staschulat et al., 2005; Altmeyer et al., 2012) when accounting for CRPDs in the WCRT analysis is given as follows:

$$R_i^{k+1} = C_i + \sum_{\forall j \in \mathrm{hp}(i)} \left\lceil \frac{R_i^k}{T_j} \right\rceil \times C_j + \sum_{\forall j \in \mathrm{hp}(i)} \gamma_{i,j}^{mul} \tag{3.14}$$

where $\gamma_{i,j}^{mul}$ is an upper bound on the total CRPD task τ_i may suffer due preemptions by a higher priority task τ_j in a time interval of length t. In general, Equation (3.13) and Equation (3.14) both can be used to incorporate CRPDs into the schedulability analysis depending the method used to compute the CRPD cost.

3.3 Other Approaches to Handle Intra- and Inter-task Cache Interference

The focus of this work is on the timing analysis techniques where caches are used without restrictions, however many approaches have been presented in literature to explicitly manage intra- and inter-task cache behavior by using different methods such as cache partitioning, cache locking, task layout optimization and enhanced scheduling models.

3.3.1 Cache Partitioning and Locking

Cache partitioning aims to eliminate potential inter-task cache conflicts by partitioning the cache between tasks. The cache is divided into several sets or partitions which might be of different sizes. These partitions are then assigned either exclusively to different tasks or are shared between a subset of tasks to reduce inter-task cache interference. Different approaches have been presented in literature for cache partitioning based on hardware (Kirk and Strosnider, 1990; Chousein and Mahapatra, 2005) and software based implementations (Wolfe, 1993; Mueller, 1995; Liedtke et al., 1997; Suhendra and Mitra, 2008). However, software-based cache partitioning is usually preferred due to its several advantages over the hardware based cache partitioning (Mueller, 1995). Software based cache partitioning approaches are either implemented in the OS or in the compiler. First software based approach to cache partitioning was proposed by Wolfe (Wolfe, 1993). His approach was similar to the page coloring approach later proposed by Liedtke et al. (Liedtke et al., 1997). Page coloring is the most commonly used software based cache partitioning technique (Mueller,

1995; Liedtke et al., 1997; Guan et al., 2009). It uses virtual to physical address translation at the OS-level to map page addresses to predefined cache regions to avoid overlap between cache space. Another approach that uses page coloring to reduce inter-task cache interference was presented by Liedtke et al. (Liedtke et al., 1997). When using cache partitioning, usually the main goal is to find the number of partitions and their respective sizes. The ideal scenario is where each task is assigned its own private cache partition, i.e., to entirely eliminate inter-task cache interference. However, this is usually not possible due to limited cache space. On the other hand, if tasks are allocated smaller amount of cache space w.r.t the sizes of the tasks, the intra-task cache interference may increase leading to an increase in the WCET of tasks. This is the reason why cache partitioning based approach are usually subjected to a trade off between intra- and inter-task cache interference (Kim et al., 2013).

Busquets et al. (Busquets-Mataix et al., 2000) proposed a cache partitioning technique that allows some tasks to share a cache partition, whereas some tasks are allocated individual partitions. They based their cache partition assignment on task priorities, i.e., higher priority tasks will be assigned to same-sized private partitions, whereas the remaining tasks with lower priorities will share one common partition. Plazar et al. (Plazar et al., 2009) proposed an approach to cache partition size selection with the goal of minimizing the total processor utilization. Cache partitioning problem was considered as an optimization problem by Bui et al. (Bui et al., 2008). The authors used genetic algorithms to minimize the worst-case system utilization such that the sum of all cache partitions cannot exceed the total cache size. Altmeyer et al. (Altmeyer et al., 2014, 2016) presented a cache partitioning algorithm that is optimal under certain cache-modeling assumptions. However, the authors concluded that the trade off between intra- and inter-task cache interference often favors sharing the cache rather than partitioning it.

Cache locking refers to the idea to prevented some cache lines from being overwritten once loaded into the cache. It is a hardware feature which is not available in all modern processors. Cache locking provides a predictable and controllable access to shared caches hence, improving the performance of real-time applications. A number of hardware and software based approaches have been proposed in literature that use the cache locking mechanism. The concept of cache locking in the context of hard real-time system was first introduced by Campoy et al. (Campoy et al., 2001). The authors proposed a genetic algorithm that selects the best instructions of tasks to be locked in the cache such as to minimize the WCET of tasks. Their approach was designed for preemptive systems and the results showed that their approach calculates the response time of tasks with negligible overestimation. Puaut and Decotigny (Puaut and Decotigny, 2002) presented two algorithms that use the memory access patterns of tasks to determine which instructions should be locked in the cache. Their approaches were designed to minimize intra-task and inter-task cache interference. The experimental results show that the algorithms presented by Puaut and Decotigny (Puaut and Decotigny, 2002) have a much better performance in comparison to the static cache analysis. Puaut and Arnaud (Puaut and Arnaud, 2006) presented a dynamic cache locking scheme. Basic blocks are extracted using the CFGs of tasks and these basic blocks are then mapped to different regions in the cache. These regions can then be locked or unlocked based

on different parameters used by the locking algorithm. Falk et al. (Falk et al., 2007) and Vera et al. (Vera et al., 2003) proposed modifications to the compiler to extract information regarding the data/instruction access pattern by tasks. This information was then used to lock cache lines. Liu et al. (Liu et al., 2009a,b) also extracted the information available at compile time to propose algorithms to perform cache locking considering instruction caches. The objective was to minimize the worst-case CPU utilization by using both static and dynamic cache locking. The approaches presented in (Liu et al., 2009a,b) are applicable to both single- and multi-tasking systems and show a better performance than the approach presented in (Falk et al., 2007). An ILP based dynamic cache locking scheme was presented by Aparicio et al. (Aparicio et al., 2011). The idea is to lock the mostly used cache lines into the instruction cache at every context switch. Their approach targeted HRT systems and deals with both intra- and inter-task cache interferences.

In more recent works, Ding et al. (Ding et al., 2014) highlighted the limitations of region-based dynamic locking and proposed a partial cache locking strategy that can exploit the benefits of the unlocked cache lines. The key idea of this work was the notion of loop-driven locking, i.e., given a series of nested loops, each line selected for locking at an inner loop can also be unlocked/locked at the exit/entry point of any of the outer loops. The approach presented by Ding et al. (Ding et al., 2014) result in a better performance in comparison to the region-based dynamic locking as proposed by Puaut and Arnaud (Puaut and Arnaud, 2006). The first work to evaluate the combination of cache partitioning and locking using a real hardware and OS in the context of multi-core real-time systems was presented by Mancuso et al. (Mancuso et al., 2013). They use a profiling mechanism to analyze the memory access pattern of tasks and obtain the most frequently accessed memory pages. Page coloring is then used to optimize the task placement in the cache. Furthermore, cache locking by-line, by-master and by-way was then used to provide intra- and inter-core cache isolation between tasks. Their approach was implemented on the Linux kernel and was evaluated on the actual hardware, i.e., ARM Cortex-A9. The work in (Mancuso et al., 2013) was extended and ported to Freescale P4080 in (Mancuso et al., 2015).

3.3.2 Task Layout Optimization

The position of a memory reference in the main memory influences its location in the cache. Task layout optimization techniques focus on reducing the intra- and inter-task cache interference by modifying the position of code segments within a task and by changing the layout of entire task in the main memory without necessarily creating cache partitions. To reduce intra-task cache conflicts, i.e., decrease in the number of cache misses during the execution of a task in isolation, Tomiyama and Yasuura (Tomiyama and Yasuura, 1997) proposed a code placement technique. The problem is formulated as an integer linear programming (ILP) problem, by which an optimal placement of task's code segment in cache is found. Focusing on intra-task cache interference, Kowarschik and Weiss (Kowarschik and Weiß, 2003) also proposed several data layout optimization techniques such as loop interchange, etc., to increase locality and reduce the number of cache misses. Lokuciejewski et al. (Lokuciejewski et al., 2008) proposed different algorithm aiming to have tighter WCET bounds. They focused on functions that are frequently called by the task and

tried to allocate them contiguous space in the main memory. In particular, they use a greedy algorithm and a heuristic to achieve such a goal. A cache-aware code positioning approach has been proposed by Falk and Kotthaus (Falk and Kotthaus, 2011). Their goal was to reduce intra-task cache interference by building conflict graphs between different code segments of a task. Tasks are split in fragments and a greedy approach-based heuristic is used to position different fragments in the main memory. Mezzetti and Vardanega (Mezzetti and Vardanega, 2013) also proposed a cache-aware approach that optimizes procedure positioning to favor incremental development for industrial needs.

Focusing on inter-task cache conflicts, Gebhard and Altmeyer (Altmeyer and Gebhard, 2007) proposed an approach to optimize task layout in memory to improve task set schedulability by minimizing the inter-task cache conflicts. Their approach showed that with different task layouts in memory inter-task cache interference can be significantly reduced. However, their approach to bound the CRPDs was pessimistic since all ECBs of a task were treated as UCBs. Lunniss et al. (Lunniss et al., 2012) later improved the work done in (Altmeyer and Gebhard, 2007) by using a more tighter approach for CRPD calculation. They proposed a simulated annealing based approach to optimize task layout in memory to reduce the inter-task cache interference and effectively improve schedulability. It has been identified in (Altmeyer et al., 2014, 2016) that an optimized layout of tasks in memory can outperform an optimal cache partitioning approach in different scenarios.

3.3.3 Enhanced Scheduling Models

To reduce the impact of preemptions on task's timing properties *limited-preemptive* scheduling techniques have also been proposed (Wang and Saksena, 1999; Baruah, 2005; Burns, 1993). Limited preemption models attempt to minimize preemption overheads by reducing the number of allowed preemptions and/or allowing preemption at program locations where the effect of preemption is minimized. Limited preemptive scheduling can be subdivided into different streams, i.e., preemption thresholds scheduling (Wang and Saksena, 1999), fixed preemption points scheduling (Burns, 1993) and deferred preemption scheduling (Baruah, 2005). In preemption threshold scheduling, each task is assigned a regular priority and a preemption threshold, and the preemption is allowed to take place only when the priority of the arriving task is higher than the threshold of the running task. In fixed preemption point scheduling, a task implicitly executes in non-preemptive mode and preemption is allowed only at predefined locations inside the task code. In deferred preemption scheduling, for each task the longest interval that can be executed non-preemptively is specified, a higher priority task can only preempt a lower priority task after the finalization of this time interval. Historically, limited preemptive scheduling methods only aim to increase schedulability by controlling preemptions without considering the preemption cost, e.g., CRPD. However, under the limited preemptive scheduling paradigm, there exist few approaches in literature that focus on incorporating intra- and inter-task cache interference into schedulability analysis.

Marinho et al. (Marinho et al., 2012a,b) computed an upper-bound on the inter-task cache interference that any task may suffer under deferred preemption scheduling. A function based on

task's execution flow is used to compute the potential inter-task cache interference at a given program point during the execution of task. The resulting cost is then incorporated into the WCET of task. Bril et al. (Bril et al., 2014) presented a schedulability analysis for preemption threshold scheduling that also accounts for inter-task cache interference. The techniques presented in (Altmeyer et al., 2012) to bound inter-task cache interference under fully preemptive scheduling are adapted to preemption threshold scheduling. Under fixed preemption point scheduling, Ramaprasad and Mueller (Ramaprasad and Mueller, 2006) proposed an approach to computer inter-task cache interference at different preemption points and due to preemption patterns in fully preemptive periodic tasks. They also proposed a WCRT analysis (Ramaprasad and Mueller, 2008) that accounts for CRPD suffered by periodic tasks having only one non-preemptive region when scheduled using fixed preemption point scheduling. The main challenge in fixed preemption point scheduling is the selection of preemption points. To this end, several preemption point selection algorithms have been proposed by Bertogna et al. (Bertogna et al., 2011), Peng et al. (Peng et al., 2014) and Cavicchio et al. (Cavicchio et al., 2015). Cavicchio et al. (Cavicchio et al., 2015) identified the relationship between inter-task cache interference and preemption point selection and proposed different algorithms for preemption point selection that reduces inter-task cache interference. In his PhD dissertation (Marković, 2020), Filipe Markovic has exclusively worked on tightening the CRPD bound for tasks scheduled using fixed preemption point scheduling producing several publications (Markovic et al., 2017; Marković et al., 2018, 2020b).

3.4 Memory Bus Contention Analysis

Several works have been presented in literature to bound additional delays that impact task executions due to memory bus contention and integrate these delays into schedulability analyses (Maiza et al., 2019).

In one of the earlier works Rosen et al. (Rosen et al., 2007a) proposed a TDMA based bus arbitration policy to bound memory bus contention. A static schedule is used to allocate different time slots to tasks that need to access the bus. A dedicated memory directly connected to the bus arbiter is used to store this schedule. The approach resulted in a reasonable performance since it prevented any deadlines miss due to contention at the bus. However, the approach used is not flexible and has many pitfalls, i.e., it assumes a table-driven arbiter which are typically not available in modern processors and it also needs to know the workload of tasks running on the system apriori, in order to avoid situations where the bus contention increases the memory access latency.

Schliecker et al. (Schliecker et al., 2010) proposed an event based model to bound the shared resource (i.e., memory bus) contention. In this approach, tasks that are concurrently executing on the processor can access the global resources using events that define the maximum and minimum access to a resource in a given time window. Every task is assigned a static priority and the time to make a resource transaction is bounded. Therefore, the worst-case interference of a task can be bounded by considering the interference from all its higher priority tasks. The problem with this

analysis is that it can overestimate the number of requests to a resource, since it always considers a minimum time interval between two accesses to a resource.

Kelter et al. (Kelter et al., 2011) and Chattopadhyay et al. (Chattopadhyay et al., 2010) proposed a WCRT analysis techniques to bound memory bus contention considering a TDMA bus and a L1 instruction cache. However, these approaches have limited applicability as they assumed separate buses and memories for both code and data which is uncommon in commodity hardware. Moreover, these methods assume non-preemptive scheduling and therefore does not account for inter-task cache interference.

Schranzofer et al. (Schranzhofer et al., 2010, 2011) presented a resource adaptive framework for the WCRT analysis of real-time tasks. In their work, the authors proposed a task model in which tasks are composed of superblocks, with each superblock having a unique entry and exit point. These superblocks execute in sequence with each superblock having a WCET and a worst-case number of access requests to a shared resource. Furthermore, these superblocks are assumed to have different execution phases, i.e., acquisition, execution and replication phases. Based on the operation of these phases different task execution models have been presented. However fitting tasks into these models is a cumbersome task. For example, the dedicated model requires to know apriori the memory access patterns of tasks in order to prefetch the required data for the computation phase. This requires the communication phases to be synchronized with the availability of the bus slot for that task, which may not hold even for a predictable arbiter employing TDMA.

In a similar work, Pellizzoni et al.(Pellizzoni et al., 2010) proposed an approach that derives arrival curves for the memory access patterns of tasks and compute an upper bound to the memory contention delay incurred. Their approach is based on the concept that tasks are composed of superblocks that are executed in pre-assigned time slots. Experimental results show that this approach can be used to bound the memory delay incurred by tasks however, with large number of tasks the proposed model limits the applicability of the solution.

Dasari et al. (Dasari et al., 2011a) presented an analysis to bound the maximum number of bus requests that can be made by a task in a given time interval using performance counters. Consequently, the bus contention suffered by the tasks is modeled as an additional term in the WCRT analysis. This work was later extended in Dasari et al. (Dasari et al., 2016). Although their analysis provides reasonably precise estimates on the memory access demand of tasks but, it uses non-preemptive scheduling and assumes partitioned caches and therefore does not take cache related effects into account. Huang et al. (Huang et al., 2016) also presented a WCRT analysis that accounts for shared memory bus contention that uses a fixed-priority arbitration scheme. Their analysis has a speedup factor of 7 when used with a simple task allocation algorithm. However, their model does not consider the impact of caches on the memory access demand of tasks which may potentially lead to optimistic results. Davis et al. (Davis et al., 2018b) explicitly modeled interference on cores, caches, memory bus and the main memory in a multicore system. The analysis in (Davis et al., 2018b) also accounts for inter-task cache interference, i.e., CRPDs, when bounding memory bus contention that can be suffered by the tasks under different bus arbitration policies.

3.5 Different Perspective of Caches

Most of the state-of-the-art approaches that focus on analyzing the impact of caches on the timing analysis of tasks look at the negative perspective of caches, i.e., due to inter-task cache interference or CRPDs that have a negative impact on schedulability. However, caches can also have a positive impact on schedulability due to *re-use* of cached content between different code segments within a task or between different job executions of the same task. This cache re-use has been considered in the existing analysis when computing the WCET of tasks in the form of persistence analysis (see Section 3.1.3) however, the notion of cache re-use can also be used across different job executions of tasks. To illustrate, consider Figure 3.7 that depicts the same example schedule as shown in Figure 3.4 but presents a different perspective. In Figure 3.7, while the red boxes show

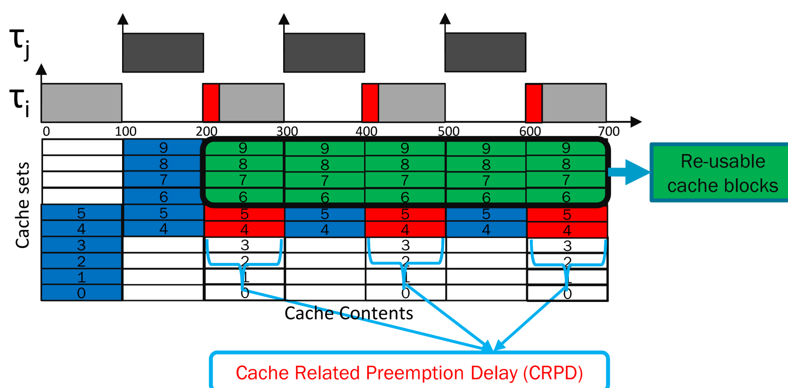


Figure 3.7: Example schedule to highlight re-usable cache blocks between different jobs of task τ_i

cache blocks of task τ_i that needs to be reloaded from the main memory due to evictions by the preempting task τ_j, i.e., CRPD, the green boxes show cache blocks of task τ_j that remain in the cache after the execution of first job of τ_j. All cache blocks represented using green boxes will be already available in the cache when second and third job of task τ_j start executing. Consequently, these existing cache blocks can be reused by the second and third job of task τ_j, which results in reducing the number of main memory accesses generated by those jobs. This reduction in main memory accesses may also lead to a reduction in the execution time of second and third job of task τ_j eventually, tightening the WCRT of task τ_i.

There exist few works in the state-of-the-art that exploit the observation shown in Figure 3.7. In one of the earlier works, Nemer et al. (Nemer et al., 2007, 2008) presented analyses for both direct-mapped and set-associative caches that consider the cache re-use between different jobs of a task and showed that inter-task cache reuse can have a significant affect on schedulability. Their approaches are based on the computation of entry and exit cache states after the execution of each job of a task that leads to a set of memory accesses that must result in cache hits due to previous instances of a task. However, their approaches are limited to non-preemptive task sets under static scheduling and do not apply to preemptive systems with commonly used priority based scheduling schemes. Recently, in his PhD dissertation (Tessler, 2019), Corey Tessler has highlighted the benefits of cache re-use between different threads of a multi-threaded task. In the

context of multi-thread real-time systems, Tessler et al. (Tessler and Fisher, 2016, 2018, 2019) has made several important contribution with focus on inter-thread cache benefits.

In the works of Nemer et al. (Nemer et al., 2007, 2008) and Tessler et al. (Tessler and Fisher, 2016, 2018, 2019) the basic idea is the same, i.e., a *positive* perspective of caches, which is also the main focus of this thesis. However, we consider a more generic system model, i.e., fixed-priority fully preemptive scheduling, with single threaded real-time tasks. We will start by exploring the impact of intra-task cache re-use on the inter-task cache interference suffered by the task considering single-level direct mapped caches. We will then extend our analysis to set-associative LRU and multi-level caches. Finally, we will demonstrate how a tighter bound on the inter-task cache interference due to inter-task cache reuse may impact the memory bus contention in multicore systems.

Part I

Analysis of Single-level Direct-mapped Caches

Chapter 4

Using Cache Persistence to Improve the Bounds on Inter-task Cache Interference

As discussed in Chapter 3 (Section 3.2) many different approaches have been presented in the state-of-the-art to bound inter-task cache interference. These approaches (Busquets-Mataix et al., 1996; Lee et al., 1998; Tomiyama and Dutt, 2000; Staschulat et al., 2005; Tan and Mooney, 2007; Altmeyer et al., 2011, 2012; Marković et al., 2020a) use the set of ECBs and UCBs of tasks to bound inter-task cache interference (or more specifically CRPD) and incorporate it into the WCRT analysis. However, all these approaches may result in pessimistic WCRT bounds due to the fact that they only consider the effect of preemptions on the memory access demand of the preempted task, but not the *variation* in memory access demand of the preempting tasks. Instead, they all assume that every job of a higher priority task τ_j preempting a lower priority task τ_i will ask for its maximum memory access demand, i.e., its worst-case memory access demand in isolation. Although this may be true for the first job released by the preempting task τ_j, subsequent jobs of τ_j may re-use most of the data and instructions that were already loaded in the cache during the execution of its previous jobs (e.g., see Figure 3.7).

In this chapter, we have addressed this issue by proposing a novel analysis that captures the re-use of cache blocks between job executions, to reduce the negative impact of caches on the WCRT bound. Our approach is orthogonal to state-of-the-art methods used for CRPD calculations and can be used independently with any of the methods described in Section 3.2. The main contributions made in this chapter are as follows:

1. We introduced the concept of *persistent cache blocks* (PCBs) in the context of WCRT analysis. PCBs are cache blocks that, once loaded into the cache by a task τ_i will never be evicted when τ_i runs *in isolation*. This concept allows us to capture the re-use of cache blocks between executions of the same task and reduce the memory access demand for subsequent jobs of a task, making its memory access demand *variable*,

2. A cache-persistence-aware WCRT analysis for fixed-priority preemptive systems that exploits the variable memory access demand of preempting tasks to tighten the WCRT bound,

3. An extension of the proposed WCRT analysis to a multi-set approach that further improves the WCRT bound by considering the total memory access demand of the preempting tasks over a task's response time rather than the worst-case memory access demand of each independent job, and

4. An experimental evaluation showing that our cache-persistence-aware WCRT analysis results in up to 13% higher task set schedulability than state-of-the-art approaches.

4.1 Assumptions on the System Model

In addition to the general system model detailed in Chapter 2, in this chapter we make the following assumptions on the system model.

- We consider a single-core platform with a single level (L1) direct-mapped instruction cache.
- We consider a task set Γ comprising n sporadic constrained deadline tasks $\{\tau_1, \tau_2, \ldots \tau_n\}$. Each task $\tau_i \in \Gamma$ is defined by a triplet (C_i, T_i, D_i), where C_i is the worst-case execution time (WCET) of τ_i, T_i is its minimum inter-arrival time and D_i is the relative deadline of each instance or job of τ_i.
- In addition to the WCET C_i, we use separate terms to measure the worst-case processing demand and memory access demand of each task. PD_i denote the worst-case processing demand of τ_i, i.e., it only accounts for execution requirements of τ_i and does not include the time spent by τ_i to perform memory operations. MD_i denote the worst-case memory access demand of any job of task τ_i, i.e., the maximum time during which any job of τ_i is performing memory operations. Note that the value of C_i, PD_i and MD_i are determined assuming τ_i executes in *isolation*. It obviously hold that $C_i > PD_i$ and $C_i > MD_i$, but it also holds that $C_i \leq PD_i + MD_i$[1] since PD_i and MD_i may result from different execution scenarios of τ_i along different execution paths (e.g., due to different inputs).
- The WCRT of task τ_i, denoted by R_i, is defined as the longest time between the arrival and the completion of any job of τ_i.
- In this work, we consider that preemption costs only refer to additional cache reloads due to those preemptions. Other overheads, e.g., due to context switches, scheduler invocations and pipeline flushes are assumed to be included in the WCET.
- We assume a timing-compositional architecture (Hahn et al., 2015), i.e., the timing contribution of memory overheads can be analyzed separately from other architectural features.

[1] It is experimentally confirmed since $C_i \leq PD_i + MD_i$ for several benchmarks from the Mälardalen Benchmark Suite (Gustafsson et al., 2010) that were analyzed using the Heptane (Hardy et al., 2017) static WCET estimation tool for a MIPS R2000/R3000 architecture.

- The worst-case time to reload on cache block from the main memory is bounded by d_{mem}.

The list of important symbols used in this chapter is provided in Table 4.1.

Table 4.1: List of important symbols used in Chapter 4

Symbol	Description
Γ	Task set of size n
τ_i	Task with index i
C_i	Worst-case execution time of task τ_i in isolation
T_i	Minimum inter-arrival time of task τ_i
D_i	Relative deadline of task τ_i
R_i	Worst-case response time of task τ_i
R_i^{un}	Worst-case response time of task τ_i computed using the CPRO-union approach
R_i^{mul}	Worst-case response time of task τ_i computed using the CPRO multi-set approach
PD_i	Worst-case processing demand of task τ_i in isolation
MD_i	Worst-case memory access demand of task τ_i in isolation
MD_i^r	Residual memory access demand of task τ_i in isolation
$\hat{MD}_i(t)$	Total memory access demand of task τ_i in a time interval of length t
$\text{hp}(i)$	The set of tasks with higher priority than τ_i
$\text{hep}(i)$	The set of tasks with higher priority than τ_i including τ_i, i.e., $\text{hep}(i) = \text{hp}(i) \cup \tau_i$.
$\text{aff}(i, j)$	The set of tasks with priorities higher than or equal to the priority of τ_i (including τ_i), but strictly lower than that of τ_j. This set contains the intermediate priority tasks, which may affect the response time of τ_i, but may also be preempted by τ_j.
d_{mem}	Time to reload one cache block from the main memory
MO_i	Memory overhead of task τ_i
$E_k(R_i)$	The maximum number of jobs any task τ_k can release during the response time R_i of task τ_i
ECB_i	The set of evicting cache blocks (ECBs) of task τ_i
UCB_i	The set of useful cache blocks (UCBs) of task τ_i
PCB_i	The set of persistence cache blocks (PCBs) of task τ_i
$nPCB_i$	The set of non-persistence cache blocks (nPCBs) of task τ_i
$M_{j,i}^{pcb}$	Multiset containing set of PCBs of task τ_j
$M_{j,i}^{ecb}$	Multiset containing set of ECBs of all task in $\text{hep}(i) \setminus \tau_j$
$M_{j,i}^{ecb-aff}$	Multiset containing set of ECBs of all task in $\text{aff}(i, j)$
$M_{j,i}^{ecb-hp}$	Multiset containing set of ECBs of all task in $\text{hep}(j) \setminus \tau_j$
$\rho_{j,i}$	Cache persistence reload overhead (CPRO) of one job of task τ_j during the response time of task τ_i.
$\hat{\rho}_{j,i}$	Total cache persistence reload overhead (CPRO) of task τ_j in an interval of length t while executing during the response time of task τ_i.
$\gamma_{i,j}(t)$	Total cache related preemption delay (CRPD) suffered by task τ_i in a time interval of length t due to preemptions by a higher priority task $\tau_j \in \text{hp}(i)$.
$\gamma_{i,j}^{mul}$	Upper bound on the total cache related preemption delay (CRPD) suffered by task τ_i in a time interval of length t due to preemptions by a higher priority task $\tau_j \in \text{hp}(i)$.
$\rho_{j,i}^{mul}(t)$	Upper bound on the total cache persistence reload overhead (CPRO) of task τ_j in an interval of length t while executing during the response time of task τ_i.

4.2 Problem Definition

In this section, first we provide a basic example to affirm the motivation behind the work presented in this chapter. Later, using this example as a base we provide some useful definitions that will be used in rest of the chapter.

4.2.1 Motivational Example

As discussed in Chapter 3 (Section 3.2), the impact of a higher priority task τ_j on the WCRT of any lower priority task τ_i can be estimated in a fairly accurate manner by analyzing the mapping of UCBs and ECBs in the cache, i.e., by computing the CRPD caused by task τ_j on task τ_i. However, the impact of τ_i on the memory access demand of τ_j is ignored during the WCRT analysis of τ_i. Yet, higher priority tasks may often execute more than one job during the response time of a lower priority task. Therefore, to accurately estimate the WCRT of a lower priority task τ_i, one must consider the impact of the preempted tasks on the memory access demand of each job released by the preempting tasks. In the literature, this is dealt with by assuming that the memory access demand for each job of a higher priority task τ_j executing within the response time of a lower priority task τ_i is always maximum, i.e, equal to the maximum memory access demand MD_j. Following that assumption, the total memory overhead MO_i that must be accounted by τ_i during its worst-case response time is upper bounded by the following equation derived in (Altmeyer et al., 2015; Davis et al., 2018b).

$$MO_i = MD_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \times (MD_j + \gamma_{i,j}) \tag{4.1}$$

There is a significant level of pessimism involved in Equation (4.1), as we will demonstrate using the example below.

Example 4.1. *Consider the two tasks τ_1 and τ_2 (where τ_1 has a higher priority than τ_2) presented in Figure 4.1. We assume that the time d_{mem} needed to access the main memory and load a memory block to the cache is equal to* 10 *time units and that the memory access demand of τ_1 and τ_2 are $MD_1 = 60$ and $MD_2 = 80$[2], respectively. We also assume that memory block $\{9\}$ accessed by τ_1 contains data that must be updated at the beginning of the execution of each of its jobs. Figure 4.1 depicts a possible schedule together with the evolution of the cache contents over time. The memory blocks that must be loaded/reloaded from the main memory after each preemption or resumption are shown in bold with a bigger font size in Figure 4.1.*

Initially, the cache is empty and τ_2 loads all its ECBs from the main memory as soon as it starts to execute. When τ_1 preempts τ_2 for the first time, it also loads all its ECBs into the cache with a memory access demand of $MD_1 = 60$. Since there is an overlap between the mapping of ECBs of τ_1 and the mapping of UCBs of τ_2 in the cache, τ_1 evicts some of the useful cache blocks

[2] Note that because the same cache block may be used by several memory blocks of the same task τ_i, the worst-case memory access demand MD_i of τ_i may be larger than the number of ECBs of τ_i multiplied by d_{mem}.

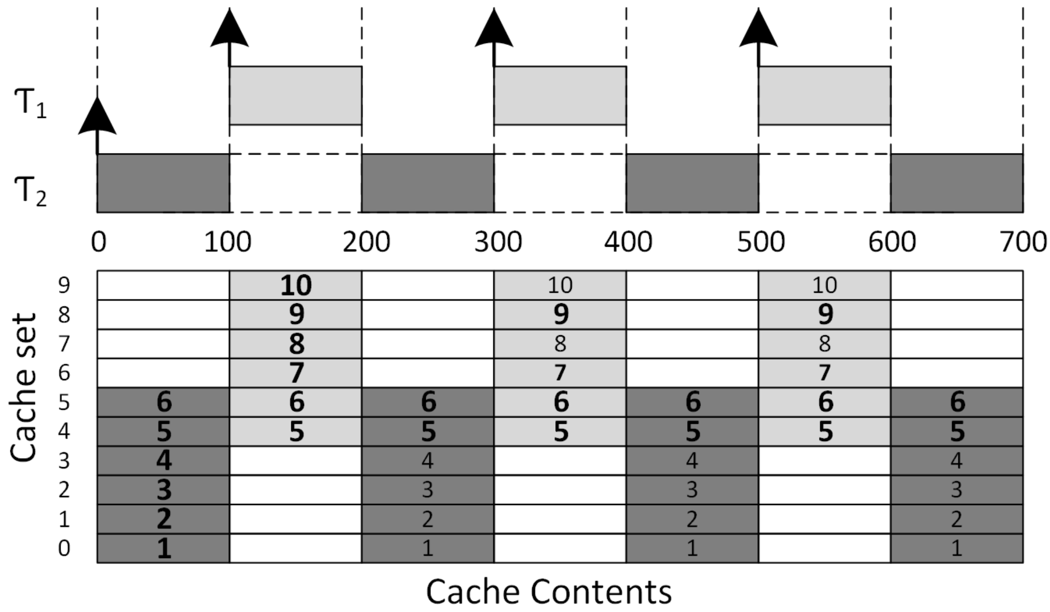


Figure 4.1: Schedule and cache contents for a taskset $\{\tau_1, \tau_2\}$ with $C_1 = 100$, $C_2 = 400$, $MD_1 = 60$, $MD_2 = 80$, $ECB_1 = \{5,6,7,8,9,10\}$, $ECB_2 = \{1,2,3,4,5,6\}$, $UCB_1 = \{6,7\}$, $UCB_2 = \{5,6\}$, $PCB_1 = \{5,6,7,8,10\}$ and $PCB_2 = \{1,2\}$. The schedule assumes that τ_1 releases its first job with an offset of 100 time units.

of τ_2. In turn, when τ_2 resumes its execution, it has to account for $\gamma_{2,1} = 2 \times d_{mem} = 20$, in order to load cache blocks $\{5,6\}$ again from main memory. However, when the second job of τ_1 preempts τ_2, one can notice that it no longer needs to reload all of its ECBs. In fact, most of the memory blocks needed by τ_1 are still in the cache. As a consequence, τ_1 must only reload memory blocks $\{5,6\}$, which have been evicted by τ_2, as well as memory block $\{9\}$ that must be reloaded for each new job execution of τ_1. The same scenario happens for all jobs released by τ_1, except the first one. The actual memory access demand for the second and third job of τ_1 is hence much less (i.e., 30) than $MD_1 = 60$, illustrating that it is not constant across all job executions.

In the presented example, memory blocks $\{5,6,7,8,10\}$ are called *persistent cache blocks* (PCBs), as they are never evicted from the cache once loaded when τ_1 executes in isolation. In contrast, cache block $\{9\}$ is a *non-persistent cache block* (nPCB). nPCBs may be cache blocks that are shared by several memory blocks of the same task, or simply data (e.g., sensor readings, value on an input port, global shared data) that must be reloaded before each access. One must note that PCBs and nPCBs are different from the notions of UCBs and ECBs in the sense that it does not matter if they are referenced more than once during a single execution of a task. However, a PCB must never be evicted from the cache by the task itself once it is fetched from main memory.

The state-of-the-art does not consider PCBs while calculating the memory overhead suffered by a task τ_i in case of preemptions. This results in pessimistic memory overhead evaluations and hence pessimistic WCRT computations. This can easily be shown using the example in Figure 4.1.

If τ_2's memory overhead is computed using Equation (4.1), one would get:

$$MO_2 = MD_2 + 3 \times MD_1 + 3 \times \gamma_{2,1} = 80 + 3 \times 60 + 3 \times 20 = 320$$

Equation (4.1) considers the worst-case memory access demand, i.e., MD_1 for each job of τ_1 that executes during the response time of τ_2. As we have shown in Example 4.1, the actual memory access demand of the second and third job of τ_1 is in fact much less. Considering the PCBs of τ_1 while calculating the memory overhead MO_2, the resulting value is given as:

$$\begin{aligned} MO_2 &= MD_2 + MD_1 + 2 \times (MD_1 - |PCB_1| \times d_{mem}) \quad + 3 \times \gamma_{2,1} \\ &= 80 + 60 + 2 \times (60 - 5 \times 10) + 3 \times 20 = 220 \end{aligned}$$

This simple example highlights the necessity to consider PCBs when calculating the memory access demand and hence the WCRT of a task.

4.2.2 Problem Formalization

The previous example casually introduced the notions of PCB and nPCB. We now formally define those two types of cache blocks associated to the execution of a task τ_i.

Definition 4.1 (Persistent cache block)**.** *A memory block of a task τ_i is persistent if once loaded by τ_i, it will never be invalidated or evicted from the cache when τ_i executes in isolation.*

Definition 4.2 (Non-persistent cache block)**.** *A non-persistent cache block (nPCB) of task τ_i is an ECB that is not a PCB. That is, it is a memory block that may need to be reloaded at some point during the execution of τ_i (in the same or different jobs), even when τ_i executes in isolation.*

The sets of PCBs and nPCBs associated to a task τ_i are denoted by PCB_i and $nPCB_i$, respectively. It follows from the two previous definitions that each cache block associated to a task τ_i (ECB_i) is either a PCB or a nPCB, hence the following two relations:

$$PCB_i \cup nPCB_i = ECB_i \tag{4.2}$$

$$PCB_i \cap nPCB_i = \emptyset \tag{4.3}$$

By Definition 4.1, if τ_i executes in isolation, a PCB is loaded only once from the main memory and hence contributes only once to the total memory access demand of τ_i. Even though all the ECBs of τ_i (i.e., PCBs and nPCBs) contribute to its worst-case memory access demand in isolation (i.e., MD_i), only the nPCBs, a subset of ECB_i, must be loaded by more than one job of τ_i. Considering the worst-case memory access demand for each job released by higher priority tasks than τ_i when computing the WCRT of τ_i, as is implicitly the case in Equations (3.13) and (3.14), is thus pessimistic. Therefore, we define the *residual memory access demand* of a task τ_i as the worst-case memory access demand of τ_i assuming that all the PCBs of τ_i are already in the cache memory and therefore result in cache hits when being accessed.

Definition 4.3 (Residual memory access demand)**.** *The residual memory access demand MD_i^r of task τ_i is the worst-case memory access demand over all the jobs of τ_i when all its PCBs are already loaded in the cache memory. Therefore, MD_i^r only accounts for the accesses to the nPCBs of τ_i and can occur during any job execution of τ_i.*

An upper bound on the total memory access demand $MD_i(t)$ of a task τ_i within a time window of length t when τ_i executes in isolation is proven in the following lemma.

Lemma 4.1. *If a task τ_i executes in isolation, then its total memory access demand $MD_i(t)$ within a time window of length t is upper bounded by $\hat{MD}_i(t)$ where*

$$\hat{MD}_i(t) \stackrel{\text{def}}{=} \min\left\{ \left\lceil \frac{t}{T_i} \right\rceil MD_i \left\lceil \frac{t}{T_i} \right\rceil MD_i^r + \mid PCB_i \mid \times d_{mem} \right\} \tag{4.4}$$

Proof. We prove that $\left\lceil \frac{t}{T_i} \right\rceil MD_i$ and $\left\lceil \frac{t}{T_i} \right\rceil MD_i^r + \mid PCB_i \mid \times d_{mem}$ are both upper bounds on the total memory access demand $MD_i(t)$ of τ_i. Thus, the minimum of those bounds is also an upper bound on $MD_i(t)$.

1. τ_i can release at most $\left\lceil \frac{t}{T_i} \right\rceil$ jobs in a time window of length t. By definition of MD_i, each of these jobs has a worst-case memory access demand MD_i. Therefore, $\left\lceil \frac{t}{T_i} \right\rceil MD_i$ is an upper bound on the total memory access demand of τ_i.

2. Recall from Equations (4.2) and (4.3) that $PCB_i \cup nPCB_i = ECB_i$ and $PCB_i \cap nPCB_i = \emptyset$. Characterizing the worst-case contribution of the PCBs and nPCBs to the total memory access demand is therefore sufficient to quantify the worst-case contribution of all the cache blocks of τ_i to $MD_i(t)$. Since by Definition 4.1, the persistent cache blocks must be loaded only once, the maximum contribution of the cache blocks in PCB_i to $MD_i(t)$ is $\mid PCB_i \mid \times d_{mem}$ (i.e., the total number of PCBs times the worst-case memory access time). By Definition 4.3, the worst-case contribution of nPCBs to the memory access demand of each job released by τ_i is MD_i^r. Since a maximum of $\left\lceil \frac{t}{T_i} \right\rceil$ jobs are released by τ_i in a time window of length t, an upper bound on the total contribution of the nPCBs to $MD_i(t)$ is given by $\left\lceil \frac{t}{T_i} \right\rceil MD_i^r$. Adding the contributions of nPCBs and PCBs, we get $\left\lceil \frac{t}{T_i} \right\rceil MD_i^r + \mid PCB_i \mid \times d_{mem}$ which is also an upper bound on the total memory access demand of τ_i in a time interval of length t.

□

Although Equation (4.4) provides an upper bound on the total memory access demand of τ_i in *isolation*, the total memory access demand of τ_i when executing concurrently with other tasks can be much larger. Indeed, as can be observed in Example 4.1, the PCBs of a task τ_j can be evicted due to the execution of any task (i.e., tasks in $\text{hep}(i) \setminus \tau_j$) between the execution of two successive jobs of τ_j. This requires the effect of all tasks in $\text{hep}(i) \setminus \tau_j$ on the memory access demand of $\tau_j \in \text{hp}(i)$ during the WCRT of τ_i to be taken into account. We refer to this extra memory access

demand caused by the eviction of PCBs of τ_j by the tasks in $\text{hep}(i) \setminus \tau_j$ as *cache persistence reload overhead* (CPRO) and denote it by $\rho_{j,i}$. CPRO is formally defined as:

Definition 4.4 (Cache persistence reload overhead)**.** *Cache persistence reload overhead, denoted by $\rho_{j,i}$, is the maximum memory overhead of any task τ_j due to eviction of its PCBs resulting from the execution of all tasks in $\text{hep}(i) \setminus \tau_j$, while τ_j is executing during the response time of τ_i.*

4.3 CPRO-union Approach

In this section, we present a simple approach similar to the state-of-the-art ECB-union (see Section 3.2.1) to calculate the CPRO (i.e., $\rho_{j,i}$). We further demonstrate how $\rho_{j,i}$ can be incorporated in the WCRT analysis of a task τ_i. Later, in Section 4.4, we extend this simple union approach into a multi-set variant to remove some of the pessimism associated with this analysis.

4.3.1 Computation of Cache Persistence Reload Overhead

As discussed in Section 4.2.2, $\rho_{j,i}$ accounts for the extra memory access demand of each job of $\tau_j \in \text{hp}(i)$ due to evictions of its persistent cache blocks by other tasks running concurrently on the processor.

As one can see in Figure 4.1, the PCBs of a task $\tau_j \in \text{hp}(i)$ can be evicted by the ECBs of any other task running on the platform between two successive jobs of τ_j. The cache persistence reload overhead $\rho_{i,j}$ can thus be upper bounded by the intersection of the set PCB_j of all PCBs of τ_j with all cache blocks (i.e., ECBs) that can be loaded by any other task between two executions of τ_j. This observation leads to the following theorem.

Theorem 4.1. *The cache persistence reload overhead imposed by the eviction of PCBs of a job of task $\tau_j \in \text{hp}(i)$ on the worst-case response time of a task τ_i is upper bounded by*

$$\rho_{j,i} = d_{mem} \times \left| PCB_j \cap \Big(\bigcup_{\forall \tau_k \in \text{hep}(i) \setminus \tau_j} ECB_k \Big) \right| \tag{4.5}$$

Proof. Since a fixed-priority scheduling algorithm is used, only tasks with priorities higher than or equal to the priority of τ_i (i.e., tasks in $\text{hep}(i)$) can execute during the response time of τ_i. Therefore, any task $\tau_k \in \text{hep}(i) \setminus \tau_j$ can execute between two subsequent jobs of τ_j and hence evict some or all the PCBs of τ_j.

The worst-case memory interference of any task $\tau_k \in \text{hep}(i) \setminus \tau_j$ on τ_j is when it reloads all its cache blocks (i.e., its ECBs) between two subsequent jobs of τ_j. Therefore, the largest set of memory blocks loaded by tasks in $\text{hep}(i) \setminus \tau_j$ between two jobs of τ_j is given by $\bigcup_{\forall \tau_k \in \text{hep}(i) \setminus \tau_j} ECB_k$.

The set of persistent cache blocks that must be reloaded by τ_j during each job execution is thus upper bounded by the intersection between τ_j's PCBs (i.e., PCB_j) and $\bigcup_{\forall \tau_k \in \text{hep}(i) \setminus \tau_j} ECB_k$.

Since each cache block reload takes at most d_{mem} time units, the CPRO due to the eviction of PCBs of τ_j by tasks in $\text{hep}(i) \setminus \tau_j$ is upper bounded by

$$d_{mem} \times \left| PCB_j \cap \Big(\bigcup_{\forall \tau_k \in \text{hep}(i) \setminus \tau_j} ECB_k \Big) \right|$$

□

Having defined an expression to calculate $\rho_{j,i}$, we now define $\rho_{j,i}(t)$, i.e., the total cache persistence reload overhead on τ_j in a time window of length t due to the eviction of its PCBs by tasks in $\text{hep}(i) \setminus \tau_j$. $\rho_{j,i}(t)$ tells us by how much the memory access demand of τ_j can vary in comparison to its memory access demand in isolation (i.e., $MD_j(t)$) due to the interference generated by the other tasks executing concurrently with τ_j. Using Theorem 4.1, $\rho_{j,i}(t)$ can be easily computed as stated in Lemma 4.2 below.

Lemma 4.2. *The total CPRO $\rho_{j,i}(t)$ on the execution time of τ_j due to the eviction of its PCBs by tasks in $\text{hep}(i) \setminus \tau_j$ in a time interval of length t is upper bounded by $\hat{\rho}_{j,i}(t)$ where*

$$\hat{\rho}_{j,i}(t) \stackrel{\text{def}}{=} \left(\left\lceil \frac{t}{T_j} \right\rceil - 1 \right) \times \rho_{j,i} \tag{4.6}$$

Proof. It directly follows from the fact that τ_j releases at most $\left\lceil \frac{t}{T_j} \right\rceil$ jobs in a time interval of length t. As a result, at most $\left(\left\lceil \frac{t}{T_j} \right\rceil - 1 \right)$ evictions can happen *between* two subsequent jobs of τ_j. Since by Theorem 4.1, the CPRO suffered by a job of τ_j is upper bounded by $\rho_{j,i}$, the total overhead $\rho_{j,i}(t)$ is upper bounded by $\left(\left\lceil \frac{t}{T_j} \right\rceil - 1 \right) \times \rho_{j,i}$. □

4.3.2 WCRT Analysis

After showing how cache persistence reload overhead $\rho_{j,i}$ of a high priority task τ_j can be computed, we now describe how it can be integrated into the WCRT analysis of any lower priority task τ_i. As mentioned in Section 3.2.4, the WCRT analysis for fixed-priority preemptive systems was first presented in (Joseph and Pandya, 1986; Audsley et al., 1993) without considering memory overheads due to preemptions. It was then extended in several works (e.g., (Staschulat et al., 2005; Altmeyer et al., 2011; Busquets-Mataix et al., 1996; Altmeyer et al., 2012)) to account for the cache related preemption delays. Some of the most prominent approaches resulted in Equations (3.13) and (3.14), previously presented in Section 3.2.4.

Although these approaches are beneficial, their WCRT analysis still rely exclusively on the WCET C_j of higher priority tasks when computing the worst-case response time of a lower priority task τ_i. That is, it assumes that each job of a task $\tau_j \in \text{hp}(i)$ executing within the response time of τ_i asks for its worst-case memory access demand MD_j. As discussed in Section 4.2, this assumption is pessimistic. In fact, due to the existence of persistent cache blocks, once τ_j loads all its ECBs (i.e., PCBs and nPCBs), subsequent jobs of τ_j will only need to reload nPCBs and some of the PCBs that may have been evicted due to the execution of tasks in $\text{hep}(i) \setminus \tau_j$. As a result, for

subsequent jobs of τ_j the memory access demand will be significantly lower than MD_j. To exploit this variable memory access demand, we present a more elaborate formulation of the WCRT analysis. We propose that for any task τ_i the WCRT of task τ_i is upper bounded by the smallest positive value R_i such that

$$R_i = C_i + \sum_{\forall j \in \text{hp}(i)} \left(PD_j(R_i) + MD_j(R_i) + \rho_{j,i}(R_i) + \gamma_{i,j}(R_i)\right) \tag{4.7}$$

In this WCRT formulation, we separately account for the maximum processing demand $PD_j(R_i)$ and the maximum memory access demand $MD_j(R_i)$ (in isolation) that can be claimed by each higher priority task τ_j within the response time R_i of τ_i. The terms $\rho_{j,i}(R_i)$ and $\gamma_{i,j}(R_i)$ denote the total cache persistence reload overhead due to the eviction of PCBs of τ_j by tasks in $\text{hep}(i) \setminus \tau_j$, and the total cache related preemption delay due to the preemptions caused by τ_j within the response time of τ_i, respectively. The terms $(PD_j(R_i) + MD_j(R_i))$ assume values obtained in isolation, while the two last terms $(\rho_{j,i}(R_i) + \gamma_{i,j}(R_i))$ account for the overheads introduced by the eviction of cache blocks by other tasks sharing the cache.

As already discussed in Section 3.2.4, $\gamma_{i,j}(R_i)$ is upper bounded by $\gamma_{i,j}^{mul}$. Furthermore, as proven in Lemmas 4.1 and 4.2, $MD_j(R_i)$ and $\rho_{j,i}(R_i)$ are upper bounded by Equations (4.4) and (4.6), respectively. Finally, because each task τ_j releases at most $\left\lceil \frac{t}{T_j} \right\rceil$ jobs in a time window of length t, $PD_j(R_i)$ is smaller than or equal to $\left\lceil \frac{R_i}{T_j} \right\rceil PD_j$.

Replacing each term with its given bound, we get that

$$R_i \leq C_i + \sum_{\forall j \in \text{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil PD_j + \sum_{\forall j \in \text{hp}(i)} \hat{MD}_j(R_i) + \sum_{\forall j \in \text{hp}(i)} \hat{\rho}_{j,i}(R_i) + \sum_{\forall j \in \text{hp}(i)} \gamma_{i,j}^{mul} \tag{4.8}$$

In systems where the number of PCBs is high and the cache interference is low, the value provided by $\hat{MD}_j(R_i) + \hat{\rho}_{j,i}(R_i)$ should always be smaller than $\left\lceil \frac{R_i}{T_j} \right\rceil MD_i$, and therefore we should often have $\left\lceil \frac{R_i}{T_j} \right\rceil PD_j + \hat{MD}_j(t) + \hat{\rho}_{j,i}(R_i)$ smaller than $\left\lceil \frac{R_i}{T_j} \right\rceil C_j$. In this case, Equation (4.8) will result in a tighter WCRT bound than Equation (3.14). However, in some situations, since $\hat{MD}_j(t)$ and $\hat{\rho}_{j,i}(R_i)$ are upper bounds and not exact values, this formulation can result in an over-estimation of the interference generated by τ_j on τ_i. In order to counter this effect, and knowing that Equation (3.14) is already an upper bound on the WCRT of τ_i, we further improve Equation (4.8) by always taking the minimum between $\left\lceil \frac{R_i}{T_j} \right\rceil C_j$ and $\left\lceil \frac{R_i}{T_j} \right\rceil PD_j + \hat{MD}_j(t) + \hat{\rho}_{j,i}(R_i)$ as the total interference caused by τ_j on τ_i (see Equation (4.9) below). Following this simple modification to Equation (4.8), Equation (4.9) will always return a value that is smaller than or equal to the solution to Equation (3.14). Our approach hence dominates the UCB union multi-set approach defined in (Altmeyer et al., 2012).

$$R_i^{un} = C_i + \sum_{\forall j \in \text{hp}(i)} \min\left\{ \left\lceil \frac{R_i^{un}}{T_j} \right\rceil C_j \, ; \left\lceil \frac{R_i^{un}}{T_j} \right\rceil PD_j + \hat{MD}_j(R_i^{un}) + \hat{\rho}_{j,i}(R_i^{un}) \right\} + \sum_{\forall j \in \text{hp}(i)} \gamma_{i,j}^{mul} \tag{4.9}$$

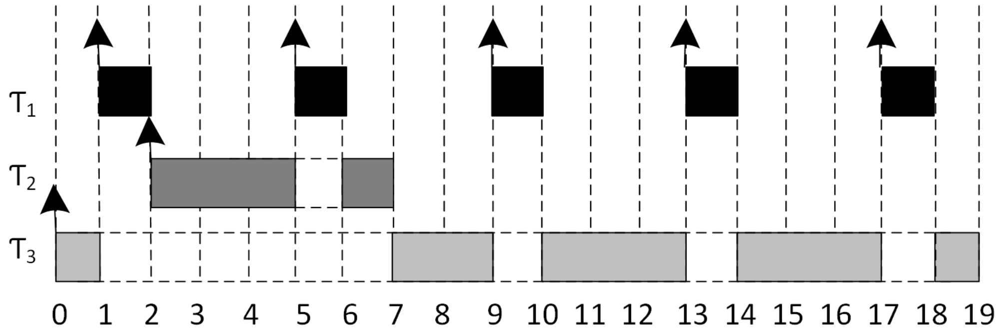


Figure 4.2: Illustration of the pessimism associated with Equation (4.6) using the task set $\{\tau_1, \tau_2\, \tau_3\}$ when τ_1 and τ_2 releasing their first jobs with an offset.

Note that Equation (4.9) is recursive. However, a solution can be found using simple fixed-point iteration on R_i^{un} initiating R_i^{un} to C_i. The iteration stops as soon as R_i^{un} does not evolve anymore or $R_i^{un} > D_i$, in which case the task is deemed unschedulable.

4.4 CPRO Multi-Set Approach

The formulation in Equations (4.5) and (4.6) considers that the ECBs of all tasks $\tau_k \in \text{hep}(i) \setminus \tau_j$ may interfere with every job of τ_j released within the response time of τ_i. This is pessimistic. Indeed, considering two different tasks τ_k and τ_l pertaining to $\text{hep}(i) \setminus \tau_j$, the number of times τ_l can execute between two successive jobs of τ_j is not necessarily equal to the number of times τ_k can execute between two successive jobs of τ_j. This situation is discussed in Example 4.2.

Example 4.2. *Let $\tau_1 = (1,4,4)$, $\tau_2 = (4,30,30)$ and $\tau_3 = (10,50,50)$, where τ_1 has the highest priority and τ_3 the lowest. Figure 4.2 presents a possible schedule that generates the worst-case response time of τ_3. As one can see, τ_1 releases 5 jobs during the response time of τ_3. As a result, Equation (4.9) upper bounds the total cache overheads on the PCBs of τ_1 with 4 times $\rho_{1,3}$. That is, it assumes that both τ_2 and τ_3 execute and reload all their ECBs between every two successive jobs of τ_1. As can be seen in Figure 4.2, this is pessimistic. In fact, τ_2 execute only twice between jobs of τ_1! Its impact on the total CPRO of τ_1 is therefore clearly overestimated.*

In order to reduce the pessimism associated with the computation of $\rho_{j,i}$, we must consider the actual number of times each task $\tau_k \in \text{hep}(i) \setminus \tau_j$ can execute between two successive jobs of τ_j. For this reason, this section presents a multi-set variant of Equation (4.6). The resulting quantity is an upper bound on $\hat{\rho}_{j,i}(t)$ denoted by $\rho_{j,i}^{mul}(t)$.

4.4.1 Computation of $\rho_{j,i}^{mul}(t)$

In this section, we first characterize the maximum number of times a task $\tau_k \in \text{hep}(i) \setminus \tau_j$ can execute between two successive jobs of τ_j. To do so, we separately analyze the tasks in $\text{hep}(j) \setminus \tau_j$ (Lemma 4.3) and $\text{aff}(i, j)$ (Lemma 4.4). We then use this information to upper bound the total cache persistence reload overhead $\rho_{j,i}(t)$ in Theorem 4.2.

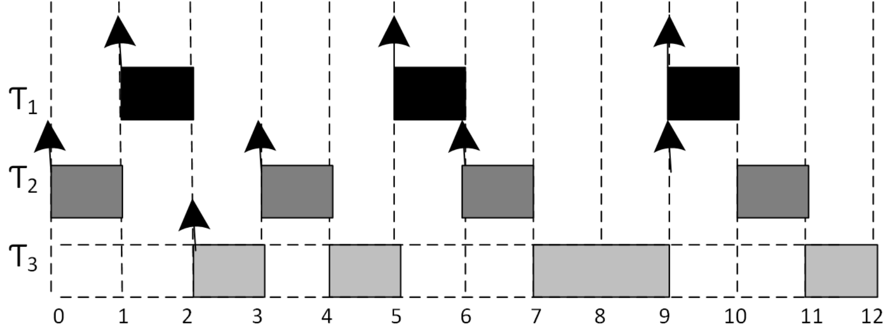


Figure 4.3: Illustration of the maximum number of times the tasks in $\text{aff}(i, j)$ and $\text{hep}(j) \setminus \tau_j$ can execute between two successive jobs of τ_j. When calculating $\rho_{2,3}$, $\tau_1 \in \text{hep}(2) \setminus \tau_2$ can release maximally 3 jobs (with each job loading all its ECBs in the worst case). In contrast, the one job released by $\tau_3 \in \text{aff}(3,2)$ can execute and load its ECBs maximum 4 times.

Lemma 4.3. *The maximum number of times a task $\tau_k \in \text{hep}(j) \setminus \tau_j$ can execute between two successive jobs of τ_j within the response time R_i of τ_i is upper bounded by $E_k(R_i)$.*

Proof. Remember that the maximum number of jobs that each higher priority task τ_k can release during the response time of a task τ_i is given by $E_k(R_i) \stackrel{\text{def}}{=} \left\lceil \frac{R_i}{T_k} \right\rceil$. Furthermore, because τ_k has a higher or equal priority than τ_j, τ_j cannot preempt τ_k. Hence, the maximum number of time τ_k can execute between two successive jobs of τ_j within a time window of length R_i is upper bounded by its number of released jobs $E_k(R_i)$ (see Figure 4.3 for an example). □

Lemma 4.4. *The maximum number of times a task $\tau_k \in \text{aff}(i, j)$ can execute between two successive jobs of τ_j within the response time R_i of τ_i is upper bounded by*

$$(E_j(R_k) + 1) \times E_k(R_i) \tag{4.10}$$

Proof. $E_j(R_k) \stackrel{\text{def}}{=} \left\lceil \frac{R_k}{T_j} \right\rceil$ provides the maximum number of jobs that τ_j can release during the response time of a task τ_k. Each of these released jobs may preempt the execution of τ_k. Considering an arrival pattern such that τ_k started to execute just before the first arrival of τ_j preempting τ_k (see Figure 4.3), the maximum number of times a job of τ_k may execute between two successive jobs of τ_j is then given by $(E_j(R_k) + 1)$. Since $E_k(R_i)$ jobs of τ_k are released within the response time of τ_i, the maximum number of times τ_k may execute between two successive jobs of τ_j within the response time of τ_i is upper bounded by $(E_j(R_k) + 1) \times E_k(R_i)$. □

Using Lemmas 4.3 and 4.4, one can derive an upper bound on $\rho_{j,i}(t)$. This upper bound is denoted by $\rho_{j,i}^{mul}(t)$ and is defined in the following theorem.

Theorem 4.2. *The total cache persistence reload overhead $\rho_{j,i}(R_i)$ on τ_j due to the eviction of its PCBs by tasks in $\text{hep}(i) \setminus \tau_j$ during the response time R_i of τ_i is upper bounded by*

$$\rho_{j,i}^{mul} \stackrel{\text{def}}{=} d_{mem} \times \left| M_{j,i}^{ecb} \cap M_{j,i}^{pcb} \right| \tag{4.11}$$

where $M^{ecb}_{j,i}$ and $M^{pcb}_{j,i}$ are multi-sets defined as

$$M^{pcb}_{j,i} = \bigcup_{E_j(R_i)-1} PCB_j \tag{4.12}$$

and

$$M^{ecb}_{j,i} = M^{ecb-aff}_{j,i} \cup M^{ecb-hp}_{j,i} \tag{4.13}$$

with

$$M^{ecb-aff}_{j,i} = \bigcup_{\forall k \in \mathrm{aff}(i,j)} \left(\bigcup_{(E_j(R_k)+1)E_k(R_i)} ECB_k \right) \tag{4.14}$$

and

$$M^{ecb-hp}_{j,i} = \bigcup_{\forall l \in \mathrm{hep}(j) \setminus \tau_j} \left(\bigcup_{E_l(R_i)} ECB_l \right) \tag{4.15}$$

Proof. The proof is based on the three following facts:
1. τ_j releases at most $\left\lceil \frac{t}{T_j} \right\rceil$ jobs in a time window of length t. At most $\left(\left\lceil \frac{t}{T_j} \right\rceil - 1 \right)$ evictions can therefore happen *between* two subsequent jobs of τ_j. The largest set of PCBs of τ_j that can be evicted between successive jobs of τ_j released during the response time of τ_i is therefore given by the multi-set $M^{pcb}_{j,i} = \bigcup_{E_j(R_i)-1} PCB_j$.
2. By Lemma 4.3, the maximum number of times a task $\tau_l \in \mathrm{hep}(j) \setminus \tau_j$ can execute between two successive jobs of τ_j during the response time of τ_i is upper bounded by $E_l(R_i)$. Hence, the largest set of ECBs that can be loaded by τ_l and interfere with the PCBs of τ_j is given by $\bigcup_{E_l(R_i)} ECB_l$ (assuming that τ_l reloads all its ECBs at each of its execution). This results in that the largest set of ECBs loaded by the tasks in $\mathrm{hep}(j) \setminus \tau_j$ between successive executions of τ_j is upper bounded by $M^{ecb-hp}_{j,i} = \bigcup_{\forall l \in \mathrm{hep}(j) \setminus \tau_j} \left(\bigcup_{E_l(R_i)} ECB_l \right)$.
3. By Lemma 4.4, the maximum number of times a task $\tau_k \in \mathrm{aff}(i,j)$ can execute between two successive jobs of τ_j during the response time of τ_i is upper bounded by $(E_j(R_k)+1) \times E_k(R_i)$. Hence, the largest set of ECBs that can be loaded by τ_k between successive jobs of τ_j during the response time of τ_i is given by $\bigcup_{(E_j(R_k)+1)E_k(R_i)} ECB_k$ (assuming that τ_k reloads all its ECBs whenever it resumes its execution). This results in that the largest set of ECBs loaded by the tasks in $\mathrm{aff}(i,j)$ between successive executions of τ_j is upper bounded by $M^{ecb-aff}_{j,i} = \bigcup_{\forall k \in \mathrm{aff}(i,j)} \left(\bigcup_{(E_j(R_k)+1)E_k(R_i)} ECB_k \right)$.

Therefore, by 2. and 3. the largest set of ECBs that can interfere with the PCBs of τ_j during the response time of τ_i is upper bounded by $M^{ecb}_{j,i} = M^{ecb-aff}_{j,i} \cup M^{ecb-hp}_{j,i}$.

Finally, the largest set of PCBs of τ_j that can be evicted by the tasks in $\mathrm{hep}(i) \setminus \tau_j$ within the response time of τ_i is upper bounded by the intersection of $M^{pcb}_{j,i}$ with $M^{ecb}_{j,i}$. Since reloading a cache block takes at most d_{mem} time units, the total cache persistence reload overhead $\rho_{j,i}(R_i)$ is upper bounded by $d_{mem} \times \left| M^{ecb}_{j,i} \cap M^{pcb}_{j,i} \right|$. □

4.4.2 Improving the Accuracy of $M^{ecb}_{j,i}$

Theorem 4.2 provides a good upper bound on the total cache persistence reload overhead $\rho_{j,i}(R_i)$ during the response time of τ_i. However, Equations (4.14) and (4.15) still consider that each job released by the tasks $\tau_k \in \text{hep}(i) \setminus \tau_j$ reload all their ECBs (i.e., PCBs and nPCBs) whenever they resume their execution. Even though this assumption may be valid for the tasks $\tau_l \in \text{hep}(j) \setminus \tau_j$, since each of their jobs contributes only once to $M^{ecb}_{j,i}$ (hence assuming that each job of τ_l accesses all its cache blocks during its execution), it is quite pessimistic for the tasks $\tau_k \in \text{aff}(i,j)$. Indeed, by Lemma 4.4 and Equation (4.14), each job of a task $\tau_k \in \text{aff}(i,j)$ is assumed to contribute $(E_j(R_k)+1)$ times to $M^{ecb}_{j,i}$. However, a PCB of task τ_k will be accessed at most once during each job execution unless this PCB is also a UCB (in which case it may be used at several program points of the task). The nPCBs must always be considered to be loaded several times during each job execution though. Indeed, since they are not persistent, it means that several memory blocks of τ_k are mapped to that same cache block, which can therefore be accessed more than once during each job execution.

It results from this discussion that $M^{ecb}_{j,i}$ can be more accurately modeled by the following equation:

$$M^{ecb}_{j,i} = M^{ecb-aff'}_{j,i} \cup M^{ecb-hp}_{j,i} \tag{4.16}$$

with

$$M^{ecb-aff'}_{j,i} = \bigcup_{\forall k \in \text{aff}(i,j)} \left[\left(\bigcup_{E_k(R_i)} (PCB_k \setminus UCB_k) \right) \cup \left(\bigcup_{(E_j(R_k)+1)E_k(R_i)} \Big(nPCB_k \cup (PCB_k \cap UCB_k) \Big) \right) \right] \tag{4.17}$$

where $(PCB_k \cap UCB_k)$ is the set of PCBs of τ_k that are also UCBs, and $(nPCB_k \cup (PCB_k \cap UCB_k))$ is therefore the set of ECBs that may be loaded more than once by each job of τ_k. All the other ECBs (those that are not in $(nPCB_k \cup (PCB_k \cap UCB_k))$ and are thus in $(PCB_k \setminus UCB_k)$ are loaded at most once per job of τ_k and are therefore accounted separately in the first term of Equation (4.17).

4.4.3 WCRT Analysis

Using the exact same argumentation as in Section 4.3.2, the worst-case response time of task τ_i can be upper bounded by the smallest positive value R^{mul}_i such that:

$$R^{mul}_i = C_i + \sum_{\forall j \in \text{hp}(i)} \min \left\{ \left\lceil \frac{R^{mul}_i}{T_j} \right\rceil C_j \; ; \left\lceil \frac{R^{mul}_i}{T_j} \right\rceil PD_j + \hat{MD}_j(R^{mul}_i) + \rho^{mul}_{j,i}(R^{mul}_i) \right\} + \sum_{\forall j \in \text{hp}(i)} \gamma^{mul}_{i,j} \tag{4.18}$$

It is important to note that, by construction, the WCRT formulation of Eq. (4.18) using the improved variant of the multi-set approach dominates the WCRT given by standard multi-set approach (Eq. (4.9)) which in turn dominates the simple union approach presented in Section 4.3.1.

4.5 Static Analysis

Having presented our proposed cache persistence-aware WCRT analysis, we proceed by explaining how the required input quantities, defined in Section 4.2.2, are obtained using standard static analysis techniques integrated in WCET estimation tools.

Static cache analysis techniques use abstract interpretation to determine the worst-case behavior with respect to caches for each memory reference. The outcome of such techniques is a classification of references given by Table 3.1 (e.g. *always-hit* when the reference will always result in a cache hit, *always-miss* when the reference will always result in a cache miss, *first-miss* when all successive occurrences of a reference but the first one will result in hits). The classification of each reference allows to determine if a reference will never require a memory access (*always-hit*) or may require an access to memory. To determine the relevant quantities required for the analysis presented in this chapter, the method presented in (Theiling et al., 2000) is used.

As we previously discussed in Section 3.2, most WCET estimation tools use IPET (*Implicit Path Enumeration Technique*) for WCET calculation. IPET is based on an Integer Linear Programming (ILP) formulation of the WCET calculation problem (Li and Malik, 1995). This formulation reflects the program structure and the possible execution flows using a set of linear constraints. The WCET estimate for a task is obtained by maximizing the following objective function:

$$\sum_{b \in BasicBlocks} E_b \times f_b \tag{4.19}$$

E_b (constant in the ILP problem) is the timing information of basic block b. f_b (variables in the ILP system, to be instantiated by the ILP solver) correspond to the number of times basic block b is executed.

For a task τ_i, quantities PD_i and MD_i are calculated using IPET by setting constant E_b accordingly for all basic blocks of τ_i. For the computation of PD_i, only the execution time of instructions is included in E_b, ignoring memory accesses. Conversely, when computing MD_i, only memory accesses (as detected by static cache analysis) are included in E_b and the execution times of instructions are ignored.

For the particular case of direct-mapped caches, determining PCB_i and ECB_i is straightforward. A memory block of task τ_i belongs to PCB_i if it is the only one mapped to a given cache block. ECB_i is simply the set of memory blocks of task τ_i. Determining UCB_i is achieved using the method presented in (Lee et al., 1998). Finally, determining MD_i^r is very similar to MD_i. IPET is applied with an execution cost of 0 and considering memory accesses, but in contrast to the computation of MD_i, only memory accesses for cache blocks in $nPCB_i$ are considered.

4.6 Experimental Evaluation

In this section, we evaluate the effectiveness of our proposed approaches in comparison to state-of-the-art techniques. We conducted three different experiments by varying the task utilizations,

number of tasks and the size of cache and evaluated their performance in terms of schedulability.

The different inputs previously defined in Section 4.5 were computed using the Heptane (Hardy et al., 2017) static WCET estimation tool. Heptane produces upper bounds on the execution times of hard real-time applications. It computes WCETs using static analysis at the binary code level. For the analysis presented in this chapter, all experiments were conducted on C-code compiled with gcc 4.1 with no optimization for MIPS R2000/R3000. The default linker memory layout is used, i.e. functions are represented sequentially in memory, and unless explicitly stated, no alignment directive is used. Without loss of generality, all instructions are assumed to execute in *1* cycle (cache access included). Each memory access, regardless of its source, results in a penalty of $d_{mem} = 100$ cycles. By default a direct-mapped instruction cache of size 2 KB with a line size of 32 B is considered.

We have integrated the results obtained from Heptane using static analysis with the MRTA framework developed by Altmeyer et al. (Altmeyer et al., 2015) for multi-core response time analysis. The MRTA tool provides a compositional framework for timing verification in multi-core systems by explicitly modeling the interferences of the different components. We modified the MRTA tool to consider task parameters that we have introduced in order to perform the analysis presented in this chapter. We have added a module in the MRTA framework that enables the calculation of the total CPRO $\rho_{j,i}(R_i)$ using the approaches detailed in Section 4.3 and 4.4. Also, as we only consider a single-core system, the preemption overhead calculation and the WCRT analysis are altered accordingly. All the experiments were performed using the Mälardalen benchmark suite (Gustafsson et al., 2010). Currently, we only consider the worst-case task layout where all benchmarks start at the same static memory address, thus maximizing the inter-task memory interference. The effect of different task layouts on our analysis will be discussed later in Chapter 6.

Evaluation is performed by randomly generating a large number of task sets and determining their schedulability using WCRT analysis for two cases:

1. WCRT analysis including only the effect of CRPD, i.e., Equation (3.14) where CRPD is computed using the UCB-Union multi-set approach (Equation (3.7)), and

2. Our proposed WCRT analysis that accounts for both CRPD and CPRO, i.e., CPRO Union (Equation (4.9)) and CPRO multi-set with/without the improvement in Equation (4.18).

Each task within the task set is randomly assigned parameters from the Mälardalen benchmarks. A subset of them is shown in Table 4.2. Also it should be clear from the numbers in Table 4.2 that the benchmark suite comprises of tasks with both small and big memory footprint (that fill the entire cache), consequently removing any bias in the results.

With the exception of parameters defined in Table 4.2, other parameters used in our experiments are defined as follows:

- The default number of tasks in each task set are 10 with task utilizations generated using UUnifast (Bini and Buttazzo, 2005).

Table 4.2: Task parameters for a selection of benchmarks from the Mälardalen Benchmark Suite (Gustafsson et al., 2010)

Name	C_i	PD_i	MD_i	$MD^r{}_i$	ECB_i	PCB_i	UCB_i	$nPCB_i$
lcdnum	3440	984	2740	192	20	20	20	0
insertsort	7574	5974	2343	752	16	16	10	0
bs	1399	203	1223	34	11	11	9	0
bsort100	712289	710289	90893	88907	20	20	15	0
ludcmp	45135	27036	21511	18442	98	30	43	68
fdct	17350	6550	11525	9327	106	22	58	84
ud	28427	20627	10415	10415	75	53	31	22
nsichneu	316409	22009	294400	294400	1377	0	110	1377
statemate	190496	10586	180110	180110	275	0	81	275

- Each task was randomly assigned one benchmark from the Mälardalen benchmark suite (Gustafsson et al., 2010) with values of C_i, PD_i, MD_i, MD_i^r along with sets of UCB_i, ECB_i, PCB_i and $nPCB_i$ obtained from the values given in Table 4.2.
- Task periods are set according the WCET assigned to each task from the benchmarks and the randomly generated utilization, i.e., $T_i = C_i/U_i$.
- Task deadlines are implicit with priorities assigned in deadline monotonic order.

4.6.1 Total Utilization

To evaluate how our proposed WCRT analysis accounting for both CPRO and CRPD (i.e., Eq (4.9) and (4.18)) performs in terms of schedulability in comparison to the UCB-union multi-set approach (Altmeyer et al., 2012) (that only considers CRPD), we randomly generated 1000 task set at different utilizations varied from 0.1 to 1 in steps of 0.025. Each task set contains 10 tasks, with benchmark parameters generated for a 2 KB cache with 64 cache sets. The WCRT analysis is performed for all approaches using the same task sets. A task set is deemed unschedulable if the calculated WCRT for any task within the task set is greater than its deadline.

Figure 4.4 shows an average number of task sets that were schedulable using all the analyzed approaches. The graph also shows a line marked as WCRT analysis with no CRPD cost (green line) that gives an upper bound on maximum number of task set that were schedulable without considering any CRPD cost. It is also important to note that we only show a cropped version of the plot starting from a utilization of 0.6 mainly because for task set utilizations less than 0.6 all approaches produced identical results. We can see from the results that the proposed WCRT analysis accounting for both CPRO and CRPD dominates the state-of-the-art WCRT analysis (UCB-union multi-set (Altmeyer et al., 2012)) that only accounts for CRPD. In fact, the three proposed approaches for CPRO calculation dominate the UCB-union multi-set approach (Altmeyer et al., 2012). This is mainly because the UCB-union multi-set approach only uses WCET (effectively

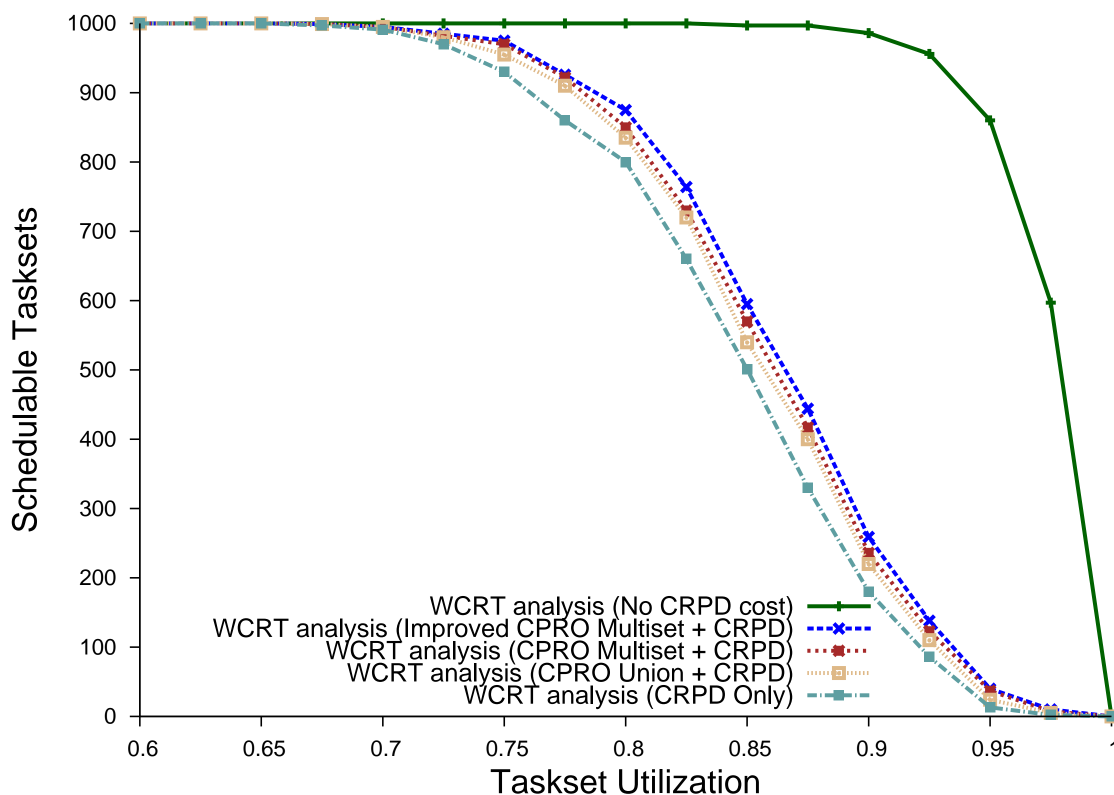


Figure 4.4: Number of tasksets that are deemed schedulable for a for a varying total utilizations.

the worst-case memory access demand) of tasks during the WCRT analysis along with the CRPD cost defined by Equation (3.7), which is very pessimistic. As a result, a high number of tasks tend to be unschedulable, especially at higher utilizations.

We can also observe from the results that the CPRO multi-set approach dominates the CPRO-union and UCB-union multi-set approaches whereas, the improved CPRO multi-set approach (Equation. (4.18)) outperforms all the other approaches. In fact, when using the improved CPRO multi-set approach we can have substantial gains in term of schedulability in comparison to the UCB-union multi-set approach, for example at a utilization of 0.85, we gain around 13% in schedulability.

4.6.2 Number of Tasks

In preemptive systems, the number of tasks adversely affects the schedulability of the task set. To analyze the performance of all approaches w.r.t number of tasks, we varied the number of tasks from 5 to 25 increasing by 5 tasks in each step. All parameters other than the number of tasks have the same values as used in the previous section. We have used the weighted schedulability measure defined by Bastoni et al. (Bastoni et al., 2010; Burns and Davis, 2014) to plot the results. The weighted schedulability measure reduces what would otherwise be a 3-dimensional plot to 2-dimensions by eliminating the axis of task set utilization. Under weighted schedulability, more

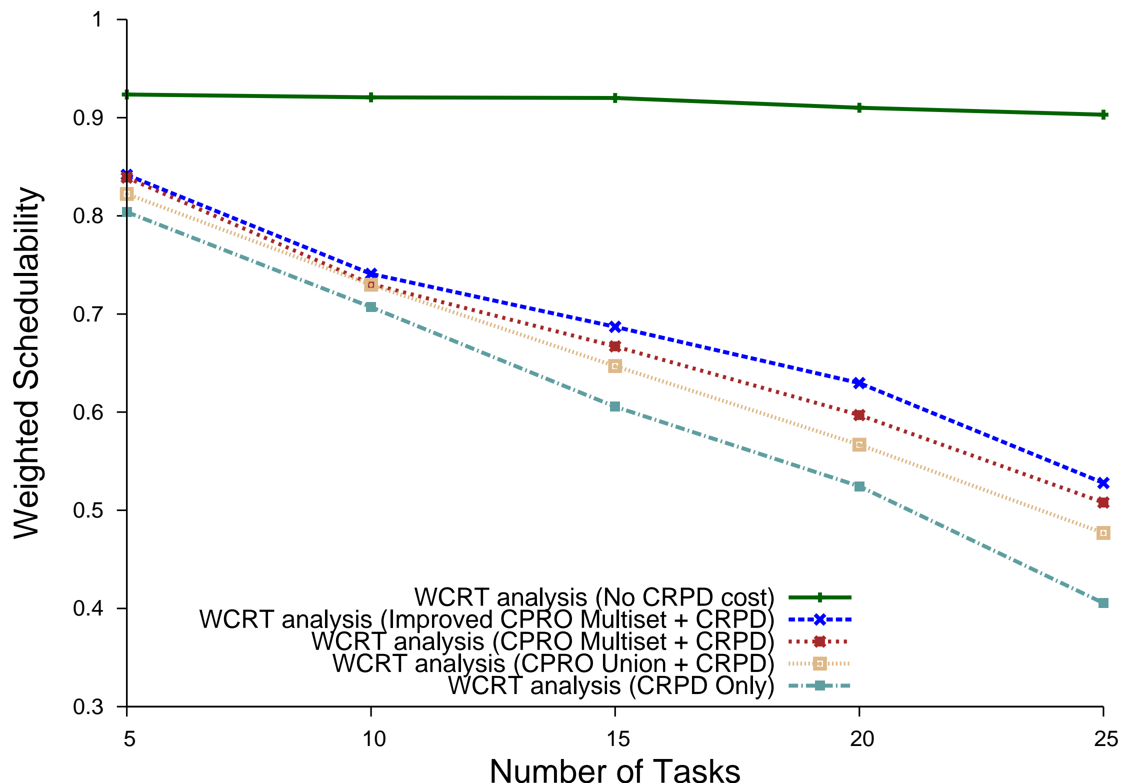


Figure 4.5: Weighted schedulability measure by varying the number of tasks from 5 to 25.

weight is given to task sets with higher utilization. Using notations from (Burns and Davis, 2014), let $S_y(\Gamma, p)$ represent the result of the schedulability test y for a given task set Γ with an input parameter p, i.e., $S_y(\Gamma, p) = 1$ if task set Γ is deemed schedulable for a given value of p and $S_y(\Gamma, p) = 0$ otherwise. Then, the weighted schedulability for that test y as a function of p, is given by $W_y(p)$, i.e.,

$$W_y(p) = \frac{\sum_{\forall \Gamma}(U(\Gamma) \times S_y(\Gamma, p))}{\sum_{\forall \Gamma} U(\Gamma)} \tag{4.20}$$

where $U(\Gamma)$ denote the core utilization of the task set Γ.

Figure 4.5 shows the results of our experiments. We can see that schedulability (varying from 0.3 to 1 by step of 0.025) for all approaches decreases as the number of tasks are increased. Indeed, this is due to an increasing number of cache evictions and reloads. On the other hand, we also observe that WCRT analysis accounting for both CPRO and CRPD performs significantly better in comparison to the other approach that only considers CRPD. The weighted schedulability using the improved CPRO multi-set approach at each point in Figure 4.5 is up to 10% higher than the UCB-union multi-set.

4.6.3 Cache Size

The cache size is an important factor that can affect the schedulability of tasks. If the cache is large enough to accommodate all the tasks without any cache reuse no additional memory accesses are

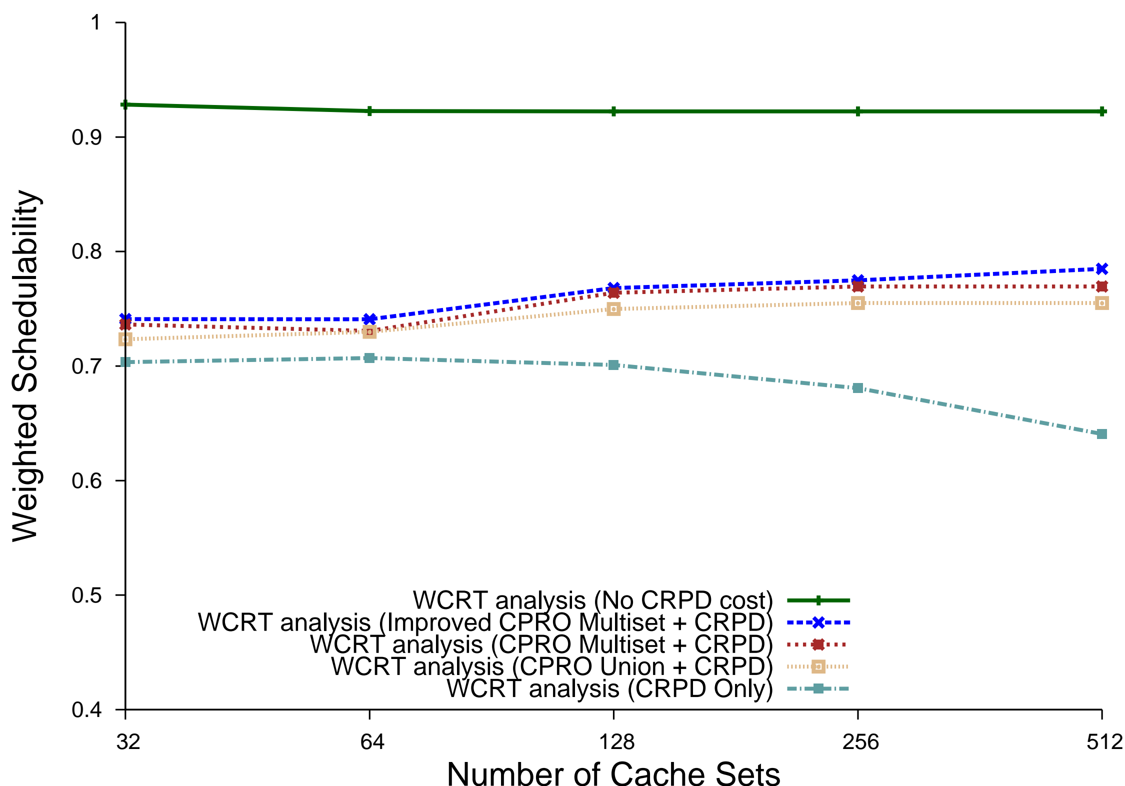


Figure 4.6: Weighted schedulability measure by varying the number of cache sets

required. In fact, in this case all the ECBs of a task will be PCBs and will never be evicted from the cache. Another case is when the cache is very small and each task can fill the entire cache during its execution. Consequently, this will result in higher memory access demand for each job of the task. To evaluate the impact of cache size on the performance of the analyses, we varied the number of cache sets from 32 to 512, keeping all other task parameters constant. Figure 4.6 shows the resulting weighted schedulability measure for each approach as a function of the number of cache sets. As the cache line size is kept constant (i.e. 32 B), increasing the number of cache sets effectively increase the cache size. Again, we can see that our proposed WCRT analysis accounting for both CPRO and CRPD dominates the UCB-union multi-set approach (Altmeyer et al., 2012) that only considers CRPD. In fact, by looking at the improved CPRO multi-set approach in Fig 4.6, we can observe that by increasing cache size the overall schedulability also increases from 0.74 (with 32 cache sets) to 0.80 (with 512 cache sets). This is due to the fact that with a bigger cache the number of PCBs for each task will also increase (hence reducing the residual memory access demand). In contrast, for the UCB-union multi-set approach (consistently with (Altmeyer et al., 2012)), the schedulability decreases due to an increase in the number of ECBs resulting in higher preemption overheads.

4.7 Chapter Summary

In this Chapter, we have presented our initial work that focus on improving the bounds on the inter-task cache interference for single-level direct-mapped caches. The proposed analysis builds upon the observation that a task can re-use cache contents between different jobs. We have presented a method to capture these persistent cache blocks (PCBs) resulting in variable memory access demand for different jobs from a task. The notion of cache persistence reload overhead (CPRO) is introduced and different approaches are presented to calculate CPRO. These approaches are orthogonal to state-of-the-art methods used for CRPD calculation and can be used independently with any of these methods. A WCRT analysis is then presented that exploits this variable memory access demand to reduce the preemption cost of higher priority tasks under fixed-priority preemptive scheduling, thereby reducing the WCRT and improving schedulability.

We evaluated the performance of our approach against a prominent approach from the state-of-the-art in terms of schedulability. Experiments were performed by varying different parameters with most of the values taken from the Mälardalen benchmarks. Experimental results show that our proposed WCRT analysis (accounting for both CPRO and CRPD) dominates the state-of-the-art approaches (that only consider CRPD) with an average improvement of around 10% in terms of schedulability.

Chapter 5

Integrated Analysis of Cache Related Preemption Delays and Cache Persistence Reload Overheads

In the work presented in Chapter 4, we derived two analyses for CPRO calculation that were integrated into an improved response time analysis for FPPS that accounts for the reduction in memory access demand of tasks due to cache persistence, along with the CRPD. The analysis considers both the CRPD and cache persistence and dominates the state-of-the-art approaches that only consider CRPD. However, the analysis presented in Chapter 4 may sometimes result in over-estimation of the task response times. This is due to the fact that CRPD and CPRO are calculated separately, providing independent upper bounds on the two classes of overheads. However, as we later show in this chapter, scenarios maximizing CRPD and those maximizing CPRO may be mutually exclusive, meaning that the total overheads can be substantially less than the sum of the two bounds.

In this chapter, we focus on two questions:

1. Is it beneficial to integrate the calculation of CRPD and CPRO to remove the over-estimation in the total overheads of tasks?
2. Under what conditions and by how much can we gain in terms of schedulability by integrating the calculation of CRPD and CPRO?

We answer these questions by:

- identifying situations where considering CRPD and CPRO separately might result in over-estimating the total memory overhead suffered by tasks due to double counting of some memory blocks that need to be reloaded,
- demonstrating how to integrate the calculation of CRPD and CPRO to include only the additional CPRO that are not already included in the CRPD calculation, and

- through experimental evaluation using a set of benchmarks to derive important observations that lead to situations where the integrated CRPD-CPRO analysis may or may not outperform separate treatment of CRPD and CPRO.

Table 5.1: List of important symbols used in Chapter 5

Symbol	Description
Γ	Task set of size n
τ_i	Task with index i
C_i	Worst-case execution time of task τ_i
T_i	Minimum inter-arrival time of task τ_i
D_i	Relative deadline of task τ_i
R_i	Worst-case response time of task τ_i
PD_i	Worst-case processing demand of task τ_i
MD_i	Worst-case memory access demand of task τ_i
MD_i^r	Residual memory access demand of task τ_i
$\hat{MD}_i(t)$	Total memory access demand of task τ_i in a time interval of length t
$\text{hp}(i)$	The set of tasks with higher priority than τ_i
$\text{hep}(i)$	The set of tasks with higher priority than τ_i including τ_i, i.e., $\text{hep}(i) = \text{hp}(i) \cup \tau_i$.
$\text{aff}(i, j)$	The set of intermediate tasks (including τ_i) that may preempt τ_i but may themselves be preempted by some higher priority task τ_j.
d_{mem}	Time to reload one cache block from the main memory
μ_i	Total memory reload overhead during the response time of task τ_i
μ_i^{sep}	Total memory reload overhead for task τ_i under the separate CRPD and CPRO analysis
μ_i^{int}	Total memory reload overhead for task τ_i under the integrated CRPD and CPRO analysis
Δ_i	Upper-bound on the portion of μ_i that is not accounted for in γ_i^{tot}
Δ_i^m	Upper-bound on the portion of μ_i that is not accounted for in γ_i^{tot-m}
$\overrightarrow{S}$	Any execution schedule of tasks
$E_k(R_i)$	The maximum number of jobs any task τ_k can release during the response time R_i of task τ_i
$N_{l,j}^{double}(R_i)$	The number of jobs of any higher priority task $\tau_l \in \text{hp}(j)$ that are already accounted for in the CRPD cost caused by another task $\tau_j \in \text{hp}(i)$ during the response time of τ_i.
ECB_i	The set of evicting cache blocks (ECBs) of task τ_i
UCB_i	The set of useful cache blocks (UCBs) of task τ_i
PCB_i	The set of persistence cache blocks (PCBs) of task τ_i
$nPCB_i$	The set of non-persistence cache blocks (nPCBs) of task τ_i

Continued on next page

Table 5.1 – continued from previous page

Symbol	Description
$b_{k,\ell}$	The ℓth cache block of a task τ_k
$\mathscr{S}_{j,i}$	The biggest set of cache blocks that can be loaded by tasks in $\text{hep}(i) \setminus \tau_j$ during the response of task τ_i and are not considered in γ_i^{tot}.
$M_{i,j}^{ucb}$	Multi-set containing set of UCBs of all tasks in $\text{aff}(i,j)$
$M_{i,j}^{ecb}$	Multi-set containing set of ECBs of task τ_j
$M_{j,i}^{pcb}$	Multi-set containing set of PCBs of task τ_j
$M_{j,i}^{ecb}$	Multi-set containing set of ECBs of all task in $\text{hep}(i) \setminus \tau_j$
$M_{j,i}^{ecb-aff}$	Multi-set containing set of ECBs of all task in $\text{aff}(i,j)$
$M_{j,i}^{hp-int}$	Multi-set containing set of cache blocks of all task in $\text{hep}(j) \setminus \tau_j$, whose evictions are not taken into account in the CRPD cost $\gamma_{i,j}^{ucb-m}$
$\rho_{j,i}$	CPRO of one job of task τ_j during the response time of task τ_i.
$\hat{\rho}_{j,i}$	Total CPRO of task τ_j in an interval of length t while executing during the response time of task τ_i.
$\delta_{j,i}$	Upper bound on the CPRO of one job of task τ_j during the response time of task τ_i, after discounting what has already been taken into account in the CRPD cost $\gamma_{i,j}^{ucb}$.
$\delta_{j,i}^{mul}$	Upper bound on the CPRO of task τ_j in a time interval t during the response time of task τ_i, after discounting what has already been taken into account in the CRPD cost $\gamma_{i,j}^{ucb-m}$.
$\gamma_{i,j}$	CRPD suffered by task τ_i due to preemptions by any higher priority task $\tau_j \in \text{hp}(i)$.
$\gamma_{i,j}^{ucb}$	CRPD suffered by task τ_i due to one preemption by any higher priority task $\tau_j \in \text{hp}(i)$, computed using the UCB-union approach (i.e., Equation (3.3)).
$\gamma_{i,j}^{ucb-m}$	Total CRPD suffered by task τ_i in a time interval of length t due to preemptions by any higher priority task $\tau_j \in \text{hp}(i)$, computed using the UCB-union multi-set approach (i.e., Equation (3.7)).
γ_i^{tot}	Upper bound on the total CRPD suffered by task τ_i during its response time under the UCB-union approach (i.e., Equation (3.3)).
γ_i^{tot-m}	Upper bound on the total CRPD suffered by task τ_i during its response time under the UCB-union multi-set approach (i.e., Equation (3.7)).

5.1 Problem Formalization

The CRPD of a task accounts for the evictions of its UCBs due to preemptions caused by higher priority tasks. Similarly, the CPRO accounts for the evictions of its PCBs between successive job executions. Therefore, the total time spent reloading cache blocks evicted during the response

time of τ_i is bounded by the sum of the CRPD and the CPRO experienced by every task executing during τ_i's response time. This overhead is denoted by μ_i and is defined as follows.

Definition 5.1 (Total Memory Reload Overhead (μ_i)). *Let $CRPD_{i,j}(\overrightarrow{S})$ and $CPRO_{i,j}(\overrightarrow{S})$ be the total* actual *CRPD and CPRO suffered by τ_j during the response time of one job of τ_i in a given schedule $\overrightarrow{S}$. The total memory reload overhead μ_i during the response time of τ_i is the maximum sum of the CRPD and CPRO of all tasks executing during τ_i's response time in any schedule $\overrightarrow{S}$. Formally,*

$$\mu_i \stackrel{\text{def}}{=} \max_{\forall \overrightarrow{S}} \left\{ \sum_{\forall \tau_j \in \text{hep}(i)} \left(CRPD_{i,j}(\overrightarrow{S}) + CPRO_{i,j}(\overrightarrow{S}) \right) \right\} \tag{5.1}$$

From the above definition, it follows that μ_i is upper-bounded by $\sum_{\tau_j \in \text{hep}(i)} (\gamma_{i,j}^{ucb-m} + \rho_{j,i}^{mul})$ where $\gamma_{i,j}^{ucb-m}$ and $\rho_{j,i}^{mul}$ are computed by Equation (3.7) and Equation (4.11), respectively. However, independently computing CRPD and CPRO may result in overestimating the actual total memory reload overhead μ_i as illustrated in the example below.

Example 5.1. *Let Γ be composed of three tasks $\{\tau_1, \tau_2, \tau_3\}$ with τ_1 having the highest priority and τ_3 the lowest. Figure 5.1 presents the task set parameters and the worst-case schedule for τ_3 together with the evolution of the cache contents over time. Cache blocks that have been evicted either due to CRPD or CPRO and must be reloaded from main memory are highlighted in red. The set of PCBs are highlighted in green.*

Initially, the cache is empty and with τ_3 being the first task to execute it loads all its ECBs into the cache. When τ_2 preempts τ_3 for the first time, it also loads its ECBs. Similarly, τ_2 is preempted by the highest priority task τ_1 at time 2. Note that ECBs of task τ_1 and UCBs/PCBs of task τ_2 are mapped to the same cache blocks, i.e., $\{7,8,9,10\}$. Therefore, when τ_2 resumes its execution after the completion of the first job of τ_1 it needs to reload all its UCBs, (highlighted in red) as they were evicted by τ_1. These additional memory accesses will be accounted for as CRPD.

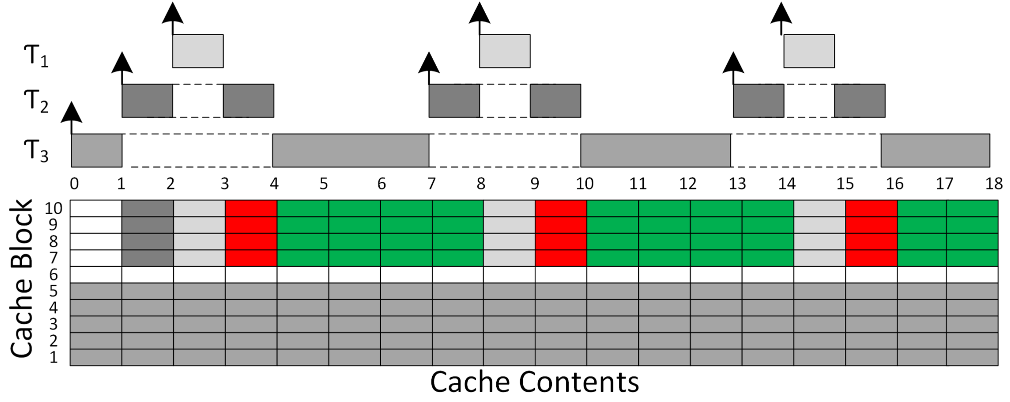
Since, the first job of τ_2 loads all of τ_2's ECBs (PCBs and nPCBs) into the cache, subsequent jobs of τ_2 may have a lower memory access demand due to the existence of PCBs in the cache, i.e., blocks $\{7,8,9,10\}$. However, some of these PCBs may be evicted due to other task executions. The additional memory accesses required to reload evicted cache blocks are accounted for as CPRO. Such a situation where the CPRO is maximized is depicted in Figure 5.1b.

Based on Figure 5.1a, the total memory reload overhead μ_3 during τ_3's response time is equal to the time needed to reload 12 cache blocks (i.e., the number of red blocks).

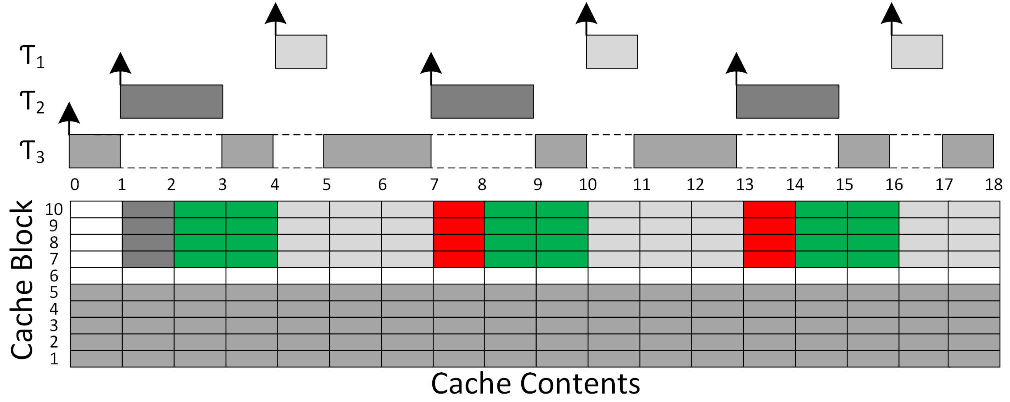
Now, if we use the UCB-union multi-set (Equation (3.7)) and the CPRO multi-set (Equation (4.11)) approaches to calculate μ_3, we have the following.

$$\mu_3 \leq \gamma_{3,1}^{ucb-m} + \gamma_{3,2}^{ucb-m} + \rho_{1,3}^{mul} + \rho_{2,3}^{mul}$$

Since τ_2 is the only task with useful cache blocks ($UCB_2 = \{7,8,9,10\}$), it is also the only task suffering from CRPD. Therefore, $\gamma_{3,2}^{ucb-m} = 0$. Using (Equation (3.7)), we have (note that



(a) Schedule maximizing CRPD during the response time of τ_3



(b) Schedule maximizing CPRO during the response time of τ_3

Figure 5.1: Schedules maximizing τ_3's response time when $C_1 = 1$, $C_2 = 2$, $C_3 = 9$, $T_1 = 6$, $T_2 = 6$, $T_3 = 25$, $ECB_1 = \{7,8,9,10\}$, $ECB_2 = \{7,8,9,10\}$, $ECB_3 = \{1,2,3,4,5\}$, $UCB_2 = \{7,8,9,10\}$, $PCB_2 = \{7,8,9,10\}$ and $UCB_1 = UCB_3 = PCB_1 = PCB_3 = \emptyset$

$E_1(R_3) = 3$, $E_1(R_2) = 1$, $E_2(R_3) = 3$, and $E_3(R_3) = 1$):

$$\gamma_{3,1}^{ucb-m} = d_{mem} \times |(3 \times UCB_3 \cup 3 \times UCB_2) \cap (3 \times ECB_1)| = d_{mem} \times 12$$

Similarly, when calculating the CPRO we can see that the set of PCBs for all tasks except τ_2 is empty. Hence, the total CPRO during the response time of task τ_3 comes only from the evictions of PCBs of task τ_2. Assuming that the CPRO is calculated using Equation (4.11) we have $\rho_{1,3}^{mul} = 0$ and

$$\rho_{2,3}^{mul} = d_{mem} \times |(2 \times PCB_2) \cap (4 \times ECB_3 \cup 3 \times ECB_1)| = d_{mem} \times 8$$

Adding CRPD and CPRO, it follows that the total memory reload overhead during the response time of τ_3 is upper-bounded by $d_{mem} \times 20$. Thus it appears that 20 cache blocks need to be reloaded during the response time of τ_3. The reason for the overestimation is that the total CRPD is indeed upper-bounded by 12 cache blocks reloads (as shown in Figure 5.1a) and the total CPRO is indeed upper-bounded by 8 cache blocks reloads (as shown on Figure 5.1b), but both scheduling

scenarios cannot happen at the same time. It is not possible for the three jobs of τ_1 *to result in the group of 4 cache block reloads three times over due to preemptions (accounted for in* $\gamma_{3,1}^{ucb-m}$*)* and *two times over due to cache persistence overheads (accounted for in* $\rho_{2,3}^{mul}$*). This observation leads to the following lemma.*

Lemma 5.1. *Let us assume that the total CRPD during the response time of task* τ_i *is computed using Equation* (3.3) *or Equation* (3.7) *and that the total CPRO during* τ_i*'s response time is computed with Equation* (4.6) *or Equation* (4.11)*. Let* $b_{k,\ell}$ *be the* ℓ^{th} *cache block of a task* $\tau_k \in \text{hp}(i)$*, i.e.,* $b_{k,\ell} \in ECB_k$*. The eviction of* $b_{k,\ell}$ *will be accounted for in both the CRPD and CPRO, only if* $b_{k,\ell}$ *is a UCB and a PCB of* τ_k*, i.e.,* $b_{k,\ell} \in UCB_k \cap PCB_k$*.*

Proof. This claim follows directly from the fact that Equation (3.3) and Equation (3.7) account for the evictions of UCBs of tasks in $\text{hep}(i)$. Therefore, the eviction of cache block $b_{k,\ell}$ will be considered in the CRPD calculation only if it is a UCB. Similarly, Equation (4.6) and Equation (4.11) account for the evictions of PCBs of tasks in $\text{hp}(i)$. Hence, the eviction of cache block $b_{k,\ell}$ will be considered in the CPRO calculation only if it is a PCB. Therefore, the eviction of $b_{k,\ell}$ may be accounted for in both the CRPD and CPRO, only if $b_{k,\ell} \in UCB_k \cap PCB_k$. □

It can also be seen in Example 5.1 that for any task $\tau_k \in \text{hp}(i)$ (e.g., τ_2) executing during the response time of a lower priority task τ_i (e.g., τ_3), only higher priority tasks than τ_k (e.g., τ_1) can participate in both the CRPD and CPRO of τ_k. This observation leads to the following lemma.

Lemma 5.2. *For any task* $\tau_k \in \text{hp}(i)$ *executing during the response time of a lower priority task* τ_i*, only the tasks in* $\text{hp}(k)$ *can contribute to both the CRPD and CPRO of* τ_k*.*

Proof. By Definition 4.4, all tasks in $\text{hep}(i) \setminus \tau_k$ can contribute to the CPRO of τ_k during the response time of τ_i.

Let τ_ℓ be any task in $\text{hep}(i) \setminus \tau_k$. Two cases must be considered:

1. If $\tau_\ell \in \text{aff}(i,k)$ then τ_ℓ has a lower priority than that τ_k. Therefore, τ_ℓ can never preempt τ_k and hence cannot contribute to τ_k's CRPD.

2. If $\tau_\ell \in \text{hp}(k)$ then τ_ℓ has a higher priority than that of τ_k. Task τ_ℓ can therefore preempt τ_k and cause CRPD.

Hence, only tasks in $\text{hp}(k)$ can contribute to both τ_k's CRPD and CPRO. □

5.2 Integrated CRPD-CPRO Analysis

In the analysis presented in Chapter 4, CRPD and CPRO are calculated independently of each other. As discussed in Section 5.1, this can lead to an overestimation of the total memory reload overhead. In this section, we present a novel approach to bound the total memory reload overhead during the response time of a task τ_i. This section builds upon the UCB-union and CPRO-union approaches for the calculation of CRPD and CPRO, respectively. In Section 5.3, we extend this

analysis to consider the more precise, but also more complex, multi-set variants of the CPRD and CPRO calculation.

It follows from Lemma 5.1 that only the cache blocks in $\bigcup_{\forall \tau_j \in \mathrm{hp}(i)} (UCB_j \cap PCB_j)$ can have their evictions counted twice during the CRPD and CPRO calculations. This double counting can be removed either (i) during the CRPD calculation by not considering the evictions of PCBs in $\bigcup_{\forall \tau_j \in \mathrm{hp}(i)} (UCB_j \cap PCB_j)$, since their eviction will be accounted for in the CPRO; or, (ii) during the CPRO calculation by not considering the eviction of UCBs in $\bigcup_{\forall \tau_j \in \mathrm{hp}(i)} (UCB_j \cap PCB_j)$, since their eviction will be considered in the CRPD. In this section, we focus on the latter solution assuming that the CRPD is computed using the UCB-union approach (i.e., using Equation (3.3)).

Lemma 5.3. *Let γ_i^{tot} be an upper-bound on the total CRPD during the response time R_i of τ_i. Further assume that γ_i^{tot} is computed using the UCB-union approach, i.e., $\gamma_i^{tot} \stackrel{\text{def}}{=} \sum_{\tau_j \in \mathrm{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \gamma_{i,j}^{ucb}$. Let Δ_i be an upper-bound on the portion of the total memory reload overhead during τ_i's response time that is not accounted for in γ_i^{tot}, that is, $\Delta_i = \mu_i - \gamma_i^{tot}$, then we have $\Delta_i \leq \sum_{\forall \tau_j \in \mathrm{hp}(i)} \left(\left\lceil \frac{R_i}{T_j} \right\rceil - 1 \right) \times \delta_{j,i}$ where*

$$\delta_{j,i} \stackrel{\text{def}}{=} d_{mem} \times \left| PCB_j \bigcap \left(\left(\bigcup_{\forall \tau_k \in \mathrm{aff}(i,j)} ECB_k \right) \bigcup \left(\bigcup_{\forall \tau_k \in \mathrm{hp}(j)} ECB_k \setminus (UCB_j \cap PCB_j) \right) \right) \right| \qquad (5.2)$$

Proof. It was proven in (Tan and Mooney, 2007) that γ_i^{tot} upper-bounds the total CRPD during τ_i's response time. Therefore, the portion of the total memory reload overhead μ_i that is not accounted for in γ_i^{tot} is a subset of the total CPRO during τ_i's response time. Similar to the calculation of the total CPRO, at most $\left(\left\lceil \frac{R_i}{T_j} \right\rceil - 1 \right)$ jobs of each higher priority task τ_j can suffer memory reload overhead $\delta_{j,i}$ not yet accounted for in γ_i^{tot}. Since the total CPRO is an upper-bound on Δ_i, using Equation (4.6) and Equation (4.5) we have $\Delta_i \leq \sum_{\tau_j \in \mathrm{hp}(i)} \left(\left\lceil \frac{R_i}{T_j} \right\rceil - 1 \right) \times \delta_{j,i}$ with $\delta_{j,i} \leq \rho_{j,i}$. We now prove the validity of $\delta_{j,i}$.

Since a fixed-priority scheduling algorithm is used, only tasks with priorities higher than or equal to the priority of τ_i (i.e., tasks in $\mathrm{hep}(i)$) can execute during the response time of τ_i. Therefore, any task $\tau_k \in \mathrm{hep}(i) \setminus \tau_j$ can execute between two subsequent jobs of another task τ_j and hence participate in τ_j's CPRO by evicting some or all its PCBs. Let τ_k be any task in $\mathrm{hep}(i) \setminus \tau_j$. Two cases need to be considered (note that $\mathrm{hep}(i) \setminus \tau_j = \mathrm{aff}(i,j) \cup \mathrm{hp}(j)$).

1. $\tau_k \in \mathrm{aff}(i,j)$. Since τ_k has a lower priority than τ_j it cannot preempt τ_j, and hence τ_k does not contribute to the CRPD of τ_j. Therefore, the memory reload overhead generated by τ_k on τ_j is not part of γ_i^{tot} and must be entirely accounted for in $\delta_{i,j}$. This worst-case interference of τ_k on τ_j is maximized when τ_k loads all its cache blocks (i.e., ECB_k).

2. If $\tau_k \in \mathrm{hp}(j)$ then, by Lemma 5.2, τ_k may contribute to both the CRPD and CPRO of τ_j. As stated in Lemma 5.1, the evictions of cache blocks of τ_j in $UCB_j \cap PCB_j$ were already considered in γ_i^{tot}. Therefore, the number of cache block evictions caused by τ_k on τ_j

that were not accounted for in γ_i^{tot} is maximized when τ_k loads all the cache blocks in $ECB_k \setminus (UCB_j \cap PCB_j)$.

From 1. and 2., the biggest set $\mathscr{S}_{j,i}$ of cache blocks that can be loaded by tasks in $\text{hep}(i) \setminus \tau_j$ and were not yet considered in γ_i^{tot} is given by:

$$\mathscr{S}_{j,i} = \left(\bigcup_{\forall \tau_k \in \text{aff}(i,j)} ECB_k \right) \bigcup \left(\bigcup_{\forall \tau_l \in \text{hp}(j)} ECB_l \setminus (UCB_j \cap PCB_j) \right)$$

The set of PCBs that must be reloaded by τ_j at each job execution is thus upper-bounded by the intersection between τ_j's PCBs (i.e., PCB_j) and the set $\mathscr{S}_{j,i}$ derived above. Since each cache block reload takes at most d_{mem} time units, the time $\delta_{j,i}$ spent by τ_j at each job execution to reload evicted PCBs that were not yet considered in γ_i^{tot} is bounded by Equation (5.2). □

As a corollary of Lemma 5.3, we can upper-bound the total memory reload overhead μ_i as stated in the following theorem:

Theorem 5.1. *The total memory reload overhead μ_i during τ_i's response time is upper-bounded by*

$$\sum_{\tau_j \in \text{hp}(i)} \left(\left(\left\lceil \frac{R_i}{T_j} \right\rceil \times \gamma_{i,j}^{ucb} \right) + \left(\left\lceil \frac{R_i}{T_j} \right\rceil - 1 \right) \times \delta_{j,i} \right) \tag{5.3}$$

Proof. Follows from Lemma 5.3 since $\mu_i = \Delta_i + \gamma_i^{tot}$. □

This directly leads to the following theorem:

Theorem 5.2. *The WCRT of τ_i is upper-bounded by the smallest positive solution to*

$$R_i = C_i + \sum_{\forall j \in \text{hp}(i)} \left(\gamma_{i,j} + \min \left\{ \left\lceil \frac{R_i}{T_j} \right\rceil C_j \, ; \left\lceil \frac{R_i}{T_j} \right\rceil PD_j + \hat{MD}_j(R_i) + \hat{\delta}_{j,i} \right\} \right) \tag{5.4}$$

where

$$\hat{\delta}_{j,i} \stackrel{\text{def}}{=} \left(\left\lceil \frac{R_i}{T_j} \right\rceil - 1 \right) \delta_{j,i} \tag{5.5}$$

and $\gamma_{i,j}$ is given by $\left\lceil \frac{R_i}{T_j} \right\rceil \gamma_{i,j}^{ucb}$ for UCB-Union.

Proof. By Theorem 5.1 and substituting $\hat{\delta}_{j,i}$ for $\hat{\rho}_{j,i}$ in Equation (4.9) □

Since, $\delta_{j,i}$ calculated using Equation (5.2) is always less than or equal to $\rho_{j,i}$ calculated using Equation (4.5), the resulting WCRT obtained using Equation (5.4) is always less than or equal to the WCRT obtained using Equation (4.9) when $\gamma_{i,j}$ is computed using the UCB-union approach. In other words, the integrated approach to CRPD and CPRO analysis given by Theorem 5.2 *dominates* the simple combination of the UCB-union and CPRO-union approaches.

Example 5.2. *We now compute the total memory reload overhead of task τ_3 in Example 5.1 using the results derived in Theorem 5.1.*

Note that the UCB-union (Equation (3.3)) and the UCB-union multi-set (Equation (3.7)) approaches would give exactly the same values for the total CRPD. Therefore, the total CRPD is upper-bounded by $d_{mem} \times 12$.

The set of PCBs for all tasks except τ_2 is empty. Therefore, based on Equation (5.2), we have $\delta_{1,3} = 0$ and

$$\begin{aligned}\delta_{2,3} &= d_{mem} \times |PCB_2 \cap (ECB_3 \cup (ECB_1 \setminus (UCB_2 \cap PCB_2)))| \\ &= d_{mem} \times |\{7,8,9,10\} \cap (\{7,8,9,10\} \setminus \{7,8,9,10\})| = 0\end{aligned}$$

According to Theorem 5.1, μ_3 is thus upper-bounded by $(12 \times d_{mem})$, which is in this case the exact overhead experienced during the response time of τ_3 as illustrated in Figure 5.1a.

5.3 Multi-set Approach to Integrated CRPD-CPRO analysis

In this section, we improve over the analysis presented in Section 5.2 by building upon the UCB-union multi-set (Equation (3.7)) and CPRO-union multi-set (Equation (4.11)) analyses that were shown to dominate the UCB-union and CPRO-union approaches.

While the UCB-union approach assumes that every job of a task $\tau_k \in \text{hp}(i)$ executing during the response time of τ_i can contribute to the total CRPD, the UCB-union multi-set approach (Equation (3.7)) considers that only a subset of τ_k's jobs actually contribute to the preemption overhead. Hence, we must also differentiate between jobs that are considered in the CRPD and those that are not, when computing the portion of the total memory reload overhead μ_i that is not yet accounted for in the total CRPD.

Example 5.3. *The example task set in Figure 5.2 has three tasks τ_1, τ_2 and τ_3 with priorities assigned in numerical order such that τ_1 has the highest priority. We want to analyze the total memory reload overhead μ_3 during the response time of τ_3. Task τ_2 is the only task with $UCB_2 \cap PCB_2 \neq \emptyset$. The sets of UCBs and PCBs of τ_1 and τ_3 are empty. Therefore, τ_2 is the only task that may suffer CRPD and CPRO. The total memory reload overhead μ_3 is thus bounded by the sum of the CRPD and CPRO suffered by τ_2 during the response time of τ_3.*

By Lemma 5.2, τ_1 is the only task that can contribute to both τ_2's CRPD and CPRO. Since τ_1 can preempt each job of τ_2 at most once (i.e., $E_1(R_2) = 1$), and because τ_2 releases three jobs during τ_3's response time (i.e., $E_2(R_3) = 3$), at most three jobs of τ_1 are preempting jobs of τ_2 during the response time of τ_3, i.e., $E_1(R_2)E_2(R_3) = 3$. Therefore, at most three jobs of τ_1 may be contributing to both τ_2's CRPD and CPRO during τ_3's response time. The one remaining job of τ_1 can only execute between two jobs of τ_2, and hence contributes only to τ_2's CPRO.

To calculate the CPRO that any task $\tau_j \in \text{hp}(i)$ can suffer during the response time of τ_i, taking into consideration what has already been accounted for in the CRPD cost, we first analyze

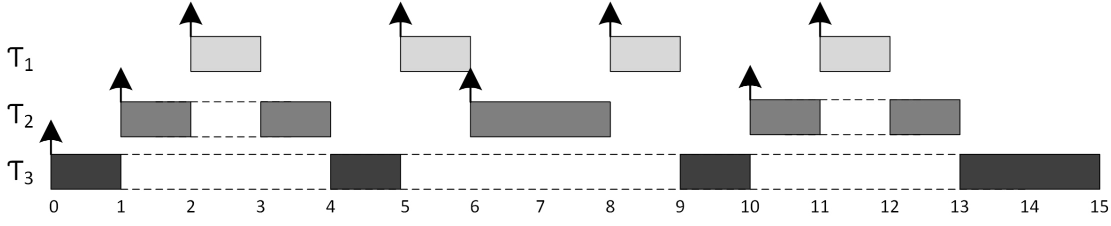


Figure 5.2: Illustrating the pessimism associated with the separate UCB-union multi-set and CPRO multi-set analysis using the task set $\{\tau_1, \tau_2, \tau_3\}$ with $C_1 = 1$, $C_2 = 2$, $C_3 = 5$, $T_1 = 3$, $T_2 = 5$ and $T_3 = 20$.

the impact of each task in $\text{hep}(i) \setminus \tau_j$ on the CPRO of τ_j. We characterize the maximum number of times a task $\tau_k \in \text{hep}(i) \setminus \tau_j$ can execute between successive jobs of τ_j. To do so, we separately analyze the tasks in $\text{aff}(i, j)$ (Lemma 4.4) and the tasks in $\text{hp}(j)$ (Lemma 4.3). We then identify how many jobs of each task contribute only to the CPRO of τ_j and how many jobs contribute to both the CRPD and the CPRO of τ_j (Lemma 5.6). We then make use of this information to derive a multi-set formulation (Lemma 5.7) that calculates the additional CPRO of a task $\tau_j \in \text{hp}(i)$ that is not already accounted for in the CRPD cost computed with Equation (3.7).

Lemma 5.4. *The maximum number of times a task $\tau_k \in \text{aff}(i, j)$ can execute between jobs of τ_j released during τ_i's response time is upper-bounded by $(E_j(R_k) + 1) \times E_k(R_i)$.*

Proof. Lemma 4.4 in Chapter 4. □

Lemma 5.5. *The maximum number of times a task $\tau_k \in \text{hp}(j)$ can execute between successive jobs of τ_j released during τ_i's response time is upper bounded by $E_k(R_i)$.*

Proof. Lemma 4.3 in Chapter 4. □

Example 5.3 shows that not all of the jobs released by a higher priority task $\tau_l \in \text{hp}(j)$ (e.g., τ_1 in Figure 5.2) during the response time of a lower priority task τ_i (e.g., τ_3 in Figure 5.2) can preempt τ_j (e.g., τ_2 in Figure 5.2). The jobs that do not preempt cannot contribute to both the CRPD and the CPRO of τ_j. This observation leads to the following Lemma:

Lemma 5.6. *For a task $\tau_j \in \text{hp}(i)$ executing during the response time of τ_i, the number of jobs of any higher priority task $\tau_l \in \text{hp}(j)$ that are already accounted for in the CRPD $\gamma_{i,j}^{ucb-m}$ is given by $N_{l,j}^{double}(R_i) = \min\{E_l(R_i)\ ;\ E_l(R_j)E_j(R_i)\}$.*

Proof. The CRPD $\gamma_{i,j}^{ucb-m}$ in Equation (3.7) is composed of the intersection of the two multi-sets $M_{i,j}^{ucb}$ and $M_{i,j}^{ecb}$.

1. The calculation of $M_{i,j}^{ecb}$ (i.e., Equation (3.9)) assumes that a task $\tau_l \in hp(j)$ can release at most $E_l(R_i)$ jobs during the response time R_i of τ_i. Therefore, at most $E_l(R_i)$ jobs of τ_l preempting τ_j are accounted for in the calculation of $\gamma_{i,j}^{ucb-m}$ in Equation (3.7).

2. The calculation of $M^{ucb}_{i,j}$ (Equation (3.8)) assumes that for any task $\tau_j \in aff(i,j)$, $E_l(R_j)E_j(R_i)$ is an upper bound on the number of times τ_j can be preempted by τ_l during τ_i's response time. Therefore, at most $E_l(R_j)E_j(R_i)$ jobs of τ_l are accounted for in $\gamma^{ucb-m}_{i,j}$ (i.e., Equation (3.7)).

It follows that the number of jobs of τ_l accounted for in $\gamma^{ucb-m}_{i,j}$ is given by $N^{double}_{l,j}(R_i)$. □

Using Lemmas 4.3-4.4, 5.2 and 5.6 we derive an upper bound on the CPRO any task $\tau_j \in hp(i)$ can suffer during τ_i's response time, discounting what has already been taken into account in the CRPD cost $\gamma^{ucb-m}_{i,j}$. This upper bound is denoted by $\delta^{mul}_{j,i}$.

Lemma 5.7. *Let γ^{tot-m}_i be an upper-bound on the total CRPD during the response time R_i of τ_i. Further assume that γ^{tot-m}_i is computed using the UCB-union multi-set approach, i.e., $\gamma^{tot-m}_i = \sum_{\tau_j \in hp(i)} \gamma^{ucb-m}_{i,j}$. Let Δ^m_i be an upper-bound on the portion of the total memory reload overhead that was not accounted for in γ^{tot-m}_i, that is, $\Delta^m_i = \mu_i - \gamma^{tot-m}_i$, then:*

$$\Delta^m_i \leq \sum_{\tau_j \in hp(i)} \delta^{mul}_{j,i} \tag{5.6}$$

where

$$\delta^{mul}_{j,i} \stackrel{\text{def}}{=} d_{mem} \times \left| M^{ecb}_{j,i} \cap M^{pcb}_{j,i} \right| \tag{5.7}$$

where $M^{ecb}_{j,i}$ and $M^{pcb}_{j,i}$ are multi-sets defined as

$$M^{pcb}_{j,i} = \bigcup_{E_j(R_i)-1} PCB_j \tag{5.8}$$

$$M^{ecb}_{j,i} = M^{ecb-aff}_{j,i} \cup M^{hp-int}_{j,i} \tag{5.9}$$

with

$$M^{ecb-aff}_{j,i} = \bigcup_{\forall k \in \text{aff}(i,j)} \left(\bigcup_{(E_j(R_k)+1)E_k(R_i)} ECB_k \right) \tag{5.10}$$

$$M^{hp-int}_{j,i} = \bigcup_{\forall l \in \text{hp}(j)} \left(\left(\bigcup_{E_l(R_i)-N^{double}_{l,j}(R_i)} ECB_l \right) \cup \left(\bigcup_{N^{double}_{l,j}(R_i)} ECB_l \setminus \left(UCB_j \cap PCB_j \right) \right) \right) \tag{5.11}$$

Proof. Since γ^{tot-m}_i upper-bounds the total CRPD during τ_i's response time calculated using Equation (3.7), the portion of μ_i that is not accounted for in γ^{tot-m}_i is a subset of the total CPRO during τ_i's response time that is,

$$\Delta^m_i \leq \sum_{\tau_j \in \text{hp}(i)} \delta^{mul}_{j,i}$$

where $\delta^{mul}_{j,i} \leq \rho^{mul}_{j,i}$.

We prove the validity of $\delta^{mul}_{j,i}$ below.

1. Since τ_j can release at most $\left\lceil \frac{t}{T_j} \right\rceil$ jobs in a time window of length t, the PCBs of τ_j can be evicted at most $\left(\left\lceil \frac{t}{T_j} \right\rceil - 1\right)$ times within the time window of length t, contributing to CPRO[1]. Therefore, the largest set of PCBs of τ_j that can be evicted during the response time of τ_i is upper bounded by the multi-set $M^{pcb}_{j,i} = \bigcup_{E_j(R_i)-1} PCB_j$ given in Equation (5.8).

2. By Lemma 4.4, the maximum number of times a task $\tau_k \in \text{aff}(i,j)$ can execute between two successive jobs of τ_j during the response time of τ_i is upper bounded by $(E_j(R_k)+1) \times E_k(R_i)$. Hence, the largest set of ECBs that can be loaded by τ_k between successive jobs of τ_j during the response time of τ_i is given by $\bigcup_{(E_j(R_k)+1)E_k(R_i)} ECB_k$. Therefore the largest set of ECBs loaded by the tasks in $\text{aff}(i,j)$ between successive executions of τ_j is upper bounded by $M^{ecb-aff}_{j,i} = \bigcup_{\forall k \in \text{aff}(i,j)} \left(\bigcup_{(E_j(R_k)+1)E_k(R_i)} ECB_k \right)$ given in Equation (5.10).

3. By Lemma 4.3, the maximum number of times a task $\tau_l \in \text{hp}(j)$ can execute between two successive jobs of τ_j during the response time of τ_i is upper bounded by $E_l(R_i)$. Hence, the largest set of ECBs that can be loaded by τ_l and interfere with the PCBs of τ_j is given by $\bigcup_{E_l(R_i)} ECB_l$. However, by Lemma 5.2, as $\tau_l \in \text{hp}(j)$ it can contribute to both the CRPD and CPRO of τ_j during the response time of τ_i. Further, by Lemma 5.6, the number of jobs of τ_l that were already considered in the CRPD of τ_j is equal to $N^{double}_{l,j}(R_i)$. Therefore, instead of assuming that all jobs released by $\tau_l \in \text{hp}(j)$ during the response time of τ_i contribute to $\delta^{mul}_{j,i}$, the multi-set $M^{hp-int}_{j,i}$ separately categorizes the impact of jobs of τ_l that can/cannot be contributing to both the CRPD and CPRO of τ_j during the response time of τ_i.

3.1 Since $N^{double}_{l,j}(R_i)$ is the number of jobs of τ_l that were already considered in the CRPD of τ_j, then $E_l(R_i) - N^{double}_{l,j}(R_i)$ jobs of τ_l only contribute to the CPRO of τ_j. The memory reload overhead generated by these $E_l(R_i) - N^{double}_{l,j}(R_i)$ jobs of τ_l on τ_j is not part of γ^{tot-m}_i and must therefore be entirely accounted for in $\delta^{mul}_{j,i}$. The worst-case interference of all these jobs is maximized when every job of τ_l loads all its cache blocks (i.e., ECB_l). Hence, the worse-case impact that these jobs of τ_l can have on the τ_j's CPRO is bounded by the multi-set $\bigcup_{E_l(R_i)-N^{double}_{l,j}(R_i)} ECB_l$ given in the first term of Equation (5.11).

3.2 For all jobs of τ_l that can contribute to both the CRPD and CPRO of τ_j, i.e., $N^{double}_{l,j}(R_i)$, then as stated in Lemma 5.1, the evictions of cache blocks of τ_j in $UCB_j \cap PCB_j$ were already considered in γ^{tot-m}_i. Therefore, the number of cache block evictions caused by these $N^{double}_{l,j}(R_i)$ jobs of τ_l on τ_j that were not accounted for in γ^{tot-m}_i is maximized when each job loads all the cache blocks in $ECB_l \setminus (UCB_j \cap PCB_j)$. Hence, the worse-case additional impact of all jobs of τ_l that contribute to both the CRPD and CPRO of τ_j is bounded by the multi-set, $\bigcup_{N^{double}_{l,j}(R_i)} ECB_l \setminus (UCB_j \cap PCB_j)$ given by the second term of Equation (5.11).

[1] Recall from Equation (4.4) that all PCBs are assumed to be loaded once anyway.

Therefore, by 2. and 3. above, the largest set of ECBs that can interfere with the PCBs of τ_j during the response time of τ_i is upper bounded by $M_{j,i}^{ecb} = M_{j,i}^{ecb-aff} \cup M_{j,i}^{hp-int}$ given by Equation (5.9). Hence, the largest set of PCBs of τ_j that can be evicted by the tasks in $\text{hep}(i) \setminus \tau_j$ within the response time of τ_i with evictions not already considered in γ_i^{tot-m}, is upper bounded by the intersection of $M_{j,i}^{pcb}$ with $M_{j,i}^{ecb}$. Since reloading a cache block takes at most d_{mem} time units, an upper bound on the total CPRO $\delta_{j,i}^{mul}$, not already included in the CRPD, is given by $d_{mem} \times \left| M_{j,i}^{ecb} \cap M_{j,i}^{pcb} \right|$ in Equation (5.7). □

As a corollary of Lemma 5.7, we can upper-bound the total memory reload overhead μ_i as stated in the following theorem:

Theorem 5.3. *The total memory reload overhead μ_i during τ_i's response time is upper-bounded by*

$$\sum_{\tau_j \in \text{hp}(i)} \left(\gamma_{i,j}^{ucb-m} + \delta_{j,i}^{mul} \right) \tag{5.12}$$

Proof. Follows from Lemma 5.7 since $\mu_i = \Delta_i^m + \gamma_i^{tot-m}$. □

This leads directly to the following theorem.

Theorem 5.4. *The WCRT of τ_i is upper-bounded by the smallest positive solution to*

$$R_i = C_i + \sum_{\forall j \in \text{hp}(i)} \left(\gamma_{i,j} + \min \left\{ \left\lceil \frac{R_i}{T_j} \right\rceil C_j \, ; \left\lceil \frac{R_i}{T_j} \right\rceil PD_j + \hat{MD}_j(R_i) + \delta_{j,i}^{mul} \right\} \right) \tag{5.13}$$

where $\gamma_{i,j}$ is given by $\gamma_{i,j}^{ucb-m}$ for UCB-Union Multi-set (i.e., Equation (3.7)).

Proof. By Theorem 5.3 and substituting $\delta_{j,i}^{mul}$ for $\hat{\rho}_{j,i}$ in Equation (4.9) □

Since, $\delta_{j,i}^{mul}$ calculated using Equation (5.7) is always less than or equal to $\rho_{j,i}^{mul}$ calculated using Equation (4.11), the resulting WCRT obtained using Equation (5.13) is always less than or equal to the WCRT obtained using Equation (4.9) when $\gamma_{i,j}$ is computed using the UCB-Union multi-set approach. In other words, the integrated multi-set approach to CRPD and CPRO analysis given by Theorem 5.4 *dominates* the seperate combination of the UCB-Union multi-set and CPRO multi-set approaches.

5.4 Experimental Evaluation

In this section, we evaluate how the integrated CRPD-CPRO analyses perform in terms of schedulability and whether it is beneficial to use the integrated approaches in comparison to the analyses that separately account for CRPD and CPRO. We performed experiments using the Mälardalen benchmark suite (Gustafsson et al., 2010) and a set of sequential benchmarks from TACLEBench (Heiko, 2016) with various parameter settings.

The tasks parameters C_i, PD_i, MD_i, MD_i^r along with the sets of UCB_i, ECB_i, PCB_i and $nPCB_i$ were extracted using the Heptane static WCET analysis tool (Hardy et al., 2017) as presented in Chapter 4. The target architecture was MIPS R2000/R3000 assuming a cache line size of 32 Bytes, a cache size of 8kB and a block reload time $d_{mem} = 8\mu$s. The memory footprint of each task was upper bounded by 256 cache sets (i.e., 100% of the cache size). Table 5.2 shows the resulting task parameters for the benchmarks used during the experiments.

The other task set parameters were randomly generated as follows. The default number of tasks was 10 with task utilizations generated using UUnifast (Bini and Buttazzo, 2005). Each task was randomly assigned the values C_i, PD_i, MD_i, MD_i^r, UCB_i, ECB_i, PCB_i and $nPCB_i$ of one of the analyzed benchmarks. Task periods were set such that $T_i = C_i/U_i$. Task deadlines were implicit and priorities were assigned in deadline monotonic order.

We conducted experiments varying the total task utilization, cache size, block reload time and task memory footprints. A WCRT based schedulability analysis is performed using the same task sets for all approaches.

5.4.1 Core Utilization.

In this experiment, we randomly generated 100 task sets (with 10 tasks each) with a total utilizations varied from 0.025 to 1 in steps of 0.025. The experiment was first performed using the Mälardalen benchmarks and then using TACLEBench's sequential benchmarks.

Figure 5.3a and 5.3b show the number of task sets that were deemed schedulable by the different analyses. Both plots also show the number of task sets that were deemed schedulable without considering any CRPD or CPRO. We only show cropped versions of the plots starting from a utilization of 0.7. All approaches produce identical results below this point.

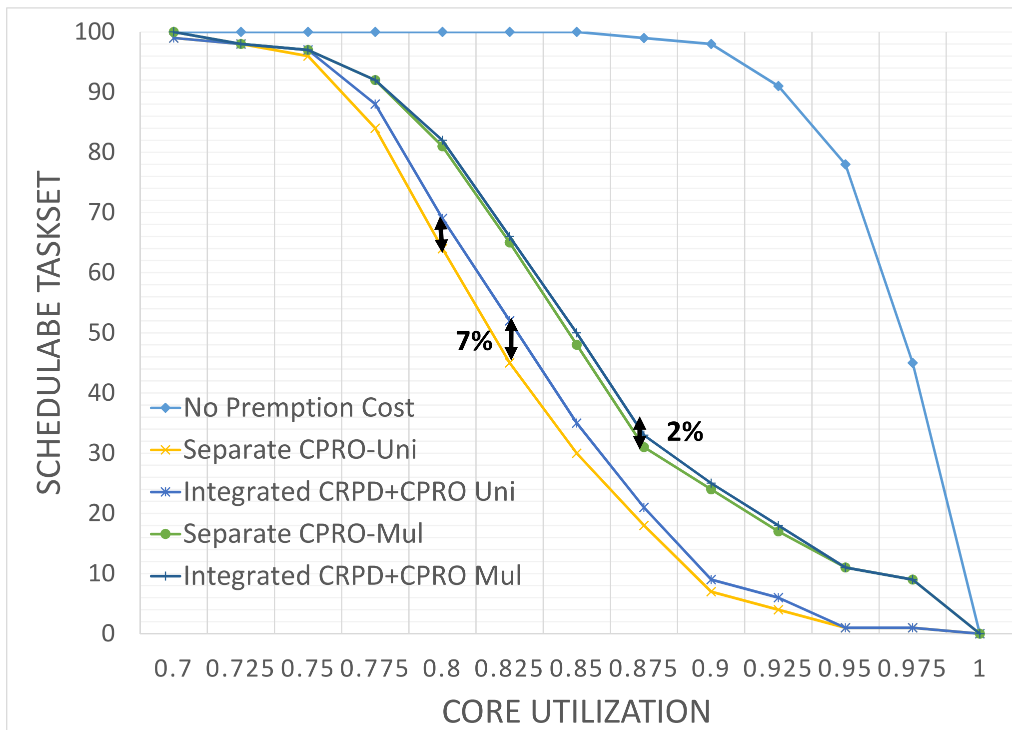
Observation 5.1. *Integrated CRPD-CPRO analyses out-perform the state-of-the-art CPRO-union and multi-set approaches that separately account for CRPD and CPRO.*

Figure 5.3a shows that when using Mälardalen benchmarks the integrated schedulability tests accepted more task sets in comparison to tests using separate CRPD and CPRO analyses. The difference between the integrated CRPD-CPRO union approach and the separate CPRO-union approach is more significant in comparison to their multi-set counterparts. The schedulability ratio is increased by up to 7%. However, as the separate CPRO multi-set approach is already much more precise the difference between the integrated CRPD-CPRO multi-set and the separate CPRO multi-set approach is only around 2%. Nevertheless, we can observe that there are task sets that were schedulable using the integrated CRPD-CPRO approaches but not with the separate CPRO-union and multi-set approaches, therefore in this case the integrated CRPD-CPRO approaches outperforms the separate CPRO-union and multi-set approaches. Note also that the schedulability gain slightly increases when the cache size increases. For instance, when there are 512 cache sets the gain is 8% for the integrated CRPD-CPRO union analysis, and 4% for the multi-set analysis.

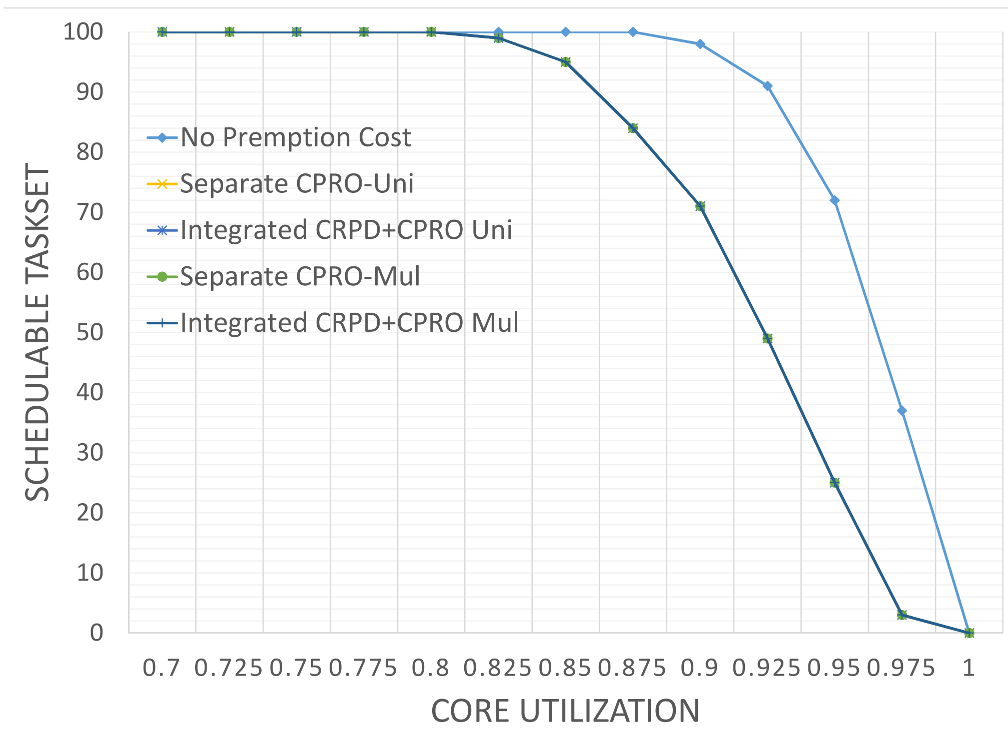
Observation 5.2. *For benchmarks (i.e., tasks) with large memory footprints, there is no gain when integrating the CRPD-CPRO calculation.*

Table 5.2: Task parameters for the benchmarks used during the experiments

Name	C_i	PD_i	MD_i	$MD^r{}_i$	ECB_i	PCB_i	UCB_i	$nPCB_i$	Benchmark Type
lcdnum	3440	984	2740	192	20	20	20	0	Mälardalen
bs	1399	203	1223	34	11	11	10	0	Mälardalen
fibcall	1585	785	886	89	8	8	7	0	Mälardalen
bsort100	712289	710289	90893	88907	20	20	18	0	Mälardalen
select	17138	11158	7858	1394	60	60	60	0	Mälardalen
sqrt	5667	2770	3242	362	26	26	25	0	Mälardalen
jfdctint	17347	7747	10473	965	96	96	96	0	Mälardalen
insertsort	7574	5974	2343	752	16	16	10	0	Mälardalen
cnt	10090	7191	3818	933	27	27	26	0	Mälardalen
prime	25891	23791	4246	2152	17	17	16	0	Mälardalen
ndes	137968	120823	31871	14834	121	75	100	46	Mälardalen
crc	143172	135796	25288	17932	44	44	43	0	Mälardalen
fdct	17350	6550	11525	9327	106	22	58	84	Mälardalen
minver	21668	4868	17265	518	167	167	159	0	Mälardalen
fft	157880	123681	45816	11888	141	141	140	0	Mälardalen
ud	28427	20627	10415	10415	75	53	31	22	Mälardalen
adpcm	230123	196131	55609	21501	240	240	237	0	Mälardalen
nsichneu	316409	22009	294400	294400	256	0	256	256	Mälardalen
statemate	190496	10586	180110	180110	256	36	256	220	Mälardalen
fmref	12117800	2143590	10148500	10063200	256	161	256	95	TACLEBench
adpcm-dec	479761	460616	84090	64892	173	173	172	0	TACLEBench
adpcm-enc	482994	462750	70921	50646	178	178	177	0	TACLEBench
h264-dec	2609630	1661910	1143780	1130800	256	133	256	123	TACLEBench
huff-dec	821956	808273	112838	97680	84	84	84	0	TACLEBench
lift	1945120	1929300	282201	265799	140	140	140	0	TACLEBench
petrinet	38532	4632	34191	9633	256	229	256	27	TACLEBench
audiobeam	1883880	1824060	310955	302240	253	75	253	178	TACLEBench



(a) Schedulability ratio using Mälardalen benchmarks



(b) Schedulability ratio using TACLEBench

Figure 5.3: Schedulability ratio with respect to total core utilization

As shown in Figure 5.3b, the integrated CRPD-CPRO analyses do not improve over the state-of-the-art for the TACLEBench benchmarks. In fact, the same number of task sets were schedulable using all the approaches. The difference with Figure 5.3a can be understood as follows. Mälardalen benchmarks consist of both light and heavy tasks (see Table 5.2) whereas the majority of tasks in TACLEBench have large memory footprints using the entire cache. Therefore, almost all tasks overlap in the cache, in which case the tasks with lower priority than a task τ_j (i.e., the tasks in $\text{aff}(i, j)$) evict the same cache blocks of τ_j as the tasks with higher priority (i.e, in $\text{hp}(j)$). Hence, according to Equation (5.2) and Equation (5.7), integrating the CRPD and CPRO analyses does not provide any gain.

From here on, we only show experimental results obtained using the Mälardalen benchmarks.

5.4.2 Cache size

For fixed priority preemptive systems, the cache size can have a significant impact on the overall schedulability of the system. In this experiment, we vary the total number of cache sets from 32 to 512. Figure 5.4a shows the resulting weighted schedulability (Bastoni et al., 2010) (see Equation (4.20)) of each approach plotted against the total cache size[2].

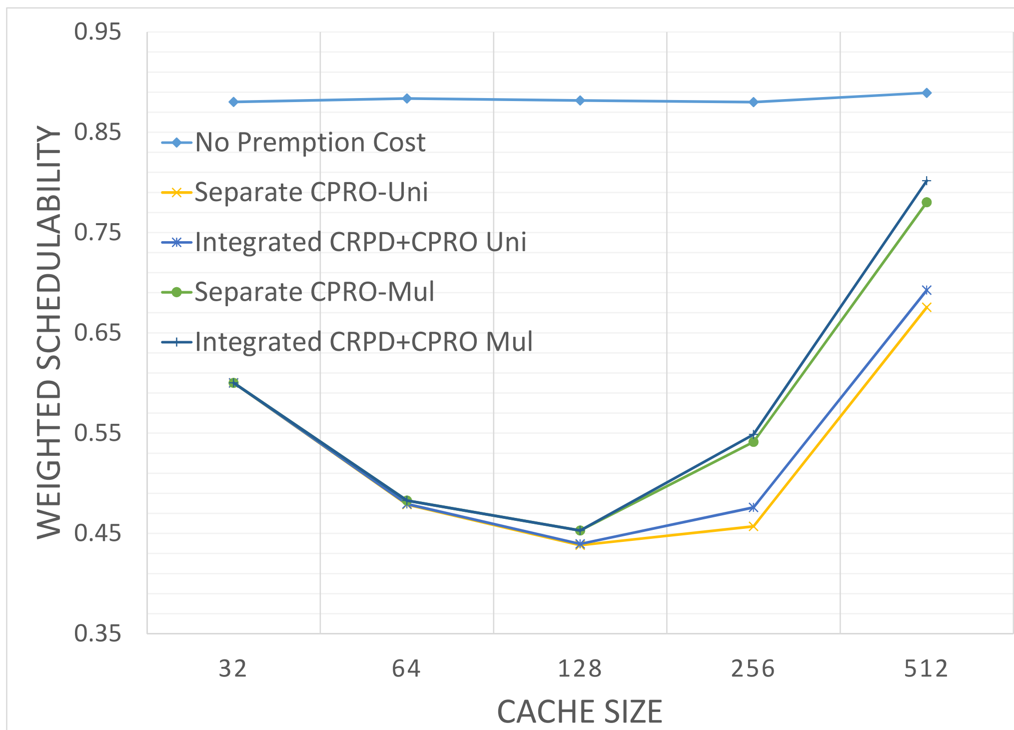
Observation 5.3. *The integrated CRPD-CPRO analyses tend to outperform the separate analyses when the cache size increases.*

We can see from the plot in Figure 5.4a, that initially increasing the cache size decreases the schedulability of all the approaches (i.e., from 32 to 128). This is mainly because most tasks use between 32 to 128 cache sets. Hence, increasing the cache size in this interval increases the number of ECBs and UCBs of tasks resulting in higher values of CRPD. Most of the cache blocks are evicted (and reloaded) for every task execution and hence we observe that all the approaches produce similar results. However, a further increase in cache size (i.e., from 128 to 512) means more tasks fit in the cache with less conflicts between tasks. Therefore, we see an increase in schedulability of all approaches. Also increasing the cache size results in increasing the number of PCBs of tasks, so the overlap between UCBs and PCBs of tasks also increase. Hence, we observe that with an increase in cache size from 128 to 512, the integrated CRPD-CPRO union and multi-set approach tend to perform better than the analyses that separately accounts for CPRD and CPRO.

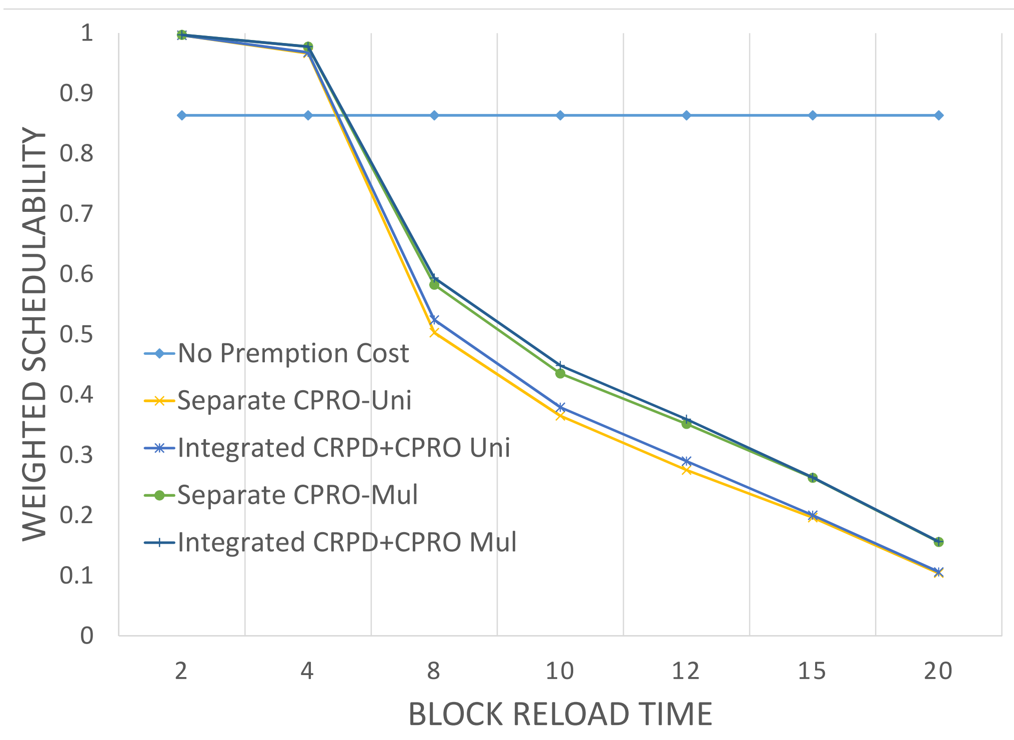
5.4.3 Block Reload Time (d_{mem})

In this experiment, we analyze the impact of block reload time d_{mem} on the performance of all the approaches by varying it between 2μs to 20μs, with all other parameters set to default values. Figure 5.4b shows the resulting weighted schedulability.

[2]When calculating weighted schedulability we only consider task set utilizations between 0.6 to 1 since for lower utilizations, all task sets are schedulable.



(a) Varying cache size



(b) Varying block reload time d_{mem}

Figure 5.4: Weighted schedulability measure by varying cache utilization, block reload time d_{mem} and cache size

Observation 5.4. *For very low or very high values of block reload time d_{mem}, the integrated and separate CRPD-CPRO analyses produce similar results.*

For smaller values of d_{mem} (i.e., between 2μs and 4μs) the impact of CRPD and CPRO on the schedulability of tasks is minimal. This means that similar results are achieved for integrated and separate union and multi-set approaches. Similarly, for higher values of d_{mem} (i.e., $d_{mem} > 15\mu$s), the CRPD becomes very high and thus negates any gain in schedulability resulting from the reduction of the CPRO cost in the integrated analysis. In contrast, for values of d_{mem} between 8μs to 12μs the impact of the overlap between CRPD and CPRO is visible. Note that for very low values of d_{mem} (i.e., between 2μs and 4μs) all analysis perform better than the "No Preemption Cost" analysis. This is mainly because the "No Preemption Cost" analysis does not account for cache persistence and only use the WCET of tasks to compute the response time.

5.4.4 Task Priority and Memory footprint

The integrated CRPD-CPRO approaches avoid double counting in the total memory reload overhead caused by the higher priority tasks. Therefore, the memory footprints of higher priority tasks can greatly affect the performance of the integrated CRPD-CPRO analysis.

To evaluate the impact of task memory footprints on the performance of the integrated CRPD-CPRO approaches, we performed a simple experiment using a single task set comprising 6 tasks (τ_1 to τ_6, where τ_1 has the highest priority). We increased the memory footprint (i.e., number of ECBs) of the highest priority task τ_1 and analyzed its impact on the total memory reload overhead μ_4 of the medium priority task τ_4. Task set parameters used in this experiment were set as follows. Core utilization was fixed at 0.7, with task utilizations generated using UUnifast algorithm. Each task was assigned parameters using the *ludcmp* benchmark[3]. Task periods were set such that $T_i = C_i/U_i$ (i.e., $T_1 = 161586$, $T_2 = 171642$, $T_3 = 220971$, $T_4 = 710848$, $T_5 = 1363503$ and $T_6 = 14533791$). Cache size was fixed to 256 cache sets with $d_{mem} = 8\mu$s.

In this experiment, we evaluate the relative performance of the integrated CRPD-CPRO approaches in terms of memory reload overhead μ. Therefore, we report the *gain* on the total memory reload overhead μ^{gain} for task τ_4, i.e., μ_4^{gain}, by increasing the number of ECBs of the highest priority task τ_1.

The relative gain μ_i^{gain} is defined as $\mu_i^{gain} \stackrel{\text{def}}{=} \frac{\mu_i^{sep}-\mu_i^{int}}{\mu_i^{sep}}$ where μ_i^{sep} is the total memory reload overhead for task τ_i under the separate CRPD and CPRO analysis and μ_i^{int} is similarly the total obtained with the integrated analysis. For the integrated CRPD-CPRO Union approach, μ_i^{int} is given by Equation (5.3), whereas for the CRPD-CPRO multi-set approach μ_i^{int} is given by Equation (5.12). For the separate approaches, in each case the value of $\rho_{j,i}$ or $\rho_{j,i}^{mul}$ is used instead of $\delta_{j,i}$ or $\delta_{j,i}^{mul}$.

Observation 5.5. *If the memory footprint of higher priority tasks increase, then the relative gain of the integrated analyses over the state-of-the-art analyses increases.*

[3] Here, we deliberately chose a benchmark with significant memory footprint to impact the memory reload overhead of other tasks.

Table 5.3: Relative gain μ_4^{gain} for the CRPD-CPRO union and multi-set approaches by increasing the number of ECBs of τ_1

Increase of τ_1's ECBs (%)	μ_4^{gain} with integrated CRPD-CPRO union	μ_4^{gain} with integrated CRPD-CPRO multi-set
No Increase	9%	12%
20%	11%	16%
40%	13%	18%
60%	14%	20%
80%	15%	20%
100%	16%	20%

Table 5.3 shows that the gain in total memory reload overhead of τ_4 increases with the τ_1's memory footprint.

This behavior can be explained as follows. If one of the higher priority tasks (e.g., τ_1) has a big memory footprint (i.e., more ECBs) it can contribute more to both CRPD and CPRO of lower priority tasks. This results in increasing the overlap between the CRPD and CPRO of those tasks. In contrast, if the higher priority tasks have small memory footprints, they will have less impact on the CRPD and CPRO of medium and lower priority tasks and hence the overlap between the CRPD and CPRO will also be small.

This observation explains the rather small average schedulability gain in the experiments presented until now. Since tasks with smaller memory footprints mostly have lower execution times, their periods are most of the time shorter. Therefore, higher priority tasks usually have smaller memory footprints in the randomly generated task sets, hence resulting in a reduced gain. Yet, we note that this relationship between memory footprint, WCET, and period does not always hold in practice. Tasks with short periods and a relatively small WCET may still have a substantial memory footprint if they implemented via straight-line code. Similarly tasks with long WCETs may have a small memory footprint in the case where they implement a small loop that is repeated many times.

5.5 Chapter Summary

In this Chapter we answer two questions: (1) Is it beneficial to integrate the calculation of CRPD and CPRO? and (2) when and to what extent can we gain in terms of schedulability by integrating the calculation of CRPD and CPRO? Our experimental evaluation, as well as theoretical dominance results, showed that integrated CRPD-CPRO analysis can, in some cases, increase the schedulability ratio by 2% to 7% by providing a tighter calculation of total memory reload overheads compared to the analyses that treat CRPD and CPRO independently. However, as pointed out using a set of observations in the experimental evaluation the gains obtained using the integrated CRPD-CPRO analysis are dependent on certain system configurations and parameter values. The average gains in terms of schedulability resulting from the integration of CRPD-CPRO

calculations may not be large; however, it is important to note that nevertheless, the integrated approaches dominate the separate treatment of CRPD and CPRO and this dominance is obtained with no increase in complexity, or need for extra information. Therefore, it is indeed beneficial to integrate the calculation of CRPD and CPRO.

Chapter 6

Evaluating the Impact of Memory Layout of Tasks on Schedulability

When computing the inter-task cache interference, i.e., CRPD and CPRO, the analysis presented in Chapter 4 considers the worst-case task layout in memory, i.e., all tasks start at the same static memory address, which maximizes the inter-task cache interference. Similarly, the analysis presented in Chapter 5, assume a sequential layout of tasks in memory. As discussed in Section 3.3.2, the position of a task in the main memory determines which cache blocks will be used by the task which in turn impacts the inter-task cache interference that can be suffered by that task. In this chapter, we will investigate the impact of different task layouts on task set schedulability.

As already discussed in Chapter 3, several different approaches have been presented in literature to bound the intra- and inter-task cache interference by means of cache partitioning (Kirk and Strosnider, 1990; Wolfe, 1993; Altmeyer et al., 2014; Busquets-Mataix et al., 1997; Bui et al., 2008; Kim et al., 2013; Altmeyer et al., 2016) or by optimizing the task layout in memory (Lunniss et al., 2012; Altmeyer and Gebhard, 2007). However, these existing cache partitioning and task layout optimization techniques focus on either the intra- or the inter-task cache interference and do not exploit the fact that both intra- and inter-task cache interference can be *interrelated*. For example, the cache partitioning approaches mainly focus on inter-task cache interference and are subjected to one basic problem: the available cache space may not be enough for each task to have its own independent (i.e., non-overlapping) cache partition. Also with cache partitioning, as the number of tasks increase, cache space that can be used for each individual task becomes always smaller. This reduced amount of cache space available to each task potentially increases its intra-task cache interference (i.e., the task may itself start to evict its own cache blocks) resulting in an increased execution time due to an increase in the number of main memory accesses. This may eventually cause the task to become unschedulable even though it does not suffer any inter-task cache interference. Similarly, the approaches focusing on optimizing task layout in memory (Lunniss et al., 2012; Altmeyer and Gebhard, 2007) changes task placements in memory to reduce the inter-task cache interference while allowing tasks an unconstrained use of the cache. However, even with an optimal layout of tasks in memory, allowing tasks an unconstrained use of cache

may still result in higher inter-task cache interference, e.g., the cache block evictions of lower priority tasks caused by a higher priority task using the whole cache will be inevitable even with an optimal layout of tasks unless the cache space used by the higher priority task is reduced (i.e., potentially increasing the intra-task cache interference of the higher priority task to decrease the inter-task cache interference it may cause).

In this chapter, we evaluate the impact of memory layout of tasks on schedulability by identifying the relationship between intra- and inter-task cache interference. First, we show how one can model intra- and inter-task cache interference in a way that allows balancing their respective contribution to tasks worst-case response times. We then propose a technique optimizing the task layout in memory that result in improved task set schedulability. The main contributions of this chapter are as follows:

1. We present a *cache coloring* approach to optimize task layout in memory such that cache colors assigned to tasks are not strictly private but may be shared between tasks;

2. we model the impact of a given cache color assignment on different task parameters and show how intra- and inter-task cache interference can be upper-bounded when using cache coloring;

3. we present a *simulated annealing* algorithm to optimize the cache color assignment to tasks by re-allocating and re-sizing the cache colors assigned to tasks such that the task set's schedulability is achieved; and

4. we perform an experimental evaluation using a set of benchmarks showing that our approach results in up to 13% higher schedulability than state-of-the-art approaches.

6.1 Cache Coloring

As discussed in Section 3.3.1, cache (or page) coloring is a software technique that is widely used to partition the cache by controlling the mapping of physical memory address to cache blocks (Liedtke et al., 1997; Guan et al., 2009; Mancuso et al., 2013; Kim et al., 2013). Cache coloring is mainly supported by systems that use *virtual* memory. The rationale behind virtual memory is to divide the address space used by tasks into blocks called *pages*. Each page represent a series of contagious memory addresses used by tasks. The Memory Management Unit (MMU) maps these pages to physical memory location by translating the virtual page addresses into physical memory addresses. Figure 6.1 shows an example mapping between physical addresses and cache entries. When virtual memory is used, each memory address referenced by a task translates into a virtual address, where g least significant bits represent a page offset. The remaining bits of the virtual address corresponds to a physical page number. The location of a memory block in the cache is determined by its physical address. We can see in Figure 6.1 that for a cache with 2^s cache sets and a cache line size of 2^l bytes, last l bits of the physical address represent the cache line offset. Whereas, the next s bits are used as a set index into the cache. In this scenario, the

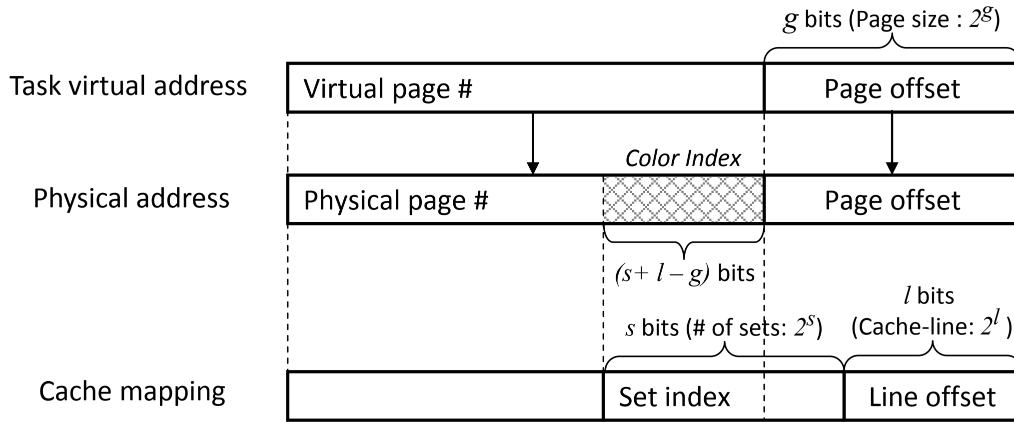


Figure 6.1: A visual representation of cache coloring (Kim et al., 2013)

overlapping bits between physical page number and the set index represent the *color* bits. Cache coloring uses these overlapping bits as a color index, which effectively divides the cache into 2^{s+l-g} cache partitions. By controlling the color of pages assigned to a task, the operating system (OS) can manipulate cache blocks at the granularity of the page size times the cache associativity. The maximum number of colors that a platform can support is usually computed as follows:

$$\text{Number of Cache Colors} = \frac{CacheSize}{CacheAssociativity \times PageSize}$$

6.2 Assumptions on the System Model

In addition to the general system model detailed in Chapter 2 and Chapter 4, in this chapter we make the following assumptions on the system model.

- The cache is assumed to be direct-mapped with k^{total} colors. Each color is uniquely numbered between 1 to k^{total}. The size of a cache color is denoted by k^{size} and is equal the number of successive sets in the cache that may be used by tasks assigned to that color. For simplicity, we assume that the size of every cache color is the same. Note that this is a common practice in real systems (Gracioli et al., 2015).

- The task set Γ comprises n tasks, i.e., $\Gamma = \{\tau_1, ..., \tau_n\}$. Each task τ_i is defined by a triplet $(C_i[k_i], T_i, D_i)$, where $C_i[k_i]$ is a vector of length k^{total} that contains the worst-case execution time of task τ_i in isolation assuming k_i contiguous cache colors are assigned to τ_i. Note that k_i represents the *number* of cache colors used by τ_i, whereas the *set* of cache colors assigned to τ_i is denoted by ck_i. The minimum inter-arrival time of τ_i is T_i and D_i is its relative deadline. We assume that the tasks have constrained deadlines, i.e., $D_i \leq T_i$.

- For each task τ_i, PD_i denotes the worst-case processing demand of τ_i considering that every memory access is a cache hit. $MD_i[k_i]$ is the worst-case memory access demand (in terms of time) of any job of task τ_i executing in isolation and assuming that k_i contiguous cache

colors are assigned to τ_i. It is usually assumed that $C_i[k_i]$ is non-increasing with k_i, i.e., $k_i < k_i + 1 \implies C_i[k_i] \geq C_i[k_i+1]$. However, we note that since PD_i is independent of the number of cache colors assigned to τ_i, it is the worst-case memory access demand $MD_i[k_i]$ which must be defined as a non-increasing function w.r.t. the number of cache colors assigned to τ_i, i.e., $k_i < k_i + 1 \implies MD_i[k_i] \geq MD_i[k_i+1]$. Furthermore, we assume that the values of $C_i[k_i]$, PD_i and $MD_i[k_i]$ can be calculated using a static timing analysis tool such as Heptane (Hardy et al., 2017).

- Similar to the definition of $C_i[k_i]$ and $MD_i[k_i]$, in this chapter we assume that the residual memory access demand (see Definition 4.3) of task τ_i also depends on the number of cache colors assigned to τ_i, i.e., k_i. Hence, in this chapter, we will denote the residual memory access demand of task τ_i by $MD_i^r[k_i]$. Consequently, Equation (4.4) used to compute the total memory access demand $\hat{MD}_i(t)$ of task τ_i within a time window of length t is adapted as follows:

$$\hat{MD}_i(t) = \min\left\{ \left\lceil \frac{t}{T_i} \right\rceil \times MD_i[k_i] \; ; \left\lceil \frac{t}{T_i} \right\rceil \times MD_i^r[k_i] + PCB_i(k_i) * d_{mem} \right\} \quad (6.1)$$

  where $PCB_i(k_i)$ denote the maximum number of PCBs of task τ_i, when τ_i is assigned k_i cache colors.

The list of important symbols used in this chapter is provided in Table 6.1.

Table 6.1: List of important symbols used in Chapter 6

Symbol	Description
Γ	Task set of size n
τ_i	Task with index i
k^{total}	The total number of colors in the cache
k^{size}	The size of one cache color
k_i	The number of cache colors allocated to task τ_i
ck_i	The set of cache colors allocated to task τ_i.
$C_i[k_i]$	Vector of length k^{total} that contains the worst-case execution time of task τ_i in isolation assuming k_i contiguous cache colors are assigned to τ_i.
C_i^{min}	Worst-case execution time of task τ_i in isolation assuming τ_i is allocated a cache of infinite size, i.e., the total cache space assigned to task τ_i is greater or equal to the size of τ_i in main memory.
T_i	Minimum inter-arrival time of task τ_i
D_i	Relative deadline of task τ_i
R_i	Worst-case response time of task τ_i
PD_i	Worst-case processing demand of task τ_i in isolation

Continued on next page

Table 6.1 – continued from previous page

Symbol	Description
$MD_i[k_i]$	Worst-case memory access demand of any job of task τ_i executing in isolation and assuming that k_i contiguous cache colors are assigned to τ_i.
MD_i^{min}	Worst-case memory access demand of task τ_i in isolation assuming τ_i is allocated an infinite cache space, i.e., the total cache space assigned to task τ_i is greater or equal to the size of τ_i in main memory.
MD_i^{max}	Maximum worst-case memory access demand of τ_i in isolation when there is no cache assigned to task τ_i (i.e., $k_i = 0$).
$\Delta MD_i[k_i]$	The change in the worst-case memory access demand $MD_i[k_i]$ of task τ_i due to an increase in the number of cache colors k_i assigned to τ_i.
$MD_i^r[k_i]$	Residual memory access demand of task τ_i in isolation assuming that k_i contiguous cache colors are assigned to τ_i.
$\Delta MD_j^r[k_j]$	The change in the residual memory access demand $MD_j^r[k_j]$ of task τ_j due to an increase in the number of cache colors k_j assigned to τ_j.
$\hat{MD}_i(t)$	Total memory access demand of task τ_i in a time interval of length t
$\text{hp}(i)$	The set of tasks with higher priority than τ_i
$\text{hep}(i)$	The set of tasks with higher priority than τ_i including τ_i, i.e., $\text{hep}(i) = \text{hp}(i) \cup \tau_i$.
$\text{aff}(i, j)$	The set of intermediate tasks (including τ_i) that may preempt τ_i but may themselves be preempted by some higher priority task τ_j.
d_{mem}	Time to reload one cache block from the main memory
CI_i^{intra,k_i}	Upper bound on the intra-task cache interference suffered by task τ_i when assigned k_i contiguous cache colors.
$CI_{i,j}^{inter,\gamma}(R_i)$	Upper bound on the inter-task cache interference in terms of CRPD that task τ_i may suffer during its response time due to preemptions by any higher priority task $\tau_j \in \text{hp}(i)$.
$CI_{j,i}^{inter,\rho}(R_i)$	Upper bound on the inter-task cache interference in terms of CPRO that each higher priority task $\tau_j \in \text{hp}(i)$ may suffer during the response time of τ_i.
$k_{i,j}$	The worst-case number of cache colors that may suffer evictions as a result of a single preemption of task τ_i by task τ_j.
$k'_{j,i}$	The maximum number of cache colors of task τ_j that can be evicted between its successive jobs due to the executions of all tasks in $\text{hep}(i) \setminus \tau_j$ during the response time of task τ_i.
$ECB_i(k_i)$	The maximum number of ECBs of task τ_i when assigned k_i cache colors
$UCB_i(k_i)$	The maximum number of UCBs of task τ_i when assigned k_i cache colors
$PCB_i(k_i)$	The maximum number of PCBs of task τ_i when assigned k_i cache colors
N_i	The average number of times each UCB of task τ_i is accessed while it is cached
N'_j	The average number of times each PCB of τ_j is accessed after it is loaded in the cache.

Continued on next page

Table 6.1 – continued from previous page

Symbol	Description
$\rho^{col}_{j,i}$	CPRO suffered by one job of a higher priority task $\tau_j \in \text{hp}(i)$ during the response time of a lower priority task τ_i, when using cache coloring.
$\rho^{one}_{j,i}$	CPRO of one job of a higher priority task $\tau_j \in \text{hp}(i)$ during the response time of a lower priority task τ_i, when considering the difference between the worst-case and the residual memory access demand of task τ_j.
$\gamma^{col}_{i,j}$	CRPD suffered by task τ_i due to one preemption by any higher priority task $\tau_j \in \text{hp}(i)$ when using cache coloring.
$\gamma^{ucb}_{i,j}$	CRPD suffered by task τ_i due to one preemption by any higher priority task $\tau_j \in \text{hp}(i)$, computed using the UCB-union approach (i.e., Equation (3.3)).
$\gamma^{tot}_{i,j}$	Total CRPD suffered by task τ_i in a time interval of length t due to preemptions by any higher priority task $\tau_j \in \text{hp}(i)$.
Sl_i	Slack of task τ_i, i.e., the difference between the relative deadline and the WCRT of τ_i.
Sl^{tot}	Total slack of task set Γ

6.3 Cache Interference Aware WCRT Analysis

In this chapter, we will calculate the WCRT of a task τ_i using a similar formulation as presented in Equation (4.9). However, we will explicitly consider the intra- and inter-task cache interference suffered by tasks during the response time R_i of task τ_i, i.e.,

$$R_i = C_i^{min} + CI_i^{intra,k_i} + \sum_{\forall j \in \text{hp}(i)} \left(\min\left\{ \left\lceil \frac{R_i}{T_j} \right\rceil \left(C_j^{min} + CI_j^{intra,k_j} \right) ; \left\lceil \frac{R_i}{T_j} \right\rceil PD_j + \hat{MD}_j(R_i) + CI_{j,i}^{inter,\rho}(R_i) \right\} + CI_{i,j}^{inter,\gamma}(R_i) \right) \tag{6.2}$$

In Equation (6.2), C_i^{min} denotes the worst-case execution time of task τ_i in isolation assuming τ_i is allocated a cache of infinite size (or more practically, the total cache space assigned to task τ_i is greater or equal to the size of τ_i in main memory). The intra-task cache interference of τ_i w.r.t the number of cache colors k_i assigned to τ_i is denoted by CI_i^{intra,k_i} as intra-task interference impacts only the execution time of τ_i itself. Similarly, the intra-task cache interference CI_j^{intra,k_j} of each higher priority task $\tau_j \in hp(i)$ executing during the response time of τ_i is considered in the higher priority interference term within the sum on higher priority tasks. Moreover, $CI_{i,j}^{inter,\gamma}(R_i)$ denotes the inter-task cache interference in terms of CRPD that task τ_i may suffer during its response time due to preemptions by any higher priority task $\tau_j \in \text{hp}(i)$ and $CI_{j,i}^{inter,\rho}(R_i)$ bounds the inter-task cache interference in terms of CPRO that each higher priority task $\tau_j \in \text{hp}(i)$ may suffer during the response time of τ_i. Note that in Equation (6.2), $\hat{MD}_j(R_i)$ will be calculated using Equation (6.1).

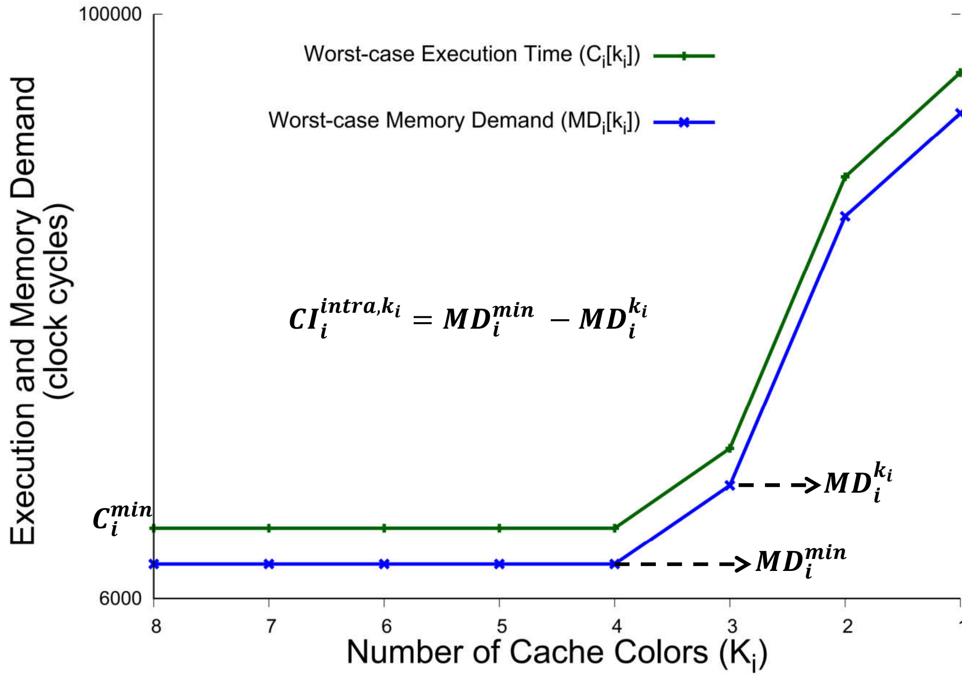


Figure 6.2: Increase in execution demand and memory access demand of task τ_i due to reduction in number of cache colors assigned to τ_i.

Since $\hat{MD}_j(R_i)$ is a function of $MD_j[k_j]$, $MD^r_j[k_j]$ and the number of PCBs of τ_j, which are in turn functions of the number of cache colors k_j assigned to τ_j therefore, $\hat{MD}_j(R_i)$ directly considers the intra-task interference of all jobs of τ_j executing during the response time of τ_i. In the following sections, we detail how the total intra- and inter-task cache interference can be bounded under the cache coloring approach considered in this chapter.

6.4 Bounding Intra-Task Cache Interference

Intra-task cache interference represents contention between different code segments of a task that are mapped to the same cache space. If the cache space allocated to a task is not sufficient to hold all its instructions/data, the task may self-evict its own cache content resulting in higher main memory access demand even when the task is executing in isolation. For a task τ_i its intra-task cache interference depends on the cache space or the number of cache colors k_i assigned to τ_i. Consider the plot of worst-case execution time ($C_i[k_i]$) and the worst-case memory access demand ($MD_i[k_i]$) of task τ_i with respect to the number of cache colors k_i assigned to τ_i shown in Figure 6.2. The plot shows the actual variation in the worst-case execution time and the worst-case memory access demand of the benchmark `fdct` of the Mälardalen benchmark suite (Gustafsson et al., 2010), i.e., represented as task τ_i, when the number of cache colors k_i assigned to τ_i are varied in a descending order from 8 to 1. The values in Figure 6.2 were obtained using Heptane (Hardy et al., 2017) for a cache with 8 cache colors, each having a size of 512 Bytes.

Figure 6.2 shows that when the number of cache colors (or cache space) assigned to task τ_i is greater or equal to the size of τ_i in main memory (i.e., for $k_i \geq 4$), the worst-case execution time ($C_i[k_i]$) and the worst-case memory access demand ($MD_i[k_i]$) of τ_i is minimum, i.e., $C_i[k_i] = C_i^{min}$ and $MD_i[k_i] = MD_i^{min}$ for $k_i \geq 4$, where MD_i^{min} represents the worst-case memory access demand of task τ_i in isolation assuming τ_i is allocated an infinite cache size. Effectively, for $k_i \geq 4$ τ_i will suffer no intra-task cache interference.

We can also observe from the plot in Figure 6.2 that by decreasing the number of cache colors k_i assigned to τ_i, its worst-case execution time ($C_i[k_i]$) and the worst-case memory access demand ($MD_i[k_i]$) tend to increase. This increase in $C_i[k_i]$ and $MD_i[k_i]$ is due to an increase in the intra-task cache interference of τ_i mainly because by reducing the number cache colors k_i, the number of UCBs of task τ_i may also decrease, i.e., by decreasing the number of cache colors k_i (or the cache space) assigned to τ_i , cache blocks of τ_i that were previously mapped to different cache sets and were reused more than once before eviction may now map to the same cache set. Consequently, loading one cache block will evict the other thus resulting in reducing the number of cache blocks of τ_i that can be reused, i.e., the number of UCBs. Effectively, this reduction of the number of UCBs results in increasing $MD_i[k_i]$ of τ_i for $k_i < 4$. Therefore, the intra-task cache interference of a task directly relates to it worst-case memory access demand in the following manner

$$CI_i^{intra,k_i} = MD_i[k_i] - MD_i^{min} \tag{6.3}$$

The resulting intra-task cache interference of task τ_i for a given cache color assignment k_i, i.e., CI_i^{intra,k_i}, is accounted for in the WCRT of τ_i (i.e., Equation (6.2)) by explicitly adding CI_i^{intra,k_i} to C_i^{min} which is the worst-case execution time of τ_i in isolation assuming τ_i is allocated an infinite cache. However, we note that because CI_i^{intra,k_i} depends on $MD_i[k_i]$ and since $MD_i[k_i]$ may not necessarily be experienced on the same execution path of τ_i for different cache color assignments ck_i, it holds that $C_i[k_i] \leq C_i^{min} + CI_i^{intra,k_i}$. Hence, Equation (6.3) provides a safe upper-bound on intra-task cache interference even for multi-paths programs.

6.5 Bounding Inter-task Cache Interference

The inter-task cache interference a task τ_i may suffer due to higher priority tasks in $\text{hp}(i)$ is mainly categorized into two types, i.e., the inter-task cache interference due to CRPDs and the inter-task cache interference due to CPROs.

The inter-task cache interference in terms of CRPD results from the eviction of UCBs of τ_i due to preemptions by a higher priority task τ_j in $\text{hp}(i)$ and is denoted by $CI_{i,j}^{inter,\gamma}$. Whereas, the inter-task cache interference in terms of CPRO results from the eviction of PCBs of the higher priority task $\tau_j \in \text{hp}(i)$ due to the executions of all other tasks in the system (while τ_j executes during the response time of τ_i) and is denoted by $CI_{j,i}^{inter,\rho}$. In the following subsections, we explain how $CI_{i,j}^{inter,\gamma}$ and $CI_{j,i}^{inter,\rho}$ can be bounded when using a cache coloring approach.

6.5.1 Inter-Task Cache Interference due to CRPDs

As discussed in Chapter 3 (Section 3.2), a number of methods have been proposed in the literature (Lee et al., 1998; Busquets-Mataix et al., 1996; Tomiyama and Dutt, 2000; Tan and Mooney, 2007; Staschulat et al., 2005; Altmeyer et al., 2011, 2012; Marković et al., 2020a) for computing the CRPD under FPPS using the set of UCBs and/or ECBs. However, in this chapter, we focus on a UCB-union-like approach (Tan and Mooney, 2007) to calculate the CRPD cost due to sharing of cache colors between several tasks. Recall, that the UCB-union approach (Tan and Mooney, 2007) uses intersection between the set of ECBs of the preempting task τ_j and the set of UCBs of all tasks in $aff(i, j)$ possibly affected by the preemption caused by τ_j to calculate CRPD cost $\gamma_{i,j}^{ucb}$ (see Equation (3.3)).

However, when cache colors are being assigned to tasks, Equation (3.3) cannot be used as is. This is mainly because when coloring tasks, any variation in the cache color of any task may potentially change the set of UCBs and ECBs of all tasks in τ. Indeed, the actual mapping of tasks within a cache color may not be known as it is handled by the cache controller.[1] Consequently, the actual set of ECBs/UCBs of tasks may not be known as they depend on the actual cache sets used by the tasks. For example, consider two tasks τ_i and τ_s sharing the same cache color ck, where ck comprises 4 cache sets, numbered from 1 to 4. If both τ_i and τ_s have 2 UCBs under this cache assignment, these UCBs can be mapped to any of the four cache sets depending on how τ_i and τ_s are mapped within ck by the cache controller, i.e., $UCB_i = \{1,2\}$ and $UCB_s = \{3,4\}$ or any other combinations with or without overlapping between UCB_i and UCB_s. Since the actual set of UCBs of tasks might not be known, using different set of UCBs of tasks in Equation (3.3) may produce different pessimistic/optimistic value for the CRPD cost $\gamma_{i,j}^{ucb}$.

In order to bound the CRPD under our cache coloring approach, we first determine the cache colors that may be affected when τ_i is preempted by a higher priority task $\tau_j \in \text{hp}(i)$. Assuming that the cache color assignment of tasks has already been done, i.e., τ_i and τ_j are assigned a set of ck_i and ck_j cache colors respectively.

We know from the UCB-union approach (Equation (3.3)), that when a task τ_i is preempted by a higher priority task τ_j, the set of UCBs of all tasks in $\text{aff}(i, j)$ can be evicted. Similarly, when a task τ_i using a set of ck_i cache colors is preempted by a higher priority task τ_j whose assigned a set of ck_j cache colors, the cache colors used by all tasks in $\text{aff}(i, j)$ may be evicted. Therefore, the maximum number of cache colors that may be affected due to a single preemption of τ_i by τ_j is bounded by $k_{i,j}$, where

$$k_{i,j} = \left| \left(\bigcup_{\forall s \in \text{aff}(i,j)} ck_s \right) \cap ck_j \right| \tag{6.4}$$

[1]Most cache controllers (Liedtke et al., 1997; Lin et al., 2008; Zhang et al., 2009; Suhendra and Mitra, 2008) work at the granularity of a memory page and can be controlled to make sure memory pages of a task map to the specified cache color. However, when sharing cache colors among tasks, memory pages of different tasks may map to the same cache color so changing the mapping of one task may affect the others, making it difficult to predict the actual placement of tasks in cache.

Here, $k_{i,j}$ gives the worst-case number of cache colors that may suffer evictions as a result of a single preemption of task τ_i by task τ_j. Therefore, the product $k_{i,j} \times k^{size}$ can be used to upper bound the number of cache sets that may be evicted due to a single preemption of τ_i by τ_j. However, this bound can obviously be very pessimistic, mainly because it does not consider the actual number of UCBs in those cache sets and hence the actual number of memory blocks that must be reloaded from main memory after eviction.

To tightly bound the CRPD cost, both the number of potentially evicted cache colors, i.e., $k_{i,j}$, and the number of ECBs/UCBs of tasks must be considered. We know that under cache coloring the actual set of ECBs/UCBs, i.e., their mapping in cache, may not be known as they depend on the actual cache sets assigned to tasks. However, their *number* only depends on the number of cache colors assigned to tasks rather than the actual cache sets assigned to those tasks. Therefore, let $UCB_i(k_i)$ and $ECB_i(k_i)$ be defined as

- $UCB_i(k_i)$: The maximum number of UCBs[2] of task τ_i when it is assigned k_i cache colors.
- $ECB_i(k_i)$: The maximum number of ECBs of task τ_i when it is assigned k_i cache colors.

Effectively, the CRPD cost due to a single preemption of τ_i by τ_j can be bounded using the notion of $UCB_i(k_i)$ and $ECB_i(k_i)$.

Lemma 6.1. *The CRPD cost due to a single preemption of a lower priority task τ_i by a higher priority task τ_j is bounded by $\gamma_{i,j}^{col}$, i.e.,*

$$\gamma_{i,j}^{col} = d_{mem} \times \min \left\{ \sum_{\forall s \in \mathrm{aff}(i,j)} (UCB_s(k_s) \times V_{s,j})\, ; ECB_j(k_{i,j}) \right\} \tag{6.5}$$

$$\textit{where} \quad V_{s,j} = 1 \quad \textit{if} \quad |ck_s \cap ck_j| > 0 \quad \textit{and} \quad V_{s,j} = 0, \ \textit{otherwise.}$$

Proof. We prove that both $\sum_{\forall s \in \mathrm{aff}(i,j)} (UCB_s(k_s) \times V_{s,j})$ and $ECB_j(k_{i,j})$ are upper bounds on the CRPD cost $\gamma_{i,j}^{col}$. Therefore, the minimum between the two is also an upper bound on $\gamma_{i,j}^{col}$.

(1). From the UCB-union approach (Equation (3.3)), it follows that when task τ_i is preempted by a higher priority task τ_j, the set of UCBs of all tasks in $\mathrm{aff}(i,j)$ may be evicted. However, when using cache coloring the actual set of UCBs of a task $\tau_s \in \mathrm{aff}(i,j)$ may not be known. Instead, we know the maximum number of UCBs of τ_s, i.e., $UCB_s(k_s)$, for a given cache color assignment ck_s with size k_s. Also due to cache coloring, τ_j can only evict UCBs of a task $\tau_s \in \mathrm{aff}(i,j)$ only when $|ck_s \cap ck_j| > 0$ (i.e., $V_{s,j} = 1$). Hence, the total number of UCBs among all tasks in $\mathrm{aff}(i,j)$ that can be evicted by τ_j is bounded by $\sum_{\forall s \in \mathrm{aff}(i,j)} (UCB_s(k_s) \times V_{s,j})$. Therefore, for a single preemption of τ_i by τ_j, $\sum_{\forall s \in \mathrm{aff}(i,j)} (UCB_s(k_s) \times V_{s,j})$ upper bounds the CRPD cost $\gamma_{i,j}^{col}$.

(2). The ECB-only approach (Busquets-Mataix et al., 1996; Tomiyama and Dutt, 2000) implies that the number of ECBs of the preempting task upper bounds the total CRPD cost that a task

[2] The maximum number of ECBs, UCBs and PCBs of a task for a given cache color assignment can be computed using any static timing analysis tool such as Heptane (Hardy et al., 2017).

may cause, i.e., for a single preemption of τ_i by τ_j the number of ECBs of τ_j also upper bounds the CRPD cost. However, due to cache coloring not all cache colors used by τ_j, i.e., k_j, may overlap with cache colors used by task τ_i (and by tasks in aff(i,j)) except for $k_{i,j}$ cache colors (i.e., Equation (6.4)).

Hence, the maximum number of ECBs of τ_j in the $k_{i,j}$ overlapping cache colors used by tasks in aff(i,j), i.e., $ECB_j(k_{i,j})$, upper bounds the CRPD cost $\gamma^{col}_{i,j}$ from τ_j's perspective.

The lemma follows. □

For a single preemption of τ_i by τ_j, the CRPD cost can be bounded using Lemma 6.1. However as we will now prove, the actual time taken to reload all UCBs of τ_i from the main memory is also bounded by the change in the worst-case memory access demand of task τ_i w.r.t the number of cache colors k_i assigned to τ_i.

To illustrate, let MD^{max}_i be the maximum worst-case memory access demand of τ_i when there is no cache assigned to τ_i (i.e., $k_i = 0$). Now, consider the example plot of main memory access demand $MD_i[k_i]$ of a task τ_i shown in Figure 6.3. The plot shows the normalized worst-case memory access demand of the `fdct` benchmark of the Mälardalen benchmark suite when the number of cache colors k_i assigned to that task varies. The values reported in Fig. 6.3 were obtained using the same cache configuration as in Figure 6.2. Figure 6.3 shows that for $k_i = 0$ the

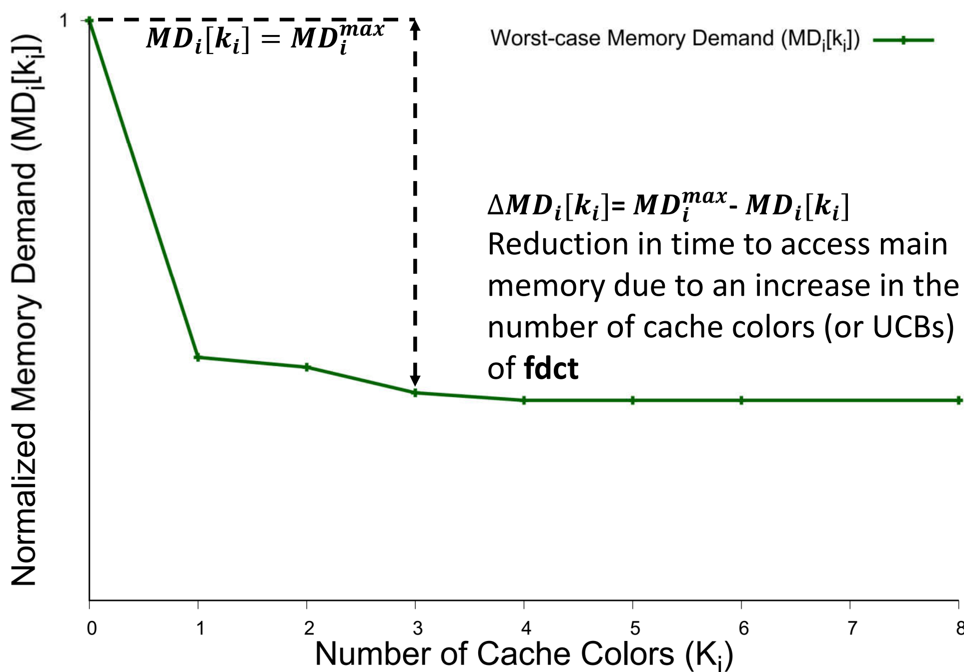


Figure 6.3: Worst-case memory access demand $MD_i[k_i]$ of task τ_i w.r.t the number of cache colors assigned to τ_i.

worst-case memory access demand of τ_i is maximum, i.e., $MD_i[k_i] = MD^{max}_i$. Also, for $k_i = 0$ since no cache space is assigned to τ_i there cannot be any useful cache blocks, i.e., $UCB_i = \emptyset$. Moreover, since $MD_i[k_i]$ is a non-increasing function w.r.t. the number of cache colors k_i, we observe that by increasing k_i, $MD_i[k_i]$ is decreasing.

This decrease in $MD_i[k_i]$ of task τ_i is due to an increase in its number of UCBs, i.e., by increasing the number of cache colors k_i (or cache space) assigned to τ_i, more instructions/data of τ_i may remain cached and therefore reused without having to reload them from main memory. This effectively increases the number of UCBs of τ_i, leading to a reduction in its worst-case memory access demand. The change in the worst-case memory access demand $MD_i[k_i]$ of τ_i due to an increase in the number of cache colors k_i assigned to τ_i can be bounded by $\Delta MD_i[k_i]$, where

$$\Delta MD_i[k_i] = MD_i^{max} - MD_i[k_i] \tag{6.6}$$

As the change in worst-case memory access demand of τ_i is due to an increase in the number of accesses to UCBs of τ_i. Formally,

$$UCB_i(k_i) \times N_i \times d_{mem} \leq \Delta MD_i[k_i] \tag{6.7}$$

where N_i is the average number of times each UCB of τ_i is accessed while it is cached.

Since $\Delta MD_i[k_i]$ bounds the time to reload all UCBs of τ_i for a given cache color assignment k_i, it also bounds the total CRPD τ_i can suffer due to eviction of its UCBs by tasks in $\text{hp}(i)$. However, we know from Lemma 6.1 that when task τ_i is preempted by a higher priority task $\tau_j \in \text{hp}(i)$, UCBs of all tasks in $\text{aff}(i,j)$ can be evicted. Therefore, to bound the total CRPD τ_i may suffer due to preemptions by a task $\tau_j \in \text{hp}(i)$ the change in the worst-case memory access demand of all tasks in $\text{aff}(i,j)$ should be considered.

Lemma 6.2. *The total CRPD cost suffered by a task τ_i due to preemptions by a higher priority task $\tau_j \in \text{hp}(i)$ is bounded by*

$$\forall j \in \text{hp}(i) : \gamma_{i,j}^{tot} \leq \sum_{\forall s \in \text{aff}(i,j)} \Delta MD_s[k_s] \tag{6.8}$$

Proof. Assuming tasks are assigned priorities in ascending order such that task τ_{i-1} has a higher priority than τ_i, we prove by induction that Equation (6.8) holds $\forall j \in \text{hp}(i)$.

Base Case: Consider τ_i and τ_{i-1} such that τ_{i-1} has a priority just above that of τ_i. Therefore, $\text{aff}(i,i-1) = \tau_i$.

The total CRPD that τ_i may suffer due to task τ_{i-1}, i.e., $\gamma_{i,i-1}^{tot}$, can never be larger than the time to reload all UCBs of τ_i N_i times from the main memory, i.e., the number of times UCBs of τ_i were accessed in cache when τ_i executes in isolation. Whereas, Equation (6.7) implies that that time is bounded by $\Delta MD_i[k_i]$. Hence, for $j = i-1$, $\gamma_{i,j}^{tot} \leq \Delta MD_i[k_i]$.

Induction step: Consider another task τ_s having a priority higher than τ_i and assume that Equation (6.8) holds for $j = s$, then Equation (6.8) also holds for $j = s-1$.

For $j = s-1$, $\text{aff}(i,s-1) = \{\tau_i, \ldots.\tau_s\}$, so using Equation (6.5) we know that when τ_{s-1} preempts task τ_i it may evict UCBs of all tasks in $\text{aff}(i,s-1)$, i.e., $\{UCB_i(k_i), \ldots.UCB_s(k_s)\}$. Also, by the same reasoning than above we know that the total CRPD every task in $\text{aff}(i,s-1) = \{\tau_i, \ldots.\tau_s\}$ may suffer due to task τ_{s-1} is bounded by $\{\Delta MD_i[k_i], \ldots.\Delta MD_s[k_s]\}$ respectively.

So, it follows that for $j = s-1$, the total CRPD τ_i may suffer due to τ_j is bounded such that $\gamma^{tot}_{i,j} \leq \sum_{\forall s \in \text{aff}(i,j)} \Delta MD_s[k_s]$.

Therefore, by induction Equation (6.8) holds for all $j \in \text{hp}(i)$. □

Since a higher priority task $\tau_j \in \text{hp}(i)$ can release $\left\lceil \frac{t}{T_j} \right\rceil$ jobs during a time window of length t and the CRPD caused by each of these jobs on τ_i can also be bounded using $\gamma^{col}_{i,j}$ (i.e., Equation (6.5)), therefore the total CRPD τ_i may suffer due to τ_j, i.e., $\gamma^{tot}_{i,j}$, during a time window of length t is bounded such that $\gamma^{tot}_{i,j} \leq \left\lceil \frac{t}{T_j} \right\rceil \times \gamma^{col}_{i,j}$.

Consequently, The total inter-task cache interference in terms of CRPD suffered by τ_i due to a higher priority task $\tau_j \in \text{hp}(i)$ in a time interval of length t is upper bounded by $CI^{inter,\gamma}_{i,j}(t)$, where

$$CI^{inter,\gamma}_{i,j}(t) = \min\left(\left\lceil \frac{t}{T_j} \right\rceil \times \gamma^{col}_{i,j};\ \sum_{\forall s \in \text{aff}(i,j)} \Delta MD_s[k_s] \right) \tag{6.9}$$

6.5.2 Inter-Task Cache Interference due to CPROs

In Chapter 4 and Chapter 6, we presented different approaches to compute the CPRO of tasks. However, in this chapter we will focus on a CPRO-union (i.e., Equation (4.5)) alike approach to bound CPRO under the proposed cache coloring approach. To calculate the CPRO of a task $\tau_j \in \text{hp}(i)$ executing during the response time of τ_i, the CPRO-union approach uses the set of PCBs of task τ_j and the set of ECBs of all tasks in $\text{hep}(i) \setminus \tau_j$ (see Equation (4.5)). However, as already discussed for the CRPD calculation (Section 6.5.1), it is not possible to directly use the CPRO-union approach (i.e., Equation (4.5)) under the cache coloring technique considered in this chapter, mainly because the actual set, i.e., their accurate placement in cache, of PCBs and ECBs may not be known. Therefore, to bound the CPRO of a task τ_j (executing during the response time of τ_i) under our cache coloring approach, we use a similar technique to the one used in Section 6.5.1. We first bound the worst-case number of cache colors that may be evicted between two subsequent jobs of τ_j.

Assuming τ_i and τ_j are assigned a set of ck_i and ck_j cache colors respectively, then the maximum number of cache colors of τ_j that can be evicted between its successive jobs due to the executions of all tasks in $\text{hep}(i) \setminus \tau_j$ during the response time of τ_i can be bounded by $k^{'}_{j,i}$ calculated as follows.

$$k^{'}_{j,i} = \left| ck_j \cap \Big(\bigcup_{\forall \tau_s \in \text{hep}(i) \setminus \tau_j} ck_s \Big) \right| \tag{6.10}$$

where $k^{'}_{j,i}$ bounds the number of cache colors that can be affected by evictions between two successive jobs of τ_j. Therefore, the product $k^{'}_{j,i} \times k^{size}$ bounds the maximum number of cache sets that can be evicted between two successive jobs of τ_j. However, that is obviously pessimistic and to have a tighter bound on the CPRO in terms of the number of PCBs of τ_j that may be evicted between its two successive jobs, we define $PCB_i(k_i)$, i.e.,

- $PCB_i(k_i)$: The maximum number of PCBs of task τ_i when it is assigned k_i cache colors.

$PCB_i(k_i)$ can also be computed in a similar manner to $ECB_i(k_i)$ and $UCB_i(k_i)$ as detailed in Section 6.5.1. Furthermore, $PCB_j(k_j)$ and $ECB_i(k_i)$ can be used to bound the CPRO of a task $\tau_j \in \text{hp}(i)$ executing during the response time of τ_i using the following lemma

Lemma 6.3. *$\rho^{col}_{j,i}$ bounds the CPRO or the maximum number of PCBs of task τ_j that may be evicted between two successive jobs of τ_j due to eviction of $k^{'}_{j,i}$ cache colors by tasks in $\text{hep}(i) \setminus \tau_j$, where*

$$\rho^{col}_{j,i} = d_{mem} \times \min\left\{ PCB_j(k_j); \sum_{\forall \tau_s \in \text{hep}(i) \setminus \tau_j} \left(ECB_s(k^{'}_{j,i}) \times V_{s,j} \right) \right\} \tag{6.11}$$

$$\text{where} \quad V_{s,j} = 1 \quad \text{if} \quad |ck_s \cap ck_j| > 0 \quad \text{and} \quad V_{s,j} = 0, \text{ otherwise.}$$

Proof. We prove that both $PCB_j(k_j)$ and $\sum_{\forall \tau_s \in \text{hep}(i) \setminus \tau_j} (ECB_s(k^{'}_{j,i}) \times V_{s,j})$ are upper bounds on the CPRO cost $\rho^{col}_{j,i}$. Therefore, a minimum between the two bounds is also an upper bound on $\rho^{col}_{j,i}$.
(1). By definition of PCBs, the CPRO of a task is upper bounded by its number of PCBs. Hence, assuming τ_j is assigned k_j cache colors, the maximum number of PCBs of τ_j are given by $PCB_j(k_j)$. Therefore, the maximum CPRO one job of τ_j can suffer during the response time of τ_i is upper bounded by $PCB_j(k_j)$.
(2). The worst-case memory interference of any task $\tau_s \in \text{hep}(i) \setminus \tau_j$ on τ_j is when it loads all its ECBs between two subsequent jobs of τ_j. With cache coloring, if k_s cache colors are assigned to a task $\tau_s \in \text{hep}(i) \setminus \tau_j$, the maximum number of ECBs of τ_s that can be loaded between two jobs of τ_j are bounded by $ECB_s(k_s)$.
(3). However, τ_s can only evict PCBs of task τ_j only if $|ck_s \cap ck_j| > 0$ (i.e., $V_{s,j} = 1$) and not all cache colors used by τ_s (i.e., ck_s) may overlap with cache color used by τ_j except for $k^{'}_{j,i}$ cache colors (see Equation (6.10)). Effectively, $ECB_s(k^{'}_{j,i}) \times V_{s,j}$ bounds the number of ECBs of τ_s that may overlap and potentially evict PCBs of τ_j.

Since all $\tau_s \in \text{hep}(i) \setminus \tau_j$ may execute between two successive jobs of τ_j potentially evicting its PCBs. The worst-case memory interference of all $\tau_s \in \text{hep}(i) \setminus \tau_j$ on PCBs of τ_j is bounded by $\sum_{\forall \tau_s \in \text{hep}(i) \setminus \tau_j} ECB_s(k^{'}_{j,i} \times V_{s,j})$. So the lemma follows. □

The CPRO (i.e., $\rho^{col}_{j,i}$) suffered by a single job of a higher priority task $\tau_j \in \text{hep}(i)$ executing during the response time of τ_i can be bounded using Lemma 6.3. However, since the CPRO accounts for the extra memory accesses of a task τ_j due to eviction of its PCBs, it may also depend on the memory access demand of τ_j given that τ_j is assigned k_j cache colors.

To further illustrate this point, consider the example plot (i.e., Figure 6.4) of a task τ_j representing the same (i.e., `fdct`) benchmark from the Mälardalen benchmark suite with the same cache configuration as used in Figure 6.3. The plot in Figure 6.4 shows two types of memory access demands of task τ_j with respect to the number of cache colors (i.e., k_j) assigned to τ_j, i.e., the worst-case memory access demand ($MD_j[k_j]$) and the residual memory access demand ($MD^r_j[k_j]$). From Figure 6.4, we observe that when the number of cache colors assigned to τ_j are less than or equal to 3 , the worst-case memory access demand $MD_j[k_j]$ of τ_j is equal to its residual memory

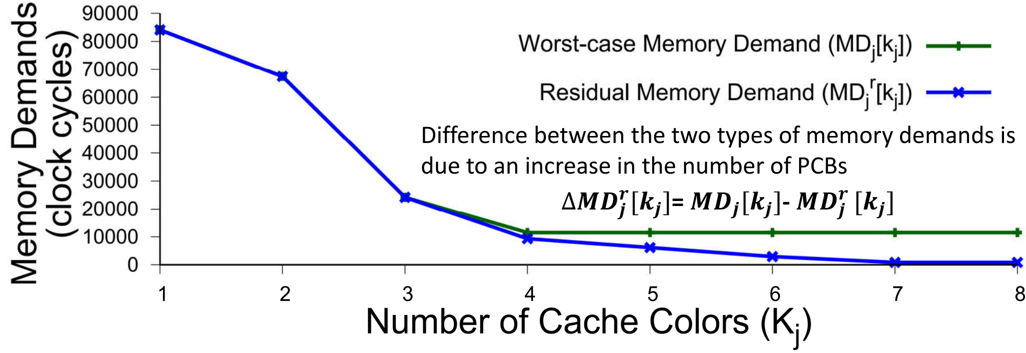


Figure 6.4: Variation in the worst-case and residual memory access demand of task τ_j w.r.t the number of cache colors assigned.

access demand $MD^r_j[k_j]$, showing that for $k_j \leq 3$, τ_j has no PCBs. However, by further increasing the cache colors assigned to τ_j (i.e., for $k_j > 3$), we can see an increasing difference between the worst-case memory access demand $MD_j[k_j]$ and the residual memory access demand $MD^r_j[k_j]$ of τ_j. This difference is due to an increase in the number of PCBs of τ_j and is denoted by $\Delta MD^r_j[k_j]$, where

$$\Delta MD^r_j[k_j] = MD_j[k_j] - MD^r_j[k_j] \tag{6.12}$$

$\Delta MD^r_j[k_j]$ corresponds to the reduction in time to access main memory due to an increase in the number of PCBs of τ_j. Therefore, $\Delta MD^r_j[k_j]$ effectively bounds the number of PCBs of τ_j given that τ_j is assigned k_j cache colors, i.e.,

$$PCB_j(k_j) \times N'_j \times d_{mem} \leq \Delta MD^r_j[k_j] \tag{6.13}$$

where N'_j is the average number of times each PCB of τ_j is accessed. Since $\Delta MD^r_j[k_j]$ bounds the number of PCBs of task τ_j, it also bounds the CPRO suffered by τ_j when it executes during the response time of a lower priority task τ_i.

Lemma 6.4. *The CPRO due to the eviction of PCBs of a job of task $\tau_j \in \text{hp}(i)$ executing during the response time of a task τ_i, i.e., $\rho^{one}_{j,i}$, is upper bounded by the difference between the worst-case and the residual memory access demand of τ_j, i.e.,*

$$\rho^{one}_{j,i} \leq \Delta MD^r_j[k_j] \tag{6.14}$$

Proof.
(1). Equation (6.11) implies that the CPRO of one job of task $\tau_j \in \text{hp}(i)$ executing during the response time of a task τ_i is upper bounded by the time to reload all PCBs of τ_j from main memory given a cache color assignment k_j, i.e., $\rho^{one}_{j,i} \leq PCB_j(k_j) \times d_{mem}$.
(2). Also, from Equation (6.13) it follows that the time to reload all PCBs of τ_j for a given cache color assignment k_j is bounded by the difference between the worst-case and the residual memory access demand of τ_j, i.e., $PCB_j(k_j) \times N'_j \times d_{mem} \leq \Delta MD^r_j[k_j]$.

(3). By definition of PCBs, $N_j^{'} \geq 1$. So the lemma follows. □

Lemma 6.4 can be used to bound the CPRO of one job of task $\tau_j \in \text{hep}(i)$ executing during the response time of a task τ_i. However, we know that task τ_j may execute several times during the execution of τ_i therefore, the total inter-task cache interference in terms of CPRO suffered by τ_j while executing during the response time of τ_i can be bounded using the following theorem

Theorem 6.1. *The total inter-task cache interference in terms of CPRO suffered by a higher priority task $\tau_j \in \text{hp}(i)$ due to evictions of its PCBs by tasks in $\text{hep}(i) \setminus \tau_j$ in a time interval of length t is bounded by $CI_{j,i}^{inter,\rho}$, where*

$$CI_{j,i}^{inter,\rho}(t) = \left(\left\lceil \frac{t}{T_j} \right\rceil - 1 \right) \times \min\left(\rho_{j,i}^{col} ; \Delta MD_j^r[k_j] \right) \tag{6.15}$$

Proof.
(1). It is proved in Lemma 4.2 that in a time interval of length t at most $\left(\left\lceil \frac{t}{T_j} \right\rceil - 1 \right)$ jobs of task τ_j can suffer CPRO.
(2). It implies that both $\rho_{j,i}^{col}$ (Equation (6.11)) and $\Delta MD_j^r[k_j]$ (by Lemma 6.4 and Equation (6.12)) upper bound the CPRO suffered by one job of τ_j executing during the response time of τ_i. Therefore, the minimum between the two bounds is also an upper bound on the CPRO suffered by a single job of τ_j during the response time of τ_i.

The theorem directly follows from the two points above. □

6.6 Optimizing Cache Color Assignment

In this section, we detail how we optimize the cache color assignment of tasks to balance the intra- and inter-task cache interference such that it results in improving task set schedulability. We have used a *Simulated Annealing* (SA) approach to optimize cache color assignment of tasks. Simulated annealing (Kirkpatrick et al., 1983) is a meta-heuristic that allows to find a near optimal solution to an optimization problem in a reasonable computational time. It starts with a randomized state and in a polling loop moves to neighboring states always accepting the moves that improve the value of the objective function. However, the SA may also accept bad moves (i.e., that does not improve upon the objective function) according to a probability distribution dependent on the“temperature”. This is different from other local search based techniques that only accept solutions that are better than the initial solution and may get stuck into a local optimum rather than global optima. Our SA-based cache coloring approach is given by Algorithm 6.1.

When allocating cache colors to tasks, Algorithm 6.1 starts by assigning sequential cache colors to all n tasks in a given task set Γ. Cache colors are assigned to tasks in priority order with the highest priority task first. Once the sequential cache color assignment is done, the algorithm checks the schedulability of each task in Γ. If all tasks in task set Γ are schedulable with the sequential cache color allocation (i.e., Γ is schedulable), no changes are made to the cache color assignment of tasks and the algorithm returns true and exit. However, if Γ was not schedulable

Algorithm 6.1 Simulated annealing based algorithm to optimize cache color assignment of tasks

Require: task set $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$; total cache colors k^{total}
Ensure: Cache color assignment $\{ck_1, ck_2, ..., ck_n\}$; $true$ if Γ is schedulable and $false$ otherwise.

```
 1: for i ← 1 to n do
 2:     {ck_i} = ∅
 3: end for
 4: AssignSequentialColors(Γ,k^total);
 5: if isSchedulable(Γ) then
 6:     return true;
 7: else
 8:     SimulatedAnnealing(Γ);
 9:     if isSchedulable(Γ) then
10:         return true;
11:     else
12:         return false;
13:     end if
14: end if
15: Function SimulatedAnnealing(Γ)
16: CurrentTemp←400;DesiredTemp←0.001; CoolingRate←0.99;
17: while CurrentTemp ≥ DesiredTemp do
18:     TaksetSlackOld = CalculateTasksetSlack(Γ);
19:     SelectRandom(ReAllocate(),ShiftLayout(),ReSize());
20:     TaksetSlackNew = CalculateTasksetSlack(Γ);
21:     ΔSlack←TaksetSlackOld − TaksetSlackNew
22:     if ΔSlack ≥ 0 then
23:         Accept new cache color assignment for Γ;
24:     else
25:         Randomprob ← rand(0,1)
26:         if Randomprob < e^(−ΔSlack/CurrentTemp) then
27:             Accept new cache color assignment for Γ;
28:         else
29:             Discard new cache color assignment of Γ;
30:         end if
31:     end if
32:     CurrentTemp = CurrentTemp ∗ CoolingRate;
33: end while
34: end function
```

with the sequential cache color allocation, cache color assignment of tasks is optimized using SA. The SA algorithm uses the sequential cache color assignment of tasks as the initial solution and then iteratively tries to improve it by randomly performing one of the following operations:

- **Re-allocate():** Swap the set of cache colors assigned to two distinct tasks. Namely two operations can be performed, (1) *swap-neighbors()*: swapping the set of cache colors assigned to two neighboring tasks. This swap is based on the order of tasks in the main memory rather than their priorities ;(2) *swap-random()*: swap the set of cache colors of two randomly chosen tasks. These tasks may or may not be adjacent in main memory. If the chosen tasks are not adjacent in memory, cache color assignment of tasks in between them is also updated.

- **Shift-layout():** Increasing/decreasing the starting offset of a randomly chosen task in the

main memory (i.e., shifting tasks right or left). To avoid creating gaps between tasks in main memory we essentially left/right shift all tasks in the main memory.

- **Re-size():** Randomly choose a task and re-allocate the number of cache colors assigned to that task, i.e., either by increasing or decreasing the number of cache colors assigned to that task.

  As we later show in Section 6.6.1, re-sizing the cache space assigned to tasks can be very beneficial especially when the tasks have large cache footprints. Also, increasing/decreasing the number of cache colors assigned to tasks effectively allows to trade between the intra- and inter-task cache interference which may result in improving task set schedulability.

To evaluate different cache color assignments, the WCRT analysis (i.e., Equation (6.2)) can be used at every iteration of the SA algorithm, i.e., checking the schedulability of all tasks in Γ after performing any of the above mentioned operations. However, this may be computationally expensive. Also, the boolean result given by Equation (6.2) can only distinguish between schedulable/unschedulable cache color assignments and does not provide any information about the impact of different cache color assignments on the intra- and inter-task cache interference suffered by the tasks. Therefore, to better quantify the quality of a cache color assignment of tasks and to guide the SA algorithm towards an optimal solution, we use the notion of *slack*. Slack Sl of a task τ_i is denoted by Sl_i and is defined as *"the difference between the relative deadline and the WCRT of τ_i"*, i.e., $Sl_i = D_i - R_i$, where R_i is calculated by considering the worst-case interference on τ_i by all higher priority tasks in hp(i), i.e., by setting $R_i == D_i$ in Equation (6.2). The total slack Sl^{tot} of task set Γ is given as

$$Sl^{tot} = \sum_{i=1}^{n} w_i \times S_i \tag{6.16}$$

where w_i is the weight assigned to every $\tau_i \in \Gamma$ such that,

$$w_i = 0 \quad \text{if} \quad Sl_i \geq 0 \quad \text{and} \quad w_i = 1, \quad \text{otherwise.}$$

Note that only the tasks with a negative slack will be assigned a non-zero weight, i.e., $w_i = 1$. This is mainly because these are the tasks that were not schedulable for a given cache color assignment but may become schedulable by changing their cache color assignment. The total task set slack is calculated after randomly performing any of the above mentioned moves during every iteration of the SA algorithm. If the change in the total task set slack from the last iteration is positive then the new cache color assignment of tasks will always be accepted. However, even if the change in task set slack is negative the new cache color assignment of tasks may still be accepted depending on how negative the change is and the current temperature of the SA algorithm, i.e., if a randomly chosen probability between 0 and 1 is less than the probability of accepting the negative change, i.e., $e^{\frac{-\Delta Slack}{CurrentTemp}}$ (see Algorithm 6.1), then the new cache color assignment for Γ will be accepted. Otherwise, the new cache color assignment will be discarded. After every iteration, the temperature of SA is reduced by multiplying it with a cooling factor until it reaches

the desired temperature. The initial temperature, desired temperature and the cooling factor defines the maximum number of iterations for the SA algorithm. In general, when the temperature is high, the SA algorithm is more open to negative changes that may be useful to escape local minima.

6.6.1 Working Example

To evaluate the effectiveness of the SA-based cache color assignment approach detailed in the previous section, we performed a small experiment using a single task set comprised of 10 tasks from the Mälardalen benchmark suite (Gustafsson et al., 2010) shown in Table 6.2, i.e., $\tau_1 = minmax$ to $\tau_{10} = bsort100$, where τ_1 has the highest priority. The selection of tasks was purely random and although these tasks may not represent a real task set, they do represent typical code found in real-time systems. For each task, the WCET $C_i[k_i]$, worst-case memory access demand $MD_i[k_i]$, worst-case processing demand PD_i and the number of ECBs (i.e., $ECB(k_i)$), UCBs (i.e., $UCB(k_i)$) and PCBs (i.e., $PCB(k_i)$) were extracted using the Heptane static WCET analysis tool (Hardy et al., 2017) as presented in Chapter 4 and 5. Note that the values for $C_i[k_i]$, $MD_i[k_i]$ and PD_i in Table 6.2 are in clock cycles. The number of cache colors used by each task, i.e., k_i, were set such that $k_i = \left\lceil \frac{ECB_i(k_i)}{k^{size}} \right\rceil$. The target architecture was MIPS R2000/R3000 assuming an instruction cache with line size of 32 Bytes and the total cache size of 16kB such that the cache has a total of 32 cache colors, i.e., $k^{total} = 32$ with each color having a size of 512 Bytes, i.e., $k^{size} = 512$ Bytes. The block reload time d_{mem} was set to 10μs.

Table 6.2: Task set parameters used in the working example

Name	$C_i[k_i]$	PD_i	$MD_i[k_i]$	k_i	T_i
minmax	2522	122	2400	2	14315
lcdnum	3440	984	2740	2	73143
cnt	10090	7191	3818	2	85816
ns	30149	28149	6172	2	169744
statemate	43344	10586	35257	18	636613
insertsort	7574	5974	2343	1	734873
nsichneu	316409	22009	294400	32	1889824
qurt	26141	9241	17713	5	2899034
fft	157880	123681	45816	9	6550339
bsort100	712289	710289	90893	2	267271122

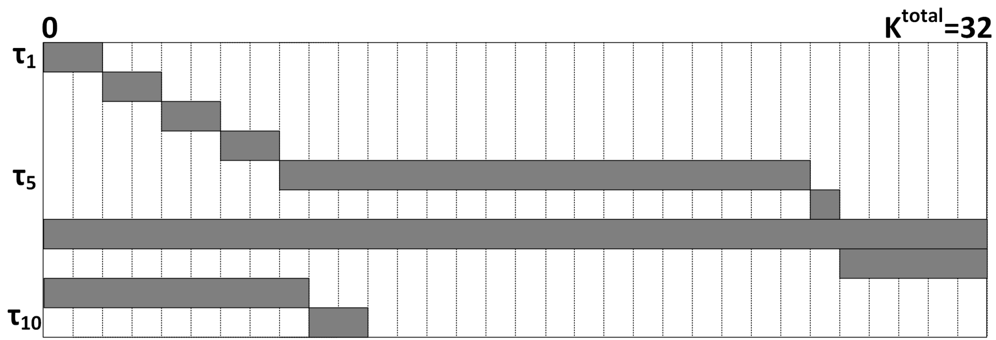
The task set was created by fixing the core utilization at 0.8, with task utilizations generated using the UUnifast algorithm (Bini and Buttazzo, 2005). Task periods were set such that $T_i = C_i[k_i]/U_i$. All tasks had implicit deadlines with priorities assigned in deadline monotonic order. We checked task set schedulability using the following approaches:

- **No preemption cost:** The WCRT analysis was performed assuming there is no preemption cost.

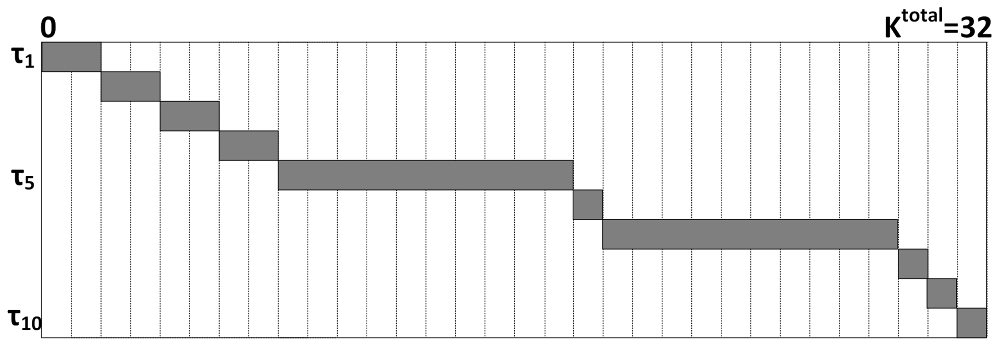
- **SA-based cache color assignment:** Cache color assignment of tasks was optimized using the SA algorithm detailed in Section 6.6.
- **SA-based cache color assignment without re-sizing:** The SA algorithm was used to optimize cache color assignment of tasks however, re-size() operation was not permitted.
- **Sequential cache color assignment:** Tasks were assigned cache colors in a sequential manner with the highest priority task first.
- **Full cache partitioning:** The cache partitioning algorithm presented in (Altmeyer et al., 2014, 2016) was used to assign independent non-overlapping cache colors (i.e., partitions) to all tasks.
- **SA-based cache color assignment without cache persistence** The SA algorithm was used to optimize cache color assignment of tasks without considering cache persistence.

We observed that the task set was schedulable only with two approaches, i.e., no preemption cost and the SA algorithm with re-sizing. All other approaches were not able to schedule the task set. The final cache color allocations for the sequential cache color assignment, full cache partitioning and the SA algorithm with re-sizing, are shown in Figure 6.5a, 6.5b and 6.5c respectively.

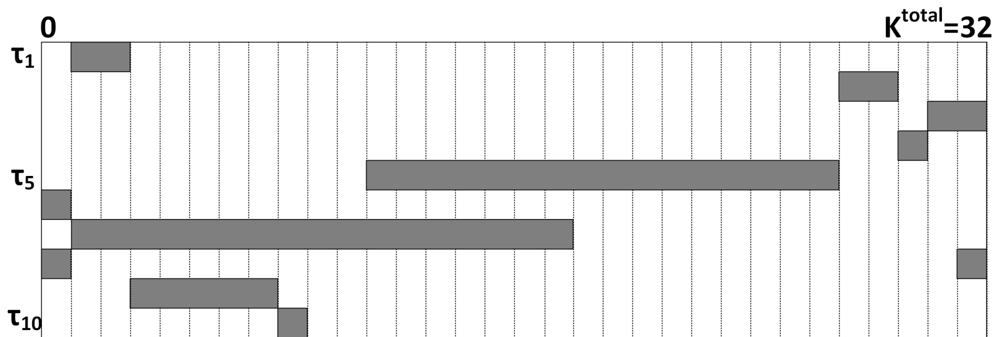
The sequential cache color assignment of tasks (see Figure 6.5a) was subjected to high inter-task cache interference (i.e., CRPD and CPRO), mainly because most cache colors were shared among tasks. This results in making the task set unschedulable. On the contrary, with full cache partitioning (see Figure 6.5b) there is no inter-task cache interference. However, the task set was still not schedulable due to an increase in the intra-task cache interference of some tasks that were assigned fewer cache colors than the actual number of cache colors needed by those tasks. The layout of tasks in cache resulting from the SA algorithm with re-sizing is shown in Figure 6.5c. The task set was schedulable mainly because the overall cache interference between tasks was reduced by trading between intra- and inter-task cache interference, e.g., the inter-task cache interference caused by τ_7 on all lower priority tasks (i.e., τ_8, τ_9 and τ_{10}) was reduced by increasing the intra-task cache interference of τ_7 (i.e., by reducing the number of cache colors used by τ_7). Note that the task set was also not schedulable using the SA algorithm without re-sizing. This shows that even with an optimized task layout, allowing tasks an unconstrained use of the cache may still result in higher inter-task cache interference that can make the task set unschedulable.



(a) Sequential cache color assignment.



(b) Full cache partitioning.



(c) SA-based cache color assignment with re-sizing.

Figure 6.5: Different cache color assignments of task set in Table 6.2.

6.7 Experimental Evaluation

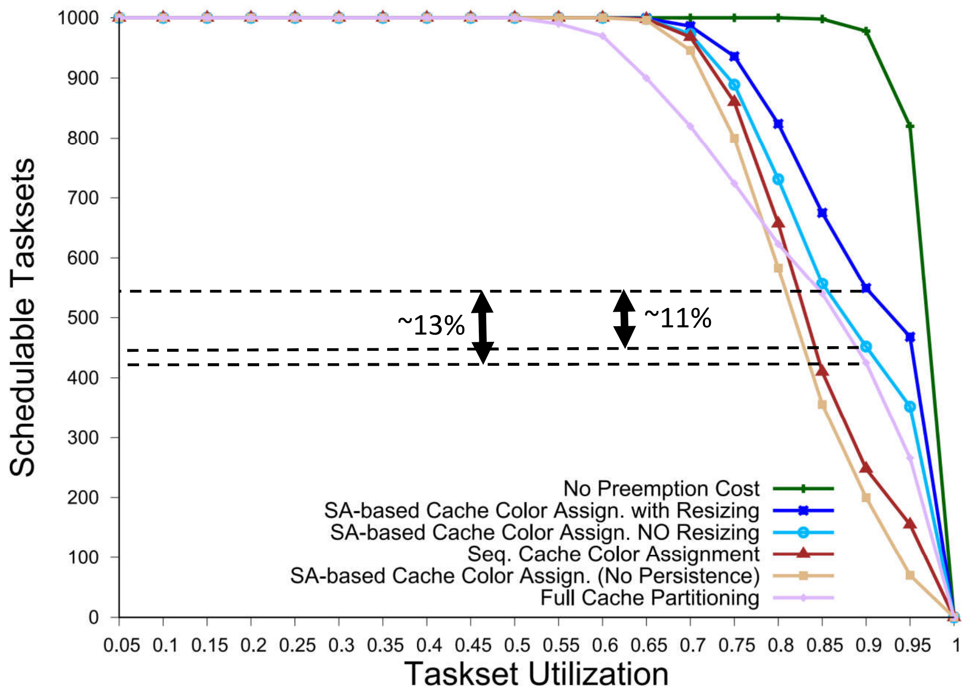
In this section, we evaluate how our proposed SA-based cache coloring approach performs in terms of schedulability in comparison to few existing task layout optimization techniques. Experiments were performed using the Mälardalen benchmark suite with parameters $C_i[k_i]$, PD_i, $MD_i[k_i]$, $MD^r_i[k_i]$, $UCB_i(k_i)$, $ECB_i(k_i)$ and $PCB_i(k_i)$ extracted using Heptane for the same cache configuration as used in Section 6.6.1. The number of cache colors used by each task, i.e., k_i, were set such that $k_i = \left\lceil \frac{ECB_i(k_i)}{k^{size}} \right\rceil$. Each task was randomly assigned the values $C_i[k_i]$, PD_i, $MD_i[k_i]$, $MD^r_i[k_i]$, $UCB_i(k_i)$, $ECB_i(k_i)$, $PCB_i(k_i)$ and k_i of one of the analyzed benchmarks. Other task set

parameters were randomly generated as follows. The default number of tasks was 10 with task utilizations generated using UUnifast (Bini and Buttazzo, 2005). Task periods were set such that $T_i = C_i[k_i]/U_i$. Task deadlines were implicit and priorities were assigned in deadline monotonic order.

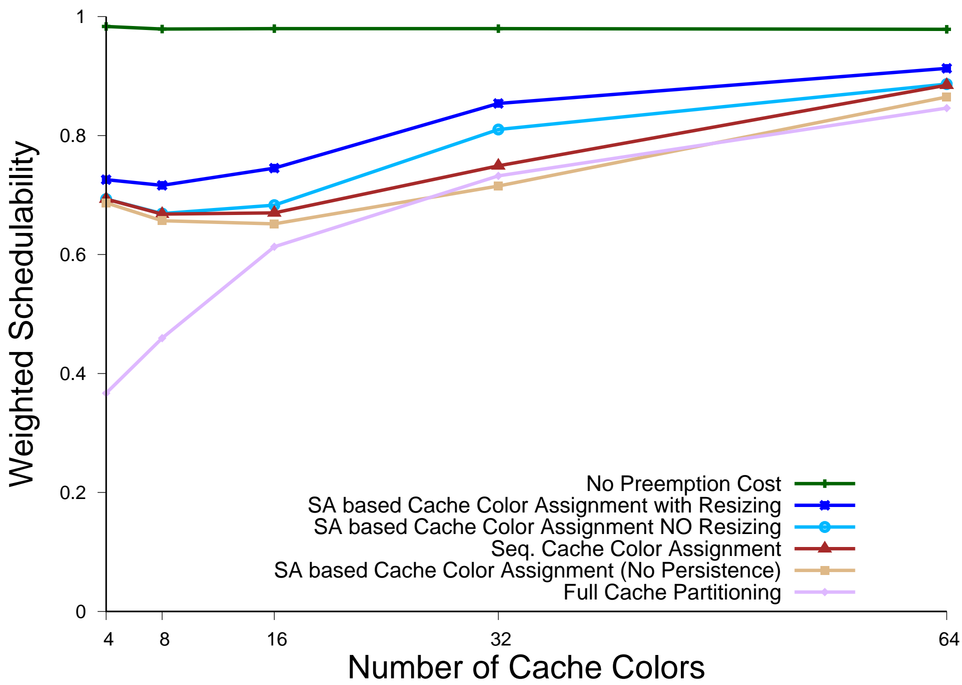
We conducted different experiments by varying core utilization, number of cache colors, size of cache colors and number of tasks. Schedulability analysis was performed using the same task sets for all the approaches detailed in Section 6.6.1 using their respective WCRT analysis.

1) Core Utilization: In this experiment, we randomly generated 1000 task set (each comprised of 10 tasks) at different core utilizations varied from 0.05 to 1 in steps of 0.05. Figure 6.6a shows the average number of task sets that were deemed schedulable using all the analyzed approaches against the total core utilization. The green line marked as “No preemption cost" is an upper bound on the maximum number of task sets that were schedulable without considering any CRPD/CPRO. Figure 6.6a shows that the proposed SA-based cache color assignment with/without re-sizing was able to schedule more task sets than all the other approaches. Also, we note that while initially the full cache partitioning approach performs worst however, at higher core utilizations it tends to outperform the sequential cache color assignment and the SA-based cache color assignment (no persistence) approach. This is mainly because at higher core utilizations, task periods become smaller resulting in higher inter-task cache interference. It is due to higher inter-task cache interference that at core utilizations of 0.85 and 0.9 the difference between the full cache partitioning approach and the SA-based cache color assignment without re-sizing is minimal. However, the SA-based cache color assignment with re-sizing counters this increase in inter-task cache interference by trading intra-task cache interference, i.e., increasing intra-task cache interference to reduce inter-task cache interference. Consequently, this results in improving task set schedulability even at higher core utilizations. For example at a utilization of 0.9, the SA-based cache color assignment with re-sizing was able to schedule up to 11 percentage points more task sets than the SA-based cache color assignment without re-sizing and up to 13 percentage points more task sets than the full cache partitioning approach.

2) Number of Cache Colors: In this experiment, we evaluate the impact of cache size on the performance of the analyzed approaches by varying the number of cache color from 4 to 64. As the size of cache colors is constant (i.e., 512 B), increasing the number of cache colors also increases the cache size. All parameters other than the number of cache colors have the same values as used in the previous experiment. We have used the weighted schedulability measure (see Equation (4.20)) defined by Bastoni et al. (Bastoni et al., 2010) to plot the results as shown in Figure 6.6b. We observe that initially increasing the number of cache colors (i.e., from 4 to 8) decreases the schedulability of all approaches except the full cache partitioning approach. This is mainly because in this interval most cache colors were shared between tasks resulting in higher inter-task cache interference. However, even in this interval the SA-based cache color assignment with re-sizing outperforms all other approaches. A further increase in the number of cache colors results in reducing the number of cache colors that are shared among tasks. Therefore, we see an increase in the schedulability of all approaches. Understandably, the performance of the full



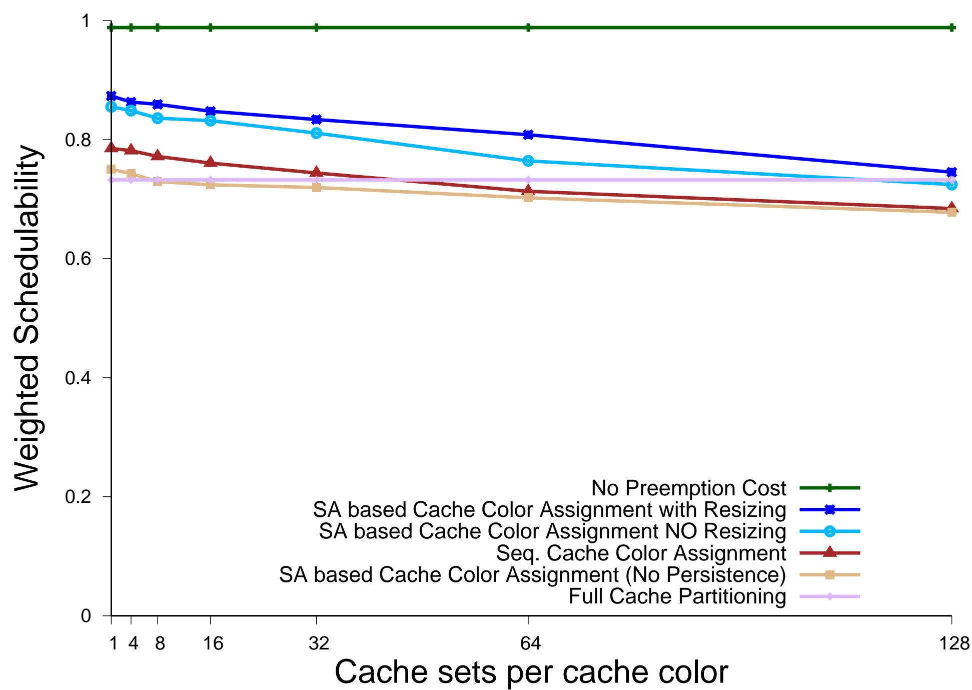
(a) Varying core utilizations



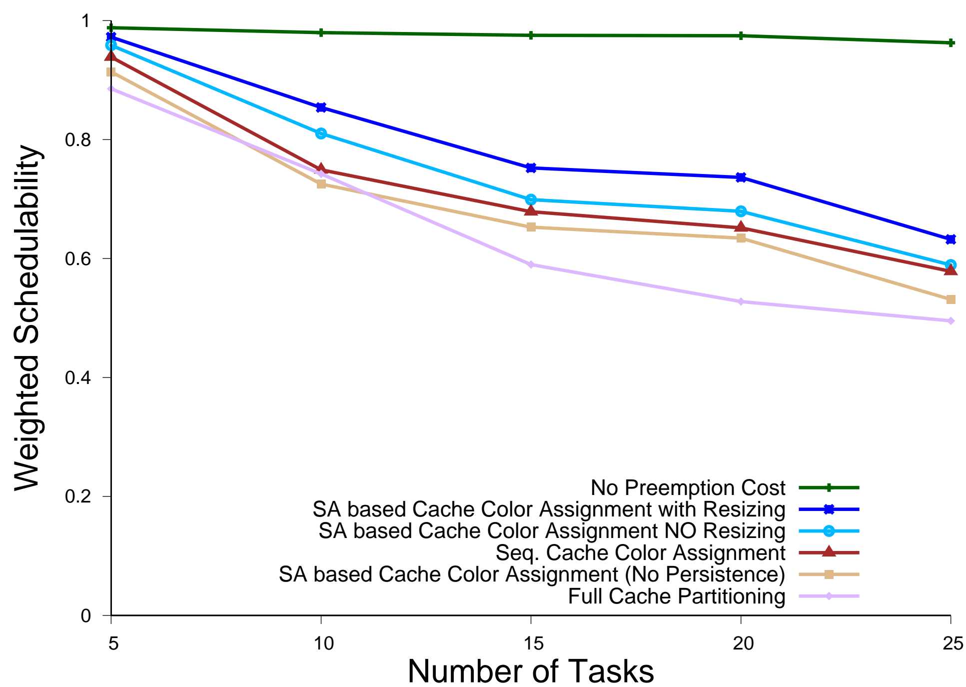
(b) Varying the number of cache colors (or cache size)

Figure 6.6: Schedulability w.r.t core utilization and cache size

cache partitioning approach is almost linear w.r.t the number of cache colors. Moreover, when the number of cache colors is large (e.g., 64) all approaches have similar results due to lower cache interference.



(a) Increasing number of cache sets per cache color



(b) Varying the number of tasks

Figure 6.7: Schedulability w.r.t number of cache sets per color and number of tasks

3) Number of cache sets per cache color: We also performed an experiment by increasing the number of cache sets per cache color whilst keeping the cache size constant. We varied the size of one cache color between 1 to 128 cache sets with all other parameters set to default values. The resulting plot of weighted task set schedulability w.r.t the number of cache sets per cache color is

shown in Figure 6.7a. We observe that when the size of a cache color is smaller all approaches were able to schedule more task sets. This is mainly because a smaller cache color size results in a tighter bound on the CRPD/CPRO suffered by the tasks. Whereas, increasing the size of a cache color decrease the total number of cache colors, potentially increasing the number of shared cache colors and the CRPD/CPRO suffered by the tasks. This results in decreasing task set schedulability for all approaches. Note that since the full cache partitioning approach uses the number of cache sets rather than cache colors, its performance is not affected by the size of a cache color.

4) Number of Tasks: To analyze the performance of all approaches w.r.t the number of tasks, we varied the number of tasks from 5 to 25 with all other parameters set to the same values as used in the core utilization experiment. Figure 6.7b shows the result of the experiment. We observe that schedulability for all approaches decreases as the number of tasks is increased. For the full cache partitioning approach this decrease in schedulability is due to an increase in intra-task cache interference, i.e., as the number of tasks increase, less cache colors can be assigned to each individual task potentially resulting in increasing its intra-task cache interference. For the other approaches, the reduction in schedulability is due an increase in inter-task cache interference due to sharing of cache colors between several tasks. However, we observe that the SA-based cache color assignment with re-sizing still achieves much higher schedulability than all the other approaches.

6.8 Chapter Summary

In this chapter, we evaluated the impact of memory layout of tasks on schedulability. We showed that intra- and inter-task cache interference can be interrelated and balancing their respective contribution to tasks WCRT may result in improving task set schedulability. We use a cache coloring approach to optimize task layout in memory such that the trade-off between intra- and inter-task cache interference can be balanced. We also showed how the intra- and inter-task cache interference can be bounded under a cache coloring approach. Lastly, a simulated annealing algorithm is proposed to optimize the cache color assignment to tasks by re-allocating and re-sizing the cache colors assigned to tasks such that the task set's schedulability is achieved. Experiments were performed by varying different parameters using values from the Mälardalen benchmarks. Experimental results show that the proposed SA based cache color assignment of tasks dominates the existing approaches used to optimize task layout in memory.

Part II

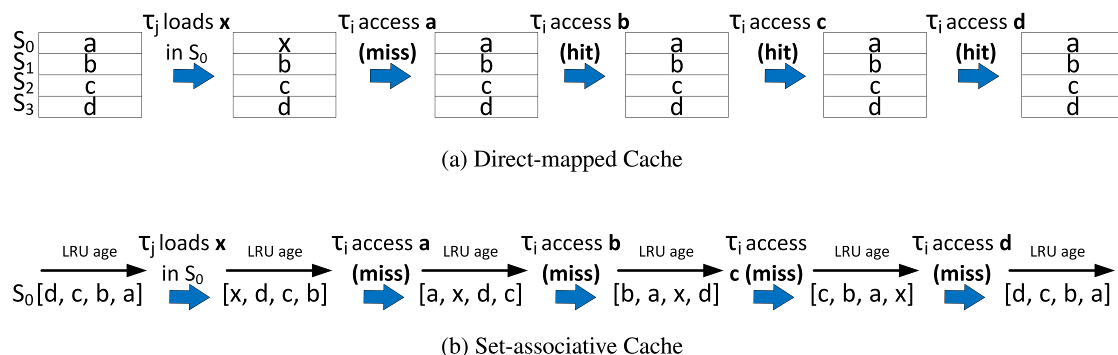
Analysis of Single- and Multi-level Set-associative Caches

Chapter 7

CPRO Analysis for Set-associative Caches

We have seen in the previous chapters that to accurately quantify the inter-task cache interference suffered by the tasks it is essential to consider both CRPDs and cache persistence. We also showed that for tasks scheduled under fixed-priority preemptive scheduling (FPPS), the worst-case response time (WCRT) analyses that account for cache persistence between jobs along with cache related preemption delays (CRPDs) dominates the analyses that only consider CRPDs. But, the approaches presented in Chapter 4 and Chapter 5 to analyze cache persistence (and compute CPROs) in the context of WCRT analysis can only support direct-mapped caches and may not work for processor architectures based on *set-associative* caches. This is mainly because in a direct-mapped cache, each cache set can hold at most one memory block whereas, in a set-associative cache, each cache set may hold as many memory blocks as there are cache *ways* (or the cache *associativity*). Therefore, in case of a cache conflict between two tasks τ_i and τ_j, each memory access performed during τ_j's execution may evict at most one PCB of τ_i in a direct-mapped cache, while it may lead to multiple evictions in a set-associative cache. This is known as the *cascading* effect. An example of such cascading effect is shown in Figure 7.1. We can see that for a direct-mapped cache (see Figure 7.1a) a single cache conflict between τ_i and τ_j (i.e., due to preemption of τ_i by τ_j) can only cause one cache miss whereas the same cache conflict leads to multiple cache misses for a set-associative cache (see Figure 7.1b). Therefore, for set-associative caches, a *sound* CPRO estimate can only be obtained by accounting for the cascading effect which is not considered by the CPRO analysis presented in the previous chapters that focus on direct-mapped caches. Therefore, in this work, we present different solutions to analyze cache persistence for set-associative caches and integrate those solutions in the WCRT analysis of FPPS. The main contributions of this chapter are as follow:

1. to propose a solution to find persistent cache blocks (PCBs) of tasks considering set-associative caches;
2. to present three different approaches to calculate CPROs on platforms implementing set



(a) Direct-mapped Cache

(b) Set-associative Cache

Figure 7.1: Example execution of a task τ_i (from left to right) considering (a) a direct-mapped cache with 4 cache sets, i.e., $\{S_0, S_1, S_2, S_3\}$ and (b) a 4-way set-associative cache having one cache set S_0 using a Least-Recently-Used (LRU) cache replacement policy. The LRU age of a block b refers to how many accesses were performed to the cache set in which b is saved since the last access to b.

associative caches. These approaches are (1) the PCB-ECB approach, that uses only the set of PCBs of the task under analysis and the set of ECBs of all other tasks in the system to evaluate the CPRO, (2) the ResilienceP analysis, that removes some of the pessimism in the PCB-ECB approach by considering the *resilience* (see Definition 3.8) of PCBs, and (3) the multi-path ResilienceP analysis, that considers the variation in the resilience of PCBs over different executions of a task in order to have an even tighter CPRO bound.

3. An experimental evaluation showing that our proposed approaches result in up to 22 percentage points higher task set schedulability than the state-of-the-art resilience analysis (Altmeyer et al., 2010) that only considers CRPDs when analyzing inter-task cache interference for architectures with set-associative caches.

7.1 Assumptions on the System Model

In addition to the general system model detailed in Chapter 2 and Chapter 4, in this chapter we make the following assumptions on the system model.

- In this chapter, we focus on single-core processor with a private set-associative single-level instruction cache (also referred to as L1) using the LRU replacement policy, i.e., on a cache miss the least recently used memory block (or equivalently the block with the largest LRU-age) within the targeted cache set is evicted. The number of memory blocks that can be stored in each cache set is referred to as the number of ways or the associativity of the cache and is denoted by W. The set of all cache sets is denoted by $\mathbb{S}$.

The list of important symbols used in this chapter is provided in Table 7.1.

Table 7.1: List of important symbols used in Chapter 7

Symbol	Description
Γ	Task set of size n
W	The number of cache ways or the associativity of the cache
$\mathbb{S}$	The set of all cache sets
τ_i	Task with index i
$\tau_{i,k}$	k^{th} job of Task τ_i
C_i	Worst-case execution time of task τ_i in isolation
T_i	Minimum inter-arrival time of task τ_i
D_i	Relative deadline of task τ_i
R_i	Worst-case response time of task τ_i
PD_i	Worst-case processing demand of task τ_i in isolation
MD_i	Worst-case memory access demand of task τ_i in isolation
MD_i^r	Residual memory access demand of task τ_i in isolation
$\hat{MD}_i(t)$	Total memory access demand of task τ_i in a time interval of length t
$\text{hp}(i)$	The set of tasks with higher priority than τ_i
$\text{hep}(i)$	The set of tasks with higher priority than τ_i including τ_i, i.e., $\text{hep}(i) = \text{hp}(i) \cup \tau_i$.
$\text{aff}(i,j)$	The set of intermediate tasks (including τ_i) that may preempt τ_i but may themselves be preempted by some higher priority task τ_j.
d_{mem}	Time to reload one cache block from the main memory
B_m^s	The set of memory blocks that are persistent in a cache set s after the execution of task τ_i
J	The maximum number of jobs any task τ_j can release in a time interval of length t
$\lvert EP_j \rvert$	The number of possible execution paths of task τ_j
ECB_i	The set of evicting cache blocks (ECBs) of task τ_i
ECB_i^s	The set of ECBs of task τ_i in a cache set s
UCB_i	The set of useful cache blocks (UCBs) of task τ_i
PCB_i	The set of persistence cache blocks (PCBs) of task τ_i
PCB_i^s	The set of PCBs of task τ_i in a cache set s
$D_{j,i}^s$	The disturbance all tasks in $\text{hep}(i) \setminus \tau_j$ may cause on PCBs of task τ_j in a cache set s
$PersistentAge_P(m_j)$	LRU-age of a PCB m_j of task τ_j at any program point P during the execution of τ_j.
$\mathbb{P}$	Set of all program points
$max\text{-}age(m_j)$	The maximum LRU-age of a PCB m_j of task τ_j over all program points during the execution of τ_j.
	Continued on next page

Table 7.1 – continued from previous page

Symbol	Description
$res_{PCB}(m_j)$	Resilience of a PCB m_j of task τ_j
$\rho_{j,i}$	CPRO of one job of a higher priority task $\tau_j \in \text{hp}(i)$ during the response time of a lower priority task τ_i
$\hat{\rho}_{j,i}$	Total CPRO of task τ_j in an interval of length t while executing during the response time of task τ_i.
$\rho^{s}_{j,i}$	CPRO suffered by one job of a higher priority task $\tau_j \in \text{hp}(i)$ during the response time of a lower priority task τ_i, due to a cache set s, computed using the PCB-ECB approach.
$\rho^{set}_{j,i}$	Total CPRO suffered by one job of a higher priority task $\tau_j \in \text{hp}(i)$ during the response time of a lower priority task τ_i under the PCB-ECB approach.
$\rho^{res,s}_{j,i}$	CPRO suffered by one job of a higher priority task $\tau_j \in \text{hp}(i)$ during the response time of a lower priority task τ_i, due to a cache set s, computed using the ResilienceP analysis.
$\rho^{res}_{j,i}$	Total CPRO suffered by one job of a higher priority task $\tau_j \in \text{hp}(i)$ during the response time of a lower priority task τ_i under the ResilienceP analysis.
$\rho_{j,m_j}(D,J)$	The maximum number of times any PCB m_j of task τ_j can contribute to the CPRO of τ_j in a time interval of length t assuming the values of disturbance suffered by τ_j, i.e., D and the number of jobs released by τ_j, i.e., J, are known.
$\rho^{M-res,s}_{j,i}$	CPRO suffered by one job of a higher priority task $\tau_j \in \text{hp}(i)$ during the response time of a lower priority task τ_i, due to a cache set s, computed using the multi-path ResilienceP analysis.
$\rho^{M-res}_{j,i}$	Total CPRO suffered by one job of a higher priority task $\tau_j \in \text{hp}(i)$ during the response time of a lower priority task τ_i under the multi-path ResilienceP analysis.

7.2 Finding PCBs for set-associative caches

For direct-mapped caches, determining the set of PCBs, i.e., PCB_i, of a task τ_i is relatively simple. as presented in Section 4.5, i.e., a memory block m_i of task τ_i belongs to PCB_i if it is the only memory block of τ_i mapped to a given cache set. However, under set-associative caches, several memory blocks of τ_i may be mapped to a single cache set and the presence of a memory block in cache depends on the LRU-age of that memory block. A W-way set-associative LRU-cache can hold up to W memory blocks in each cache set and the LRU-age of each memory block can be between 0 and $W-1$ (respectively representing the most-recently and the least-recently accessed

memory block in the cache set). Given a program point P, if the LRU-age of a memory block at P is greater than or equal to W then an access to that memory block at P will be a cache miss, i.e., the memory block is not in the cache anymore. We can leverage this information to find PCBs of a task τ_i under a set-associative cache. By definition, once loaded into the cache by task τ_i all PCBs of τ_i will not be evicted or invalidated by τ_i while executing in isolation. Therefore, all memory blocks used by τ_i that have an LRU-age less than or equal to $W-1$ at every program point P in τ_i can be PCBs of τ_i. However, knowing that PCBs are potentially reused by the same but also by every next job executed by task τ_i, using only one job execution of τ_i to bound the maximum LRU-age of PCBs may not be sufficient. To illustrate this last property, consider the control-flow graph (CFG) and mapping of the memory blocks of two successive jobs of task τ_i (i.e., $\tau_{i,1}$ and $\tau_{i,2}$) shown in Figure 7.2. In that example, all memory blocks, i.e., m_1, m_2, m_3, and m_4, used by τ_i map to the same cache set s in a 4-way set-associative cache. We can see in Figure 7.2a that when considering only one job of τ_i, i.e., $\tau_{i,1}$, the maximum LRU-age, i.e., the maximum number of distinct accesses between two references to a memory block, of memory blocks m_1, m_2, m_3, and m_4 is 3, 2, 1, and 0 respectively (see Section 3.2.2 for details on the computation of maximum LRU-age of memory blocks). However, if we consider the execution sequence $\tau_{i,1}$ followed by $\tau_{i,2}$, we can see that the maximum LRU-age of all the memory blocks is 3. Note that underestimating the maximum LRU-age of memory blocks may lead to false positives, i.e., a memory block may be categorized as a PCB (i.e., LRU-age $\leq W-1$ over the execution of one job) while it is not (i.e., LRU-age $> W-1$ over the execution of a sequence of jobs). Therefore, in order to soundly estimate the set of PCBs of a task τ_i, it is important to calculate the maximum LRU-age of all memory blocks used by τ_i after any execution sequence of jobs of τ_i (in isolation). This can be done by assuming that τ_i is cyclic, i.e., a loop is assumed between the end point E and start point S of τ_i (see Figure 7.2b). The cyclic assumption ensures that the maximal number of different cache accesses between the last use of a memory block m_i in one job of τ_i and the first access of m_i in the next job of τ_i are considered when determining the maximum LRU-age of m_i.

Formally, the analysis to find PCBs of a task τ_i is performed as follows:

1. Apply the standard *persistence* analysis (Theiling et al., 2000; Cullmann, 2013) on the code of task τ_i to determine the set of memory blocks B_m^s that are persistent in each cache set s at the end of τ_i. The Persistence analysis determines if a memory block will not be evicted after it has been loaded in the cache, i.e., the first reference to that memory block may result in a cache miss but all subsequent references to that memory block will be cache hits (see Section 3.1.3 for a detailed description on the persistence analysis). A memory block m_i is persistent in a cache set s if its LRU-age in s is less than or equal to $W-1$ at the end of τ_i's execution.

2. Apply the persistence analysis again (i.e., to account for the cyclic assumption) on τ_i assuming that each cache set s already contains the B_m^s memory blocks at the start of τ_i's execution and that each of those blocks has its maximum LRU-age derived in step 1. All memory blocks in B_m^s that are in every *Abstract Cache State* (ACS) (see Definition 3.1)) of the sec-

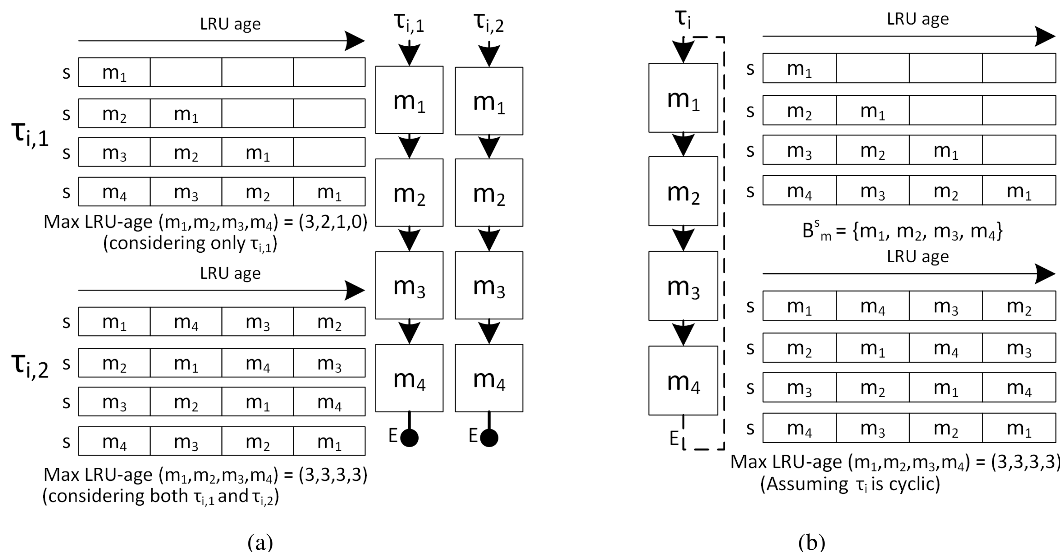


Figure 7.2: Maximum LRU-age of memory blocks of task τ_i (a) over the execution of two jobs of τ_i, and (b) under the assumption that τ_i is cyclic

ond persistence analysis (i.e, memory block with LRU-age$\leq W-1$ in all ACSs) are PCBs of τ_i in cache set s and are denoted by PCB_i^s. The final set of PCBs of task τ_i is then given by

$$PCB_i = \bigcup_{s \in \mathbb{S}} PCB_i^s \tag{7.1}$$

7.3 CPRO Analysis for Set-Associative Caches

In this section, we present two approaches for the calculation of the CPRO for set-associative caches, namely, the PCB-ECB approach and the ResilienceP analysis.

7.3.1 PCB-ECB Approach

As we have previously explained in Chapter 4 and Chapter 5, for direct-mapped caches, the CPRO can be computed by using the intersection between the set of PCBs of the task under analysis and the set of ECBs of all other tasks that may evict PCBs of that task. For example, under the CPRO-union approach (i.e., Equation (4.5)) presented in Section 4.3, the CPRO $\rho_{j,i}$ one job of a higher priority task $\tau_j \in \text{hp}(i)$ may suffer while executing during the response time of a lower priority task τ_i is computed using the set of PCBs of task τ_j, i.e., PCB_j, the set of ECBs of all tasks in $\text{hep}(i) \setminus \tau_j$, i.e., $\bigcup_{\forall \tau_k \in \text{hep}(i) \setminus \tau_j} ECB_k$, and d_{mem} which is the time required to reload one PCB from the main memory (see Theorem 4.1, for a formal proof of Equation (4.5)). The bound resulting from Equation (4.5) is sound for a direct-mapped cache where each ECB of tasks in $\text{hep}(i) \setminus \tau_j$ can evict at most one PCB of τ_j. However, using Equation (4.5) to calculate the CPRO of τ_j under a set-associative cache may result in underestimating the CPRO of τ_j due to the cascading effect

mentioned earlier using Figure 7.1, i.e., several/all PCBs of task τ_j may be evicted due to a single ECB of another task mapped to the same cache set. Considering that in a set-associative cache, each cache set s can be analyzed independently, a sound estimate of the CPRO suffered by one job of task τ_j due to a cache set $s \in \mathbb{S}$ can be obtained by using the two following properties:

1. PCBs of task τ_j mapped in cache set s may be evicted and hence participate to the CPRO of τ_j during the response time of another task τ_i, only if one or more ECB(s) of tasks in $\text{hep}(i) \setminus \tau_j$ are mapped to the same cache set s. To formally characterize the impact of ECB(s) of tasks in $\text{hep}(i) \setminus \tau_j$ on PCBs of task τ_j in a cache set s, we define the notion of *Disturbance*, i.e.,

   Definition 7.1 (Disturbance)**.** *The disturbance suffered by a task τ_i on a cache set s due to another set of tasks $\mathscr{T}$ is defined as the total number of ECBs of tasks in $\mathscr{T}$ that are mapped to the cache set s. The disturbance due to $\mathscr{T}$ is thus the maximum number of memory blocks of tasks in $\mathscr{T}$ that compete with τ_i for space in cache set s.*

   Based on the above definition, PCBs of task τ_j mapped in cache set s can be evicted and hence participate to the CPRO of τ_j, only if the disturbance task τ_j may suffer due to all tasks in $\text{hep}(i) \setminus \tau_j$ in s is greater than or equal to one.

2. The participation of a cache set s to the CPRO of τ_j is upper bounded by the number of PCBs of τ_j in that cache set multiplied by d_{mem}, i.e., $d_{mem} \times |PCB^s_j|$.

Therefore, the CPRO one job of task τ_j may suffer due to cache set s is upper bounded by $\rho^{set,s}_{j,i}$, where

$$\rho^{set,s}_{j,i} = d_{mem} \times \begin{cases} |PCB^s_j| & \text{if } D^s_{j,i} \geq 1 \\ 0 & \text{otherwise} \end{cases} \tag{7.2}$$

where $D^s_{j,i}$ is the disturbance all tasks in $\text{hep}(i) \setminus \tau_j$ may cause on PCBs of τ_j in a cache set s. By definition, the maximum disturbance task τ_j may suffer due to all tasks in $\text{hep}(i) \setminus \tau_j$ is upper bounded by $\sum_{\forall \tau_k \in \text{hep}(i) \setminus \tau_j} |ECB^s_k|$, i.e., the total number of ECBs of tasks in $\text{hep}(i) \setminus \tau_j$ mapped to cache set s. Therefore, $D^s_{j,i} \leq \sum_{\forall \tau_k \in \text{hep}(i) \setminus \tau_j} |ECB^s_k|$. The rationale of using ECBs of all tasks in $\text{hep}(i) \setminus \tau_j$ to bound $D^s_{j,i}$ is to account for nested/multiple execution of tasks in $\text{hep}(i) \setminus \tau_j$ between two jobs of task τ_j. Since, in the worst-case all tasks in $\text{hep}(i) \setminus \tau_j$ may sequentially execute between two jobs of τ_j, the cumulative impact of tasks in $\text{hep}(i) \setminus \tau_j$ on PCBs of τ_j is upper bounded by $\sum_{\forall \tau_k \in \text{hep}(i) \setminus \tau_j} |ECB^s_k|$.

Note that Equation (7.2) accounts for the cascading effect by considering that a single ECB of tasks in $\text{hep}(i) \setminus \tau_j$ mapped to a cache set s may evict all PCBs of τ_j in that cache set.

The total CPRO one job of τ_j may suffer during the response time of τ_i is given by

$$\rho^{set}_{j,i} = \sum_{\forall s \in \mathbb{S}} \rho^{set,s}_{j,i} \tag{7.3}$$

7.3.2 ResilienceP Analysis

The PCB-ECB approach presented above assumes that if one ECB of any task in hep$(i) \setminus \tau_j$ is mapped to a cache set s then all the PCBs of τ_j in s will be evicted. This assumption is safe but very pessimistic. To illustrate, consider the example depicted in Figure 7.3. It shows a sequence

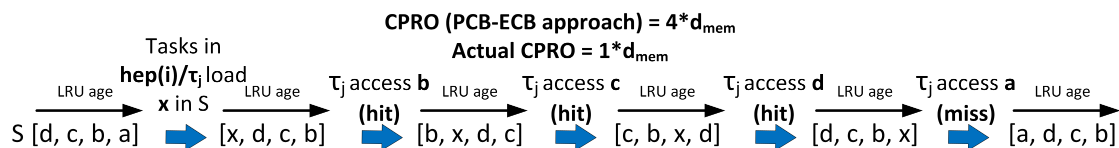


Figure 7.3: Example scenario to highlight the pessimism in the PCB-ECB approach

of cache references during the execution of a task τ_j (from left to right) assuming that τ_j has 4 PCBs in cache set s, i.e., $PCB_j^s = \{a, b, c, d\}$. We also assume that the value of disturbance $D_{j,i}^s$ is equal to 1, i.e., only one ECB of tasks in hep$(i) \setminus \tau_j$ is mapped to cache set s. Using $|PCB_j^s| = 4$ and $D_{j,i}^s = 1$ in Equation (7.2) (i.e., the PCB-ECB approach) the CPRO of τ_j due to cache set s is calculated to be $4 \times d_{mem}$. However, we can see in Figure 7.3 that only one cache reference of τ_j will be a miss after the execution of tasks in hep$(i) \setminus \tau_j$. Therefore, the actual CPRO of τ_j due to cache set s is only $1 \times d_{mem}$.

The ResilienceP analysis removes excessive pessimism in the PCB-ECB approach by finding PCBs of task τ_j that may remain cached (and therefore does not contribute to the CPRO) even after the execution of tasks in hep$(i) \setminus \tau_j$ thanks to their resilience. Based on the definition of resilience, i.e., Definition 3.8, the resilience of a PCB m_j of task τ_j is given by the maximum value of disturbance $D_{j,i}^s$ that m_j can endure before being evicted from the cache due to the execution of tasks in hep$(i) \setminus \tau_j$.

As already explained in Chapter 3 (Section 3.2.2), the resilience-analysis proposed in (Altmeyer et al., 2010) can be used to determine the resilience of all memory blocks used by a task τ_j, at every program point P during the execution of τ_j. However, using the state-of-the-art resilience-analysis (Altmeyer et al., 2010) to determine the resilience of PCBs may result in overestimating the resilience of PCBs. This is mainly because the resilience-analysis (Altmeyer et al., 2010) was proposed to calculate the resilience of UCBs instead of PCBs. By definition, UCBs of a task τ_j may only be reused during the same job execution of τ_j and hence it is sufficient to consider the execution of only one job of τ_j when bounding the maximum LRU-age of its UCBs. However, PCBs are different from UCBs considering that PCBs may be reused during the execution of the same job and/or any next job of τ_j. Therefore, to have a sound estimate of the resilience of PCBs of τ_j it is necessary to calculate the maximum LRU-age of PCBs after any execution sequence of jobs of τ_j. See Figure 7.2 that shows that considering only one job of task τ_i may result in underestimating the maximum LRU-age (i.e., overestimating the resilience) of memory blocks m_2, m_3 and m_4.

The ResilienceP analysis uses the approach described in Section 7.2 to determine the resilience of PCBs of each task. Let $PersistentAge_P(m_j)$ denote the LRU-age of a PCB m_j at any program

point P during the execution of task τ_j resulting from the analysis detailed in Section 7.2. Then, the maximum LRU-age $max\text{-}age(m_j)$ of m_j is calculated by maximizing $PersistentAge_P(m_j)$ over all program points during the execution of τ_j, i.e.,

$$max\text{-}age(m_j) = \max_{\forall P \in \mathbb{P}} PersistentAge_P(m_j) \tag{7.4}$$

where $\mathbb{P}$ is the set of all program points. Consequently, the resilience of PCB m_j is then given by $res_{PCB}(m_j) = (W-1) - max\text{-}age(m_j)$.

Therefore, the ResilienceP analysis upper bounds the CPRO that may be suffered by one job of task τ_j due to cache set s by $\rho_{j,i}^{res,s}$, where $\rho_{j,i}^{res,s}$ is computed as follows

$$\rho_{j,i}^{res,s} = d_{mem} \times \left| PCB_j^s \setminus \{ m_j | res_{PCB}(m_j) \geq D_{j,i}^s \} \right| \tag{7.5}$$

where $res_{PCB}(m_j)$ is the resilience of a PCB $m_j \in PCB_j^s$ and $D_{j,i}^s$ is the maximum disturbance all tasks in $\text{hep}(i) \setminus \tau_j$ may cause to a cache set s.

Note that Equation (7.5) excludes PCBs of τ_j that remain cached after the execution of tasks in $\text{hep}(i) \setminus \tau_j$ (i.e., those for which $res_{PCB}(m_j) \geq D_{j,i}^s$) from the CPRO. Therefore, it provides a tighter bound on the CPRO than the PCB-ECB approach.

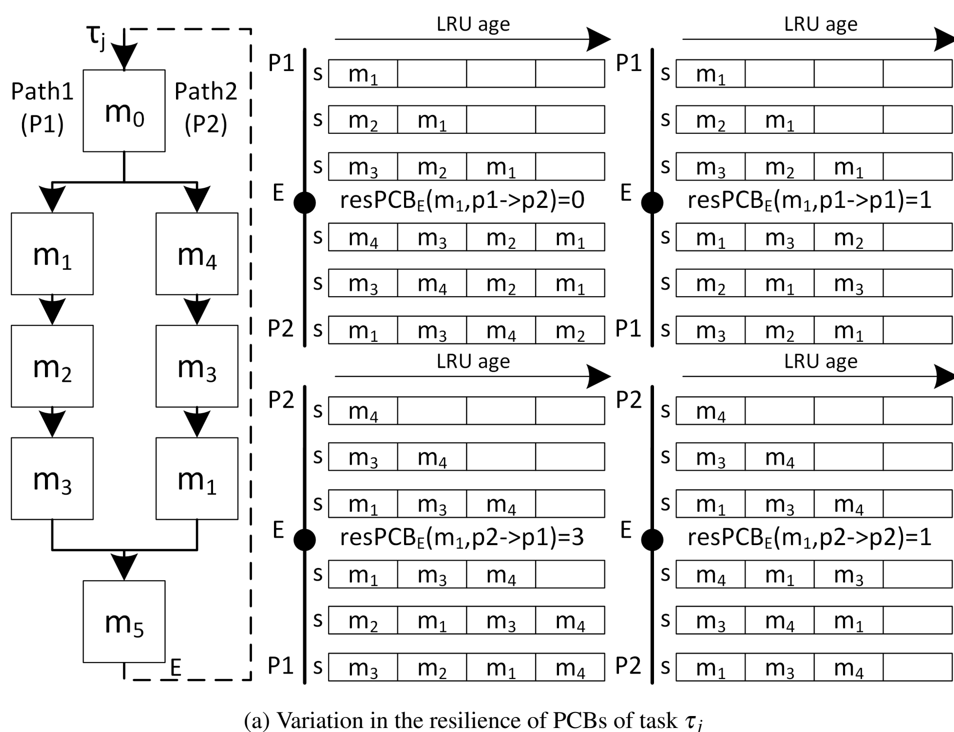
The total CPRO of one job of task τ_j executing during the response time of another task τ_i is thus bounded by

$$\rho_{j,i}^{res} = \sum_{\forall s \in \mathbb{S}} \rho_{i,j}^{res,s} \tag{7.6}$$

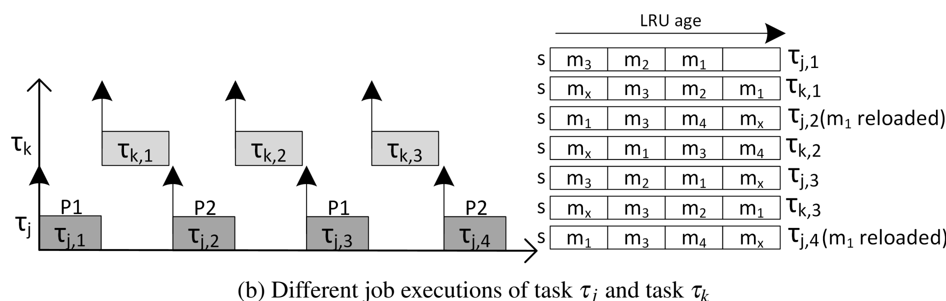
Finally, knowing from Lemma 4.2 that in a time interval of length t at most $\left\lceil \frac{t}{T_j} \right\rceil - 1$ jobs of τ_j may suffer CPRO. Therefore, the total CPRO of task τ_j in a time interval of length t can be bounded by $\hat{\rho}_{j,i}(t)$ (i.e., given by Equation (4.6)), where $\rho_{j,i}$ can be calculated either using the PCB-ECB approach or the ResilienceP analysis (i.e., by using Equation (7.3) or Equation (7.6)).

7.4 Multi-path ResilienceP Analysis

The ResilienceP analysis always considers the worst-case (i.e., minimum) resilience for every PCB and for every job of τ_j that may execute in a time interval of length t. This assumption is exact in the case where τ_j has only a single execution path as shown in Figure 7.2b. However, if τ_j has multiple execution paths, the resilience of PCBs may vary depending on the actual execution paths taken by two successive jobs of τ_j. Therefore, always considering the minimum resilience of PCBs over all job executions of τ_j may overestimate the total CPRO τ_j may suffer. To illustrate this, Figure 7.4a shows the CFG of a task τ_j with two execution paths and four possible execution flows between two jobs of τ_j, i.e., $p1 \rightarrow p2$, $p2 \rightarrow p1$, $p1 \rightarrow p1$ and $p2 \rightarrow p2$. The cache content along each execution flow is also shown in Fig. 7.4a. We assume that all memory blocks of τ_j except m_0 and m_5 map to the same cache set s of a 4-way set-associative cache. For simplicity, we only focus on PCB m_1.



(a) Variation in the resilience of PCBs of task τ_j



(b) Different job executions of task τ_j and task τ_k

Figure 7.4: Highlighting the pessimism in the ResilienceP analysis

We can see in Figure 7.4a that the resilience of m_1 is minimum, i.e., $res_{PCB}(m_1) = 0$, if the first job of τ_j follows path $p1$ and the next job follows path $p2$. Now consider the example schedule shown in Figure 7.4b showing four jobs of τ_j executed together with three jobs of a task $\tau_k \in \text{hep}(i) \setminus \tau_j$ such that $ECB_k^s = \{m_x\}$, i.e, $D_{j,k}^s = 1$. Figure 7.4b shows the contents of cache set s after the execution of every job of task τ_j and task τ_k.

Since the minimum resilience of m_1 is 0 and $D_{j,k}^s > res_{PCB}(m_1)$, the ResilienceP analysis (i.e., Equation (7.5)) implies that every time τ_k preempts τ_j or executes between two subsequent jobs of τ_j, m_1 will be evicted. This results in a CPRO equal to $3 \times d_{mem}$. However, we can see in Figure 7.4b that this is not true. In fact even when we maximize the number of jobs of τ_j following the execution flow with the minimum resilience (i.e., $p_1 \rightarrow p_2$), m_2 is evicted and reloaded only two times resulting in a CPRO of $2 \times d_{mem}$. The reason behind this result is that if the first two jobs of τ_j execute according to the execution flow $p_1 \rightarrow p_2$, then the second and third jobs of τ_j can either follow the execution flow $p_2 \rightarrow p_1$ or $p_2 \rightarrow p_2$ In both cases, the actual resilience of m_1

Table 7.2: CPRO-table for every PCB m_j of task τ_j

		Number of jobs of τ_j (J)			
		2	3	...	$\lceil \frac{t}{T_j} \rceil$
Disturbance D	1	$\rho_{j,m_j}(1,2)$	$\rho_{j,m_j}(1,3)$	...	$\rho_{j,m_j}(1,\lceil \frac{t}{T_j} \rceil)$
	2	$\rho_{j,m_j}(2,2)$	$\rho_{i,j}(2,3)$	...	$\rho_{j,m_j}(2,\lceil \frac{t}{T_j} \rceil)$
	...	...	...	...	...
	$\geq W$	1	2	...	$\lceil \frac{t}{T_j} \rceil - 1$

is equal to 1 (instead of 0 as assumed by the ResilienceP analysis).

The multi-path ResilienceP analysis reduces the pessimism of the ResilienceP analysis by considering the variation in the resilience of PCBs across different job execution flows of a same task τ_j. It computes the total CPRO task τ_j may suffer in a time interval of length t by first creating a CPRO-table (See Table 7.2) for all PCBs of τ_j in each cache set. The CPRO-table determines how many times each PCB $m_j \in PCB_j^s$ can be evicted in an interval of length t considering a given disturbance D and the maximum number of jobs J released by τ_j in the interval of length t. Given the values of D and J, one entry in Table 7.2 tells us how many times PCB m_j may be evicted and must therefore be reloaded.

7.4.1 Building the CPRO-table

In this subsection we discuss how a CPRO-table can be built for PCBs.

First, we make use of a couple of simple properties to bound the size of that table:

1. It is proved in Lemma 4.2 that if a task τ_j releases J jobs in a time interval of length t, the maximum number of times each PCB $m_j \in PCB_j^s$ can be evicted is upper bounded by $J-1$.
2. If the disturbance D suffered by a PCB m_j is greater than or equal to the number of ways W in the cache (i.e., $D \geq W$) then the entire cache set s will be filled by the ECBs of disturbing tasks and PCB m_j will be evicted after every job execution i.e., it will be evicted $J-1$ times.

We use that information to fill all the CPRO-table entries such that $D \geq W$ (see Table 7.2).

The remaining entries (noted $\rho_{j,m_j}(D,J)$) are calculated using Algorithm 7.1. Algorithm 7.1 uses the set of all possible execution paths EP_j of task τ_j and the maximum length t of the interval in which its PCB m_j may be evicted as input. It then fills row-wise all entries in Table 7.2. The function `PathsPermutations(·)` at line 1 returns a set that contains all possible executions paths combinations between two jobs of τ_j. Given that task τ_j has $|EP_j|$ possible execution paths, the number of possible execution flows between two jobs of τ_j is given by $Flows$, where the size of $Flows$ is $2^{|EP_j|}$.

The external loop (lines 2 to 17) is used to iterate over all disturbance values D between 1 and $W-1$ (all table entries for $D \geq W$ are already filled). As previously discussed, the resilience

Algorithm 7.1 Building the CPRO-table for PCB m_j of task τ_j

Input: Interval length t; PCB m_j; Set of all possible execution paths EP_j of τ_j.
Output: All $\rho_{j,m_j}(D,J)$ entries in Table 7.2.

```
1:  Flows :=PathsPermutations(τj, |EPj|)
2:  for D := 1 to W − 1 do
3:     PossibleFlows :=FindPathsCombinations(mj, Flows, D)
4:     L :=FindLongestFlow(PossiblePaths)
5:     for J := 2 to ⌈t/Tj⌉ do
6:        if |PossibleFlows| = 0 then
7:           ρj,mj(D,J) := 0;
8:        else
9:           MaxCPRO := J − 1
10:          if L ≥ MaxCPRO then
11:             ρj,mj(D,J) := MaxCPRO;
12:          else
13:             ρj,mj(D,J) := MaxCPRO − ⌊J/(L+1)⌋;
14:          end if
15:       end if
16:    end for
17: end for
```

of a PCB m_j may vary depending on the specific combination of execution paths taken by two successive jobs of τ_j (see Figure 7.4a). Therefore, the function `FindPathsCombinations(·)` (line 3) returns the set of paths combinations of two successive jobs of τ_j for which the resilience of m_j is less than D. By the definition of resilience, the memory block m_j may be evicted only for those paths combinations. Function `FindLongestFlow(·)` then generates (at line 4) the longest execution flow composed of path combinations in $PossibleFlows$. For example, assuming $PossibleFlows$ contains the three paths combinations $p_1 \to p_2$, $p_3 \to p_1$ and $p_3 \to p_2$. The longest execution flow that may be generated by `FindLongestFlow(·)` is $p_3 \to p_1 \to p_2$. The function thus returns 2 as the length L of that flow. Note that by definition of $PossibleFlows$, there exists a (possibly different) program point P in each path composing that execution flow for which the resilience of m_j is less than D. Therefore, if the maximum disturbance D is applied at each of those program points, then m_j will be evicted L times. Moreover, since `FindLongestFlow(·)` generates the longest such execution flow, there cannot be more than L successive evictions of m_j. The nested loop (lines 5 to 16) is then used to upper bound how many times m_j will be evicted for every possible value of J (note that for sporadic tasks, at most $\left\lceil \frac{t}{T_j} \right\rceil$ jobs of τ_j may be released in any interval of length t, thus $J \leq \left\lceil \frac{t}{T_j} \right\rceil$). If the set of possible paths combinations returned by the function `FindPathsCombinations(·)` is empty, then m_j cannot be evicted for the disturbance value D and hence $\rho_{j,m_j}(D,J)$ is equal to 0 (line 6). Otherwise, if there exists some paths combinations for which m_j may be evicted with a disturbance D, i.e., $|PossibleFlows| > 0$, then two cases must be considered:

1. If $L \geq J-1$, then we know from Lemma 4.2 that the CPRO suffered by J successive jobs of a task τ_j is upper bounded by $J-1$ and thus $\rho_{j,m_j}(D,J) = J-1$ (line 11).

Algorithm 7.2 Computing the total CPRO of task τ_j in a time interval of length t

Input: Interval length t; CPRO-table of every PCB m_j of task τ_j; Disturbance $D^s_{j,i}$ for every $s \in \mathbb{S}$
Output: The total CPRO of task τ_j in a time interval of length t, i.e., $\rho^{M-res}_{i,j}(t)$.

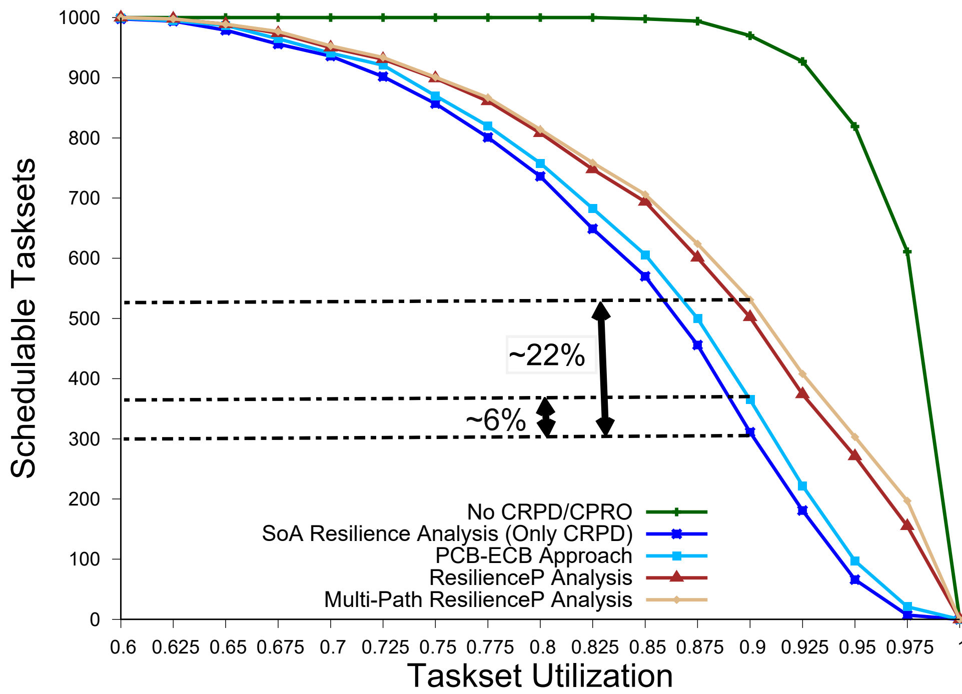
1: $J := \left\lceil \frac{t}{T_j} \right\rceil$
2: $\rho^{M-res}_{i,j}(t) := 0$
3: **for** $\forall s \in \mathbb{S}$ **do**
4: $\quad \rho^{M-res,s}_{i,j} := 0$
5: $\quad D := D^s_{j,i}$
6: $\quad$ **for** $\forall m_j \in PCB^s_j$ **do**
7: $\quad\quad \rho^{M-res,s}_{i,j} := \rho^{M-res,s}_{i,j} + \rho_{j,m_j}(D,J)$
8: $\quad$ **end for**
9: $\quad \rho^{M-res}_{i,j}(t) := \rho^{M-res}_{i,j}(t) + \rho^{M-res,s}_{i,j}$
10: **end for**

2. If $L < J-1$, then, by the definition of L, m_j may be evicted at most L times in every execution flow composed of $L+1$ successive jobs of τ_j. Therefore, the maximum number of times m_j may be evicted for a succession of J jobs is bounded by $(J-1) - \left\lfloor \frac{J}{L+1} \right\rfloor$ (line 13).

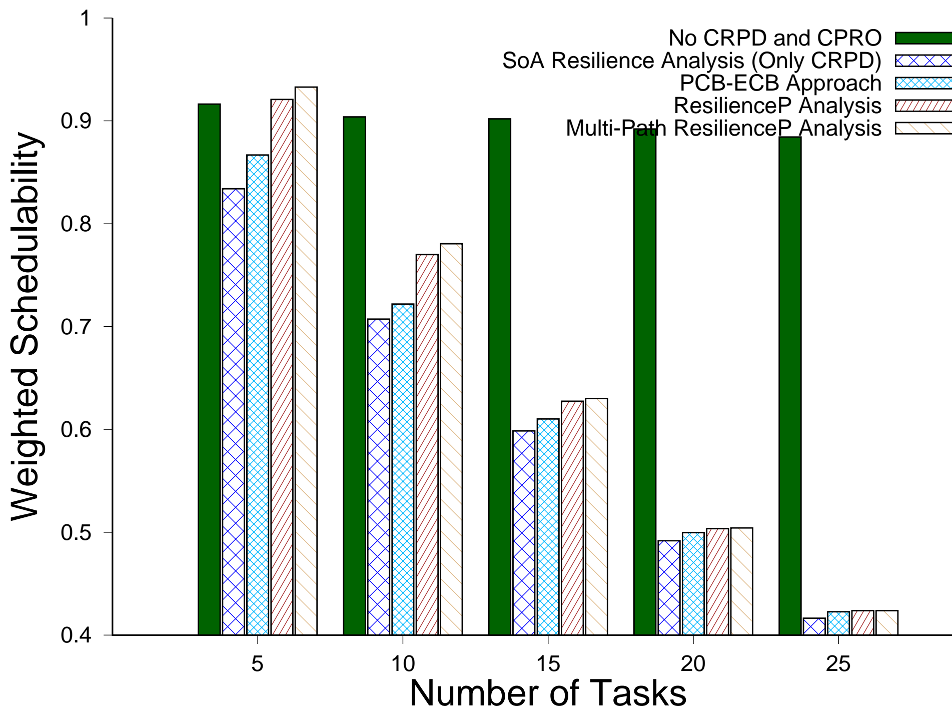
Example 7.1. *Consider memory block m_1 in Figure 7.4. By applying Algorithm 7.1 with $D^s_{j,k} = 1$ (i.e., $|ECB^s_k| = 1$), L=1 (i.e., the execution flow $p_1 \rightarrow p_2$) and J=$\{2,3,4\}$, we get $\rho_{j,m_1}(1,2) = 1$, $\rho_{j,m_1}(1,3) = 1$ and $\rho_{j,m_1}(1,4) = 2$ which is consistent with the scenario depicted in Figure 7.4b.*

7.4.2 Bounding the CPRO

After creating the CPRO-table of every PCB of task τ_j using Algorithm 7.1, the total CPRO that task τ_j may suffer in a time interval of length t can be bounded using Algorithm 7.2. The inputs to Algorithm 7.2 are the CPRO-tables of every PCB m_j of task τ_j, the maximum disturbance task τ_j may suffer for every $s \in \mathbb{S}$ due to the execution of tasks in $\text{hep}(i) \setminus \tau_j$, i.e., $D^s_{j,i}$, and the length of time interval t. The output of Algorithm 7.2 is the total CPRO of task τ_j in a time interval of length t denoted by $\rho^{M-res}_{i,j}(t)$. Given the length t of the time interval, $J := \left\lceil \frac{t}{T_j} \right\rceil$ upper bounds the number of jobs τ_j may execute in t (line 1). For every cache set $s \in \mathbb{S}$, the value of D is set using $D^s_{j,i}$ (line 5). Given the values of D and J the inner loop (lines 6 to 8) iteratively computes the CPRO of task τ_j in every cache set $s \in \mathbb{S}$, i.e., $\rho^{M-res,s}_{i,j}$, using values from the CPRO-table of every PCB $m_j \in PCB^s_j$. The outer loop (lines 3 to 10) then sums up the values of $\rho^{M-res,s}_{i,j}$ for all $s \in \mathbb{S}$ to bound $\rho^{M-res}_{i,j}(t)$.



(a)



(b)

Figure 7.5: Task sets schedulability by varying (a) total task set utilization and (b) the total number of tasks in a task set

7.5 WCRT Analysis

The WCRT analysis that accounts for both CRPDs and CPROs considering direct-mapped caches is given by Equation (4.9) and (4.18). The same equations can also be used to calculate the WCRT R_i of a task τ_i when considering set-associative caches; (i) by calculating the CPRO using any of the approaches presented in Sections 7.3 and 7.4, and (ii) by calculating the CRPD using the state-of-the-art resilience-analysis (Altmeyer et al., 2010) (i.e., Equation (3.12)). The WCRT R_i of task τ_i is then calculated by using simple fixed-point iteration on R_i, where R_i is initialized to C_i. In every iteration, the values of CPRO are updated based on the chosen approach. For example, if multi-path ResilienceP analysis is used, Algorithm 7.2 is executed at every iteration to update the total CPRO suffered by the tasks based on the CPRO-tables previously built using Algorithm 7.1. To reduce computation time, the CPRO-table of every PCB of each task can be built only once by setting $t = D_n$ in Algorithm 7.1, where D_n denote the deadline of the lowest priority task τ_n in the task set. The iteration stops as soon as R_i does not evolve anymore or $R_i > D_i$ (i.e., the task is deemed unschedulable).

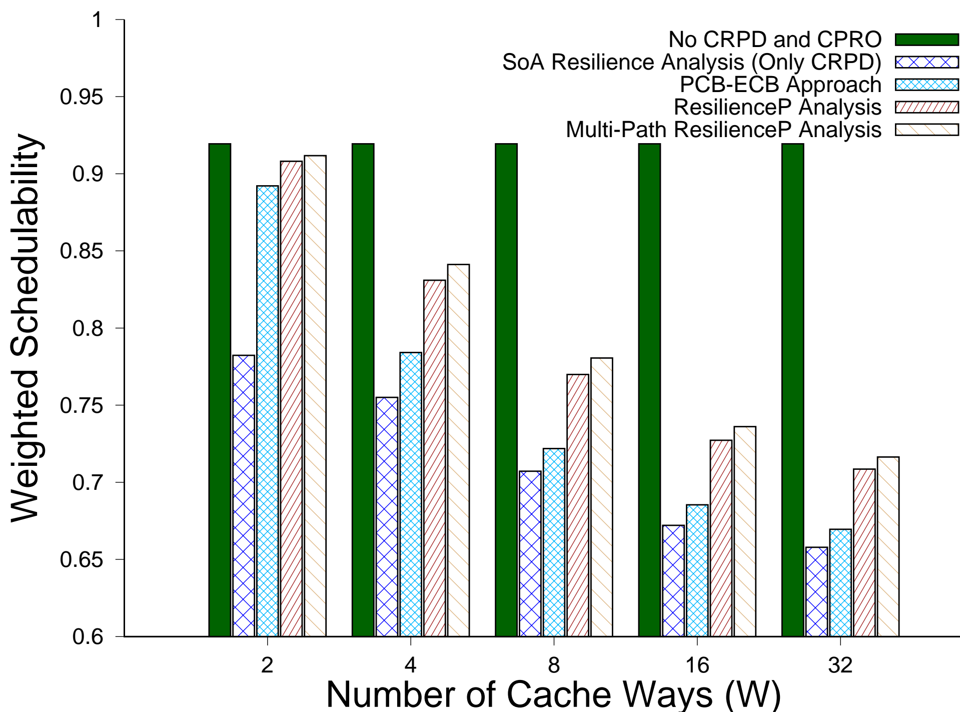
7.6 Experimental Evaluation

In this section, we evaluate how our proposed approaches that account for both cache persistence (i.e., CPROs) and CRPDs perform in terms of schedulability in comparison to the state-of-the-art resilience-analysis (Altmeyer et al., 2010) that only considers CRPDs when analyzing set-associative caches. We performed experiments using synthetic task sets where tasks parameters C_i, PD_i, MD_i, UCB_i, ECB_i and PCB_i were taken from Table 5.2. Each task in the task set was randomly assigned the values C_i, PD_i, MD_i, UCB_i, ECB_i and PCB_i of one of the benchmarks referred in that table. The system was setup to model a MIPS R2000/R3000 architecture assuming a 8-way set-associative cache with 64 sets, a line size of 32 Bytes (i.e., a total cache size of 16kB) and a memory reload time $d_{mem} = 10\mu$s.

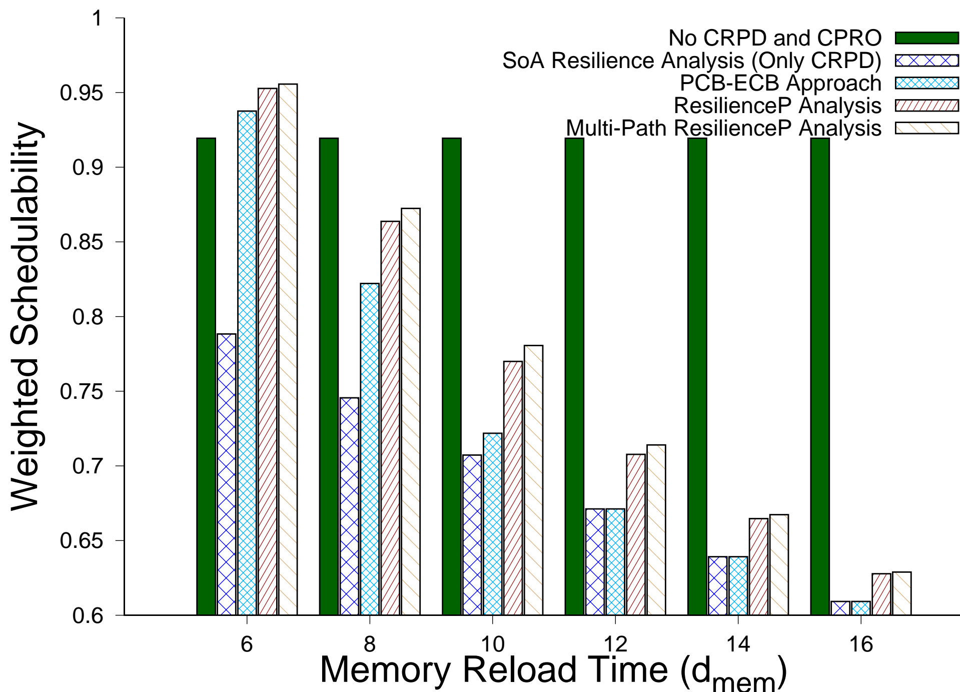
The default number of tasks in a task set was 10 with task utilizations generated using UUnifast (Bini and Buttazzo, 2005). Task periods and deadlines were set such that D_i=T_i=C_i/U_i. Task priorities were assigned in a deadline monotonic order. Furthermore, to evaluate the performance of the multi-path ResilienceP analysis each task was randomly assigned between 1 to 4 execution paths.

We performed different experiments by varying the total core utilization, number of tasks, number of cache ways and memory reload time d_{mem}. A WCRT based schedulability analysis is performed using the same task sets for all the analyzed approaches.

1. Core Utilizations: In this experiment, we varied the total core utilization from 0.025 to 1 in steps of 0.025 and randomly generated 1000 tasksets at every value of the core utilization. Figure 7.5a show the number of task sets that were deemed schedulable by all the analyzed approaches. The plot also show the number of task sets that were deemed schedulable without considering any CRPD and CPRO (i.e., green line). Note that we only show cropped version of



(a) Varying the number of cache ways



(b) Varying memory reload time d_{mem}

Figure 7.6: Weighted schedulability results by varying (a) number of cache ways W and (b) memory reload time d_{mem}

the plot starting from a utilization of 0.6 as all approaches produce identical results below this utilization. Figure 7.5a shows that our approaches that also account for cache persistence (i.e.,

CPROs) along with CRPDs dominate the state-of-the-art resilience-analysis that only consider CRPDs and does not account for cache persistence. Among the three proposed approaches, the PCB-ECB approach has the least number of task sets that were deemed schedulable. This is intuitive, since the PCB-ECB approach pessimistically assume that every PCB of a task in a cache set will be evicted if one or more ECBs of any other task are mapped to the same cache set. This pessimism is reduced by the ResilienceP analysis by considering the resilience of PCBs which results in accepting more task sets. Finally, the multi-path ResilienceP analysis was able to schedule even more task sets than the ResilienceP analysis by considering the variation in the resilience of PCBs over multiple job executions. Our proposed approaches improve task set schedulability by 6 to 22 percentage points over the state-of-the-art analysis. Note that on an Intel core i-7 processor (3.4GHz) the average computation time to generate the plot shown in Figure 7.5a was 210 seconds.

2. Number of Tasks: In this experiment, we varied the total number of tasks in a task set between 5 to 25 in steps of 5 keeping default values of all other parameters. Since we varied both the number of tasks and core utilizations we have used the weighted schedulability measure defined by Equation (4.20) to plot the results in Figure. 7.5b. We can see in Figure. 7.5b that by increasing the number of tasks, the total number of task sets that were deemed schedulable by all the approaches decreases. Indeed, this is due to an increasing number of cache evictions and reloads leading to higher CRPD and CPRO. However, we can still observe that our approaches always dominate the state-of-the-art analysis. Note that by increasing the number of tasks, all the three proposed approaches tend to produce similar results. This is mainly because by increasing the number of tasks, the number of ECBs of tasks sharing cache space with PCBs of other tasks also increases, i.e., the disturbance is increased. Therefore, even if PCBs of some tasks have a greater resilience they might still be evicted due to a higher disturbance.

3. Number of Cache Ways (W): The number of ways W defines how many memory blocks can be mapped to one cache set. We increased the number of cache ways from 2 to 32, keeping default values for all other parameters. The results are shown in Figure 7.6a. We can see in Figure 7.6a that by increasing the number of cache ways the total number of schedulable task sets decreases. This is mainly because we assume that the total cache size is constant hence by increasing the number of cache ways the number of cache sets decreases. This results in more tasks sharing the same cache sets which in turns leads to higher CRPD and CPRO. However, we can still see that all the three proposed approaches dominate the state-of-the-art resilience-analysis. In fact due to lower CPRO, for a cache associativity of 2 and 4, the performances of our analyses are considerably better. Note that for all the experiments in this chapter, we assume a sequential layout of tasks in memory (and in cache), however, task layout optimization techniques, e.g., as proposed in Section 6.6, can also be used to further improve task set schedulability.

4. Memory Reload Time d_{mem}: we varied the value of memory reload time d_{mem} from $6\mu s$ to $16\mu s$ in step of $2\mu s$. The results are presented in Figure 7.6b. We can see in Figure 7.6b that for lower values of d_{mem} the difference between the weighted schedulability of our approaches and the state-of-the-art analysis is significantly higher. This is mainly because for lower values of d_{mem}

the reduction in memory access demand due to cache persistence dominates the CPRO. In fact, we can see in Figure 7.6b that when the value of d_{mem} is equal to 6μs, analysis that account for cache persistence even outperform the “No CRPD and CPRO" analysis. This is mainly because the “No CRPD and CPRO" analysis does not account for cache persistence and only use the WCET of tasks to compute the response time.

We also note that for $d_{mem} \geq 12\mu$s the performances of the PCB-ECB approach and the state-of-the-art analysis are identical. This is due to the excessive pessimism in PCB-ECB approach which is removed by the ResilienceP (and multi-path ResilienceP) analysis.

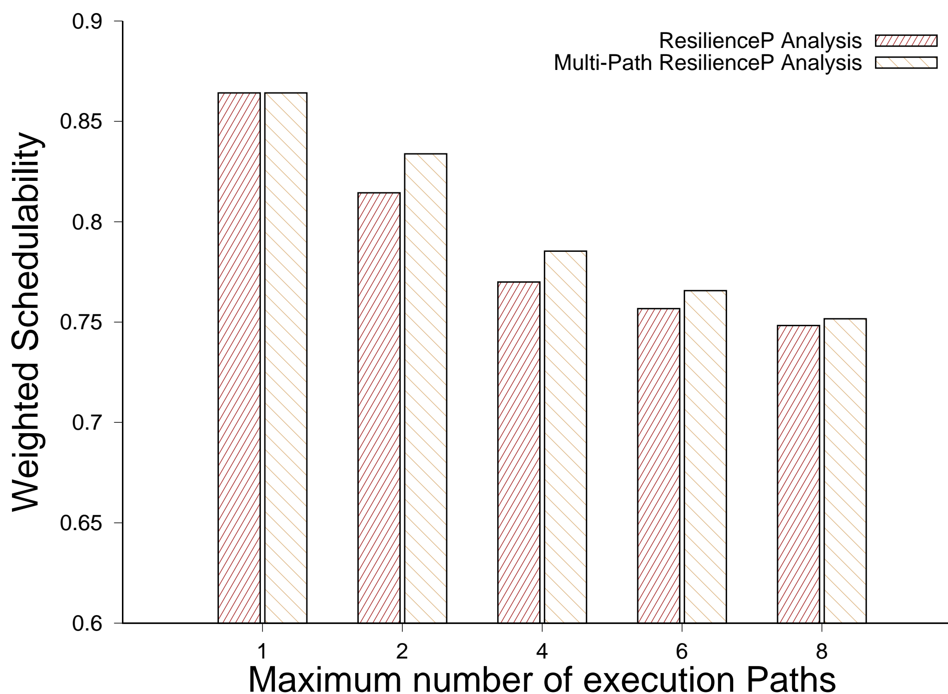


Figure 7.7: Performance of ResilienceP and multi-path ResilienceP analysis w.r.t the number of execution paths

5. Number of execution paths: It is obvious from the results that the multi-path ResilienceP analysis dominates all other approaches. However, the performance of the multi-path ResilienceP analysis depends on the number of execution paths of tasks and the resilience of PCBs along those paths. To evaluate this, we varied the maximum number of execution paths in each task between 1 to 8 and compared the performance of ResilienceP and multi-path ResilienceP analyses. The results are presented in Figure 7.7. We can see in Figure 7.7 that if tasks are only allowed to have a single execution path, both ResilienceP and multi-paths ResilienceP analyses produce identical results. Moreover, for number of execution paths between 2 to 6 the multi-path ResilienceP analysis tend to produce better results than the ResilienceP analyses. However, for a further increase in number of paths, the difference between the ResilienceP and multi-paths ResilienceP analyses tend to disappear. The is due to the fact that when the number of paths increase, it becomes more probable that there exist several execution flows for which the resilience of the PCBs is low. The function `FindPathsCombinations(·)` of Algorithm 7.1 will then more easily return very

long execution flows with low resilience. This eventually leads to account for the same number of evictions of PCBs as under the ResilienceP analysis.

7.7 Chapter Summary

In this chapter, we proposed a solution to analyze cache persistence for set-associative caches in the context of the WCRT analysis of FPPS. We showed how persistent cache blocks of tasks can be determined when considering set-associative caches. We then presented three different approaches to calculate the CPRO under set-associative caches. Our first analyses, i.e., the PCB-ECB approach, is very coarse-grain. To improve the analysis performance, we explained how the resilience of PCBs can be computed and factored in the analysis. Therefore, the resulting ResilienceP analysis performs considerably better. Lastly, the multi-path ResilienceP analysis uses the variation in the resilience of PCBs over different job execution flows to derive an even tighter CPRO bound. The experimental evaluation shows that our proposed approaches result in up to 22 percentage point higher task set schedulability than the state-of-the-art analyses.

Chapter 8

Tightening the Bound on Inter-task Cache Interference for Multilevel Caches

In Chapter 4-7, we focused on bounding the inter-task cache interference considering only a single-level cache. However, modern computing platforms usually use a hierarchy of cache memories which poses additional challenges during the analysis of WCET/WCRT of tasks. These challenges stem from the computation of intra- and inter-task cache interference for multiple cache levels and their integration when performing timing analysis of tasks. As previously mentioned in Section 3.1.4, several different approaches have been proposed in literature to bound the intra-task cache interference considering multilevel caches (Ferdinand and Wilhelm, 1999; Theiling, 2002; Hardy and Puaut, 2008; Sondag and Rajan, 2010) however, the literature on the computation of inter-task cache interference is relatively scarce. In Section 3.2.3, we have briefly discussed the only two existing approaches (Chattopadhyay and Roychoudhury, 2014; Zhang and Koutsoukos, 2016) in the state-of-the-art that focus on the computation of inter-task cache interference (more specifically CRPDs). It has been shown in (Chattopadhyay and Roychoudhury, 2014; Zhang and Koutsoukos, 2016) that when considering multilevel caches, a precise CRPD bound can only be obtained by accurately quantifying the indirect effect of preemption (see Definition 3.9) and because of the indirect effect of preemption, the traditional UCB concept used to analyze CRPD in single-level caches is hard to use in case of multilevel caches. Consequently, the works in (Chattopadhyay and Roychoudhury, 2014; Zhang and Koutsoukos, 2016) introduce the notion of multilevel UCBs (see Definition 3.10) and that of useful positive references (see Definition 3.11) to compute CRPDs. However, these existing approaches (Chattopadhyay and Roychoudhury, 2014; Zhang and Koutsoukos, 2016) may still result in generating imprecise CRPD bounds mainly due to two reasons, i.e.,

- Due to an overestimation in the number of memory blocks that can contribute to the indirect effect of preemption suffered by a memory block. This overestimation in the indirect effect of preemption suffered by the tasks may lead to imprecise CRPD bounds.

- When accounting for CRPD due to a memory block that has multiple references categorized as L2-hits in the absence of preemption (but may result in L2-misses after preemption), the existing analysis (Chattopadhyay and Roychoudhury, 2014; Zhang and Koutsoukos, 2016) assume that all references to that memory block can be impacted due to a single preemption and therefore contribute to CRPD. This leads to pessimistic CRPD bounds considering that multiple references to the same memory block may result in L2 cache hits but not all those references are impacted due to a single preemption and therefore may not contribute to the CRPD.

Building on the above two points, in this chapter we aim to reduce the pessimism in the computation of inter-task cache interference for multilevel caches by providing a tighter bound on the CRPD suffered by the tasks. The main contributions of this chapter are as follows:

1. We define the notion of useful cache blocks (UCBs) for multilevel caches based on the cache level from which those UCBs may be re-used. We then show how these UCBs can be determined considering a two-level non-inclusive cache hierarchy;

2. we present a new approach to bound the indirect effect of preemption suffered by memory blocks. We show that a tighter bound on the indirect effect of preemption can be obtained by calculating the indirect effect of preemption that can be caused instead of calculating the indirect effect of preemption that can be suffered due to preemptions;

3. we present a new analysis to compute the CRPD due to memory blocks that were categorized as L2-hits in the absence of preemption but may become L2-misses due to preemption. Our approach identifies how many references to such memory blocks can be impacted due to a preemption and therefore may contribute to the CRPD;

4. we incorporate the CRPD bounds resulting from our proposed approach into a WCRT based schedulability analysis and perform an extensive experimental evaluation that compares the performance of our analysis against the state-of-the-art CRPD analysis presented in (Chattopadhyay and Roychoudhury, 2014). The results show that our proposed CRPD analysis results in up to 20 percentage points higher task set schedulability than the CRPD analysis in (Chattopadhyay and Roychoudhury, 2014).

8.1 Assumptions on the System Model

In this chapter, we make the following assumptions on the systems model:

- We focus on a single-core processor having two-level *non-inclusive* cache hierarchy (i.e., we consider only L1 and L2 caches). Non-inclusive cache hierarchy implies that the content in the L1 cache may or may not be duplicated in the L2 cache. We only consider instruction references and assume that both L1 and L2 caches are set-associative and use the Least-Recently-Used (LRU) cache replacement policy. W_1 and W_2 respectively denote the number

of cache ways or cache associativity of L1 and L2 caches. The set of all L1/L2 cache sets is denoted by $\mathbb{S}_1/\mathbb{S}_2$. The total number of cache sets in the L1 and L2 are given by $|\mathbb{S}_1|$ and $|\mathbb{S}_2|$ respectively. Specifically, we focus on the following cache configuration, i.e., $|\mathbb{S}_1| \leq |\mathbb{S}_2|$ and $W_1 \leq W_2$.

- We consider a task set Γ comprising of n sporadic tasks $\{\tau_1, \tau_2, ...\tau_n\}$. Each task $\tau_i \in \Gamma$ is defined using a triplet, i.e., C_i, T_i and D_i. C_i denote the worst-case execution time (WCET) of task τ_i, T_i is its minimum inter-arrival time and D_i is the relative deadline of each job of τ_i. We assume tasks have constrained deadlines, i.e., $D_i \leq T_i$. R_i denote the WCRT of τ_i. Furthermore, tasks can be scheduled using any fixed-priority preemptive scheduling (FPPS) algorithm such as Rate or Deadline Monotonic (Liu and Layland, 1973).

- We assume that the set of all memory blocks accessed by a task τ_i during its execution is given by $\mathbb{M}_i = \{m_{1,i}, m_{2,i}, ..., m_{z,i}\}$. For any memory reference, L1 cache is always accessed. If a memory block is not available in the L1 cache but it is available in the L2 cache (i.e., an L1 cache miss but a L2 cache hit), that memory block will be first loaded into the L1 cache. The time needed to load that block from L2 cache to L1 cache is given by d_{L1}. If the required memory block is not present in both L1 and L2 caches (i.e. an L2 cache miss), it is loaded from the main memory to both cache levels. The time needed to load/reload that block from the main memory to both cache levels is given by $d_{L1} + d_{L2}$, where d_{L2} denote the L2 cache miss penalty.

The list of important symbols used in this chapter is provided in Table 8.1.

Table 8.1: List of important symbols used in Chapter 8

Symbol	Description
Γ	Task set of size n
W_1	The number of cache ways or the associativity of the L1 cache
W_2	The number of cache ways or the associativity of the L2 cache
$\mathbb{S}_1$	The set of all cache sets in the L1 cache
$\mathbb{S}_2$	The set of all cache sets in the L2 cache
τ_i	Task with index i
C_i	Worst-case execution time of task τ_i in isolation
T_i	Minimum inter-arrival time of task τ_i
D_i	Relative deadline of task τ_i
R_i	Worst-case response time of task τ_i
$\mathbb{M}_i$	Set of all memory blocks accessed by task τ_i during its execution
$m_{x,i}$	Any memory block x used by task τ_i during its execution
$\text{hp}(i)$	The set of tasks with higher priority than τ_i

Continued on next page

Table 8.1 – continued from previous page

Symbol	Description
$\mathrm{hep}(i)$	The set of tasks with higher priority than τ_i including τ_i, i.e., $\mathrm{hep}(i) = \mathrm{hp}(i) \cup \tau_i$.
$\mathrm{aff}(i,j)$	The set of intermediate tasks (including τ_i) that may preempt τ_i but may themselves be preempted by some higher priority task τ_j.
d_{L1}	L1 cache miss penalty
d_{L2}	L2 cache miss penalty
S	Any set in the L1/L2 caches
$CU_{m_{x,i}}^{P,1}$	Maximum LRU-age of a memory block $m_{x,i}$ of task τ_i in the L1 cache w.r.t a program point P
$CU_{m_{x,i}}^{P,2}$	Maximum LRU-age of a memory block $m_{x,i}$ of task τ_i in the L2 cache w.r.t a program point P
$MayACS_{e,j,1}(S)$	Set of all memory blocks that may be cached in the L1 cache set S during the execution of task τ_j
$MayACS_{e,j,2}(S)$	Set of all memory blocks that may be cached in the L2 cache set S during the execution of task τ_j
$S_{m_{x,i},1}$	Cache set where memory block $m_{x,i}$ of task τ_i is mapped in the L1 cache
$S_{m_{x,i},2}$	Cache set where memory block $m_{x,i}$ of task τ_i is mapped in the L2 cache
$ECB_j^{S_{m_{x,i},1}}$	Set of ECBs of task τ_j that map to the same L1 cache set as memory block $m_{x,i}$ of task τ_i
$ECB_j^{S_{m_{x,i},2}}$	Set of ECBs of task τ_j that map to the same L2 cache set as memory block $m_{x,i}$ of task τ_i
$ID_{m_{y,i}}^{r,P}$	Set of memory blocks that contribute to the indirect effect of preemption suffered by any memory block $m_{y,i} \in \mathbb{M}_i$ (accessed at any program point r) due to preemption of task τ_i by a higher priority task $\tau_j \in \mathrm{hp}(i)$ at a preemption point P.
$\mathbb{R}$	Set of program locations where a memory block $m_{y,i} \in \mathbb{M}_i$ may be accessed by task τ_i after its preemption by a higher priority task $\tau_j \in \mathrm{hp}(i)$ at a preemption point P. All program location in $\mathbb{R}$ must be reachable from the preemption point P.
$D_{f,r}$	Set of all those memory blocks of task τ_i that have at least one reference categorized as an L1 cache hit starting from the entry node of τ_i and ending at any program location r.
$ID_{m_{y,i}}^{max,P}$	The maximum indirect effect of preemption suffered by any memory block $m_{y,i} \in \mathbb{M}_i$ (accessed at any program point r) due to preemption of task τ_i by a higher priority task $\tau_j \in \mathrm{hp}(i)$ at a preemption point P.

Continued on next page

Table 8.1 – continued from previous page

Symbol	Description
$CRT_{i,j}^{P,1}$	CRPD cost due to all the memory blocks of task τ_i that may suffer only a L1 cache miss penalty due to preemption by a higher priority task $\tau_j \in \text{hp}(i)$ at a program point P.
$CRT_{i,j}^{P,2}$	CRPD cost due to all the memory blocks of task τ_i that may suffer both L1 and L2 cache miss penalties due to preemption by a higher priority task $\tau_j \in \text{hp}(i)$ at a program point P.
$ICRT_{i,j}^{P,1}$	CRPD cost due to first access to all the memory references of task τ_i that were L1 cache misses and L2 cache hits in the absence of preemption but may suffer L2 cache miss penalties due to preemption by a higher priority task $\tau_j \in \text{hp}(i)$ at a program point P.
$ICRT_{i,j}^{P,2}$	CRPD cost due to subsequent accesses (excluding the first access) to all the memory references of task τ_i that were L1 cache misses and L2 cache hits in the absence of preemption but may suffer L2 cache miss penalties due to preemption by a higher priority task $\tau_j \in \text{hp}(i)$ at a program point P.
$\mathbb{P}$	Set of all program points in task τ_i
$\mathbb{P}_2$	Set of all program location of task τ_i with L1 cache misses and L2 cache hits, that are reachable from the preemption point P.
$MustAge(m_{y,i}, P, l)$	Function that returns the LRU-age of a memory block $m_{y,i}$ in the level-l cache at any program location P in task τ_i. The LRU-age is derived using the Must-cache analysis (Hardy and Puaut, 2008).
$IL2_{ind}$	Upper bound on the number of L2 cache misses to any memory reference solely due to the indirect effect of preemption.
$UCB_{i,1}^{P}$	Set of L1-UCBs of task τ_i at a program point P.
$UCB_{i,2}^{P}$	Set of L2-UCBs of task τ_i at a program point P.
$\gamma_{i,i}^{H}$	CRPD cost suffered by task τ_i due to a single preemption by any higher priority task $\tau_j \in \text{hp}(i)$ when considering a two-level non-inclusive cache hierarchy. $\gamma_{i,i}^{H}$ is computed using the CRPD analysis proposed in (Chattopadhyay and Roychoudhury, 2014).
$Ind_{m_{y,i}}^{P}$	Set of memory blocks that contribute to the indirect effect of preemption suffered by any memory block $m_{y,i} \in \mathbb{M}_i$ due to a single preemption of task τ_i by any higher priority task $\tau_j \in \text{hp}(i)$ at a preemption point P.
$FA_{m_{x,i}}^{P}$	Program location where any memory block $m_{x,i}$ of task τ_i is first accessed after the preemption point P.
$\mathbb{P}_1$	Set of all program locations between the preemption point P and program point $FA_{m_{x,i}}^{P}$

Continued on next page

Table 8.1 – continued from previous page

Symbol	Description
$Ind^{mul,P}_{m_{y,i}}$	Set of memory blocks that contribute to the indirect effect of preemption suffered by any memory block $m_{y,i} \in \mathbb{M}_i$ due to multiple preemptions of task τ_i by any higher priority task $\tau_j \in \text{hp}(i)$ w.r.t a preemption point P.
$\gamma^{P,L_1}_{i,j}$	CRPD cost due to all L1-UCBs of task τ_i that are evicted from the L1 cache (but are available in the L2 cache) due a preemption of task τ_i by any higher priority task $\tau_j \in \text{hp}(i)$ at a preemption point P
$\gamma^{P,L_{12}}_{i,j}$	CRPD cost due to all L1-UCBs of task τ_i that are evicted from both L1 and L2 caches due a preemption of task τ_i by any higher priority task $\tau_j \in \text{hp}(i)$ at a preemption point P
$\gamma^{P,L_2}_{i,j}$	CRPD cost due to all L2-UCBs of task τ_i that are evicted from L2 cache due a preemption of task τ_i by any higher priority task $\tau_j \in \text{hp}(i)$ at a preemption point P
$\mathbb{P}_2$	Set of all program locations with L2 cache hits between the preemption point P (including P) and end point e of a task τ_i
$\mathbb{R}^P_{m_{y,i}}$	Set that contains all program locations after the preemption point P where a reference to memory block $m_{y,i}$ is a L2 cache hit
$R^{1,P}_{m_{y,i}}$	First program location after the preemption point P where a reference to memory block $m_{y,i}$ is a L2 cache hit
$\gamma^{\overline{H}}_{i,i}$	CRPD cost suffered by task τ_i due to a single preemption by any higher priority task $\tau_j \in \text{hp}(i)$ when considering a two-level non-inclusive cache hierarchy. $\gamma^{\overline{H}}_{i,i}$ is computed using our proposed CRPD analysis.
$\gamma^{\overline{H},max}_{i,i}$	Maximum CRPD cost suffered by task τ_i due to a preemption by any higher priority task $\tau_j \in \text{hp}(i)$ when considering a two-level non-inclusive cache hierarchy.

8.2 State-of-the-Art CRPD Analysis for Multilevel Caches (Chattopadhyay and Roychoudhury, 2014)

As mentioned previously in Section 3.2.3, there exist only two works in the state-of-the-art (Chattopadhyay and Roychoudhury, 2014; Zhang and Koutsoukos, 2016) that focus on the computation of CRPDs for multilevel caches. However, as we consider a non-inclusive cache hierarchy, we will briefly explain the CRPD analysis of Chattopadhyay et al. (Chattopadhyay and Roychoudhury, 2014). Chattopadhyay et al. (Chattopadhyay and Roychoudhury, 2014) defined the notion of UCBs for two-level caches (see Definition 3.10) and used that notion of UCBs to analyze CRPD for multilevel non-inclusive caches. The sets of UCBs of tasks are computed using the Must-cache analysis (Theiling, 2002; Hardy and Puaut, 2008) along with a backward flow analysis. Recall

that the Must-cache analysis (see Section 3.1.1) determines the set of memory blocks that are in the cache at any given program point under all circumstances, i.e., the reference to such memory blocks will always be cache hits w.r.t that program point. The result of the UCB analysis presented in (Chattopadhyay and Roychoudhury, 2014) is a tuple $CU^{P}_{m_{x,i}} = (CU^{P,1}_{m_{x,i}}, CU^{P,2}_{m_{x,i}})$ that captures the fixed-point on the maximum LRU-age of a memory block $m_{x,i}$ of task τ_i at a program point P in both L1 and L2 caches. If memory block $m_{x,i}$ is not present in L1 cache w.r.t a program point P, $CU^{P}_{m_{x,i}}$ is given by the tuple $(\infty, CU^{P,2}_{m_{x,i}})$, where ∞ represent all scenarios where $m_{x,i}$ may not be present in the L1 cache, i.e., $CU^{P,1}_{m_{x,i}} \geq W_1$. Similarly, if $m_{x,i}$ is not present in the L2 cache, $CU^{P}_{m_{x,i}}$ is given by $(CU^{P,1}_{m_{x,i}}, \infty)$. Note that according to the state-of-the-art definition of two-level UCBs (i.e., Definition 3.10), any memory block $m_{x,i}$ of task τ_i can only be categorized as a UCB at a program point P if $CU^{P}_{m_{x,i}} \neq (\infty, \infty)$.

To compute the set of ECBs of the preempting task, e.g., τ_j, the analysis in (Chattopadhyay and Roychoudhury, 2014) use the May-cache analysis (Hardy and Puaut, 2008). Recall that the May-cache analysis (see Section 3.1.2) determines all memory blocks that may be in the cache at a given program point, i.e., it over-approximate the content in the cache w.r.t a program point. For any cache set S, the set of ECBs of the preempting task τ_j in S is computed by applying the May-cache analysis at the end point e of task τ_j. The May-cache state of cache set S at e will always include all possibly accessed memory blocks by τ_j in S. The output of the May-cache analysis w.r.t a cache set S is given by the tuple $MayACS_{e,j}(S) = (MayACS_{e,j,1}(S), MayACS_{e,j,2}(S))$, where $MayACS_{e,j,1}(S)/MayACS_{e,j,2}(S)$ represent the abstract caches state (see Definition 3.1) of a L1/L2 cache set S w.r.t the end point e of task τ_j. Effectively, $MayACS_{e,j,1}(S)/MayACS_{e,j,2}(S)$ contain all memory blocks that may be cached in a L1/L2 cache set S during the execution of task τ_j.

Assuming that the mapping of a memory block $m_{x,i}$ of task τ_i in L1(L2) cache is defined by the tuple $S_{m_{x,i},1}(S_{m_{x,i},2})$ such that $S_{m_{x,i},1}$ denote the cache set where $m_{x,i}$ is mapped in the L1 cache and $S_{m_{x,i},2}$ denote the cache set where $m_{x,i}$ is mapped in the L2 cache. Then, the number of ECBs of a higher priority task $\tau_j \in \text{hp}(i)$ that may overlap with $m_{x,i}$ in L1(L2) cache are given by

$$\begin{aligned} |ECB_j^{S_{m_{x,i},1}}| &= |MayACS_{e,j,1}(S_{m_{x,i},1})| \quad \text{and} \\ |ECB_j^{S_{m_{x,i},2}}| &= |MayACS_{e,j,2}(S_{m_{x,i},2})| \end{aligned} \tag{8.1}$$

where $ECB_j^{S_{m_{x,i},1}}$ $(ECB_j^{S_{m_{x,i},2}})$ is the set of ECBs of task τ_j that map to the same L1(L2) cache set as memory block $m_{x,i}$ of task τ_i.

Using the set of ECBs defined by Equation (8.1) the CRPD analysis in (Chattopadhyay and Roychoudhury, 2014) determines if a memory block $m_{x,i}$ of task τ_i will be evicted from L1(L2) cache due to preemptions. Effectively, it is assumed that $m_{x,i}$ can be evicted from the L1(L2) cache due to a preemption of task τ_i by task $\tau_j \in \text{hp}(i)$ at a program point P, only if the sum of the maximum LRU-age of $m_{x,i}$ in L1(L2) cache at P, i.e., $CU^{P,1}_{m_{x,i}}(CU^{P,2}_{m_{x,i}})$, and the number of ECBs of τ_j that may overlap with $m_{x,i}$ in L1(L2) cache, i.e., $|ECB_j^{S_{m_{x,i},1}}|(|ECB_j^{S_{m_{x,i},2}}|)$, is greater than or equal to the associativity of the L1(L2) cache, i.e., $W_1(W_2)$. Formally, if $CU^{P,l}_{m_{x,i}} + |ECB_j^{S_{m_{x,i},l}}| \geq W_l$

then memory block $m_{x,i}$ of τ_i will be evicted from the level-l cache due to a preemption of task τ_i by any higher priority task $\tau_j \in \text{hp}(i)$ at a preemption point P.

8.2.1 Calculating the Indirect Effect of Preemption

By definition (see Definition 3.9) indirect effect of preemption is experienced when a memory reference that was a L1 cache hit in the absence of preemption access the L2 cache after preemption (e.g., the second reference to memory block A during the preempted execution of task τ_i shown in Figure 3.6). Due to this extra reference to the L2 cache, the amount of intra-task cache conflicts generated in the L2 cache may increase after preemption. This increase in intra-task L2 cache conflicts may result in generating L2 cache misses for memory references that were categorized as L2 cache hits in the absence of preemption (e.g., the second reference to memory block m in the preempted execution of task τ_i shown in Figure 3.6). Consequently, it can be stated the indirect effect of preemption is suffered by all those memory blocks that may be accessed from the L2 cache after preemption. Under the CRPD analysis presented in (Chattopadhyay and Roychoudhury, 2014), the indirect effect of preemption any memory block $m_{y,i} \in \mathbb{M}_i$ (accessed at any program point r) can suffer due to preemption of task τ_i by a higher priority task $\tau_j \in \text{hp}(i)$ at a preemption point P is given by $|ID^{r,P}_{m_{y,i}}|$, where $ID^{r,P}_{m_{y,i}}$ is a set comprising of all memory blocks $m_{x,i} \in \mathbb{M}_i$ that satisfy the following constraint, i.e.,

$$ID^{r,P}_{m_{y,i}} = \left\{ m_{x,i} | m_{x,i} \neq m_{y,i} \wedge m_{x,i} \in D_{f,r} \wedge (S_{m_{x,i},2} = S_{m_{y,i},2}) \wedge CU^{P}_{m_{x,i}} \neq (\infty,\infty) \wedge CU^{P,1}_{m_{x,i}} + |ECB^{S_{m_{x,i},1}}_{j}| > W_1 \right\} \tag{8.2}$$

Equation (8.2) states that any memory block $m_{x,i} \in \mathbb{M}_i$ can cause an indirect effect of preemption on any other memory block $m_{y,i} \in \mathbb{M}_i$ accessed at any program point r after preemption, if $m_{x,i}$ satisfy all four conditions in Equation (8.2). These conditions are explained as follows:

- First condition, i.e., $m_{x,i} \neq m_{y,i}$, implies that the same memory blocks can not cause an indirect effect of preemption on each other, i.e., if $m_{x,i} = m_{y,i}$ and the reference to $m_{x,i}$ was a L1 cache hit in the non-preempted execution of τ_i but the same reference becomes a L1 cache miss after preemption of τ_i by τ_j. Then, $m_{x,i}$ can be reloaded either from the L2 cache or from the main memory. However, in both cases the state of L1/L2 cache w.r.t $m_{x,i}$ will be the same, i.e., $m_{x,i}$ will be the youngest element in both L1 and L2 cache after the reload from L2 cache or main memory. Therefore, if $m_{x,i} = m_{y,i}$, $m_{y,i}$ can not suffer an indirect effect of preemption due to $m_{x,i}$.

- The second condition, i.e., $m_{x,i} \in D_{f,r}$, implied that $m_{x,i}$ can only cause an indirect effect of preemption on $m_{y,i}$, i.e., increasing the LRU-age of $m_{y,i}$ in L2 cache, if $m_{x,i} \in D_{f,r}$. Where $D_{f,r}$ is a set of all those memory blocks; (i) that must be accessed along some path starting from the entry node of τ_i and ending at r, and (ii) that must have at least one reference categorized as a L1 cache hit (by the intra-task cache analysis) in the absence of preemption and that reference must be reachable from program point r. Effectively, $D_{f,r}$ includes all memory blocks of task τ_i that have at least one reference categorized as a L1 cache hit along

any execution path of τ_i starting from the entry node of τ_i and ending at r. In (Chattopadhyay and Roychoudhury, 2014) $D_{f,r}$ is computed using a forward-flow analysis (along with the Must-cache analysis (Theiling, 2002)) which starts from the entry point of task τ_i and ends at r.

- Third condition, i.e., $S_{m_{x,i},2} = S_{m_{y,i},2}$, implies that any memory block $m_{x,i} \in \mathbb{M}_i$ can only cause an indirect effect of preemption on any other memory block $m_{y,i} \in \mathbb{M}_i$ if both $m_{x,i}$ and $m_{y,i}$ are mapped to the same L2 cache set. This is intuitive, since each cache set of a set-associative cache can be analyzed independently. Therefore, $m_{y,i}$ can only suffer the indirect effect of preemption from all memory blocks that are mapped to the same L2 cache set as $m_{y,i}$.

- $CU^{P}_{m_{x,i}} \neq (\infty,\infty)$ implies that for $m_{x,i}$ to cause an indirect effect after a preemption at P, $m_{x,i}$ must be a UCB at P. The last condition, i.e., $CU^{P,1}_{m_{x,i}} + |ECB_j^{S_{m_{x,i},1}}| > W_1$, states that for $m_{x,i}$ to cause an indirect effect of preemption on $m_{y,i}$, $m_{x,i}$ must be evicted from the L1 cache due to preemption at P by the higher priority task τ_j. If $m_{x,i}$ is evicted from the L1 due to preemption at P, the first access to $m_{x,i}$ after preemption will result in a L1 miss. Hence, $m_{x,i}$ will be accessed either from L2 or is reloaded from the main memory. In both cases, $m_{x,i}$ will cause an indirect effect on all memory blocks (including $m_{y,i}$) that map to the same L2 cache set as $m_{x,i}$. Finally, the set $ID^{r,P}_{m_{y,i}}$ contains all memory blocks $m_{x,i} \in \mathbb{M}_i$ that satisfy all the above mentioned conditions.

Considering that a memory block $m_{y,i}$ can be accessed at several different program locations (i.e., r) that are reachable from the preemption point P, the worst-case indirect effect of preemption that $m_{y,i}$ can suffer is computed by maximizing Equation (8.2) over all program locations where $m_{y,i}$ may be accessed and are reachable from the preemption point P. Let $\mathbb{R}$ denote the set of all such program locations then, the worst-case indirect effect $m_{y,i}$ may suffer due to a preemption at program point P is given by $ID^{max}_{m_{y,i},P}$, where

$$ID^{max,P}_{m_{y,i}} = \max_{\forall r \in \mathbb{R}} |ID^{r,P}_{m_{y,i}}| \tag{8.3}$$

Example 8.1. *To demonstrate how the indirect effect of preemption is computed for the CRPD analysis of (Chattopadhyay and Roychoudhury, 2014), let us use Equation (8.2) to calculate the indirect effect of preemption suffered by memory block m, i.e., $ID^{r,P}_{m}$, for the execution scenario shown in Figure 3.6. We have $D_{f,r} = \{A\}$ and memory block A satisfy all constraints in Equation (8.2) in case of a preemption at P. Therefore, we have $ID^{r,P}_{m} = \{A\}$ and $ID^{max,P}_{m} = 1$.*

More details on the formulation of Equation (8.2) and (8.3) can be found in (Chattopadhyay and Roychoudhury, 2014).

8.2.2 CRPD Computation

Under the analysis presented in (Chattopadhyay and Roychoudhury, 2014), the CRPD task τ_i may suffer due to a single preemption by a higher priority task $\tau_j \in \text{hp}(i)$ at any arbitrary preemption point P comprises of four components, i.e., $CRT_{i,j}^{P,1}$, $CRT_{i,j}^{P,2}$, $ICRT_{i,j}^{P,1}$, and $ICRT_{i,j}^{P,2}$. $CRT_{i,j}^{P,1}$ captures the CRPD due to all those memory blocks of τ_i that are evicted from the L1 cache due to preemption by τ_j at P but may still be available in the L2 cache, i.e.,

$$CRT_{i,j}^{P,1} = d_{L1} \times |\mathbb{M}_{i,L1}| \tag{8.4}$$

where $\mathbb{M}_{i,L1}$ is the set of all memory blocks in $\mathbb{M}_i$ that might be L1 cache hits in the absence of preemption, but may suffer L1 miss penalties after preemption, i.e.,

$$\mathbb{M}_{i,L1} = \left\{ m_{x,i} | CU_{m_{x,i}}^{P} \neq (\infty,\infty) \wedge CU_{m_{x,i}}^{P,1} + |ECB_j^{S_{m_{x,i},1}}| \geq W_1 \wedge CU_{m_{x,i}}^{P,2} + |ECB_j^{S_{m_{x,i},2}}| + ID_{m_{x,i}}^{max,P} < W_2 \right\} \tag{8.5}$$

Similarly, $CRT_{i,j}^{P,2}$ accounts for the CRPD due to all those memory blocks of τ_i that are evicted from both L1 and L2 caches due to preemption, i.e.,

$$CRT_{i,j}^{P,2} = (d_{L1} + d_{L2}) \times |\mathbb{M}_{i,L1L2}| \tag{8.6}$$

where

$$\mathbb{M}_{i,L1L2} = \left\{ m_{x,i} | CU_{m_{x,i}}^{P} \neq (\infty,\infty) \wedge CU_{m_{x,i}}^{P,1} + |ECB_j^{S_{m_{x,i},1}}| \geq W_1 \wedge CU_{m_{x,i}}^{P,2} + |ECB_j^{S_{m_{x,i},2}}| + ID_{m_{x,i}}^{max,P} \geq W_2 \right\} \tag{8.7}$$

By the definition of UCBs provided in (Chattopadhyay and Roychoudhury, 2014) (Definition 3.10), all memory blocks in $\mathbb{M}_i$ that are L2 cache hits in the absence of preemption are not categorized as UCBs. Therefore, to account for the CRPD due to all those memory blocks, the analysis in (Chattopadhyay and Roychoudhury, 2014) checks all program locations with memory references that were L2 cache hits in the absence of preemption but may become L2 cache misses due to preemption. The set of all such program location of task τ_i that are reachable from the preemption point P is denoted by $\mathbb{P}_2$.

The CRPD analysis in (Chattopadhyay and Roychoudhury, 2014) assume that any reference to a memory block $m_{y,i} \in \mathbb{M}_i$ at any program point $r \in \mathbb{P}_2$, can suffer multiple L2 cache misses due to preemptions. The first reference to $m_{y,i}$ after preemption may suffer a L2 cache miss due to the combined affect of the preempting task, i.e., due to ECBs of the preempting task τ_j that map to the same cache set as $m_{y,i}$ given by $|ECB_j^{S_{m_{y,i},2}}|$, and the indirect effect of preemption suffered by $m_{y,i}$ at that program point $r \in \mathbb{P}_2$, i.e., $|ID_{m_{y,i}}^{r,P}|$. Consequently, $ICRT_{i,j}^{P,1}$ captures the resulting CRPD cost for the first reference to memory block $m_{y,i}$ after preemption, i.e.,

$$ICRT_{i,j}^{P,1} = \sum_{r \in \mathbb{P}_2} \begin{cases} 0 & \text{if } MustAge(m_{y,i}, r, 2) + |ECB_j^{S_{m_{y,i},2}}| + |ID_{m_{y,i}}^{r,P}| < W_2 \\ d_{L2} & \text{otherwise.} \end{cases} \tag{8.8}$$

where $MustAge(m_{y,i}, r, 2)$ is the LRU-age of memory block $m_{y,i}$ in the L2 cache immediately before the program point $r \in \mathbb{P}_2$ and is computed using the Must-cache analysis (Hardy and Puaut, 2008). The CRPD analysis in (Chattopadhyay and Roychoudhury, 2014) assume that the same memory block, i.e., $m_{y,i}$, can also be evicted solely due to the indirect effect of preemption. Consequently, $ICRT^{P,2}_{i,j}$ captures the CRPD cost due to references to $m_{y,i}$ that may suffer an L2 cache miss penalty after preemption only due to the indirect effect of preemption, where $ICRT^{P,2}_{i,j}$ is given as

$$ICRT^{P,2}_{i,j} = IL2_{ind} \times \sum_{r \in \mathbb{P}_2} \begin{cases} 0 & \text{if } MustAge(m_{y,i}, r, 2) + |ID^{r,P}_{m_{y,i}}| < W_2 \\ d_{L2} & \text{otherwise.} \end{cases} \tag{8.9}$$

In Equation (8.9), $IL2_{ind}$ denotes an upper bound on the number of L2 cache misses to any memory reference solely due to the indirect effect of preemption. It is proved in (Chattopadhyay and Roychoudhury, 2014) that if the cache configuration is such that $|\mathbb{S}_1| \leq |\mathbb{S}_2|$ and $W_1 \leq W_2$ then the value of $IL2_{ind}$ is upper bounded by 1, i.e., $IL2_{ind} \leq 1$.

Finally, the worst-case CRPD suffered by task τ_i due to a single preemption by task τ_j is given by maximizing Equation (8.4)-(8.9) over the set of all program point $\mathbb{P}$, i.e.,

$$\gamma^H_{i,j} = \max_{P \in \mathbb{P}} \left(CRT^{P,1}_{i,j} + CRT^{P,2}_{i,j} + ICRT^{P,1}_{i,j} + ICRT^{P,2}_{i,j} \right) \tag{8.10}$$

Readers are direct to (Chattopadhyay and Roychoudhury, 2014) for details on the formulation of Equation (8.1)-(8.10).

8.3 Multilevel Useful Cache Blocks

The state-of-the-art definition of UCBs for two-level caches (i.e., Definition 3.10) assume that any memory block $m_{x,i}$ of task τ_i can only be categorized as a UCB at a program point P, if $m_{x,i}$ is not evicted from both L1 and L2 caches before being reused at a later program point Q. However, considering that multilevel non-inclusive caches do not strictly enforce content inclusion, it is likely that memory block $m_{x,i}$ can be available in only one cache level (e.g., L1 or L2) at program point Q and hence will be re-used from that cache level. For example, in Figure 3.6 both memory blocks A and m are cached in L1 and L2 at program point P, however, only memory block A remains cached in both L1 and L2 before its next reuse. Therefore, by Definition 3.10 only memory block A will be categorized as a UCB at program point P even though memory block m is only evicted from L1 cache and is later reused from the L2 cache.

In a non-inclusive multilevel cache, a memory block can be available in any of the cache levels. Therefore, using this insight, we can re-define the notion of UCBs for multilevel caches based on the cache level from which a memory block might be reused. For example, in a two-level cache a memory block $m_{x,i}$ can be re-used either from L1 or L2 cache (i.e., L1 or L2 cache hit). Therefore, $m_{x,i}$ can be categorized as a L1- or L2-UCB. Formally, for a cache with two levels, UCBs can be categorized into:

Definition 8.1 (L1-Useful Cache Blocks (L1-UCBs):)**.** *A memory block $m_{x,i}$ of task τ_i is a L1-UCB w.r.t a program point P if (i) $m_{x,i}$ is cached in L1 at P and (ii) $m_{x,i}$ is reused at a program point Q that must be reachable from P without eviction of $m_{x,i}$ from the L1 cache, i.e., the reference to $m_{x,i}$ at program point Q should be categorized as a L1 hit. The set of memory blocks of task τ_i categorized as L1-UCBs w.r.t a program point P is given by $UCB^P_{i,1}$.*

Definition 8.2 (L2-Useful Cache Blocks (L2-UCBs):)**.** *A memory block $m_{y,i}$ of task τ_i is a L2-UCB at program point P if (i) $m_{y,i}$ is cached at P in L2 and (ii) $m_{y,i}$ is reused at a program point Q that must be reachable from P without eviction of $m_{y,i}$ from the L2 cache, i.e., the reference to $m_{y,i}$ at program point Q is categorized as a L2 cache hit. The set of memory blocks of task τ_i categorized as L2-UCBs w.r.t a program point P is denoted by $UCB^P_{i,2}$.*

Based on the above definitions, a memory block can be both a L1-UCB and a L2-UCB w.r.t a program point P, i.e., the reference to that memory block at any program point Q that is reachable from P can be a cache hit in both L1 and L2 caches. However, in this case, that memory block will only be considered as a L1-UCB as it will always be accessed from the L1 cache during non-preempted execution of a task. Note that the above definitions of UCBs can also be generalized to l-level caches.

It is argued in the existing works (Chattopadhyay and Roychoudhury, 2014; Zhang and Koutsoukos, 2016) that the concept of UCBs is difficult to use for the analysis of CRPDs for multilevel cache. However, as we later show in Section 8.5, by categorizing UCBs based on the cache level from which they might be re-used, the UCB concept can be used to compute CRPD for multilevel caches.

8.3.1 Finding L1/L2-UCBs

The set of L1-UCBs and L2-UCBs of a task τ_i at a program point P can be determined by using the Must-cache analysis (Hardy and Puaut, 2008) along with a backward flow analysis as proposed in (Chattopadhyay and Roychoudhury, 2014). As mentioned earlier, the result of the UCB analysis in (Chattopadhyay and Roychoudhury, 2014) is a tuple $CU^P_{m_{x,i}} = (CU^{P,1}_{m_{x,i}}, CU^{P,2}_{m_{x,i}})$ that captures the fixed-point on the maximum LRU-age of a memory block $m_{x,i}$ of task τ_i at a program point P. Consequently, the set of L1-UCBs of a task τ_i at a program point P can be computed using the following expression

$$UCB^P_{i,1} = \left\{ m_{x,i} | m_{x,i} \in \mathbb{M}_i \wedge \left(\left(CU^{P,1}_{m_{x,i}} \neq \infty \wedge CU^{P,2}_{m_{x,i}} \neq \infty \right) \vee \left(CU^{P,1}_{m_{x,i}} \neq \infty \wedge CU^{P,2}_{m_{x,i}} = \infty \right) \right) \right\} \tag{8.11}$$

Equation (8.11) implies that given a program point P, all memory blocks $m_{x,i} \in \mathbb{M}_i$ with a maximum LRU-age in L1 $\neq \infty$ are L1-UCBs of τ_i w.r.t P.

Similarly, the set of L2-UCBs of task τ_i at a program point P can be computed as follows

$$UCB^P_{i,2} = \left\{ m_{y,i} | m_{y,i} \in \mathbb{M}_i \wedge \left(CU^{P,1}_{m_{y,i}} = \infty \wedge CU^{P,2}_{m_{y,i}} \neq \infty \right) \right\} \tag{8.12}$$

Equation (8.12) implies that any memory block $m_{y,i} \in \mathbb{M}_i$ is a L2-UCBs at a program point P, if $m_{y,i}$ may be evicted from L1 along some path reachable from P, i.e., $CU_{m_{y,i}}^{P,1} = \infty$, but it remains cached in L2, i.e., $CU_{m_{y,i}}^{P,2} \neq \infty$. Similarly, to compute the number of ECBs of a higher priority task $\tau_j \in \mathrm{hp}(i)$ that may map to the same L1(L2) cache set as the L1/L2-UCBs of task τ_i, we will also use the May-cache analysis (Hardy and Puaut, 2008), i.e., Equation (8.1).

8.4 Tightening the Bound on the Indirect Effect of Preemption

The state-of-the-art approach to calculate the indirect effect of preemption (i.e., Equation. (8.2)) provides a sound estimate on the indirect effect of preemption that can be suffered by the memory blocks. However, that approach may result in overestimating the indirect effect of preemption. This overestimation is due to an over-approximation on the set of memory blocks that can cause the indirect effect of preemption. To illustrate consider the following example:

Example 8.2. *Figure 8.1 shows a sequence of memory reference (from left to right) during non-preempted (top) and preempted (bottom) execution of task τ_i. We assume that L1 and L2 are two-way set-associative LRU-caches, i.e., $W_1 = W_2 = 2$. All memory blocks used by task τ_i, i.e., A, B and m, are mapped to the same L1 and L2 cache set. For clarity, we only focus on the computation of indirect effect of preemption suffered by memory block m due to preemption at program point P. We can see in Figure 8.1 that the second reference to memory block m is a*

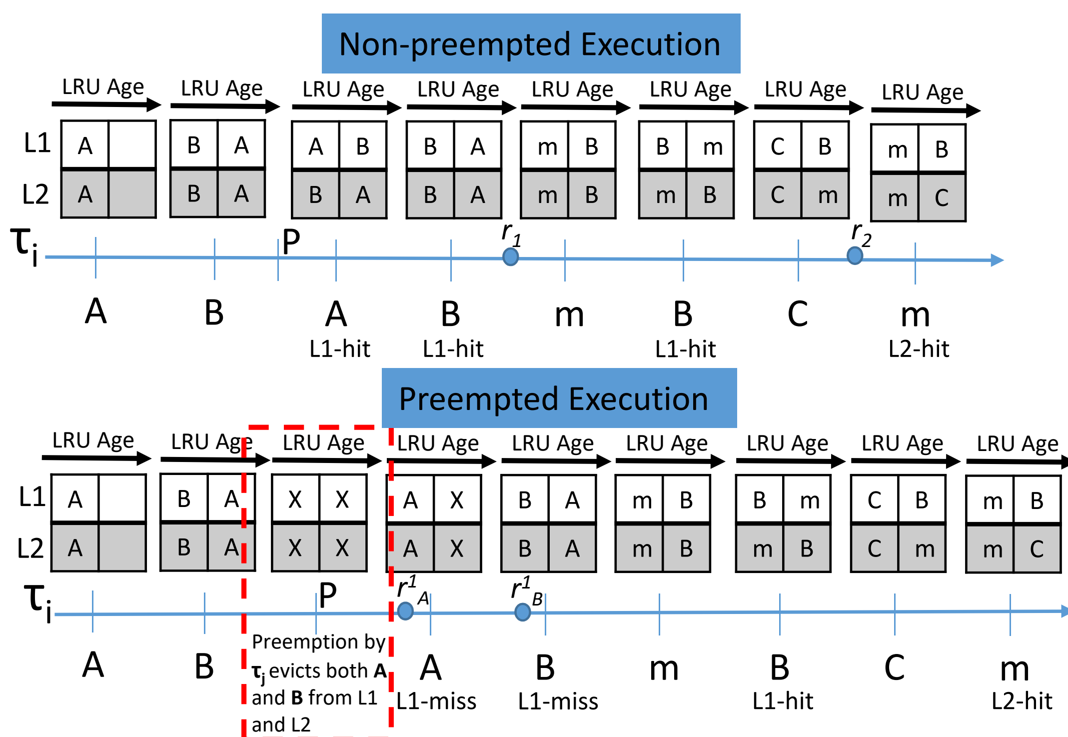


Figure 8.1: Highlighting the pessimism in the calculation of indirect effect of preemption by (Chattopadhyay and Roychoudhury, 2014).

L2-hit in both non-preempted and preempted execution of τ_i. However, if we use analysis of (Chattopadhyay and Roychoudhury, 2014) to calculate the indirect effect of preemption that can be suffered by memory block m due to a preemption at P, we will get $D_{f,r_1} = D_{f,r_2} = \{A,B\}$ w.r.t program locations r_1 and r_2 where m is accessed after preemption. Moreover, both memory blocks A and B satisfy all constraints in Equation (8.2), i.e., (i) A and B are UCBs at program point P; (ii) both A and B will be evicted from L1 (and L2) cache due to preemption at program point P by a higher priority task $\tau_j \in \text{hp}(i)$ and (iii) both A and B map to the same L2 cache set as m. Therefore, the analysis in (Chattopadhyay and Roychoudhury, 2014) will conclude that both memory blocks A and B can cause an indirect effect of preemption on memory block m, i.e., $ID_m^{r_1,P} = ID_m^{r_2,P} = \{A,B\}$ and $ID_m^{max,P} = 2$ (by using Equation (8.2) and (8.3)) after a preemption at P. Consequently, if we use this bound on the indirect effect of preemption of m in Equation (8.9), it results that m will be evicted from L2 solely due to indirect effect of preemption, i.e., $MustAge(m,r_2,2) + |ID_m^{r_2,P}| = 2+2 > W_2$. However, we can see in Figure 8.1 that this does not happen. In fact, the second reference to memory block m remains a L2 cache hit even after preemption because m will not suffer any indirect effect of preemption.

Example 8.2 shows that overestimating the indirect effect of preemption may lead to pessimistic CRPD values. Therefore, we propose an improved analysis that computes a tighter bound on the indirect effect of preemption using Algorithm 8.1. Instead of calculating the indirect effect of preemption that can be suffered by memory blocks, Algorithm 8.1 bounds the indirect effect that can be caused by memory blocks as a result of a preemption. We prove the correctness of Algorithm 8.1 using the following Lemma

Algorithm 8.1 Calculating the indirect effect of preemption caused due to preemption of task τ_i by τ_j at a program point P

Output: The indirect effect of preemption suffered by every memory block $m_{y,i} \in \mathbb{M}_i$ due to preemption at program point P, i.e., $Ind^P_{m_{y,i}}$.

1: **for** $\forall m_{y,i} \in \mathbb{M}_i$ **do**
2: $\quad Ind^P_{m_{y,i}} := \emptyset$
3: **end for**
4: **for** $\forall m_{x,i} \in UCB^P_{i,1}$ **do**
5: $\quad$ **if** $CU^{P,1}_{m_{x,i}} + |ECB_j^{S_{m_{x,i},1}}| \geq W_1$ **then**
6: $\quad\quad FA^P_{m_{x,i}} := GetFirstAccess(m_{x,i},P)$
7: $\quad\quad$ **for** $\forall m_{y,i} \in \mathbb{M}_i \setminus m_{x,i}$ **do**
8: $\quad\quad\quad$ **if** $((MustAge(m_{y,i},FA^P_{m_{x,i}},2) \neq \infty) \wedge (S_{m_{y,i},2} == S_{m_{x,i},2}) \wedge ((MustAge(m_{x,i},FA^P_{m_{x,i}},2) = \infty) \vee (MustAge(m_{x,i},FA^P_{m_{x,i}},2) > MustAge(m_{y,i},FA^P_{m_{x,i}},2))))$ **then**
9: $\quad\quad\quad\quad Ind^P_{m_{y,i}} := Ind^P_{m_{y,i}} \cup m_{x,i};$
10: $\quad\quad\quad$ **end if**
11: $\quad\quad$ **end for**
12: $\quad$ **end if**
13: **end for**

Lemma 8.1. *The indirect effect suffered by any memory block $m_{y,i} \in \mathbb{M}_i$ of task τ_i due to a preemption at any program point P is upper bounded by $|Ind^P_{m_{y,i}}|$, where $Ind^P_{m_{y,i}}$ is a set of memory blocks that can cause the indirect effect of preemption on $m_{y,i}$ and is computed using Algorithm 8.1.*

Proof. **(1).** By the definition of indirect effect of preemption (i.e., Definition 3.9), it can be caused by only those memory blocks that are used from the L1 cache in the absence of preemption (i.e., L1-UCBs) but may be accessed from the L2 or the main memory after preemption. Therefore, for any preemption point P, the set of memory blocks that may cause an indirect effect of preemption on any memory block $m_{y,i} \in \mathbb{M}_i$ is upper bounded by the set of L1-UCBs of τ_i at that preemption point P, i.e., $UCB^P_{i,1}$. Consequently, the external loop in Algorithm 8.1 (i.e., lines 4 to 13) iterates over all memory blocks $m_{x,i} \in UCB^P_{i,1}$.

(2). Any L1-UCB $m_{x,i} \in UCB^P_{i,1}$ can cause an indirect effect of preemption on any other memory block $m_{y,i} \in \mathbb{M}_i$ due to preemption at a program point P only if $m_{x,i}$ is evicted from the L1 cache due to that preemption, i.e., $CU^{P,1}_{m_{x,i}} + |ECB^{S_{m_{x,i},1}}_j| > W_1$. In Algorithm 8.1 line 5 checks this condition for all L1-UCBs in $UCB^P_{i,1}$.

(3). For any memory block $m_{x,i}$ that is a L1-UCB at P the first reachable references to $m_{x,i}$ after P (e.g., at program location $FA^P_{m_{x,i}}$ computed using function $GetFirstAccess(m_{x,i}, P)$ in Algorithm 8.1) determines whether $m_{x,i}$ is evicted from the L1 cache or not and hence it may or may not contribute to the indirect effect of other memory blocks. If $m_{x,i}$ is evicted from the L1 cache due to preemption (i.e., **(2)** holds) then the first access to $m_{x,i}$ after P, i.e., at $FA^P_{m_{x,i}}$, can either be a L2 cache hit or a L2 cache miss. However, in both cases an access to $m_{x,i}$ at program point $FA^P_{m_{x,i}}$ may generate an indirect effect on all memory blocks that are in L2 cache at $FA^P_{m_{x,i}}$, i.e., reloading $m_{x,i}$ from the main memory or from the L2 cache may increase the LRU-age of memory blocks that are already in L2 cache at program point $FA^P_{m_{x,i}}$.

(4). Since $m_{x,i}$ can cause an indirect effect of preemption on only those memory blocks that are in the L2 cache at program point $FA^P_{m_{x,i}}$. The nested loop in Algorithm 8.1 (lines 7 to 11) determines all memory blocks $m_{y,i} \in \mathbb{M}_i \setminus m_{x,i}$ that can suffer indirect effect of preemption due to $m_{x,i}$. The computation is based on four conditions that are explained as follows:

(4.1). $m_{x,i}$ can only cause an indirect effect of preemption on any other memory block $m_{y,i} \in \mathbb{M}_i \setminus m_{x,i}$ if $m_{y,i}$ is in the L2 cache at program point $FA^P_{m_{x,i}}$. This is determined using the Must-age (Hardy and Puaut, 2008) of $m_{y,i}$ at program point $FA^P_{m_{x,i}}$, i.e., $m_{y,i}$ can only suffer an indirect effect of preemption due to $m_{x,i}$ if $(MustAge(m_{y,i}, FA^P_{m_{x,i}}, 2) \neq \infty)$.

(4.2). $m_{x,i}$ can cause an indirect effect of preemption on $m_{y,i}$ if both $m_{x,i}$ and $m_{y,i}$ map to the same L2 cache set, i.e., $S_{m_{x,i},2} == S_{m_{y,i},2}$.

(4.3). If the access to $m_{x,i}$ at program point $FA^P_{m_{x,i}}$ is a L2 cache miss, i.e., $MustAge(m_{x,i}, FA^P_{m_{x,i}}, 2) = \infty$, then $m_{x,i}$ will be reloaded from the main memory to both L2 and L1 caches. This will increase the LRU-age of all existing memory blocks in the L2 cache including $m_{y,i}$. Hence, $m_{y,i}$ will suffer an indirect effect of preemption due to $m_{x,i}$.

(4.4). But, if the access to $m_{x,i}$ at program point $FA^P_{m_{x,i}}$ is a L2 cache hit, then $m_{x,i}$ will be reloaded from L2 to L1 cache. Moreover, in this case an access to $m_{x,i}$ can only change the LRU-ages of memory blocks that are younger than $m_{x,i}$ in the L2 cache. So, $m_{y,i}$ can only suffer an indirect effect of preemption from $m_{x,i}$ if $MustAge(m_{x,i}, FA^P_{m_{x,i}}, 2) > MustAge(m_{y,i}, FA^P_{m_{x,i}}, 2)$ holds. Note that **(4.3)** and **(4.4)** are mutually exclusive, therefore only one of them needs to be true for $m_{x,i}$ to cause an indirect effect of preemption on $m_{y,i}$.

Finally, if all the above conditions hold for any memory block $m_{y,i} \in \mathbb{M}_i \setminus m_{x,i}$, $m_{x,i}$ will be added to the set of memory blocks that can cause an indirect effect of preemption on $m_{y,i}$, i.e., $Ind^P_{m_{y,i}} \cup m_{x,i}$. Consequently, the cardinality of the set $Ind^P_{m_{y,i}}$ upper bounds the indirect effect that can be suffered by any memory block $m_{y,i} \in \mathbb{M}_i$ due to preemption at any program point P. □

Example 8.3. *Using Algorithm 8.1 to calculate the indirect effect of preemption of memory block m in Figure 8.1, we can see that both memory blocks A and B are L1-UCBs at P, i.e., $\{A,B\} \in UCB^P_{i,1}$. Also, both A and B will be evicted from L1 due to preemption and also map to the same L2 cache set as memory block m. However, m is not in the L2 at the first access of memory block A (and B) after preemption point P, i.e., $MustAge(m, FA^P_A, 2) = \infty$ (and $MustAge(m, FA^P_B, 2) = \infty$). Therefore, m will not suffer any indirect effect due to eviction of A and B at preemption point P, i.e., $|Ind^P_m| = 0$.*

8.4.1 Handling Nested/Multiple Preemptions

Algorithm 8.1 computes the indirect effect of preemption that may be suffered by memory blocks of task τ_i due to a single preemption by a higher priority task $\tau_j \in \text{hp}(i)$. However, in reality, τ_i can be preempted several times by the same task or by different tasks during its execution. It has been shown in the state-of-the-art CRPD analysis for single-level set-associative caches (Altmeyer et al., 2010) that multiple preemptions of task τ_i by the same task, e.g., τ_j, does not pose any additional challenges in the computation of CRPD for single-level caches. This is mainly because the May-cache analysis (Theiling et al., 2000) used to compute the set of ECBs of task τ_j that may overlap with the set of UCBs of task τ_i in the L1 cache, over-approximate the set of ECBs of τ_j, i.e., even if certain memory blocks may not be accessed in one job execution of τ_j they do appear in the set of ECBs. Therefore, the contribution every preemption by τ_j can make to the CRPD of τ_i can be analyzed independently. Similarly, it is also shown in (Altmeyer et al., 2010; Altmeyer, 2013) that in case of multiple preemption of task τ_i by different tasks in $\text{hp}(i)$, the CRPD cost can be computed by simulating nested preemptions, i.e., when computing the CRPD due to a single preemption of task τ_i by any higher priority task $\tau_j \in \text{hp}(i)$, it is assumed that τ_j has itself already been preempted by all higher priority tasks in $\text{hp}(j)$. Consequently, the analysis in (Altmeyer et al., 2010; Altmeyer, 2013) that accounts for multiple preemptions, use the union of set of ECBs of all tasks in $\text{hep}(j)$, i.e., $|\bigcup_{\forall h \in \text{hep}(j)} ECB_h^{S_{m_{x,i},1}}|$, instead of only using the set of ECBs of task τ_j, i.e.,$|ECB_j^{S_{m_{x,i},1}}|$, when computing the CRPD due to a single preemption of task τ_i by any higher priority task $\tau_j \in \text{hp}(i)$.

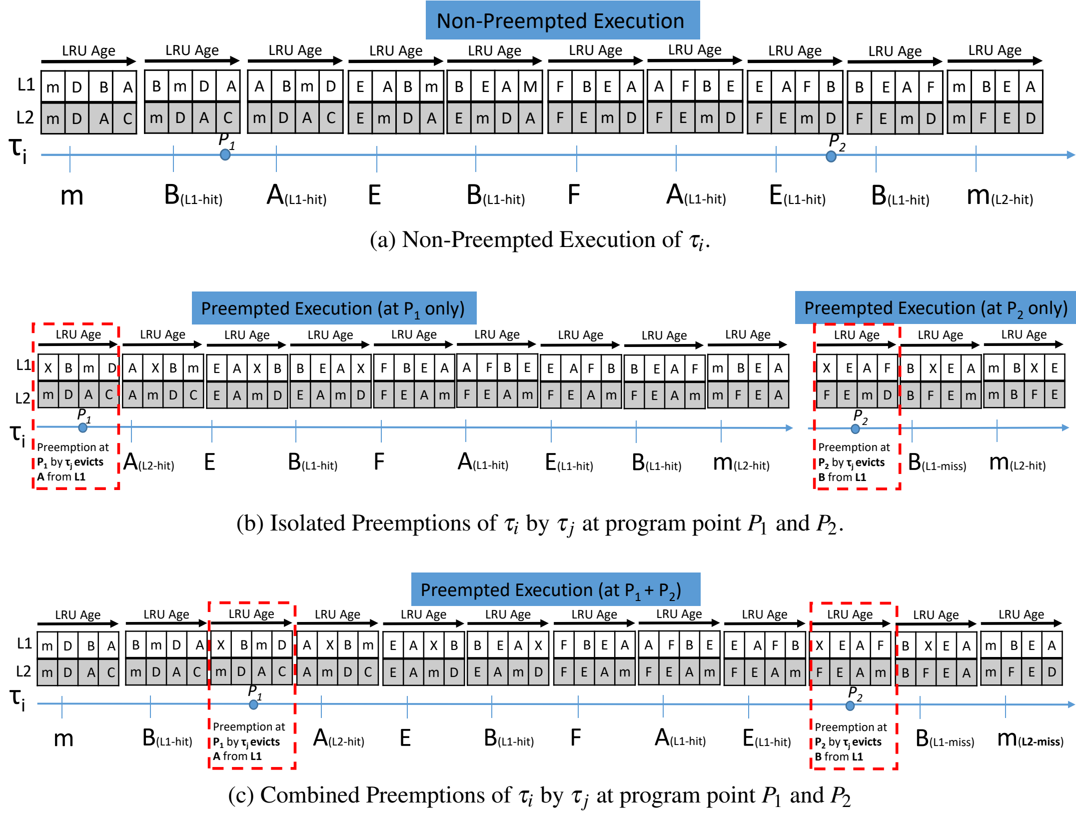
However, when computing CPRD for multilevel caches in the presence of multiple preemptions, only simulating nested preemptions of tasks, i.e., using the union of ECBs of the preempting tasks, may not be enough. This is mainly due to the indirect effect of preemption that exists in multilevel caches, i.e., multiple preemptions by the same or different task(s) may “collaborate" to cause more indirect effect than they would in “isolation". To illustrate this point, consider the example given below:

Example 8.4. *Figure 8.2 shows a sequence of memory references during the execution of a task τ_i considering different preemption scenarios. We assume that L1 and L2 are 4-way set-associative LRU caches and all memory blocks used by task τ_i and the preempting task, e.g., τ_j, map to the same L1 and L2 cache set. Note that we only focus on computing the indirect effect of preemption that can be suffered by memory block m due to preemptions at program points P_1 and P_2.*

During the non-preempted execution of τ_i (see Figure 8.2a), first reference to memory block A after P_1 and the first reference to memory block B after P_2 are L1 cache hits. Moreover, the second reference to memory block m is also a L2 cache hit. We assume that task τ_i can be preempted by a higher priority task $\tau_j \in \text{hp}(i)$ at program points P_1 and P_2 such that τ_j only has one L1 cache conflict with τ_i, i.e., τ_j only loads ECB X in the L1 cache. When considering preemptions of τ_i by τ_j at P_1 and P_2 independently or in isolation (see Figure 8.2b), we can see that by just considering the preemption at P_1, only the first reference to memory block A will be impacted and it will result in a L1 cache miss (but a L2 cache hit). Similarly, a preemption at P_2 in isolation can only impact the first reference to memory block B after P_2 which results in a L2 cache miss. Furthermore, since both preemptions, i.e., at P_1 and P_2, can only evict one L1-UCB, the maximum indirect effect (computed using Algorithm 8.1) that memory block m may suffer due to a preemption at P_1 or P_2 will be 1. Consequently, we can see in Figure 8.2b that the second reference to memory block m will remain a L2 cache hit when considering both preemptions in isolation.

However, when considering consecutive preemptions of task τ_i by task τ_j (see Figure 8.2c) we can see that the preemption at program point P_1 can collaborate with the preemption at P_2. This collaboration generates an indirect effect of 2 on memory block m which results in the eviction of memory block m from the L2 cache after a preemption at P_2 as shown in Figure 8.2c.

Example 8.4 shows that Algorithm 8.1 may underestimate the indirect effect of preemption suffered by memory blocks in the presence of multiple preemptions. This is mainly because when using Algorithm 8.1, the indirect effect of preemption that can be caused at a preemption point P is upper bounded by the number of L1-UCBs of tasks that may be evicted from the L1 cache only due to preemption at P. However, as we just demonstrated in Example 8.4, two consecutive accesses to the memory block under analysis, i.e., the memory block for which the indirect effect is being computed (e.g., *m* in Figure 8.2), may enclose two or more preemption points (e.g., P_1 and P_2 in Figure 8.2) and all L1-UCBs that be may be evicted due to preemptions between those preemption points may contribute to the indirect effect suffered by the memory block under analysis. Therefore, to have a sound estimate on the indirect effect that can be caused due to multiple preemptions w.r.t a program point P, we need to consider all L1-UCBs that may be evicted from the L1 cache between the preemption point under analysis, e.g., P and the program point where the memory block under analysis is first accessed after P, e.g., r. For example, if $m_{y,i}$ is the memory block under analysis which is accessed at any program point r after the preemption point P. Then, the indirect effect of preemption that $m_{y,i}$ can suffer due to one or more preemptions by any higher priority task $\tau_j \in \text{hp}(i)$ at P is upper bounded by the maximum number of L1-UCBs of task τ_i that may be evicted from the L1 cache between program points P and r. To compute the indirect effect of preemption in the presence of multiple preemptions we propose Algorithm 8.2



(a) Non-Preempted Execution of τ_i.

(b) Isolated Preemptions of τ_i by τ_j at program point P_1 and P_2.

(c) Combined Preemptions of τ_i by τ_j at program point P_1 and P_2

Figure 8.2: Multiple preemption scenarios with collaborating and isolated preemptions. The indirect effect of preemption suffered by memory block m due to consecutive preemptions, i.e., at P_1 and P_2, is higher than the indirect effect caused by individual preemptions.

which is an inclusive version of Algorithm 8.1. The major difference between Algorithm 8.1 and 8.2 is the function $GetProgamPoints(P, FA^{P}_{m_{y,i}})$ that computes $\mathbb{P}_1$ (line 4), which is the set of all program locations $P^{'}$ (including P) that are between the preemption point under analysis, i.e., P, and the program point where memory block $m_{y,i}$ is first accessed after P, i.e., $FA^{P}_{m_{y,i}}$. Algorithm 8.2 then computes the indirect effect of preemption for all program locations $P^{'} \in \mathbb{P}_1$ using the exact same steps as used in Algorithm 8.1. Note that the additional condition $m_{x,i} \notin Ind^{mul,P}_{m_{y,i}}$ ensures that every memory block $m_{x,i}$ that is a L1-UCB at any program point between P and $FA^{P}_{m_{y,i}}$ and is evicted from the L1 cache due to preemption, will only contribute once to the indirect effect of preemption of memory block $m_{y,i}$. This is mainly because, if an access to a memory block $m_{x,i}$ at any program location in $\mathbb{P}_1$ results in a L1 cache miss then, $m_{x,i}$ will be reloaded in the L1 cache from the L2 cache or from the main memory. In both cases, $m_{x,i}$ will become the youngest element in the L1 and L2 caches and therefore can not cause any more indirect effect of preemption on memory block $m_{y,i}$ that has higher LRU-age in the L2 cache than $m_{x,i}$.

Finally, the indirect effect of preemption that can be suffered by a memory block $m_{y,i} \in \mathbb{M}_i$ due to one or more preemptions by a higher priority task $\tau_j \in \mathrm{hp}(i)$ is bounded by $|Ind^{mul,P}_{m_{y,i}}|$, where $Ind^{mul,P}_{m_{y,i}}$ is a set that contains all memory blocks that can cause an indirect effect of preemption on $m_{y,i}$ even in the presence of multiple preemptions.

Algorithm 8.2 Calculating the indirect effect of preemption that can be suffered by all memory blocks used by task τ_i when considering multiple preemptions by higher priority tasks in $\in \text{hp}(i)$ w.r.t preemption point P

Output: Upper bound on indirect effect of preemption suffered by every $m_{y,i} \in \mathbb{M}_i$ w.r.t a preemption point P, when considering multiple preemptions by a higher priority task $\tau_j \in \text{hp}(i)$, i.e., $|Ind^{mul,P}_{m_{y,i}}|$.

1: **for** $\forall m_{y,i} \in \mathbb{M}_i \setminus m_{x,i}$ **do**
2: $Ind^{mul,P}_{m_{y,i}} := \emptyset$
3: $FA^{P}_{m_{y,i}} := GetFirstAccess(m_{y,i}, P)$
4: $\mathbb{P}_1 := GetProgamPoints(P, FA^{P}_{m_{y,i}})$
5: **for** $\forall P^{'} \in \mathbb{P}_1$ **do**
6: **for** $\forall m_{x,i} \in UCB^{P^{'}}_{i,1}$ **do**
7: **if** $(CU^{P^{'},1}_{m_{x,i}} + |\bigcup_{\forall h \in \text{hep}(j)} ECB^{S_{m_{x,i},1}}_{h}| > W_1) \wedge (m_{x,i} \notin Ind^{mul,P}_{m_{y,i}})$ **then**
8: $FA^{P^{'}}_{m_{x,i}} := GetFirstAccess(m_{x,i}, P^{'})$
9: **if** $((MustAge(m_{y,i}, FA^{P^{'}}_{m_{x,i}}, 2) \neq \infty) \wedge (S_{m_{y,i},2} == S_{m_{x,i},2}) \wedge$ $((MustAge(m_{x,i}, FA^{P^{'}}_{m_{x,i}}, 2) = \infty) \vee (MustAge(m_{x,i}, FA^{P^{'}}_{m_{x,i}}, 2) > MustAge(m_{y,i}, FA^{P^{'}}_{m_{x,i}}, 2))))$ **then**
10: $Ind^{mul,P}_{m_{y,i}} := Ind^{mul,P}_{m_{y,i}} \cup m_{x,i};$
11: **end if**
12: **end if**
13: **end for**
14: **end for**
15: **end for**

8.5 Improved CRPD Analysis for Multilevel caches

In this section, we will demonstrate how the notion of multilevel UCBs, i.e., L1-UCBs and L2-UCBs, can be used to compute the CRPD for multilevel non-inclusive caches.

8.5.1 CRPD due to Eviction of L1-UCBs

L1-UCBs of a task τ_i can be evicted from the L1 cache only due to a direct preemption by any higher priority task $\tau_j \in \text{hp}(i)$. This is mainly because the L1 cache is always accessed by the tasks so the amount of intra-task cache interference that can be suffered by any memory block stored in the L1 cache will not be changed due to preemptions. However, when analyzing the L2 cache conflicts to a L1-UCBs $m_{x,i}$ of task τ_i both the direct effect of preemption, i.e., ECBs of higher priority tasks, and the indirect effect of preemption suffered by $m_{x,i}$ must be considered. This is mainly because the indirect effect of preemption is only suffered by memory blocks that may be accessed from the L2 cache only during the preempted execution of tasks.

To compute the CRPD due to eviction of L1-UCBs of task τ_i as a result of a preemption by any higher priority task $\tau_j \in \text{hp}(i)$ at any arbitrary program point P, we use a similar approach as presented in (Chattopadhyay and Roychoudhury, 2014) (i.e., Eq. (8.4) and (8.6)). L1-UCBs of task τ_i can be evicted from the L1 cache because of a preemption but may still be available in the L2 cache. We use $\gamma^{P,L_1}_{i,j}$ to denote the CRPD cost due to all those L1-UCBs of task τ_i, where $\gamma^{P,L_1}_{i,j}$

is computed as follows:

$$\gamma_{i,j}^{P,L_1} = d_{L1} \times \Big| \Big\{ m_{x,i} | m_{x,i} \in UCB_{i,1}^{P} \wedge CU_{m_{x,i}}^{P,1} + \Big| \bigcup_{h \in \text{hep}(j)} ECB_h^{S_{m_{x,i},1}} \Big| \geq W_1 \wedge CU_{m_{x,i}}^{P,2} + \Big| \bigcup_{h \in \text{hep}(j)} ECB_h^{S_{m_{x,i},2}} \Big| + Ind_{m_{x,i}}^{mul,P} < W_2 \Big\} \Big| \tag{8.13}$$

Note that when computing the L2 cache conflicts to a memory block $m_{x,i} \in UCB_{1,i}^{P}$, Equation (8.13) also considers the indirect effect of preemption that may be suffered by $m_{x,i}$, i.e., $Ind_{m_{x,i}}^{mul,P}$, which is computed using Algorithm. 8.2. The union of set of ECBs of all tasks in $\text{hep}(j)$ is used to account for nested/multiple preemptions of task τ_i by tasks in $\text{hp}(i)$.

Similarly, some L1-UCBs of task τ_i might be evicted from both L1 and L2 caches due to preemptions. We use $\gamma_{i,j}^{P,L_{12}}$ to denote the CRPD cost due to all those L1-UCBs of task τ_i, where $\gamma_{i,j}^{P,L_{12}}$ is computed as follows:

$$\gamma_{i,j}^{P,L_{12}} = (d_{L1} + d_{L2}) \times \Big| \Big\{ m_{x,i} | m_{x,i} \in UCB_{i,1}^{P} \wedge CU_{m_{x,i}}^{P,1} + \Big| \bigcup_{h \in \text{hep}(j)} ECB_h^{S_{m_{x,i},1}} \Big| \geq W_1 \wedge CU_{m_{x,i}}^{P,2} + \Big| \bigcup_{h \in \text{hep}(j)} ECB_h^{S_{m_{x,i},2}} \Big| + Ind_{m_{x,i}}^{mul,P} \geq W_2 \Big\} \Big| \tag{8.14}$$

8.5.2 CRPD due to Eviction of L2-UCBs

All memory blocks used by task τ_i with one or more memory references categorized as L2 cache hits in the absence of preemption, i.e., L2-UCBs of τ_i (see Definition 8.2), may be evicted from the L2 cache; (i) directly due to preemptions by the preempting tasks in $\text{hp}(i)$ or (ii) due to the indirect effect of preemption or (iii) due to a combination of both (i) and (ii). However, before presenting our analysis to compute the CRPD due to evictions of L2-UCBs of tasks, we highlight a source of pessimism in the existing analysis (Chattopadhyay and Roychoudhury, 2014). The pessimism lies in the use of Equation (8.8) and (8.9) that are used by the analysis in (Chattopadhyay and Roychoudhury, 2014) to compute the CRPD due to memory references that were L2 cache hits in the absence of preemption but may become L2 cache misses after the preemption. When calculating the CRPD due to a memory reference, e.g., to a memory block $m_{y,i}$ of task τ_i, that was a L2 cache hit in the absence of preemption but may result in a L2 cache miss after preemption, the analysis in (Chattopadhyay and Roychoudhury, 2014) assume that each reference to memory block $m_{y,i}$ after preemption may suffer up to two L2 misses after preemption, i.e., by using both Equation (8.8) and (8.9) to check for the eviction of same memory reference. However, this is not true, as we demonstrate using the following example.

Example 8.5. *We calculate the CRPD costs* $ICRT_{i,j}^{P,1}$ *and* $ICRT_{i,j}^{P,2}$*, i.e., using Equation* (8.8) *and* (8.9)*, for the example scenario shown in Figure* 8.3*. We have* $W_1 = 2$ *and* $W_2 = 3$ *and we assume that all memory blocks used by task* τ_i*, i.e., A,B,C,D and m, map to the same L1/L2 cache set. We can see in Figure* 8.3 *that three references to memory block m are L2 cache hits in the*

non-preempted execution of task τ_i, i.e., at program points r_1, r_2 and r_3. Therefore, the set of program locations with L2 cache hits w.r.t to a program point P, i.e., $\mathbb{P}_2$, is given by $\mathbb{P}_2 = \{r_1, r_2, r_3\}$. The Must-age of m at r_1, r_2 and r_3 is computed to be $MustAge(m, r_1, 2) = MustAge(m, r_2, 2) = MustAge(m, r_3, 2) = 2$. The number of ECBs of the preempting task, i.e., τ_j, that map the same L2 cache set as m are given by $|ECB_j^{S_{m,2}}| = 2$. Using Equation (8.2) to calculate the indirect effect of preemption suffered by m at r_1, r_2 and r_3 we get $|ID_m^{r_1,P}| = 0$ and $|ID_m^{r_2,P}| = |ID_m^{r_3,P}| = 1$. Moreover, since we have $\mathbb{S}_1 \leq \mathbb{S}_2$ and $W_1 \leq W_2$ we set $IL2_{ind} = 1$ in Equation (8.9). Now using all these values in Equation (8.8), we have $MustAge(m, r, 2) + |ECB_j^{S_{m,2}}| + |ID_m^{r,P}| > W_2$ for every $r \in \mathbb{P}_2$. Hence, the resulting value of $ICRT_{i,j}^{P,1}$ will be $3 \times d_{L2}$. Similarly, using the values of Must-age and the indirect effect of preemption of m in Equation (8.9) we have $MustAge(m, r, 2) + |ID_m^{r,P}| > W_2$ for r_2 and r_3 in $\mathbb{P}_2$. Therefore, the resulting value of $ICRT_{P,2}$ will be $2 \times d_{L2}$. Consequently, the total CRPD cost due to L2 cache misses resulting from preemption is calculated to be $ICRT_{i,j}^{P,1} + ICRT_{i,j}^{P,2} = (3+2) \times d_{L2} = 5 \times d_{L2}$. However, we can see from the preempted execution scenario shown in Figure 8.3 that this bound on the CRPD is very pessimistic and the actual CRPD cost due to all references to memory block m after preemption is only $2 \times d_{L2}$.

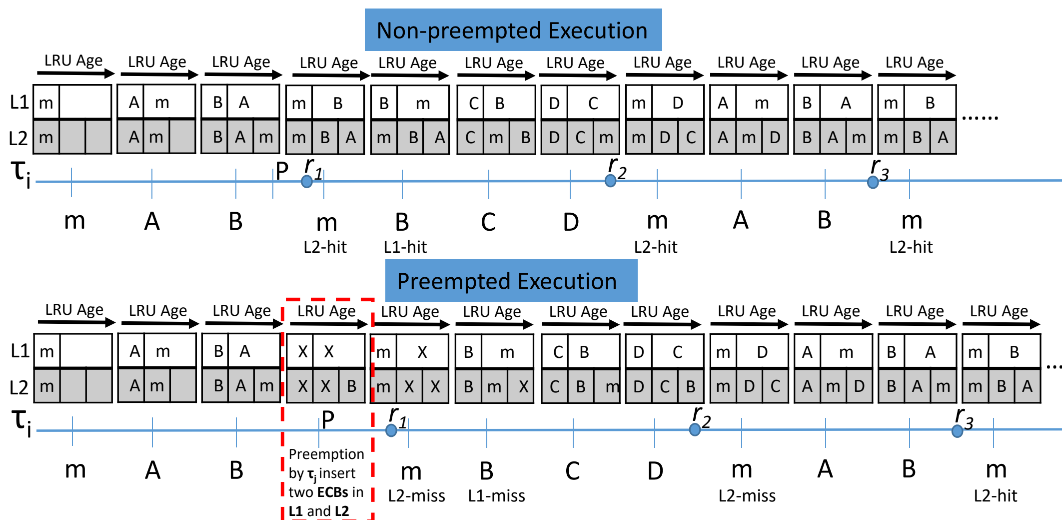


Figure 8.3: Example scenario to demonstrate the pessimism of (Chattopadhyay and Roychoudhury, 2014) when calculating the CRPD due to L2 cache misses resulting from preemption.

There are two main reasons why the analysis of (Chattopadhyay and Roychoudhury, 2014) overestimates the CRPD due to L2 cache misses resulting from preemptions; (1) the analysis does not differentiate between memory references that may be accessed inside a loop and the memory references that are not in a loop, i.e., the same memory reference may lead to more than one cache miss only if it is accessed in a loop, and (2) the analysis assume that all memory references that were L2 cache hit during non-preempted execution of a task may be impacted both directly and indirectly due to preemptions, i.e., it evaluates Equation (8.8) and (8.9) for all program locations with L2 cache hits. Although, it is true that multiple references to the same memory block (e.g.,

memory block m in Figure 8.3) may result in a L2 cache hit, not all those references can be impacted directly/indirectly due to preemptions.

To reduce the pessimism in the existing analysis (Chattopadhyay and Roychoudhury, 2014), we will focus on bounding the number of references to memory blocks that may be impacted directly/indirectly due to preemption. We start by computing the set of L2-UCBs of a task τ_i w.r.t a program point P, i.e., $UCB^P_{i,2}$. $UCB^P_{i,2}$ is the set of memory blocks that have at least one reference categorized as a L2 cache hit, starting from the program point under analysis, i.e., P, until the end point e of task τ_i. Formally,

$$UCB^P_{i,2} = \bigcup_{\forall r \in \mathbb{P}_2} UCB^r_{i,2} \tag{8.15}$$

where $\mathbb{P}_2$ is the set of all program locations between P and e with L2 cache hits.

Let $UCB^P_{i,2} = \{m_{1,i}, m_{2,i}, m_{3,i}, \ldots, m_{n,i}\}$ be the result of Equation (8.15), then for every memory block $m_{y,i}$ in $UCB^P_{i,2}$ we define a set $\mathbb{R}^P_{m_{y,i}}$ that contains all program locations after the preemption point P where a reference to $m_{y,i}$ is a L2 cache hit, i.e., $\mathbb{R}^P_{m_{y,i}} = \{R^{1,P}_{m_{y,i}}, R^{2,P}_{m_{y,i}}, \ldots, R^{k,P}_{m_{y,i}}\}$. Effectively, any memory block $m_{y,i}$ can have up to k accesses classified as L2 cache hits w.r.t a program point P. The rationale of defining $\mathbb{R}^P_{m_{y,i}}$ is to investigate how many references to memory block $m_{y,i}$ can be impacted directly or indirectly due to preemptions and therefore may contribute to CRPD. According to the CRPD analysis in (Chattopadhyay and Roychoudhury, 2014) all references to a memory block $m_{y,i} \in UCB^P_{i,2}$ of task τ_i can be impacted both directly and indirectly due to a preemption by a higher priority task $\tau_j \in \text{hp}(i)$ at a program point P. However, this is not true, in fact for any memory block $m_{y,i} \in UCB^P_{i,2}$ it is only the first reference to $m_{y,i}$ after preemption, i.e., at $R^{1,P}_{m_{y,i}}$, that can be directly impacted due to a preemption. All subsequent references to $m_{y,i}$ after program point $R^{1,P}_{m_{y,i}}$, i.e., at $\{R^{2,P}_{m_{y,i}}, \ldots, R^{k,P}_{m_{y,i}}\}$, can only be impacted due to the indirect effect of preemption.

Lemma 8.2. *For any memory block $m_{y,i} \in UCB^P_{i,2}$, it is only the first reference to $m_{y,i}$ after the preemption point P, i.e., at program point $R^{1,P}_{m_{y,i}}$, that can be directly impacted due to preemption. All subsequent references to $m_{y,i}$ after $R^{1,P}_{m_{y,i}}$, i.e., at $\{R^{2,P}_{m_{y,i}}, \ldots, R^{k,P}_{m_{y,i}}\}$, can only be impacted due to the indirect effect of preemption suffered by $m_{y,i}$.*

Proof. By definition if $m_{y,i} \in UCB^P_{i,2}$ then the reference to memory block $m_{y,i}$ at program point $R^{1,P}_{m_{y,i}}$ will be a L2 cache hit during the non-preempted execution of task τ_i, i.e., after an access to $m_{y,i}$ at $R^{1,P}_{m_{y,i}}$, $m_{y,i}$ will be the youngest element in both L1 and L2 caches. Now, after the preemption at program point P, the access to $m_{y,i}$ at $R^{1,P}_{m_{y,i}}$ may result in a L2 cache hit/miss. If the reference to $m_{y,i}$ at $R^{1,P}_{m_{y,i}}$ is a L2 cache hit after preemption, the sate of the L1 and L2 caches w.r.t $m_{y,i}$ will remain the same as in case of the non-preempted execution, i.e., $m_{y,i}$ will be the youngest element in both L1 and L2 caches after $R^{1,P}_{m_{y,i}}$. Similarly, even when the reference to $m_{y,i}$ at $R^{1,P}_{m_{y,i}}$ results in a L2 cache miss after preemption, $m_{y,i}$ will be reloaded from the main memory into both L1 and L2 caches at $R^{1,P}_{m_{y,i}}$. This will again make $m_{y,i}$ the youngest element in both L1 and L2 caches after program point $R^{1,P}_{m_{y,i}}$. Therefore, the direct impact of preemption on $m_{y,i}$ will be neutralized after a L2 cache hit/miss at $R^{1,P}_{m_{y,i}}$ and all subsequent references to $m_{y,i}$ can only be impacted due to the indirect effect of preemption suffered by $m_{y,i}$. □

Using Lemma 8.2 we can compute the CRPD cost task τ_i may endure due to the eviction of one of it's L2-UCB, e.g., $m_{y,i} \in UCB^P_{i,2}$, resulting from a preemption by any higher priority task $\tau_j \in \mathrm{hp}(i)$ at a program point P. We use two equations to compute that CRPD cost, i.e., Equation (8.16) and (8.17). Equation (8.16) computes the CRPD cost due to the first reference to memory block $m_{y,i} \in UCB^P_{i,2}$ after the preemption point P, i.e.,

$$\begin{cases} 0 & \text{if } MustAge(m_{y,i}, R^{1,P}_{m_{y,i}}, 2) + |\bigcup_{h \in \mathrm{hep}(j)} ECB_h^{S_{m_{y,i},2}}| + |Ind^{mul,P}_{m_{y,i}}| < W_2 \\ d_{L2} & \text{otherwise.} \end{cases} \tag{8.16}$$

Similarly, the CRPD cost due to all subsequent reference to memory block $m_{y,i} \in UCB^P_{i,2}$ after the program point $R^{1,P}_{m_{y,i}}$ can be calculated as follows

$$\sum_{\forall r \in \mathbb{R}^P_{m_{y,i}} \setminus R^{1,P}_{m_{y,i}}} \begin{cases} 0 & \text{if } MustAge(m_{y,i}, r, 2) + |Ind^{mul,P}_{m_{y,i}}| < W_2 \\ d_{L2} & \text{otherwise.} \end{cases} \tag{8.17}$$

It is straightforward to see that if any reference to a memory block $m_{y,i} \in UCB^P_{i,2}$ is not in a loop, it can contribute only once to the CRPD. However, the problem emerges when one or more references to memory block $m_{y,i}$ are inside a loop, in which case we need to bound how many times each reference to $m_{y,i}$ can contribute to the CRPD.

Lemma 8.3. *Any reference to a memory block $m_{y,i} \in UCB^P_{i,2}$ which is inside a loop, e.g., $Ref(m_{y,i})$, can contribute at most two L2 cache misses to the CRPD suffered by task τ_i due to preemption at a program point P.*

Proof. Any reference to a memory block $m_{y,i} \in UCB^P_{i,2}$ which is inside a loop, e.g., $Ref(m_{y,i})$, can be classified as a L2 cache hit in the absence of preemption w.r.t a program point P under two conditions; (1) if there are fewer than W_2 conflicting L2 memory blocks accessed between two references to $m_{y,i}$ w.r.t to program point P and (2) if at least W_2 conflicting L2 memory blocks are accessed between two access to $m_{y,i}$ w.r.t program point P, however there exist at least one memory reference which is a L1 cache hit, i.e., it does not generate an L2 cache conflict to $m_{y,i}$. If (1) is true, then the reference $Ref(m_{y,i})$ can only become a L2 cache miss after preemption if $m_{y,i}$ is directly evicted due to preemption at P and $Ref(m_{y,i})$ can suffer at most one L2 miss after preemption. Also, if (2) holds then the reference $Ref(m_{y,i})$ may also become a L2 miss after preemption due to the indirect effect of preemption caused by memory references that were L1 cache hits in the absence of preemption but may access the L2 after preemption. However, it is proved in (Chattopadhyay and Roychoudhury, 2014) that if the cache configuration is such that $|\mathbb{S}_1| \leq |\mathbb{S}_2|$ and $W_1 \leq W_2$, then any memory reference that was a L2 cache hit in the absence of preemption, e.g., $Ref(m_{y,i})$, can lead to at most one L2 miss due to the indirect effect after preemption. Knowing that $Ref(m_{y,i})$ is in a loop so both (1) and (2) can be true along different paths reachable to $Ref(m_{y,i})$. Therefore, we can deduce that if reference $Ref(m_{y,i})$ is inside a loop, it may cause up to two L2 cache misses after preemption. The lemma follows. □

We also know from Lemma 8.2 that after a preemption at a program point P, it is only the first reference to memory block $m_{y,i}$, i.e., at program point $R^{1,P}_{m_{y,i}}$, that can be directly impacted due to preemption. This leads to the following Lemma

Lemma 8.4. *Any reference to a memory block $m_{y,i} \in UCB^{P}_{i,2}$ which is inside a loop, i.e., $Ref(m_{y,i})$, can cause up to two L2 cache misses after a preemption at any program point P only when $Ref(m_{y,i})$ is the first reference to $m_{y,i}$ after P, i.e., at $R^{1,P}_{m_{y,i}}$.*

Proof. We prove this lemma by contradiction. Let us assume there exist a memory reference to $m_{y,i}$, i.e., $\hat{Ref}(m_{y,i})$, which is inside a loop and can cause up to two L2 cache misses after the preemption at P but $\hat{Ref}(m_{y,i})$ is not the first reference to $m_{y,i}$ after the preemption point P, i.e., $\hat{Ref}(m_{y,i})$ is performed at $R^{x,P}_{m_{y,i}} \neq R^{1,P}_{m_{y,i}}$.

For $\hat{Ref}(m_{y,i})$ to cause two L2 cache misses after the preemption at P, there must be at least two paths reachable to $\hat{Ref}(m_{y,i})$ after P where $m_{y,i}$ will be evicted either directly or indirectly. However, we know from Lemma 8.2 that it is only the first reference of $m_{y,i}$ after preemption, i.e., at $R^{1,P}_{m_{y,i}}$, that can cause an L2 cache miss directly due to a preemption at P and all subsequent references to $m_{y,i}$ after the program point $R^{1,P}_{m_{y,i}}$ can only be evicted from the L2 cache due to the indirect effect of preemption. Moreover, it is proved in (Chattopadhyay and Roychoudhury, 2014) that any reference to a memory block $m_{y,i}$ can lead to at most one L2 miss solely due to the indirect effect of preemption. Therefore, if $\hat{Ref}(m_{y,i})$ is not the first reference to $m_{y,i}$ after preemption it can cause at most one L2 cache miss due to the indirect effect of preemption suffered by $m_{y,i}$. Hence, we reach a contradiction. □

Finally, by using Lemmas 8.2, 8.3 and 8.4 we can bound the maximum number of times any memory block $m_{y,i} \in UCB^{P}_{i,2}$ can contribute to the CRPD suffered by task τ_i due to a preemption by any higher priority task $\tau_j \in \mathrm{hp}(i)$ at an arbitrary program point P.

Lemma 8.5. *The contribution of a memory block $m_{y,i} \in UCB^{P}_{i,2}$ to the CRPD suffered by task τ_i due to a preemption by any higher priority task $\tau_j \in \mathrm{hp}(i)$ at an arbitrary program point P is upper bounded by $\min(k, |Ind^{mul,P}_{m_{y,i}}| + 1) \times d_{L2}$. Where k is the cardinality of the set $\mathbb{R}^{P}_{m_{y,i}}$.*

Proof. We prove that for a memory block $m_{y,i} \in UCB^{P}_{i,2}$ both k and $Ind^{mul,P}_{m_{y,i}} + 1$ are upper bounds on the number of additional L2 cache misses that can be generated due to preemption at a program point P. Therefore, $\min(k, |Ind^{mul,P}_{m_{y,i}}| + 1) \times d_{L2}$ upper bounds the contribution of $m_{y,i}$ to the CRPD suffered by task τ_i at a program point P.

If memory block $m_{y,i}$ has k memory references classified as L2 cache hits in the absence of preemption after the program point P, then, in the worst-case all k references to $m_{y,i}$ may result in L2 cache misses due to preemption at P. Considering that the penalty of a single L2 cache miss is d_{L2}, therefore, the product $k \times d_{L2}$ upper bounds the contribution of $m_{y,i}$ to the CRPD due to preemption at a program point P.

From Lemma 8.2, we know that only the first reference to memory block $m_{y,i}$ after preemption may result in a L2 cache miss directly due to preemption and all subsequent reference to $m_{y,i}$ after the first reference can result in a L2 cache miss only due to indirect effect of preemption. We

also know that $|Ind^{mul,P}_{m_{y,i}}|$ is an upper bound on the number of memory blocks that can cause an indirect effect of preemption on $m_{y,i}$ after a preemption at P. So, the worst-case scenario is when each memory block in $Ind^{mul,P}_{m_{y,i}}$ is accessed between every two references to $m_{y,i}$ and every access to a memory block in $Ind^{mul,P}_{m_{y,i}}$ leads to a L2 cache miss for $m_{y,i}$. Consequently, $m_{y,i}$ can suffer up to $|Ind^{mul,P}_{m_{y,i}}|$ L2 cache misses due to the indirect effect of preemption caused by memory blocks in $Ind^{mul,P}_{m_{y,i}}$. Moreover, from Lemma 8.4, we know that if the first reference to memory block $m_{y,i}$ after preemption is in a loop we can have one additional L2 cache miss after preemption. Consequently, a total of $|Ind^{mul,P}_{m_{y,i}}|+1$ L2 cache misses can be generated for memory block $m_{y,i} \in UCB^{P}_{i,2}$ after the preemption at P. Therefore, $(|Ind^{P}_{m_{y,i}}|+1) \times d_{L2}$ is also an upper bound on the contribution of $m_{y,i}$ to the CRPD suffered by task τ_i due to a preemption at program point P. The lemma follows. □

8.5.2.1 CRPD Computation

We now show how to compute the total CRPD cost that task τ_i may bear due to all its L2-UCBs in $UCB^{P}_{i,2}$ that may be evicted from the L2 cache due a preemption at program point P by any higher priority task $\tau_j \in \text{hp}(i)$. We use Algorithm 8.3 to calculate that CRPD cost. The working of Algorithm 8.3 is explained as follows: The output of Algorithm 8.3 is the maximum CRPD cost, i.e., denoted by $\gamma^{P,L_2}_{i,j}$, that can be suffered by task τ_i due to eviction of all its L2-UCBs in $UCB^{P}_{i,2}$. $\gamma^{P,L_2}_{i,j}$ is computed by first computing the CRPD cost due to every memory block $m_{y,i} \in UCB^{P}_{i,2}$, i.e., denoted by $\gamma^{P,L_2}_{m_{y,i},j}$. For every L2-UCB $m_{y,i} \in UCB^{P}_{i,2}$ (external loop line 4 to 27), the algorithm starts by extracting the L2 cache hit locations using the function $GetHitLocations(.)$ (line 5). The output of the function $GetHitLocations(.)$ is the set $\mathbb{R}^{P}_{m_{y,i}}$, i.e., $\mathbb{R}^{P}_{m_{y,i}} = \{R^{1,P}_{m_{y,i}}, R^{1,P}_{m_{y,i}}, ..., R^{k,P}_{m_{y,i}}\}$. The algorithm then checks if the first reference to memory block $m_{y,i}$ after preemption point P, i.e., at $R^{1,P}_{m_{y,i}}$, is in a loop. If $R^{1,P}_{m_{y,i}}$ is in a loop, reference to $m_{y,i}$ at $R^{1,P}_{m_{y,i}}$ may result in up to two L2 cache misses (see Lemmas 8.3 and 8.4). So, the algorithm checks for the eviction of $m_{y,i}$ from the L2 cache at program point $R^{1,P}_{m_{y,i}}$ using both Equation (8.16) and (8.17) (lines 6 to 13). Similarly, if $R^{1,P}_{m_{y,i}}$ is not in a loop, reference to $m_{y,i}$ at $R^{1,P}_{m_{y,i}}$ can only be evicted from the L2 cache due to the combination of ECBs of the preempting tasks in $\text{hp}(i)$ and the indirect effect of preemption (i.e., lines 14 to 16). From Lemma 8.2, we know that all subsequent reference to $m_{y,i}$ after $R^{1,P}_{m_{y,i}}$, i.e., at $\{R^{2,P}_{m_{y,i}}, ..., R^{k,P}_{m_{y,i}}\}$, may only result in a L2 cache miss due to the indirect effect of preemption suffered by $m_{y,i}$. Therefore, the algorithm checks for the eviction of $m_{y,i}$ at all references except $R^{1,P}_{m_{y,i}}$ using only Equation (8.17) (lines 19 to 21). Furthermore, from Lemma 8.5, we know that the maximum CRPD task τ_i can suffer due to any L2-UCB $m_{y,i} \in UCB^{P}_{i,2}$ is upper bounded by $\min(|\mathbb{R}^{P}_{m_{y,i}}|, |Ind^{mul,P}_{m_{y,i}}|+1) \times d_{L2}$, which is considered in the last construct of Algorithm 8.3 (i.e., lines 23 and 24). Finally, the CRPD cost suffered by task τ_i due to any L2-UCB $m_{y,i} \in UCB^{P}_{i,2}$ is given by $\gamma^{P,L_2}_{m_{y,i},j}$ and the total CRPD cost task τ_i may suffer due to eviction of all of its L2-UCBs in $UCB^{P}_{i,2}$ is summed up in $\gamma^{P,L_2}_{i,j}$.

Algorithm 8.3 Algorithm to calculate the total CRPD cost due to eviction of L2-UCBs of task τ_i w.r.t a preemption point P

Output: The total CRPD cost, i.e., denoted by $\gamma_{i,j}^{P,L_2}$, that can be suffered by task τ_i due to the eviction of all its L2-UCBs in $UCB_{i,2}^{P}$, in case of a preemption at program point P by any higher priority task $\tau_j \in \mathrm{hp}(i)$.

1: $\gamma_{i,j}^{P,L_2} := 0$
2: **for** $\forall m_{y,i} \in UCB_{i,2}^{P}$ **do**
3: $\quad \gamma_{m_{y,i},j}^{P,L_2} := 0$
4: **end for**
5: **for** $\forall m_{y,i} \in UCB_{i,2}^{P}$ **do**
6: $\quad \mathbb{R}_{m_{y,i}}^{P} = GetHitLocations(m_{y,i}, P)$
7: $\quad$ **if** $R_{m_{y,i}}^{1,P}$ is in loop **then**
8: $\quad\quad$ **if** $MustAge(m_{y,i}, R_{m_{y,i}}^{1,P}, 2) + |\bigcup_{h \in \mathrm{hep}(j)} ECB_h^{S_{m_{y,i},2}}| + |Ind_{m_{y,i}}^{mul,P}| \geq W_2$ **then**
9: $\quad\quad\quad \gamma_{m_{y,i},j}^{P,L_2} := \gamma_{m_{y,i},j}^{P,L_2} + d_{L2}$
10: $\quad\quad$ **end if**
11: $\quad\quad$ **if** $MustAge(m_{y,i}, R_{m_{y,i}}^{1,P}, 2) + |Ind_{m_{y,i}}^{mul,P}| \geq W_2$ **then**
12: $\quad\quad\quad \gamma_{m_{y,i},j}^{P,L_2} := \gamma_{m_{y,i},j}^{P,L_2} + d_{L2}$
13: $\quad\quad$ **end if**
14: $\quad$ **else**
15: $\quad\quad$ **if** $MustAge(m_{y,i}, R_{m_{y,i}}^{1,P}, 2) + |\bigcup_{h \in \mathrm{hep}(j)} ECB_h^{S_{m_{y,i},2}}| + |Ind_{m_{y,i}}^{mul,P}| \geq W_2$ **then**
16: $\quad\quad\quad \gamma_{m_{y,i},j}^{P,L_2} := \gamma_{m_{y,i},j}^{P,L_2} + d_{L2}$
17: $\quad\quad$ **end if**
18: $\quad$ **end if**
19: $\quad$ **for** $\forall r \in \mathbb{R}_{m_{y,i}}^{P} \setminus R_{m_{y,i}}^{1,P}$ **do**
20: $\quad\quad$ **if** $MustAge(m_{y,i}, r, 2) + |Ind_{m_{y,i}}^{mul,P}| \geq W_2$ **then**
21: $\quad\quad\quad \gamma_{m_{y,i},j}^{P,L_2} := \gamma_{m_{y,i},j}^{P,L_2} + d_{L2}$
22: $\quad\quad$ **end if**
23: $\quad$ **end for**
24: $\quad$ **if** $\gamma_{m_{y,i},j}^{P,L_2} > \min(|\mathbb{R}_{m_{y,i}}^{P}|, |Ind_{m_{y,i}}^{mul,P}| + 1) \times d_{L2}$ **then**
25: $\quad\quad \gamma_{m_{y,i},j}^{P,L_2} := \min(|\mathbb{R}_{m_{y,i}}^{P}|, |Ind_{m_{y,i}}^{mul,P}| + 1) \times d_{L2}$
26: $\quad$ **end if**
27: $\quad \gamma_{i,j}^{P,L_2} := \gamma_{i,j}^{P,L_2} + \gamma_{m_{y,i},j}^{P,L_2}$
28: **end for**

8.5.3 Computation of total CRPD and WCRT Analysis

The sum of Equation (8.13) and (8.14) upper bounds the CRPD any task τ_i may suffer due to evictions of its L1-UCBs by any higher priority task $\tau_j \in \mathrm{hp}(i)$. Similarly, Algorithm 8.3 can be used to upper bound the CRPD of τ_i because of the eviction of its L2-UCBs by task τ_j. Therefore, an upper bound on the CRPD of task τ_i due to a preemption by task τ_j can be obtained by maximizing Equation (8.13), (8.14) and Algorithm 8.3 over all program points in τ_i, i.e.,

$$\gamma_{i,j}^{\overline{H}} = \max_{P \in \mathbb{P}} (\gamma_{i,j}^{P,L_1} + \gamma_{i,j}^{P,L_{12}} + \gamma_{i,j}^{P,L_2}) \tag{8.18}$$

However, we know that any higher priority task $\tau_j \in \text{hp}(i)$ can preempt any task $\tau_k \in \text{aff}(i,j)$ during the response time of task τ_i. Therefore, to ensure that the maximum CRPD cost is considered for a preemption of task τ_i by task $\tau_j \in \text{hp}(i)$, we maximize Equation (8.18) over all tasks in $\text{aff}(i,j)$, i.e.,

$$\gamma_{i,j}^{\overline{H},max} = \max_{\forall k \in \text{aff}(i,j)} \gamma_{i,j}^{\overline{H}} \tag{8.19}$$

Equation (8.19) is safe since it accounts for both nested preemptions (i.e., τ_i preempted by τ_k which is preempted by τ_j) and consecutive preemptions (of τ_i by τ_k and τ_j). The CRPD cost of a direct preemption of task τ_i by task $\tau_j \in \text{hp}(i)$ is accounted for in the term $\gamma_{i,j}^{\overline{H},max}$, whereas the indirect CRPD cost task τ_j may generate due to preemption of any task $\tau_k \in \text{aff}(i,j)$ during the response time of τ_i (i.e., a nested preemption) will be accounted for in the terms $\gamma_{i,k}^{\overline{H},max}$, i.e., due to the use of union of ECBs of all tasks $\text{hep}(k)$ when computing $\gamma_{i,k}^{\overline{H}}$. Finally, $\gamma_{i,j}^{\overline{H},max}$ can be incorporated into computation of WCRT R_i of a task τ_i as follows

$$R_i = C_i + \sum_{\forall j \in \text{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \times \left(C_j + \gamma_{i,j}^{\overline{H},max} \right) \tag{8.20}$$

Note that Equation (8.20) is recursive. However, a solution can be found using simple fixed-point iteration on R_i by initializing R_i to C_i. The iteration stops as soon as R_i converges or $R_i > D_i$, in which case the task is deemed unschedulable.

8.6 Experimental Evaluation

In this section, we will explain how our proposed WCRT analysis that provides a tighter CRPD bound for non-inclusive multilevel cache compares against the state-of-the-art analysis in (Chattopadhyay and Roychoudhury, 2014). First, we explain how the input quantities required for the analysis in (Chattopadhyay and Roychoudhury, 2014) and our proposed analysis can be computed. We then perform experiments by varying different parameters to compares the performance of both analyses.

8.6.1 Deriving Parameters for the Analyses

To derive task parameters needed to compare the CRPD analysis of (Chattopadhyay and Roychoudhury, 2014) against our proposed analysis we have used the Heptane (Hardy et al., 2017) static WCET analysis tool. Heptane is an open source WCET analysis tool that supports cache hierarchies. Specifically, it supports multilevel non-inclusive caches and implements the WCET analysis presented in (Hardy and Puaut, 2008). However, currently the tool only output the WCET of the analyzed benchmark and few cache statistics such as the total number of references to each cache level and the number of cache hits/misses for each cache level. Therefore, we have modified Heptane to compute the parameters needed for our analysis.

We have added a new module named `MultiCRPDAnalysis` to Heptane that enables us to compute different parameters needed for our multilevel CRPD analysis. The set of L1- and L2-

UCBs w.r.t a benchmark are computed using the Must-cache analysis along with a backward flow analysis on the control flow graph. The backward flow analysis computes the abstract cache state at the exit of a basic block by using the join operation on all the abstract cache states at the entry of its successors. For every memory block $m_{x,i}$ used by a task τ_i the analysis starts by assuming $CU^P_{m_{x,i}} = (\infty, \infty)$. Then for each program point P, the analysis checks the accessed memory block and update the abstract cache state using the Must-update and Must-join operations defined in (Hardy and Puaut, 2008). A memory block $m_{x,i}$ is considered a L1-UCB at program point P if it satisfies Equation (8.11). Similarly, all memory blocks that satisfy Equation (8.12) w.r.t a program point P are considered L2-UCBs at that program point. Note that our analysis to derive the set of L1-UCBs for multilevel caches is similar to the UCB analysis in (Chattopadhyay and Roychoudhury, 2014) however, we additionally derive the set of L2-UCBs w.r.t every program point P. The set of ECBs of task τ_i are computed using the May-cache analysis that determines the set of all memory blocks used by task τ_i at each cache level. To compute the indirect effect of preemption, we use a forward flow analysis along with the Must-cache analysis (Theiling et al., 2000). Since, the indirect effect of preemption is caused by memory blocks that were L1 cache hits in the absence of preemption but may be accessed from the L2 cache or main memory after preemption, the forward flow analysis (along with Must-case analysis) upper bounds the set of memory blocks that have one or more reference categorized as L1 cache hits in the absence of preemption and can cause the indirect effect of preemption on any memory block $m_{x,i}$ of task τ_i. For the analysis in (Chattopadhyay and Roychoudhury, 2014), the forward flow analysis is performed starting from the entry point of the program and ending at program point r where $m_{x,i}$ may be accessed. For our analysis, the forward flow analysis is performed for each pair of program locations between two access to memory block $m_{x,i}$. In both cases, the largest set of memory blocks is used when computing the indirect effect of preemption on any memory block $m_{x,i}$. Since Heptane allows to analyze each cache level, other parameters needed for the implementation of Equation (8.2) and Algorithm 8.2 are extracted using the Must-cache analysis. Similarly, the `cfglib` used by Heptane allows to compute loop bound for each basic block. This information is then used in Algorithm 8.3 to compute the CRPD due to the eviction of L2-UCBs.

8.6.2 Experiments

To evaluate the performance of our proposed CRPD analysis against the existing analysis, we conducted different experiments with various parameter settings. All experiments were performed using the Mälardalen benchmark suite (Gustafsson et al., 2010). For every benchmark, parameters such as the WCET, set of L1- and L2-UCBs, set of L1- and L2-ECBs, maximum LRU-ages of memory blocks, total number of references, number of references in loops etc., were extracted using Heptane. The target architecture was MIPS R2000/R3000 with a two level instruction cache hierarchy such that, L1 cache is 2-way set-associative with 32 sets and line size of 32-bytes, and L2 cache is 4-way set-associative with 64 sets and line size of 64-bytes. The L1 cache miss penalty was 10 processor cycles, i.e., $d_{L1} = 10$, and the L2 cache miss penalty was 100 processor cycles. Table 8.2 shows some benchmark parameters used in the experiments. Also, some task

Table 8.2: Benchmarks parameters from the Mälardalen Benchmark Suite (Gustafsson et al., 2010) used during the experimental evaluation

Name	C_i	L1-ECBs	L2-ECBs	L1-UCBs	L2-UCBs
bs	4020	11	6	11	2
bsort100	5811344	20	10	19	3
crc	1782419	43	22	43	10
expint	647343	19	11	18	3
fibcall	14023	8	4	8	3
insertsort	52245	16	8	16	4
lcdnum	8640	12	6	12	1
matmult	1795585	28	14	28	3
ns	129598	20	10	20	5
qurt	111554	53	33	53	24
fir	96215	22	11	22	4
prime	248299	17	9	17	7
select	145160	26	14	26	3
sqrt	20159	26	13	25	5
minmax	4435	17	9	16	6
ud	145170	75	38	74	6
minver	58155	167	84	167	58
fft	1252363	141	71	140	13
statemate	229575	261	133	254	37
fdct	101944	106	53	106	2
jfdctint	100331	96	48	96	2
ludcmp	341583	98	49	98	8
nsichneu	515015	1377	512	1377	422

set parameters were randomly generated as follows. The default number of tasks in each task set were 10 with task utilization generated using UUnifast (Bini and Buttazzo, 2005). Each task was randomly assigned values of one of the benchmark in Table 8.2. Task deadlines were implicit with priorities assigned in a deadline monotonic order. Task periods were set such that $T_i = C_i/U_i$.

We performed several experiments by varying the total task set utilizations, number of tasks per task set, L1 cache miss penalty, L2 cache miss penalty, number sets in the L1 cache, number of sets in the L2 cache, number of ways in the L1 cache and the number of ways in the L2 cache. A WCRT based schedulability analysis is performed using the same task set for all the analyzed approaches.

8.6.2.1 Task set Utilizations

In this experiment, we varied the total task set utilization from 0.025 to 1 in steps of 0.025 and randomly generated 1000 task sets per utilization point. Figure 8.4 shows the number task sets that were deemed schedulable using the “SoA Multilevel CRPD analysis", i.e., the CRPD analysis of (Chattopadhyay and Roychoudhury, 2014), and our “Proposed Multilevel CRPD analysis". The green line marked as “No Preemption cost" provides an upper bound on the number of task sets that were deemed schedulable without considering any CRPD cost. For clarity, we only show a cropped version of the plot in Figure 8.4 starting from a utilization of 0.6. All approaches produce identical results below this point. Figure 8.4 shows that our proposed approach performs

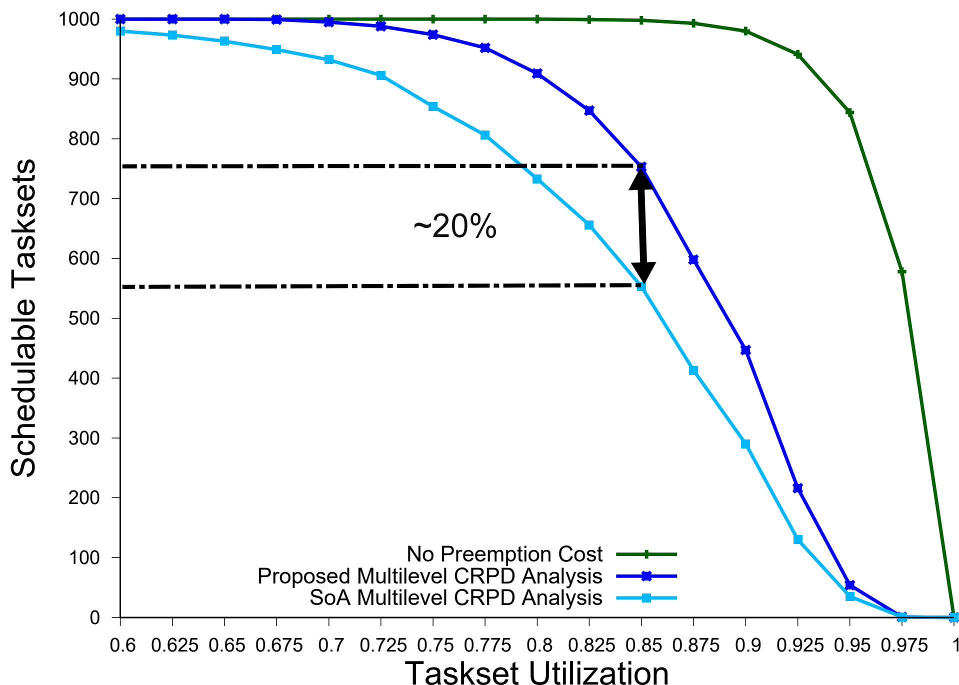


Figure 8.4: Number of task set deemed schedulable by varying total task set utilization

significantly better in comparison to the SoA analysis. The proposed analysis dominates the SoA analysis mainly due to two reasons: (i) it provides a tighter bound on the indirect effect of preemption that can be suffered by UCBs of tasks and (ii) it accurately estimates the CRPD suffered by tasks due to memory blocks that were L2 cache hits in the absence of preemption (i.e., L2-UCBs), but may suffer L2 cache misses after preemption. Although, the major improvement in the CRPD computation results from the treatment of L2 cache hits, however, we can see that the number of L2-UCBs of tasks (see Table 8.2) is very small in comparison to the number of L1-UCBs of tasks. But, still our proposed analysis results in improving task set schedulability by up to 20% percentage points over the existing analysis.

8.6.2.2 Number of Tasks

To evaluate how total number of tasks in a task set may impact the analyzed approaches, we performed an experiment by varying the total number of tasks in a task set between 5 to 25 in steps of 5. For all other parameters default values were considered. Since, we varied both the total task set utilizations and the number of tasks, we have used the weighted schedulability measure (see Equation (4.20)) to generate the plot shown in Figure 8.5. Intuitively, increasing the number of tasks tends to decrease task set schedulability because of an increase in the number of preemptions (which leads to an increase in the overall CRPD cost). The same can be confirmed from the plot

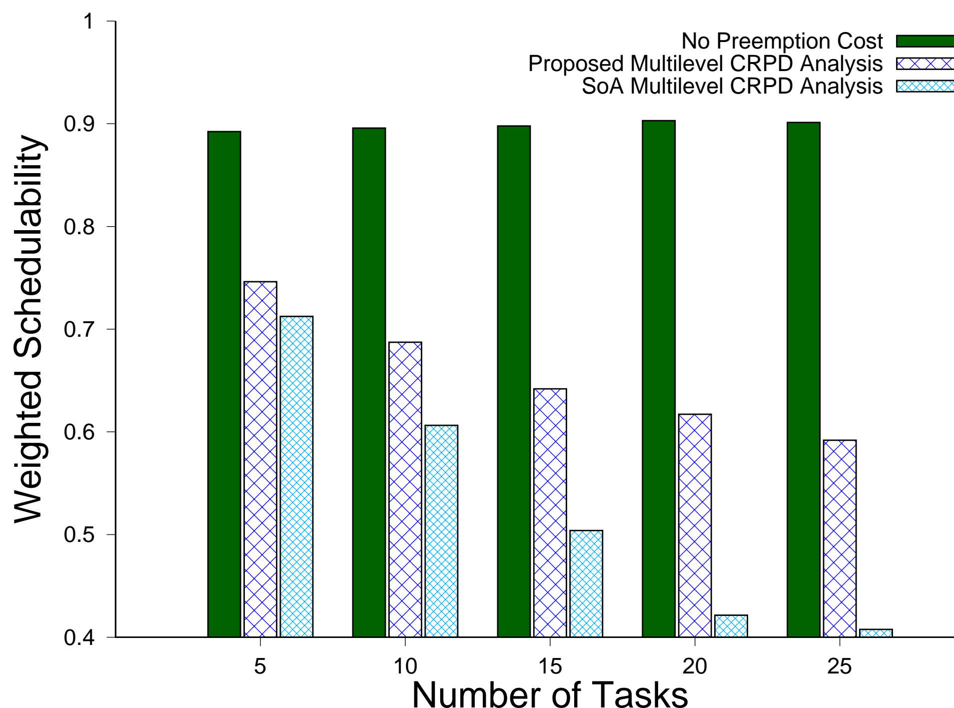


Figure 8.5: Wighted schedulability measure by varying the total number of tasks in a task set

shown in Figure 8.5. However, we can also see that our proposed CRPD analysis always dominates the SoA CRPD analysis. In fact, for higher number of tasks per task set (e.g., for 20 or 25 tasks per task set) the difference between the weighted schedulability of both approaches tends to increase. This is due to an excessive pessimism in the SoA CRPD analysis which may count the evictions of the same memory blocks several times in the CRPD cost. This pessimism is reduced by our analysis by bounding the number of times each memory block can contribute to the CRPD cost.

8.6.2.3 Number of Ways in the L1 Cache (W_1)

In this experiment, we varied the L1 cache associativity, i.e., the number of ways in the L1 cache W_1, and evaluated its impact on the performance of all the analyzed approaches. All other parameters were set to their default values. However, since we focus on a cache configuration where L1 cache associativity is always less than or equal to the L2 cache associativity, i.e., $W_1 \leq W_2$. There-

fore, for this experiment, we also fixed the L2 cache associativity to 32, i.e., $W_2 = 32$. We then varied the number of ways in the L1 cache between 2 to 32 and plotted the weighted schedulability for both approaches as shown in Figure 8.6. Note that increasing the number of ways in the L1 cache will also increase the size of the L1 cache.

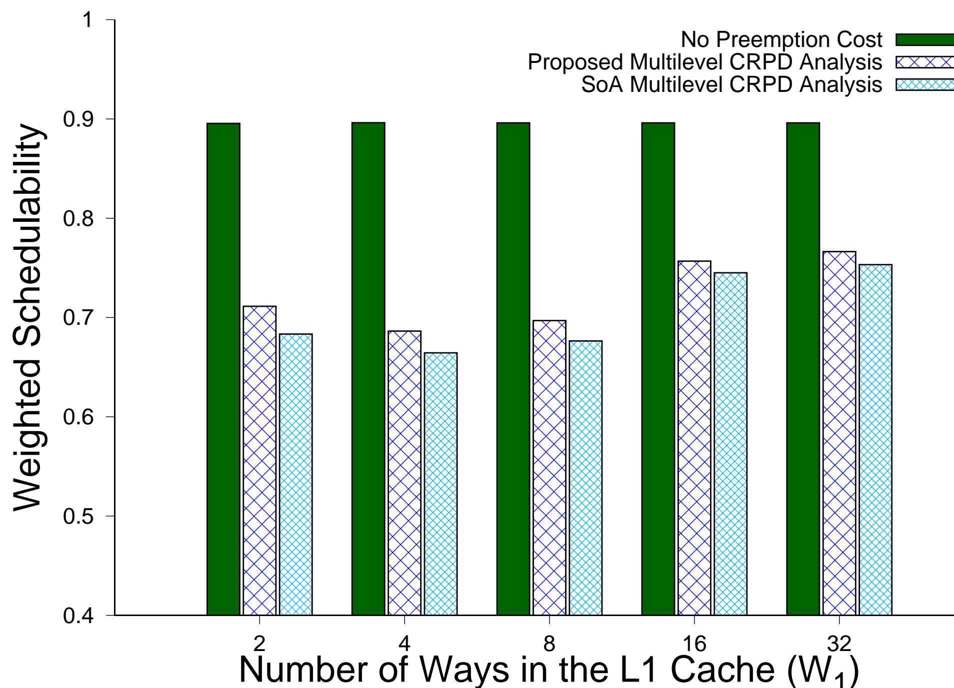


Figure 8.6: Weighted schedulability measure by varying number of ways in the L1 cache. The number of ways in the L2 cache were set to 32, i.e., $W_2 = 32$

We can see in Figure 8.6 that by varying the number of ways in the L1 cache (i.e., L1 cache size), both approaches produce similar results with the proposed approach marginally improving task set schedulability. This is mainly because, for both approaches the CRPD analysis for the L1 cache is very similar except for the computation of the indirect effect of preemption. Moreover, with the number of ways in the L2 cache set to 32, the L2 cache size becomes relatively larger w.r.t the analyzed benchmarks, which leads to almost no CRPD due to the L2 cache. Therefore, we observe that increasing the number of ways in the L1 cache has a similar effect on both analyses.

8.6.2.4 Number of Ways in the L2 Cache (W_2)

We also performed an experiment by varying the number of ways in the L2 cache, i.e., W_2, between 2 to 32 and evaluated their impact on task set schedulability. Default values were used for all the other parameters. The resulting plot is shown in Figure 8.7. Note that increasing the number of ways in the L2 cache also increases its size.

Figure 8.7 shows that when varying the number of ways in the L2 cache (i.e., increasing the L2 cache size), the difference between the performance of both analyses is very clear, with the proposed analysis clearly outperforming the existing analysis. This is because our analysis

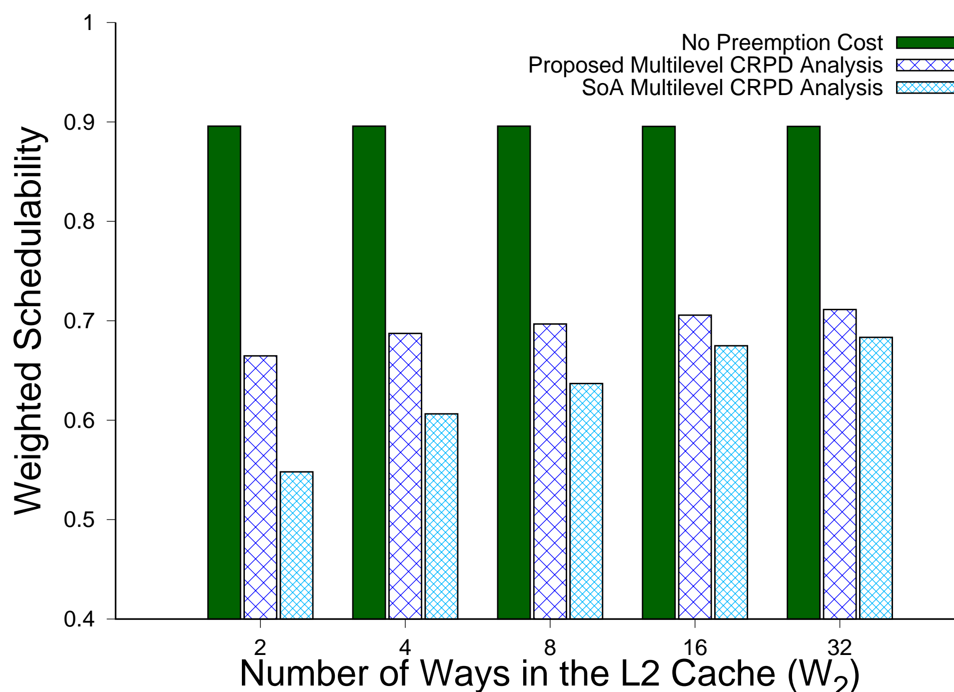


Figure 8.7: Weighted schedulability measure by varying number of ways in the L2 cache

provides much tighter bound on the L2 CRPD cost than the existing analysis. We can see in Figure 8.7 that when the L2 cache is smaller, i.e., the potential CRPD due to the L2 cache is higher, our approach performs significantly better than the existing analysis. However, by increasing the number of ways in the L2 cache, the difference between the performance of both analysis tends to reduce. This is mainly due to an overall reduction in the L2 CRPD due to a larger L2 cache.

8.6.2.5 Number of Sets in the L1 Cache ($|\mathbb{S}_1|$)

In this experiment, we varied the number of sets in the L1 cache, i.e., $|\mathbb{S}_1|$, between 16 to 256 and plotted the resulting weighted schedulability measure in Figure 8.8. Note that to ensure that the number of sets in the L1 cache are always less than or equal to the number of sets in the L2 cache, i.e., $|\mathbb{S}_1| \leq |\mathbb{S}_2|$, for this experiment we set $|\mathbb{S}_2| = 512$. Default values were used for all other parameters.

We can see a similar trend in Figure 8.8 which was observed by increasing the L1 cache associativity (i.e., Figure 8.6). Also, due to the same reasoning as previously explained in experiment 3, i.e., due to the similarities in the L1 CRPD analysis and a relatively larger L2 cache, both approaches tend to behave similarly when the L1 cache size is increased.

8.6.2.6 Number of Sets in the L2 Cache ($|\mathbb{S}_2|$)

Figure 8.9 shows the weighted schedulability measure resulting from an increase in the number of sets in the L2 cache. As an increase in the number of sets in the L2 cache also increases its size,

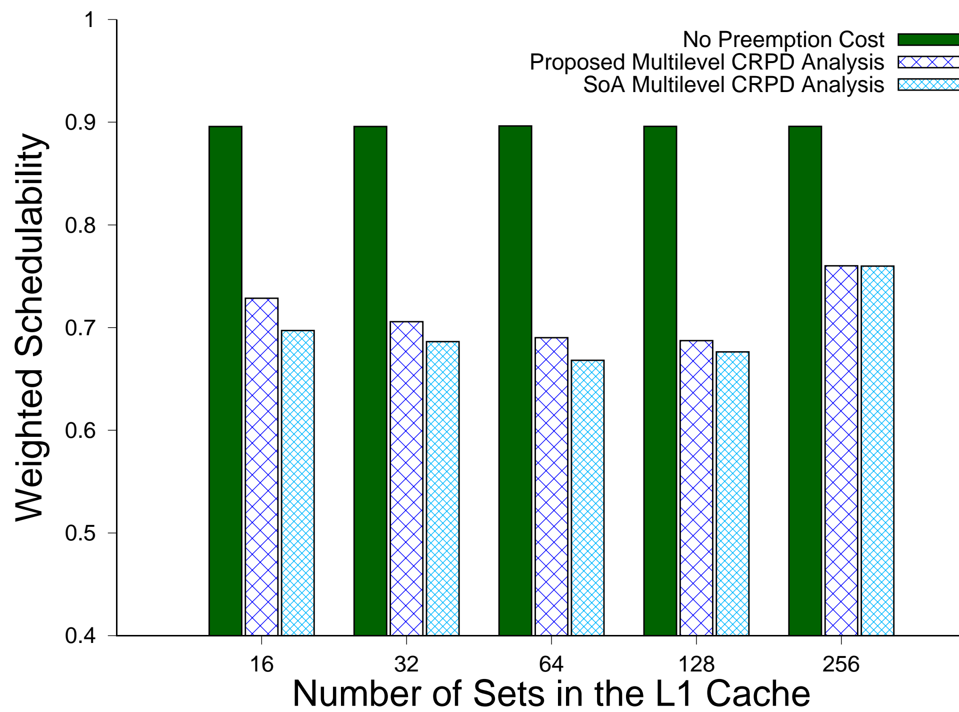


Figure 8.8: Weighted schedulability measure by varying number of sets in the L1 cache. The number of sets in the L2 cache were fixed to 512, i.e., $|\mathbb{S}_2| = 512$

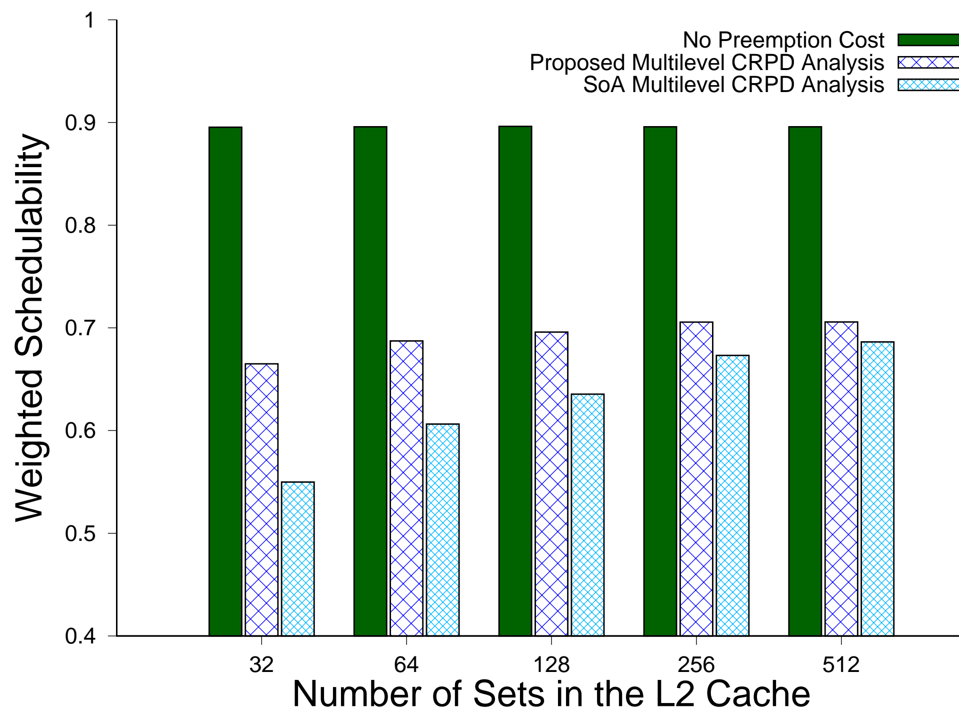
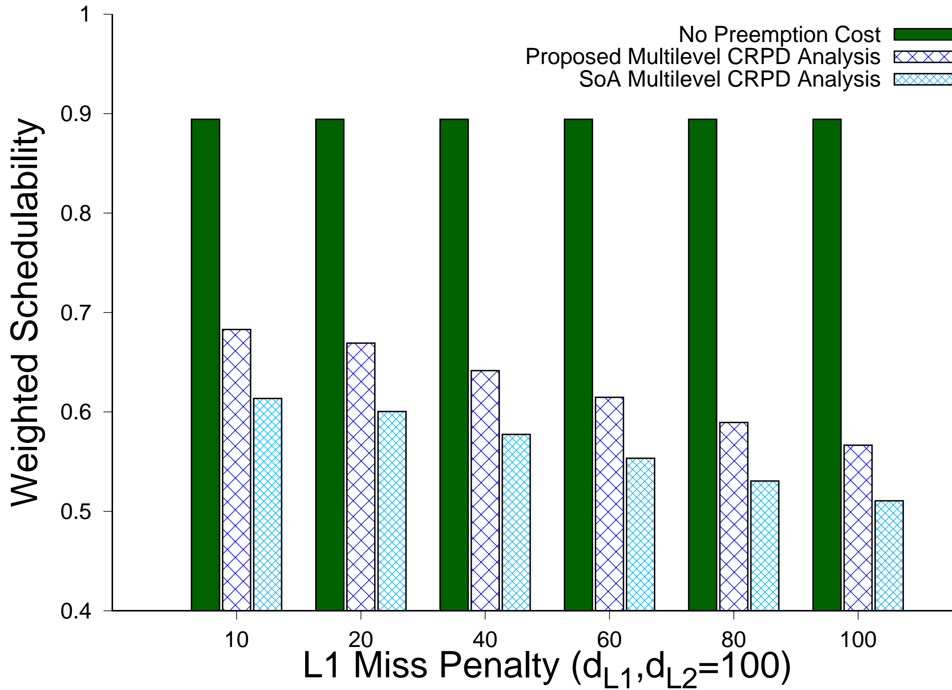
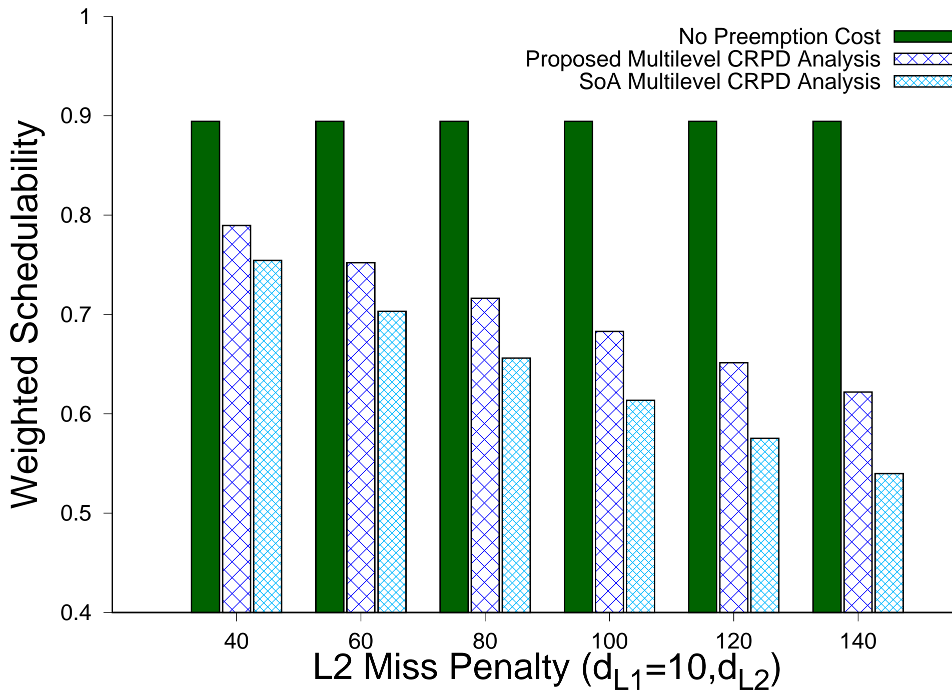


Figure 8.9: Weighted Schedulability measure by varying number of sets in the L2 cache. The number of sets in the L1 cache were set to their default value, i.e., $|\mathbb{S}_1| = 32$

we can see a similar trend in Figure 8.9 which was previously observed in Figure 8.7. Therefore, this behavior can be explained using the exact same reasoning described in experiment 4.



(a) Varying the L1 cache miss penalty (d_{L1})



(b) Varying the L2 cache miss penalty (d_{L2})

Figure 8.10: Weighted schedulability results by varying d_{L1} and d_{L2}

8.6.2.7 Varying the L1 and L2 Cache Miss Penalties (d_{L1} and d_{L1})

We conducted two more experiments by varying the L1 and L2 cache miss penalties. Default values were used for all other parameters.

In the first experiment, we varied the L1 cache miss penalty between 10 to 100 processor cycles and the resulting weighted schedulability measure is shown in Figure 8.10a. Since, the L1 miss penalty can not be larger than the L2 miss penalty, for this experiment, we set $d_{L2} = 100$ cycles. We can see in Figure 8.10a that by increasing the L1 cache miss penalty the weighted schedulability for both approaches decreases. However, difference between the performance of both analysis remains nearly constant due to a similar L1 CRPD analysis.

For the second experiment, we varied the L2 cache miss penalty between 40 to 140 processor cycles keeping the default value for L1 cache miss penalty. The results are shown in Figure 8.10b. We can see that for lower values of L2 cache miss penalty the difference between the weighted schedulability of both approaches is smaller. However, by increasing the value of L2 miss penalty the difference between the performance of both approaches also increases which is due to a tighter bound on the L2 CRPD by the proposed analysis.

8.7 Chapter Summary

In this chapter, we presented a CRPD analysis for multilevel non-inclusive caches. We redefined the notion of UCBs for multilevel caches, showed how these UCBs can be computed and used them to compute the CRPD. We showed that a tighter bound on the indirect effect of preemption can be obtained by calculating the indirect effect of preemption that can be caused instead of calculating the indirect effect of preemption that can be suffered by memory blocks. We then presented a new analysis to compute the CRPD due to memory blocks that were categorized as L2 cache hits in the absence of preemption but may become L2 cache misses due to preemptions. Our analysis provides a tighter CRPD bound than the existing analysis by identifying how many references to a memory block can be impacted due to preemptions and therefore may contribute to the CRPD.

We evaluated the performance of our proposed CRPD analysis against an existing analysis from the state-of-the-art in terms of schedulability. Experiments were performed by varying different parameters with most values taken from the Mälardalen benchmarks. Experimental results show that our proposed CRPD analysis for multilevel non-inclusive caches dominate the state-of-the-art analysis and results in up to 20% percentage points higher task schedulability.

Part III

Extension to Multicore Platforms

Chapter 9

Evaluating the Impact of Inter-task Cache Interference on Memory Bus Contention in Multicore Systems

In the previous chapters, we have discussed how to derive a tighter bound on the inter-task cache interference considering different cache configurations, i.e., direct-mapped, set-associative and multi-level caches. This chapter now evaluates how a sound estimate on the inter-task cache interference may impact the contention due to sharing of memory bus in multicore systems.

In a multicore system, data and instructions are transferred from the main memory to the requesting core over a *shared* memory bus. Due to the use of a shared memory bus, main memory requests by a task τ_i running on one core may be delayed by tasks executing on other cores, thereby increasing the WCRT of τ_i. This increase in the WCRT of τ_i depends on many factors such as (i) the number of main memory requests generated by τ_i and all other tasks running on the same core, (ii) the number of main memory requests generated by all tasks executing on different cores than τ_i and (iii) the memory bus arbiter. One of the main aspects that impact (i) and (ii) is the number of *cache misses* suffered by each task during its execution. Indeed, the number of main memory requests generated by a task strongly depends on whether the instructions and data it requires are available in the cache memory (cache hit) or not (cache miss), which in turn depends on the intra- and inter-task cache interference suffered by the task. The number of cache misses or the number of main memory requests generated by a task τ_i when executing in isolation can be bounded by using the intra-task cache interference analysis (see Section 3.1). However, when task τ_i executes concurrently with other tasks, the number of bus/main memory requests generated by τ_i may also depend on the inter-task cache interference (i.e., CRPD and CPRO) suffered by the task. Consequently, the total memory bus contention suffered by task τ_i during its execution depends mainly on the inter-task cache interference suffered by task τ_i and all other tasks running on the same core as well as the inter-task cache interference suffered by all tasks executing in parallel with τ_i on different cores than τ_i.

There exist few approaches in literature that account for CRPDs (Davis et al., 2018b; Altmeyer

et al., 2015) when bounding the memory bus contention in multicore systems. However, as we showed in Chapter 4 and Chapter 7, only considering CRPDs when computing the inter-task cache interference of tasks may lead to pessimistic WCRT bounds and the analysis that accounts for both CRPDs and cache persistence dominates the analysis that only consider CRPDs. In this chapter, we evaluate how a tighter bound on the inter-task cache interference may impact memory bus contention in multicore platforms considering both work conserving and non-working conserving bus arbitration policies. We analyze multicore architecture considering both single-level and multilevel caches. For architectures with single-level caches, we built on the analysis presented in Chapter 7 to compare the performance of memory bus contention analysis that only accounts for CRPDs against the memory bus contention analysis that accounts for both CRPDs and cache persistence. For architectures with multiple cache levels, we evaluate how the two CRPD analysis discussed in Chapter 8 may impact the memory bus contention suffered by the tasks under different bus arbitration policies.

9.1 Assumptions on the System Model

In this chapter, we make the following assumptions on the system model:

- We consider multicore platforms with m identical timing-compositional cores π_1 to π_m. By timing-compositional we mean that it is safe to separately account for interference from different sources such as cores, caches and memory bus (Hahn et al., 2013).
- When considering multicore architectures having a single cache level (i.e., L1) we will consider the system model detailed in Section 7.1.
- For multicore architectures that support a two-level non-inclusive cache hierarchy (i.e., comprising of L1 and L2 caches) we will consider the system model and assumptions detailed in Section 8.1.
- We assume that the cache(s) is/are set-associative and use the Least-Recently-Used (LRU) cache replacement policy. The last level cache is connected via a shared bus to the global main memory. The worst-case time for one access to the main memory is given by d_{mem}. Note that when considering multicore architectures with two-level caches, the worst-case time for one access to the main memory is given by the sum of L1 cache miss penalty d_{L1} and the L2 cache miss penalty d_{L2}, i.e., $d_{mem} = d_{L1} + d_{L2}$.
- We consider a set Γ of n sporadic constrained deadline tasks $\Gamma = \{\tau_1, \tau_2, \ldots \tau_n\}$. Each task $\tau_i \in \Gamma$ is defined by a quadruple (PD_i, MD_i, D_i, T_i) where PD_i is the worse-case execution time of a job of τ_i considering that every memory access is a cache hit. Consequently, it only accounts for execution requirements of the task and does not include the time needed to fetch data and instructions from main memory. MD_i is the worst-case memory access demand of a job of τ_i, i.e., the maximum number of main memory request generated by

any job of τ_i. Note that the values of PD_i and MD_i are calculated assuming τ_i executes in *isolation*. D_i is the relative deadline of τ_i and T_i is the minimum-inter arrival time between two jobs of τ_i. We assume that tasks are scheduled with a partitioned task-level fixed priority scheduling algorithm where each task is statically assigned to a core at design time. Tasks assigned to a core π_x are denoted by Γ_x. Tasks can be assigned priorities using any fixed priority assignment scheme (e.g., Rate or Deadline Monotonic (Liu and Layland, 1973)). Furthermore, we assume that the priority of each task is unique thus providing a global priority order such that τ_1 has the highest priority and τ_n the lowest. R_i denotes the WCRT of task τ_i and is defined as the longest time between the arrival and the completion of any of its jobs.

The list of important symbols used in this chapter is given in Table 9.1.

Table 9.1: List of important symbols used in Chapter 9

Symbol	Description
Γ	Task set of size n
π_x	Core x of a multicore processor
Γ_x	Set of tasks assigned to core π_x of a multicore processor
τ_i	Task with index i
PD_i	Worst-case processing demand of task τ_i in isolation
MD_i	Worst-case memory access demand of task τ_i in isolation
T_i	Minimum inter-arrival time of task τ_i
D_i	Relative deadline of task τ_i
R_i	Worst-case response time of task τ_i
MD_i^r	Residual memory access demand of task τ_i in isolation
$\hat{MD}_i(n_i)$	The maximum number of bus accesses generated by n_i jobs of task τ_i while executing in isolation
$\mathrm{hp}(i)$	The set of tasks with higher priority than τ_i
$\mathrm{hep}(i)$	The set of tasks with higher priority than τ_i including τ_i, i.e., $\mathrm{hep}(i) = \mathrm{hp}(i) \cup \tau_i$.
$\mathrm{aff}(i,j)$	The set of intermediate tasks (including τ_i) that may preempt τ_i but may themselves be preempted by some higher priority task τ_j.
d_{L1}	L1 cache miss penalty
d_{L2}	L2 cache miss penalty
d_{mem}	Total time needed to reload one block from the main memory to cache(s), i.e., $d_{mem} = d_{L1} + d_{L2}$.
$\gamma_{i,j,x}$	Additional bus accesses resulting from the CRPD suffered by task τ_i due to preemptions by a higher priority task $\tau_j \in \mathrm{hp}(i)$ executing on the same core π_x

Continued on next page

Table 9.1 – continued from previous page

Symbol	Description
$BAS_i^x(t)$	Upper bound on the total number of bus accesses that can occur due to task τ_i and all higher priority tasks in $\text{hp}(i)$ executing on core π_x during a time interval of length t
$BAO_k^y(t)$	Upper bound on the total number of bus accesses by all tasks having priority k or higher executing on core π_y during a time interval of length t
$BAT_i^x(t)$	The total number of bus accesses that may delay the execution of τ_i on core π_x during a time interval of length t
sl	Slot size for Round-Robin (RR) and TDMA bus arbitration policy
$\rho_{j,i,x}$	Additional bus accesses due to CPRO suffered by one job of a higher priority task $\tau_j \in \text{hp}(i)$ executing during the response time of a lower priority task τ_i on a core π_x
$\hat{\rho}_{j,i,x}(n_i)$	Additional bus accesses due to CPRO suffered by n_i jobs of a higher priority task $\tau_j \in \text{hp}(i)$ executing during the response time of a lower priority task τ_i on a core π_x
$E_i(t)$	The maximum number of jobs any task τ_i can release in a time interval of length t
ECB_i	The set of evicting cache blocks (ECBs) of task τ_i
UCB_i	The set of useful cache blocks (UCBs) of task τ_i
PCB_i	The set of persistence cache blocks (PCBs) of task τ_i

9.2 CRPD-aware Memory Bus Contention Analysis

The maximum number of main memory accesses that can be generated by a task τ_i in isolation is upper bounded by the worst-case memory access demand of task τ_i, i.e., MD_i. However, when task τ_i executes concurrently with other tasks, it may generate additional main memory requests due to the CRPD it may suffer due to preemptions by higher priority tasks in $\text{hp}(i)$. As we have mentioned previously, there exist approaches in literature that account for CRPDs when bounding the memory bus contention in multicore systems. One such approach is presented in (Altmeyer et al., 2015; Davis et al., 2018b). The analysis presented in (Altmeyer et al., 2015; Davis et al., 2018b) bounds the memory bus contention that can be suffered by a task τ_i executing on core π_x of a multicore processor in a time window of length t by first computing two values; (i) $BAS_i^x(t)$, which is an upper bound on the total number of bus accesses that can occur due to task τ_i and all higher priority tasks in $\text{hp}(i)$ executing on core π_x during t and (ii) $BAO(t)$, which is an upper on the total number of bus accesses generated by all tasks running on other cores than π_x during the same time interval of length t. Under the bus contention analysis presented in (Davis et al., 2018b; Altmeyer et al., 2015) $BAS_i^x(t)$ is upper bounded such that

$$BAS_i^x(t) \leq MD_i + \sum_{\forall \tau_j \in \Gamma_x \cap \text{hp}(i)} \left\lceil \frac{t}{T_j} \right\rceil \times (MD_j + \gamma_{i,j,x}) \tag{9.1}$$

where MD_i and MD_j are the worst-case memory access demands of task τ_i and task $\tau_j \in \text{hp}(i)$ respectively, and $\gamma_{i,j,x}$ accounts for the additional memory accesses due the CRPD suffered by task τ_i due to preemptions by a higher priority task $\tau_j \in \text{hp}(i)$ executing on the same core π_x. Note that when using Equation (9.1), $\gamma_{i,j,x}$ can be computed using any of the CRPD analysis discussed in Section 3.2.

When bounding the total number of bus accesses generated by all tasks running on cores other than π_x, no assumption can be made about the synchronization of tasks w.r.t the release of τ_i on core π_x. Therefore, for a task τ_l executing on some core $\pi_y \neq \pi_x$, the worst-case number of bus accesses generated by τ_l in a time interval of length t are obtained by assuming that the first job of τ_l executes as late as possible, i.e., just before its WCRT, while all subsequent jobs of τ_l execute as early as possible. Let $BAO_k^y(t)$ be an upper bound on the total number of bus accesses due to all tasks having priority k or higher executing on core π_y. It is proven in (Davis et al., 2018b; Altmeyer et al., 2015) that $BAO_k^y(t)$ can be upper bounded using the following equation:

$$BAO_k^y(t) \leq \sum_{\tau_l \in \Gamma_y \cap \text{hep}(k)} W_{k,l}^y(t) + W_{k,l,cout}^y(t) \tag{9.2}$$

where $W_{k,l}^y(t)$ is an upper bound on the total number of bus accesses by the *carry-in* and *body* jobs of task τ_l that execute during a time interval of length t. The carry-in job of a task τ_l w.r.t a time interval of length t is a job that is released sometime before the start of t but has its deadline in t. Whereas, all jobs of task τ_l with both release times and deadlines in the time interval t are body jobs of τ_l. Let $N_{k,l}^y(t)$ be an upper bound on the maximum number of jobs of τ_l that may fully execute in an interval of length t on core π_y and considering that the maximum number of bus accesses each of those jobs can generate is upper bounded by $MD_l + \gamma_{k,l,y}$, we have

$$W_{k,l}^y(t) = N_{k,l}^y(t) \times (MD_l + \gamma_{k,l,y}) \tag{9.3}$$

where $N_{k,l}^y(t)$ is given by

$$N_{k,l}^y(t) = \left\lfloor \frac{t + R_l - (MD_l + \gamma_{k,l,y}) \times d_{mem}}{T_l} \right\rfloor \tag{9.4}$$

In Equation (9.2), $W_{k,l,cout}^y$ denote the maximum number of bus accesses by the *carry-out* job of task τ_l that may execute during the time interval t. The carry-out job of a task τ_l w.r.t a time interval of length t is a job that is released during the time interval t but has a deadline after t. The maximum number of bus accesses that can be generated by the carry-out job of task τ_l are computed as follows:

$$W_{k,l,cout}^y = \min\left(\left\lceil \frac{t + R_l - (MD_l + \gamma_{k,l,y}) \times d_{mem} - N_{k,l}^y(t) \times T_l}{d_{mem}} \right\rceil ; MD_l + \gamma_{k,l,y} \right) \tag{9.5}$$

The bus contention analysis presented in (Altmeyer et al., 2015; Davis et al., 2018b) uses Equation (9.1) and (9.2), to compute the total number of bus accesses BAT_i^x that may delay the execution

of τ_i on core π_x under different bus arbitration policies. For example, if the bus arbitration policy is Fixed-priority (FP) based, i.e., bus accesses inherit the priority of the task that generate them, then $BAT_i^x(t)$ is given by

$$BAT_i^x(t) = BAS_i^x(t) + \sum_{\forall \pi_y \neq \pi_x} BAO_i^y(t) + 1 + \min\left(BAS_i^x(t); \sum_{\forall \pi_y \neq \pi_x} BAO_{i,low}^y(t)\right) \tag{9.6}$$

Where $BAS_i^x(t)$ and $BAO_i^y(t)$ are calculated using Equation. (9.1) and (9.2) respectively. The sum $\sum_{\forall \pi_y \neq \pi_x} BAO_i^y(t)$ represents the worst-case bus delay of task τ_i, i.e., when all bus accesses from all tasks in hep(i) executing on other cores are served before the last bus access of τ_i. Typically, memory bus requests are non-preemptive therefore if the main memory receives a request from a lower priority task before the request from the higher priority task arrives, the memory request from the higher priority task may be served after the completion of the request from the lower priority task. However, in this scenario the maximum delay the higher priority task can suffer can only be of one memory access. Consequently, $+1$ in Equation. (9.6) accounts for that one bus access from any lower task in lp(i) executing on the same core as τ_i and can only occur at the start of busy period w.r.t. τ_i. Finally, the term $\min\left(BAS_i^x(t); \sum_{\forall \pi_y \neq \pi_x} BAO_{i,low}^y(t)\right)$ in Equation. (9.6) upper bounds the bus interference due to accesses by tasks in lp(i) executing on cores other than π_x. Note that $BAO_{i,low}^y(t)$ is calculated in a similar manner to $BAO_i^y(t)$ (i.e., Equation. (9.2)), but considering bus accesses from tasks having a lower priority than τ_i, i.e., $BAO_{i,low}^y(t) = \sum_{\forall \tau_l \in \Gamma_y \cap \text{lp}(i)} W_{i,l}^y(t) + W_{i,l,cout}^y$.

Similarly, it is shown in (Altmeyer et al., 2015; Davis et al., 2018b) that if the bus arbitration policy is Round-Robin (RR) then $BAT_i^x(t)$ can be computed as follows:

$$BAT_i^x(t) = BAS_i^x(t) + \sum_{\forall \pi_y \neq \pi_x} \min\left(BAO_n^y(t); sl \times BAS_i^x(t)\right) + 1 \tag{9.7}$$

where sl denote the number of memory access slots per core. Under a RR bus, the worst-case delay is suffered by task τ_i when each accesses in $BAS_i^x(t)$ is delayed by all cores other than π_x for sl slots. However, since n represents the lowest priority in the system, $BAO_n^y(t)$ also upper bounds the bus accesses due to all tasks executing on core π_y. Therefore, the maximum number of bus access by all tasks executing on core π_y that may delay the execution of $\tau_i \in \pi_x$ are upper bounded by $\min\left(BAO_n^y(t); sl \times BAS_i^x(t)\right)$.

If the bus arbitration policy is non-work conserving, i.e., TDMA, then $BAT_i^x(t)$ is upper bounded by

$$BAT_i^x(t) = BAS_i^x(t) + ((L-1) \times sl) \times BAS_i^x(t) + 1 \tag{9.8}$$

where the length of one TDMA cycle is $L \times sl$. Since TDMA is non-work conserving, it assumes that each bus access in $BAS_i^x(t)$ will always be delayed by $(L-1) \times sl$ bus accesses by other cores irrespective of whether these slots are used or not (in contrast to Round-Robin). For more details on the formulation of Equation (9.1) -(9.8) readers are referred to (Altmeyer et al., 2015; Davis et al., 2018b).

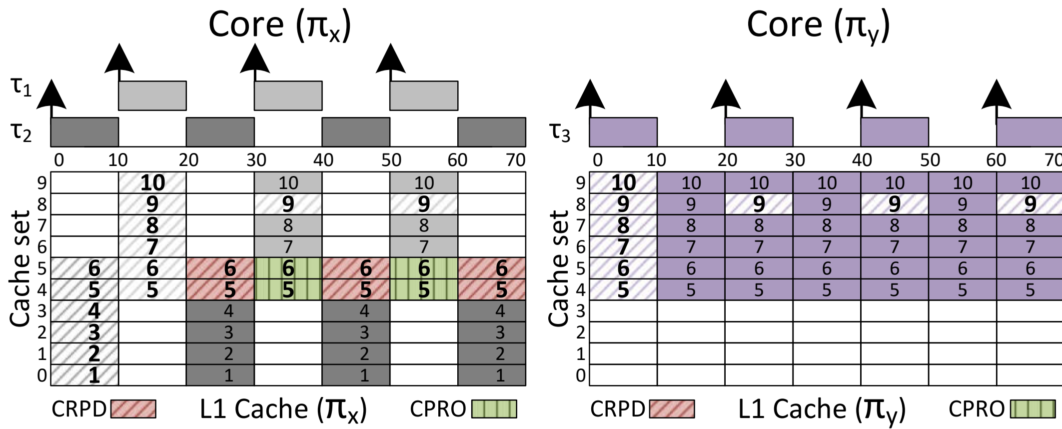


Figure 9.1: Execution of task τ_1 and τ_2 on core π_x and task τ_3 on core π_y. Task parameters of interest are: PD_1=PD_3 = 4, PD_2= 32, MD_1=MD_3 = 6, MD_2 = 8, MD^r_1=MD^r_3 = 1, ECB_1=ECB_3 = $\{5,6,7,8,9,10\}$, $ECB_2 = \{1,2,3,4,5,6\}$, PCB_1=PCB_3 = $\{5,6,7,8,10\}$ and $UCB_2 = \{5,6\}$.

9.3 Cache Persistence-aware Memory Bus Contention Analysis

The memory bus contention analysis presented in (Altmeyer et al., 2015; Davis et al., 2018b) (i.e., Equation (9.1) and (9.2)) provides a safe upper bound on the memory bus delay suffered by tasks executing on a multicore platform. However, since the analysis in (Altmeyer et al., 2015; Davis et al., 2018b) does not consider the variation in the memory access demand of tasks due to cache persistence, it may overestimate the actual number of accesses that compete for the bus during the response time of the task under analysis. Recall that cache persistence refers to the re-use of PCBs between different job executions of the same task. If all PCBs of a task τ_i were loaded in the cache by a previous job of τ_i, the memory access demand of all subsequent jobs of τ_i can be much lower than the worst-case memory access demand of τ_i in isolation. This type of memory demand is called the residual memory access demand of τ_i and is denoted by MD^r_i (see Definition 4.3). According to Definition 4.3, a PCB is loaded only once from the main memory when a task τ_i executes in isolation. Therefore, the total number of bus accesses generated by n_i jobs of τ_i executing in isolation can be computed by defining Equation (4.4) as a function of number of jobs n_i, i.e.,

$$\hat{MD}_i(n_i) = \min\left(n_i \times MD_i\,; n_i \times MD^r_i + |PCB_i|\right) \tag{9.9}$$

We now consider the schedule and task parameters shown in Figure 9.1 to show how Equation (9.9) can be used to reduce the pessimism of (Altmeyer et al., 2015; Davis et al., 2018b). We have three tasks τ_1, τ_2 and τ_3 with τ_1 and τ_2 executing on core π_x and τ_3 executing on core π_y. We assume τ_1 has the highest priority and τ_3 the lowest. The worst-case main memory access demands MD_1, MD_2 and MD_3 are 6, 8 and 6 respectively. Memory blocks in Figure 9.1 that are pattern filled are those that are loaded/reloaded from the main memory during the task executions. We focus on τ_2 and use $BAT^x_2(R_2)$ to denote the total number of bus accesses that may be generated during its response time. Assuming that the memory bus arbitration policy is Round-Robin (RR)

with a slot size sl equal to 1, $BAT_2^x(R_2)$ can be bounded using Equation (9.7) such that

$$BAT_2^x(R_2) = BAS_2^x(R_2) + \min\left(BAO_3^y(R_2); BAS_2^x(R_2)\right) \tag{9.10}$$

where

$$BAS_2^x(R_2) = MD_2 + 3 \times (MD_1 + \gamma_{2,1,x}) = 8 + 3 \times (6+2) = 8 + 18 + 6 = 32 \tag{9.11}$$

$$BAO_3^y(R_2) = N_{3,3}^y(R_2) \times MD_3 = 4 \times (6) = 24 \tag{9.12}$$

Note that in Equation (9.11) $\gamma_{2,1,x}$ is derived using Equation (3.4) however, any other CRPD analysis can also be used to compute $\gamma_{2,1,x}$. Moreover, since τ_2 is the lowest priority task on core π_x, Equation (9.11) does not have a trailing +1 as in Equation (9.7).

Now, if we compare the result of Equation (9.11) with the cache contents of core π_x shown in Figure 9.1 we can see that Equation (9.11) overestimates the value of $BAS_2^x(R_2)$. Figure 9.1 shows that only the first job of τ_1 needs to load all its ECBs from the main memory and hence has a worst-case memory access demand $MD_1 = 6$. Moreover, since all PCBs of τ_1 were loaded in the cache by the first job of τ_1, the memory access demand of the next two jobs of τ_1 only corresponds to the reloading of memory block $\{9\}$, i.e., $MD_1^r = 1$. Consequently, the actual number of memory accesses made by the three jobs of τ_1 executing during the response time of τ_2 are respectively $MD_1 + MD_1^r + MD_1^r = 6 + 1 + 1 = 8$, which is much lower than $3 \times MD_1 = 18$ accounted for in Equation (9.11).

Figure 9.1 also shows an overlap between PCBs $\{5,6\}$ of τ_1 and ECBs $\{5,6\}$ of τ_2 in cache. This overlap may lead to additional bus accesses due to CPRO, i.e., to reload PCBs $\{5,6\}$ from the main memory between two subsequent executions of τ_1. The additional bus accesses due to CPRO suffered by n_i jobs of a higher priority task $\tau_j \in \mathrm{hp}(i)$ executing during the response time of a lower priority task τ_i on a core π_x is given by $\hat{\rho}_{j,i,x}(n_i)$, i.e.,

$$\hat{\rho}_{j,i,x}(n_j) = (n_j - 1) \times \rho_{j,i,x} \tag{9.13}$$

where $\rho_{j,i,x}$ denote the additional bus accesses due to CPRO suffered by on job of task τ_j during the response time of task τ_i and it can be computed using any of the CPRO analysis presented in Chapter 4, 5 or Chapter 7. For example, if we use the CPRO-union approach (see Section 4.3) to compute the additional bus accesses due to CPRO suffered by task τ_1 during the response of τ_2, for the schedule shown in Figure 9.1 we will get $\hat{\rho}_{1,2,x}(3) = 2 \times 2 = 4$. Therefore, due to cache persistence, the actual number of bus accesses during the response time task τ_2 on core π_x are given by

$$MD_2 + MD_1 + 2 \times MD_1^r + \hat{\rho}_{1,2,x}(3) + 3 \times \gamma_{2,1,x} = 26 \tag{9.14}$$

which is much lower than the value of $BAS_2^x(R_2) = 32$ calculated using Equation (9.11). This leads to the following lemma.

Lemma 9.1. *The total number of bus accesses by a single job of task $\tau_i \in \Gamma_x$ and all higher priority tasks in $\Gamma_x \cap \mathrm{hp}(i)$ executing in a time interval of length t are upper bounded by $B\hat{A}S_i^x(t)$, where*

$$B\hat{A}S_i^x(t) = MD_i + \sum_{\forall \tau_j \in \Gamma_x \cap \mathrm{hp}(i)} \min\Big(E_j(t) \times MD_j; \hat{MD}_j(E_j(t)) + \hat{\rho}_{j,i,x}(E_j(t))\Big) + \sum_{\forall \tau_j \in \Gamma_x \cap \mathrm{hp}(i)} E_j(t) \times \gamma_{i,j,x} \text{ with } E_j(t) = \left\lceil \frac{t}{T_j} \right\rceil \tag{9.15}$$

Proof. We prove that in a time interval of length t, $B\hat{A}S_i^x(t)$ is an upper bound on the total number of bus accesses generated by task $\tau_i \in \Gamma_x$ and all higher priority tasks in $\Gamma_x \cap \mathrm{hp}(i)$.
1. By assumption, only one job of τ_i must be considered. Hence, the total number of bus accesses generated by τ_i are upper bounded by its worst-case memory access demand MD_i.
2. Any task $\tau_j \in \Gamma_x \cap \mathrm{hp}(i)$ can release at most $E_j(t) = \left\lceil \frac{t}{T_j} \right\rceil$ jobs in a time window of length t. Therefore, it follows from Equation (9.1) that $E_j(t) \times MD_j$ is an upper bound on the total number of bus accesses generated by task $\tau_j \in \Gamma_x \cap \mathrm{hp}(i)$ in isolation. Moreover, the additional bus accesses due to preemptions of task τ_i by task τ_j in a time interval of length t are upper bounded by $E_j(t) \times \gamma_{i,j,x}$ (see Equation 9.1). Hence, the sum $MD_i + E_j(t) \times MD_j + E_j(t) \times \gamma_{i,j,x}$ is an upper bound on the total number of bus accesses generated by task $\tau_i \in \Gamma_x$ and any higher priority task τ_j in $\Gamma_x \cap \mathrm{hp}(i)$ in a time interval of length t.
3. Recall from Equation (9.9) and (9.13) that $\hat{MD}_j(E_j(t))$ is an upper bound on the total number of bus accesses due to $E_j(t)$ jobs of τ_j executing in isolation and $\hat{\rho}_{j,i,x}(E_j(t))$ is an upper bound on the additional bus accesses due to CPRO suffered by all those jobs of τ_j. Therefore, the sum $MD_i + \hat{MD}_j(E_j(t)) + \hat{\rho}_{j,i,x}(E_j(t)) + E_j(t) \times \gamma_{i,j,x}$ is also an upper bound on the total number of bus accesses generated by task $\tau_i \in \Gamma_x$ and any higher priority task τ_j in $\Gamma_x \cap \mathrm{hp}(i)$ in a time window of length t considering both CRPD and CPRO. Thus, the minimum between $MD_i + E_j(t) \times MD_j + E_j(t) \times \gamma_{i,j,x}$ and $MD_i + \hat{MD}_j(E_j(t)) + \hat{\rho}_{j,i,x}(E_j(t)) + E_j(t) \times \gamma_{i,j,x}$ is also an upper bound. The lemma follows. □

Continuing the example depicted in Figure 9.1, we can see that Equation (9.12) also overestimates the value of $BAO_3^y(R_2)$. In fact, due to cache persistence, the actual number of bus accesses generated by task $\tau_3 \in \Gamma_y$ that may contend for the bus during the response time of task $\tau_2 \in \Gamma_x$ are: $MD_3 + 3 \times MD_3^r = 6 + 3 \times 1 = 9$, which is much lower than the value of $BAO_3^y(R_2) = 24$ calculated using Equation (9.12). This observation leads to the following lemma.

Lemma 9.2. *The total number of bus accesses by all tasks $\in \Gamma_y$ with priority k or higher that may contend for bus access with task $\tau_i \in \Gamma_x$ during a time window of length t is upper bounded by*

$$B\hat{A}O_k^y(t) = \sum_{\forall \tau_l \in \Gamma_y \cap \mathrm{hep}(k)} \hat{W}_{k,l}^y(t) + W_{k,l,cout}^y \tag{9.16}$$

where

$$\hat{W}_{k,l}^y(t) = \min\Big(N_{k,l}^y(t) \times MD_l; \hat{MD}_l(N_{k,l}^y(t)) + \hat{\rho}_{k,l,y}(N_{k,l}^y(t))\Big) + N_{k,l}^y(t) \times \gamma_{k,l,y} \tag{9.17}$$

and $W^{y}_{k,l,cout}$ and $N^{y}_{k,l}(t)$ are given by Equation (9.5) and (9.4) respectively.

Proof. Since Equation (9.5) (i.e., $W^{y}_{k,l,cout}$) is proved in (Altmeyer et al., 2015; Davis et al., 2018b), we only need to prove that $\hat{W}^{y}_{k,l}(t)$ is an upper bound on the total number of bus accesses by jobs of $\tau_l \in \Gamma_y \cap \text{hep}(k)$ that fully execute in a time interval of length t.
1. It follows from Equation (9.4) that $N^{y}_{k,l}(t)$ is an upper bound on the number of jobs that may be fully executed in a time interval of length t by any task $\tau_l \in \Gamma_y \cap \text{hep}(k)$. Therefore, $N^{y}_{k,l}(t) \times MD_l$ upper bounds the total number of bus accesses generated by task τ_l in a time interval of length t in isolation. Moreover, $N^{y}_{k,l}(t) \times \gamma_{k,l,y}$ is an upper bound on the additional bus accesses due to CRPD suffered by task $\tau_l \in \Gamma_y \cap \text{hep}(k)$ in a time window of length t. Therefore, the sum $N^{y}_{k,l}(t) \times MD_l + N^{y}_{k,l}(t) \times \gamma_{k,l,y}$ is an upper bound on the total number of bus accesses by jobs of $\tau_l \in \Gamma_y \cap \text{hep}(k)$ that fully execute in a time interval of length t.
2. Using $N^{y}_{k,l}(t)$ in Equation (9.9) and (9.13) we get $\hat{MD}_l(N^{y}_{k,l}(t))$ which is an upper bound on the total number of bus accesses due to $N^{y}_{k,l}(t)$ successive jobs of τ_j executing in isolation and $\hat{\rho}_{k,l,y}(N^{y}_{k,l}(t))$ which is an upper bound on the additional bus accesses due to CPRO suffered by all those jobs. Hence, the sum $\hat{MD}_l(N^{y}_{k,l}(t)) + \hat{\rho}_{k,l,y}(N^{y}_{k,l}(t)) + N^{y}_{k,l}(t) \times \gamma_{k,l,y}$ is also an upper bound on the total number of bus accesses by jobs of $\tau_l \in \Gamma_y \cap \text{hep}(k)$ that fully execute in a time interval of length t considering both CRPD and CPRO. Consequently, the minimum between $N^{y}_{k,l}(t) \times MD_l + N^{y}_{k,l}(t) \times \gamma_{k,l,y}$ and $\hat{MD}_l(N^{y}_{k,l}(t)) + \hat{\rho}_{k,l,y}(N^{y}_{k,l}(t)) + N^{y}_{k,l}(t) \times \gamma_{k,l,y}$ is also an upper bound. The lemma follows. □

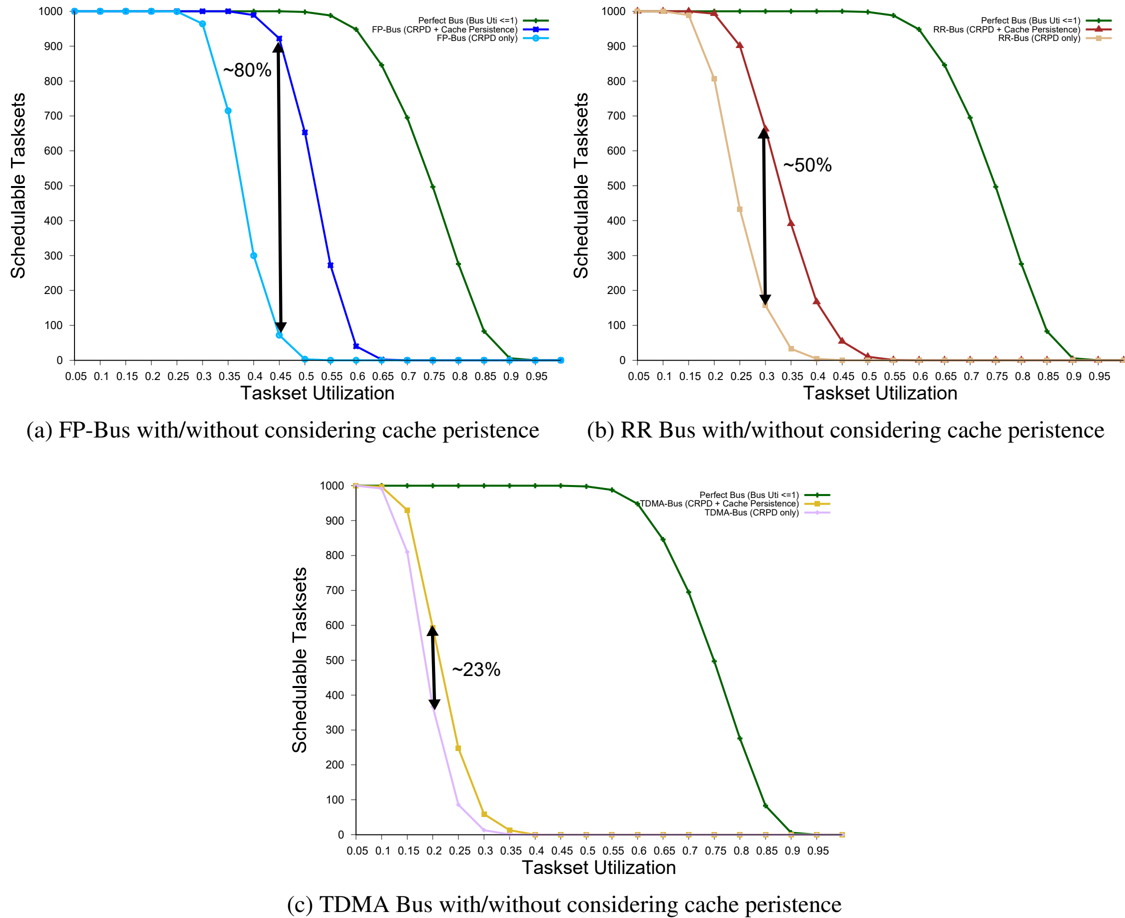
Note that having bounded the values of $B\hat{A}S^{x}_{i}(t)$ (i.e., Lemma 9.1) and $B\hat{A}O^{y}_{k}(t)$ (i.e., Lemma 9.2) these value can be directly used in Equation (9.6), (9.7) or (9.8) to bound the value of $BAT^{x}_{i}(t)$ due to cache persistence for any given bus arbitration policy.

9.4 Bus Contention-Aware Worst-case Response Time (WCRT) Analyses

If the memory bus contention that can be suffered by a task $\tau_i \in \Gamma_x$ during its execution is explicitly considered then the worst-case response time R_i of a task τ_i is given by the smallest possible solution of the following expression

$$R_i = PD_i + \sum_{\forall \tau_j \in \Gamma_x \cap \text{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \times PD_j + BAT^{x}_{i}(R_i) \times d_{mem} \tag{9.18}$$

where PD_i and PD_j are the worst-case processing demands of task τ_i and τ_j respectively. The term $\sum_{\forall \tau_j \in \Gamma_x \cap \text{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \times PD_j$ upper bounds the total core interference suffered by task τ_i due to preemptions by higher priority tasks executing on the same core, whereas the total memory bus interference that τ_i may suffer during R_i is upper bounded by $BAT^{x}_{i}(R_i) \times d_{mem}$. Depending on the bus arbitration policy, $BAT^{x}_{i}(R_i)$ can be calculated using Equation. (9.6), (9.7) or (9.8). When considering the bus contention analysis that only accounts for CRPDs the values of $BAS^{x}_{i}(t)$ and



(a) FP-Bus with/without considering cache peristence

(b) RR Bus with/without considering cache peristence

(c) TDMA Bus with/without considering cache peristence

Figure 9.2: Schedulability ratio of different bus arbitration policies by varying total core utilizations

$BAO_k^y(t)$ are computed using Equation (9.1) and (9.2) respectively. For cache persistence-aware bus contention analysis the values of $BAS_i^x(t)$ and $BAO_k^y(t)$ are computed using Lemma 9.1 and Lemma 9.2, respectively. Note that the response time of each task may depend on the response times of other tasks. This circular dependency is solved using a fixed point iteration over all response times, first initiating the response time of each task to $PD_i + MD_i \times d_{mem}$ and stopping as soon as all response times remain constant or there is one task with $R_i > D_i$, in which case the task is deemed unschedulable.

9.5 Experimental Evaluation

In this section, we evaluate the impact of inter-task cache interference on memory bus contention in multicore platforms. As we have mentioned previously, we analyze multicore architectures with two types of cache configurations, i.e., (i) with only one cache level and (ii) with multiple cache levels. When analyzing architectures with single-level caches, we compare the performance of different bus arbitration policies with/without considering cache persistence. For architectures

that support multiple cache levels, we evaluate how our proposed CRPD analysis for multilevel caches (presented in Chapter 8) can impact memory bus contention in comparison to the state-of-the-art CRPD analysis for multilevel caches presented in (Chattopadhyay and Roychoudhury, 2014).

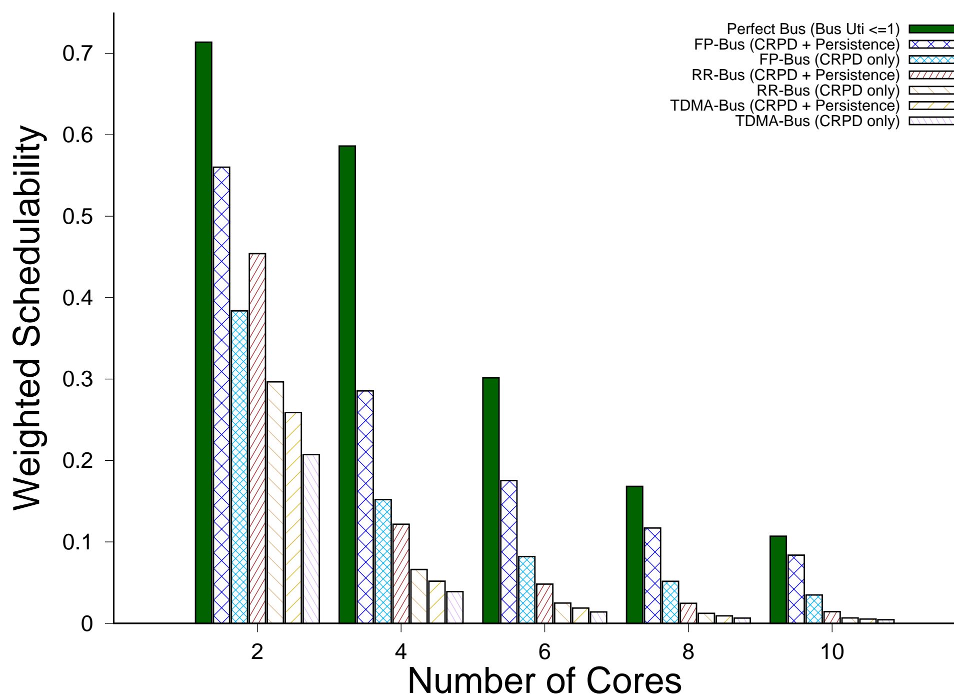


Figure 9.3: Wighted schedulability measure by varying the total number of cores

9.5.1 Multicore Platforms with Single-level Caches

To evaluate how a tighter bound on the inter-task cache interference can impact memory bus contention in multicore platforms with single-level caches, we compare the performance of FP, RR and TDMA bus arbitration policies under two approaches; (i) that only accounts for CRPDs and (ii) that accounts for both CRPDs and cache persistence. For bus contention analysis that only accounts for CRPDs we compute the CRPDs using the Resilience analysis (Altmeyer et al., 2010) (see Section 3.2.2). Cache persistence-aware bus contention analysis computes the CRPDs using the Resilience analysis and CPROs using the Multi-path ResilienceP analysis (see Section 7.4).

We model a multicore platform with 4 cores each having a private L1 instruction cache with 4-ways, 64 cache sets and a block size of 32 Bytes. The default value of d_{mem} is $5\mu s$. All experiments were performed using the Mälardalen benchmark suite (Gustafsson et al., 2010) with the values of PD_i, MD_i, MD_i^r, UCB_i, ECB_i and PCB_i taken from Table 5.2. The default task set size was 24 with 6 tasks per core. Each task within the task set is randomly assigned parameters from one of the benchmarks of the Mälardalen benchmark suite. Task utilizations were generated using UUnifast (Bini and Buttazzo, 2005) assuming an equal utilization for each core. Task periods

and deadlines were set such that $T_i = D_i = (PD_i + MD_i \times d_{mem})/U_i$. Task priorities were assigned according to deadline monotonic.

We randomly generated task sets and determined their schedulability using Equation (9.18) for FP, RR, and TDMA buses with and without considering cache persistence under different settings, i.e., by varying core utilizations, number of cores, memory reload time d_{mem}, cache size and RR/TDMA slot size sl that has a default value of 2.

1. Core Utilizations: In this experiment, we varied the per core utilization between 0.05 to 1 in steps of 0.05. For every value of core utilization, 1000 task sets were generated. Figure 9.2 shows the number of task sets that were deemed schedulable by FP, RR and TDMA bus arbitration policies with and without considering cache persistence. Figure 9.2 also shows a line marked as "perfect bus" which assumes that there is no interference on the memory bus when the bus utilization ≤ 1. That line provides an upper bound on the actual number of schedulable task sets at a particular core utilization. We can see in Figure 9.2 that bus arbitration policies that account for cache persistence dominate their counterparts that do not consider cache persistence. This improved performance mainly results from a tighter bound on the number of bus request generated by tasks executing on different cores. We can see in Figure 9.2a that for a FP bus, up to 80% more task sets were schedulable when considering cache persistence. Similarly, we can also see huge improvements for both RR (up to 50% more schedulable task sets) and TDMA (up to 23% more schedulable task sets). Note that the FP bus outperforms the RR and TDMA buses since it provides a tightly bounded bus latency for single accesses which is not the case with RR and TDMA.

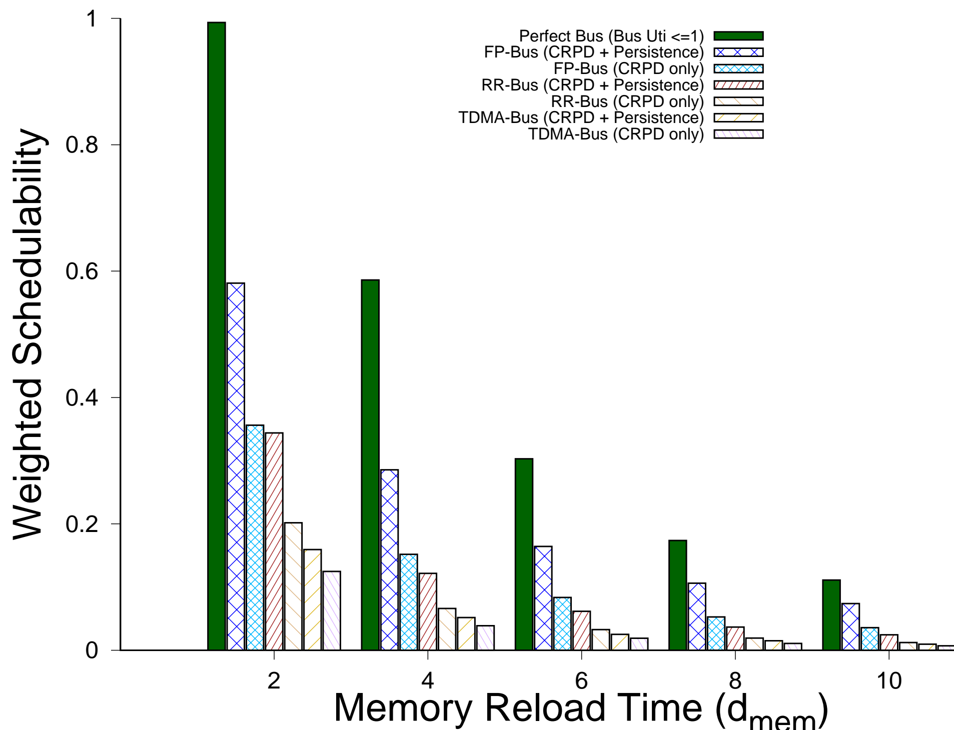


Figure 9.4: Wighted schedulability measure by varying the value of memory reload time d_{mem}

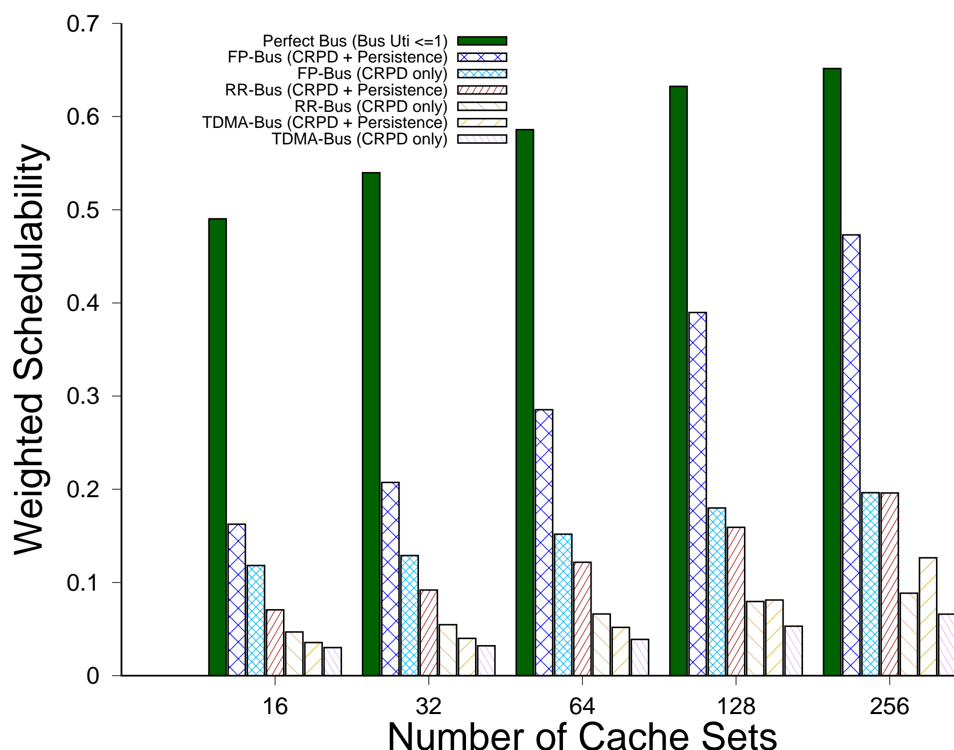


Figure 9.5: Wighted schedulability measure by increasing cache size between 2kB to 32kB

2. Number of Cores: In this experiment, we varied the number of cores between 2 and 10 in steps of 2 with all other parameters set to the default values. We have used the weighted schedulability measure defined in (Bastoni et al., 2010) (i.e., Equation (4.20)) to plot the results in Figure 9.3. We can see in Figure 9.3 that by increasing the number of cores the total number of schedulable task sets decreases. This is mainly because by increasing the number of cores the number of tasks also increases. This leads to an increases in the interference on the memory bus. However, we can see that analyses that account for cache persistence always dominate their counterparts that do not account for cache persistence.

3. Memory Reload Time d_{mem}: For this experiment, we varied the value of memory reload time d_{mem} from $2\mu s$ to $10\mu s$ in step of $2\mu s$ and evaluated its impact on the performance of all bus arbitration policies. The results are presented in Figure 9.4. We can see in Figure 9.4 that for lower values of d_{mem} the difference between the weighted schedulability of cache persistence-aware analyses and their respective counterparts is higher. However, for higher value of d_{mem} the time spent by tasks in performing memory operations increases and hence the schedulability of all approaches decreases.

4. Cache Size: To evaluate the impact of cache size on the performance of the analyses, we varied the size of the L1 cache of each core between 2kB to 32kB by increasing the number of sets in the cache from 16 to 256. Default values were used for all other parameters. The results are shown in Figure 9.5. We can see in Figure 9.5 that by increasing the cache size, the number of schedulable task sets under bus arbitration policies that account for cache persistence also increases. This is

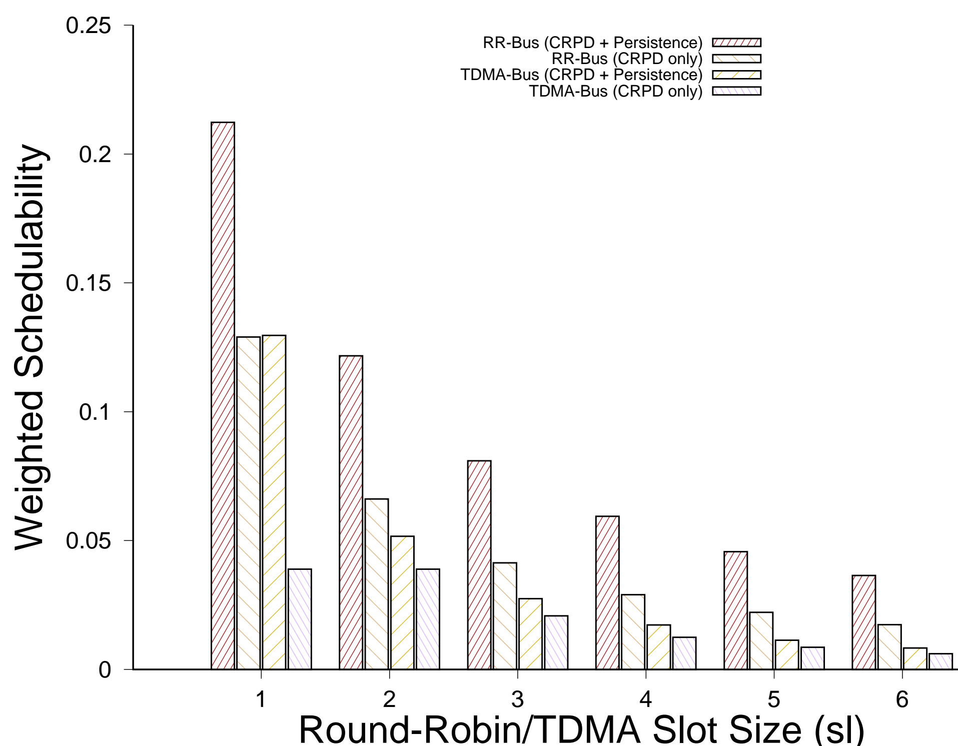


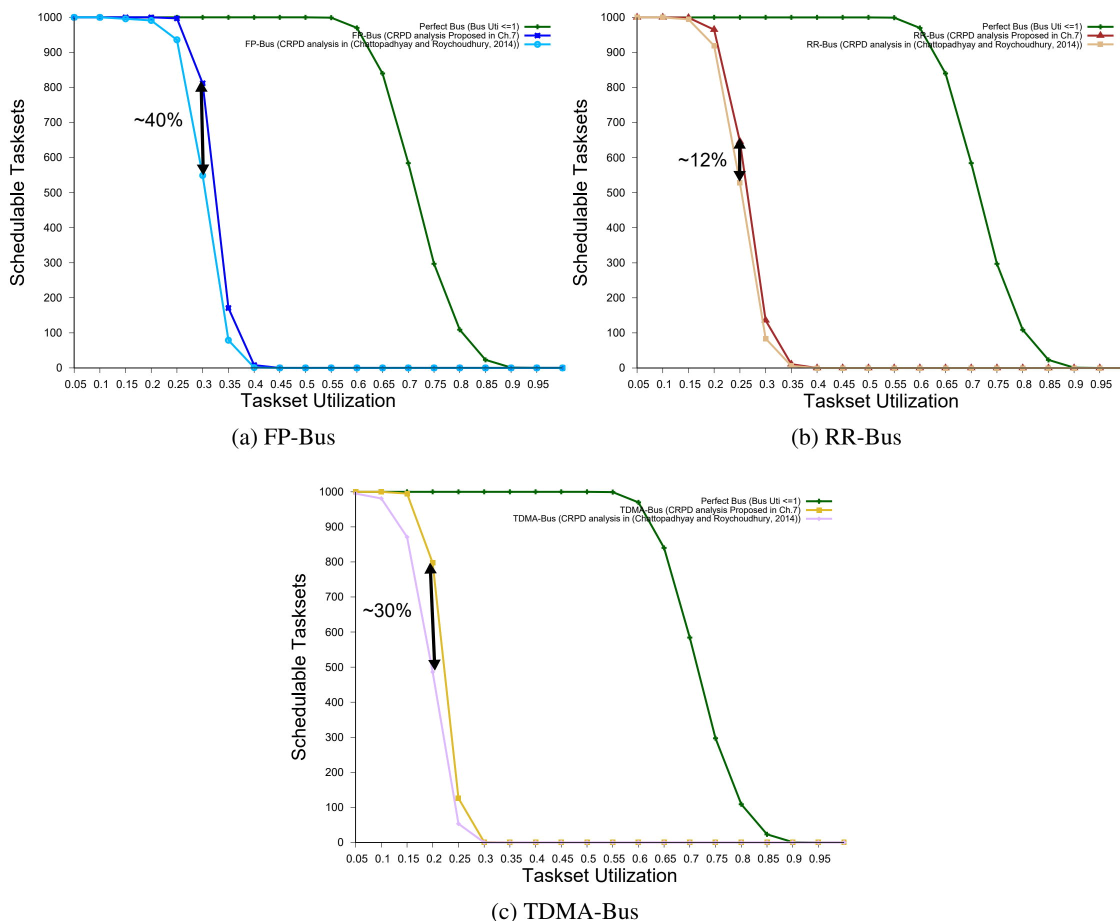
Figure 9.6: Wighted schedulability measure by varying the RR/TDMA slot size (*sl*)

mainly because by increasing the cache size the number of PCBs of each task also increases, which results in gradually improving the performance of the analyses that account for cache persistence. Note that the increase in cache size also reduce CRPD and thus increase the task set schedulability for analyses that do not account for cache persistence, but at a slower rate than for persistence aware analyses.

5. RR/TDMA slot size *sl*: For RR/TDMA buses the number of memory access slots per core, i.e., *sl*, can greatly affect the memory bus contention suffered by tasks. To evaluate this, we varied the value of slot size *sl* between 1 to 6 and plotted the results in Figure 9.6. The results show that for smaller values of *sl*, the difference between the weighted schedulability of cache persistence aware analyses and their respective counterparts is much higher. However, by increasing the value of *sl* the task set schedulability of all approaches decrease, which is intuitive, considering the formulation of Equation (9.7) and (9.8).

9.5.2 Multicore Platforms with Multilevel Caches

To evaluate the impact of inter-task cache interference on memory bus contention in multicore platforms that support multilevel caches, we compared the performance of different bus arbitration policies considering two approaches; (i) that computes the inter-task cache interference using our proposed CRPD analysis for multilevel caches presented in Chapter 8 (i.e., Equation (8.19)) and (ii) that computes the inter-task cache interference using the CRPD analysis of (Chattopadhyay and Roychoudhury, 2014) (i.e., Equation (8.10)). Effectively, the goal is to conclude if a tighter



(a) FP-Bus

(b) RR-Bus

(c) TDMA-Bus

Figure 9.7: Schedulability ratio of different bus arbitration policies by varying total core utilizations for multicore architectures with two-level caches.

bound on the CRPD can improve schedulability of tasks executing on a multicore platform that support multilevel caches.

We model a multicore platform having 4 cores each supporting private two-level instructions caches, i.e., L1 and L2. L1 cache is 2-way set-associative with 32 sets and line size of 32-bytes. L2 cache is 4-way set-associative with 64 sets and line size of 64-bytes. The L1 cache miss penalty was 10 processor cycles, i.e., $d_{L1} = 10$, and the L2 cache miss penalty was 100 processor cycles. Consequently, the total time to reload one memory block from the main memory to both cache levels is computed such that $d_{mem} = d_{L1} + d_{L2} = 110$. All experiments were performed using the Mälardalen benchmark suite (Gustafsson et al., 2010) with the values of C_i, L1-ECBs, L2-ECBs, L1-UCBs and L2-UCBs taken from Table 8.2. The values of worst-case processing demand PD_i and worst-case memory access demand MD_i were chosen randomly such that $MD_i = rand(0.1, 0.6) * C_i$ and $PD_i = C_i - MD_i$. The default task set size was 32 with 8 tasks per core. Each task within the task set is randomly assigned parameters from one of the benchmarks of the Mälardalen benchmark suite. Task utilizations were generated using UUnifast (Bini and Buttazzo, 2005) assuming an equal utilization for each core. Task periods and deadlines were set such that

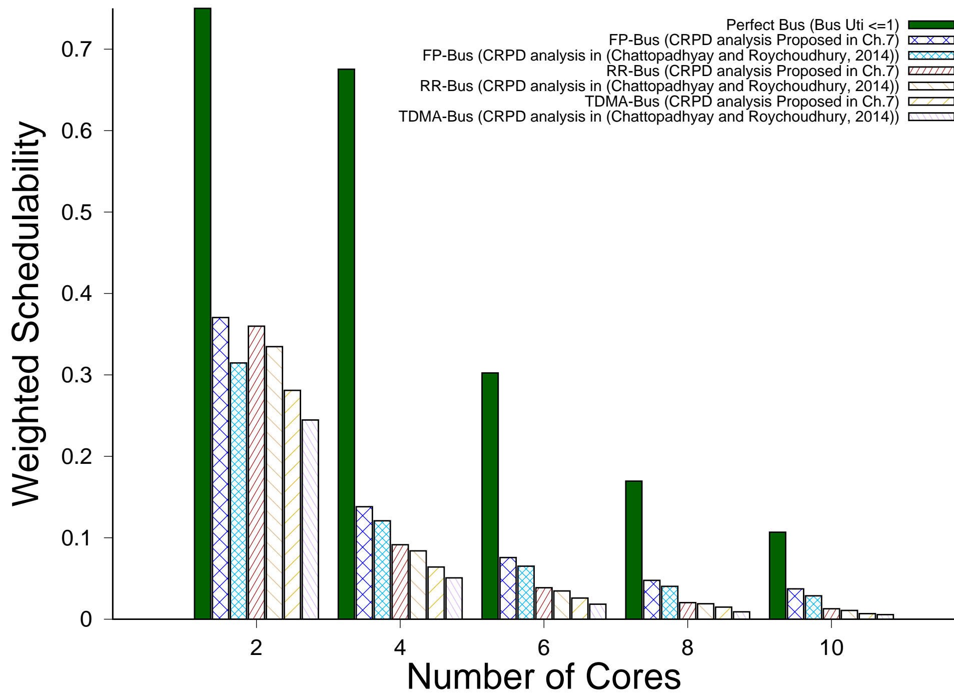


Figure 9.8: Wighted schedulability measure by varying the total number of cores in multicore platforms with two-level caches

$T_i = D_i = (PD_i + MD_i \times d_{mem})/U_i$. Task priorities were assigned according to deadline monotonic.

We randomly generated a large number of task sets and determined their schedulability using Equation (9.18) for FP, RR, and TDMA buses. Depending on the chosen bus arbitration policy, upper bound on the memory bus contention, i.e., $BAT_i^x(R_i)$ in Equation (9.18), can be computed using Equation (9.6), (9.7) or (9.8). For every bus arbitration policy the CRPD is either computed using our proposed CRPD analysis for multilevel caches, i.e., Equation (8.19)), or by using the state-of-the-art CRPD analysis for multilevel caches presented in (Chattopadhyay and Roychoudhury, 2014) (i.e., Equation (8.10)). The schedulability analysis is then performed under different settings. Note that for RR/TDMA bus, the default value of slot size sl was 2.

1. Core Utilizations: For this experiment, we varied the per core utilization between 0.05 to 1 in steps of 0.05 and generated 1000 task sets at each step. Figure 9.7 shows the number of task sets that were deemed schedulable at each step by FP, RR and TDMA bus arbitration policies that use our proposed CRPD analysis for multilevel caches, i.e., Equation (8.19)), and the state-of-the-art CRPD analysis for multilevel caches presented in (Chattopadhyay and Roychoudhury, 2014) (i.e., Equation (8.10)). In Figure 9.7, we can see that the performance of all bus arbitration policies is similar to what was observed in Figure 9.2. However by comparison, the total number of task sets that were deemed schedulable by each approach is lower. This is mainly because the analysis results plotted in Figure 9.7 only considers CRPDs when computing the memory bus contention of tasks and does not account for cache persistence which is considered by the analysis results shown in Figure 9.2. Overall, we can see that the bus arbitration policies that consider a tighter

CRPD bound (i.e., given by the CRPD analysis presented in Chapter 8) in the computation of memory bus contention, dominate their counterparts that use the state-of-the-art CRPD analysis presented in (Chattopadhyay and Roychoudhury, 2014). This shows inter-task cache interference can also have a significant impact on the schedulability of tasks executing on a multicore platform that support multilevel caches.

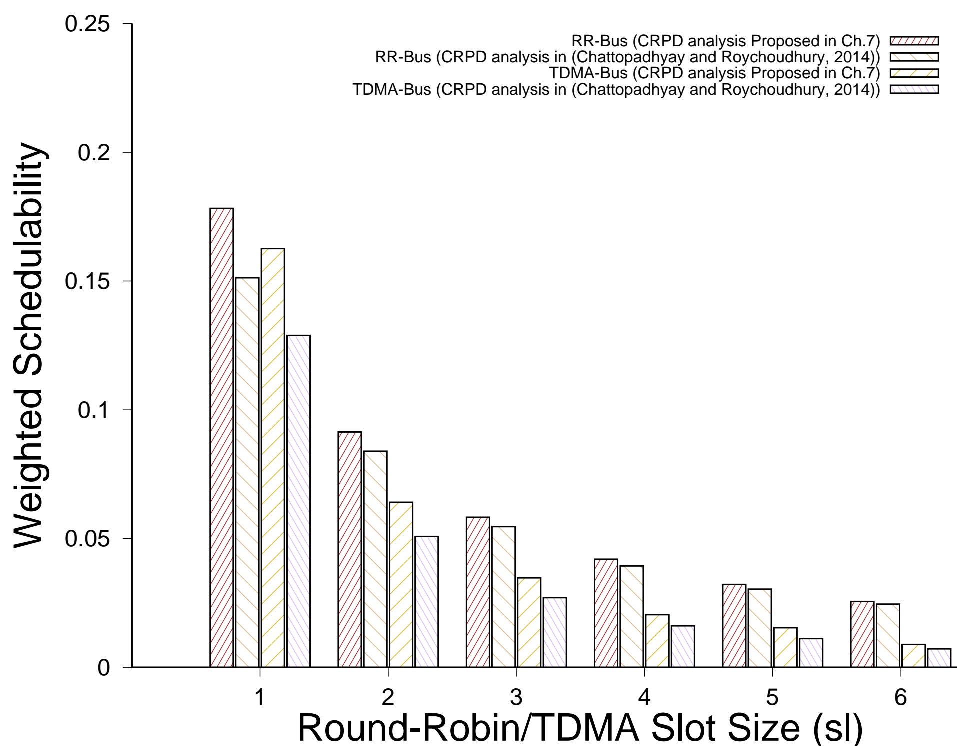


Figure 9.9: Wighted schedulability measure by varying the RR/TDMA slot size (sl) for multicore platforms with two-level caches.

2. Number of Cores: In this experiment, we increased the number of cores in the platform between 2 to 10 in steps of 2 and plotted the resulting weighted schedulability measure in Figure 9.8. We can see a similar trend in Figure 9.8 which was observed for the results plotted in Figure 9.3, i.e., by increasing the number of cores (which also increases the total number of tasks) the total number of task sets deemed schedulable by all approaches decreases. This is due to an increase in the inter-task cache interference due to CRPDs and since the analyses considered in Figure 9.8 does not account for cache persistence we can see a much faster decrease in schedulability in comparison to Figure 9.3. However, we can see that our proposed CRPD analysis for multilevel caches still dominates the state-of-the-art analysis.

3. RR/TDMA slot size sl: In another experiment, we varied the value of RR/TDMA slot size sl from 1 to 6 and plotted the resulting weighted schedulability in Figure 9.9. Again, we can see that the plot shown in Figure 9.9 is similar to the plot shown in Figure 9.6 but, the overall weighted schedulability of all approaches is lower. We can also see that for smaller values of sl a tighter CRPD bound improves task set schedulability. However, for higher values of sl the inter-task

cache interference from tasks executing on other cores also increases. This results in decreasing the number of task sets deemed schedulable by all approaches.

From the results presented in Section 8.6.2, we know that our proposed CRPD analysis for multilevel caches mainly benefits from a tighter bound on the CRPD due to L2 cache. Therefore, we also performed experiments by varying L2 cache miss penalty and L2 cache size and observed a similar trend which is shown in Figure 9.4 and 9.5 respectively.

9.6 Chapter Summary

In this chapter, we evaluate the impact of inter-task cache interference on task set schedulability in multicore platforms. We show that the memory bus contention suffered by tasks executing on a multicore platform heavily depends on the inter-task cache interference suffered by the tasks. We perform different experiments by considering multicore platforms with single- and multi-level caches. The experimental results show that the analyses that provides a tighter bound on the inter-task cache interference significantly reduce the memory bus contention suffered by tasks executing on multicore platforms, thereby improving schedulability.

Chapter 10

Thesis Summary, Limitations and Future Directions

For hard real-time systems that allow task preemptions, a sound estimate of inter-task cache interference is a prerequisite for an accurate schedulability analysis. However, bounds on the inter-task cache interference must also be as precise as possible such that system resources are not underutilized. In this thesis, we have identified the sources of pessimism in the state-of-the-art computation of inter-task cache interference and proposed solutions that improve the accuracy of the schedulability analysis by providing tighter bounds on the inter-task cache interference. We also provide a holistic insight into the relationship between inter-task cache interference and memory bus contention and show how a tighter bound on inter-task cache interference can impact the memory bus contention suffered by tasks executing on a multicore platform.

10.1 Summary of Contributions

The work done in this dissertation is divided into three parts. In the first part, i.e., Chapter 4-6, we focus on systems with single-level direct-mapped caches. In Chapter 4, we show that the existing analysis in literature that only focus on *cache-related preemption delays* may overestimate the inter-task cache interference suffered by the tasks. To remove this overestimation, we introduce the notion of *cache persistence* that enables the reuse of cache content between different task instances, thus, providing a tighter bound on the inter-task cache interference. However, we also identify that persistent cache content of tasks can also be evicted due to inter-task cache conflicts which may lead to additional memory reload overhead called *cache persistence reload overhead* (CPRO). We present different approaches to compute CPRO and show how to correctly account for both CRPD and cache persistence in the schedulability analysis for *fixed-priority preemptive systems*. In Chapter 5, we identify that a *separate* computation of CRPD and CPRO may lead to an overestimation in the total inter-task cache interference suffered by the tasks. We present an *integrated* analysis that considers cache evictions that are already counted for in the CRPD analysis, when computing CPRO. The integrated analysis provides a tighter bound on the total

inter-task cache interference compared to the separate treatment of CRPD and CPRO. In Chapter 6, we evaluate the impact of memory layout of tasks on schedulability. We show that the intra- and inter-task cache interference can be interrelated and balancing their respective contribution to tasks WCRT may result in improving task set schedulability. We present an approach to optimize task layout in memory such that the trade-off between intra- and inter-task cache interference can be balanced and the task set's schedulability is achieved. Evaluation has shown that the new methods strongly improve upon former approaches.

In the second part of this thesis, i.e., Chapter 7 and 8, we focus on the analysis of inter-task cache interference considering single-level and multilevel set-associative caches. In Chapter 7, we provide solutions that analyze the impact of cache persistence on the schedulability analysis considering *set-associative* LRU-caches. We show how *persistent cache blocks* (PCBs) of tasks can be determined when considering set-associative caches and present three different approaches to calculate CPRO for set-associative caches. In Chapter 8, we present a CRPD analysis for *multilevel non-inclusive* caches. We identify challenges in the computation of CRPD for multilevel caches and propose solutions that provide a tighter CRPD bound than an existing analysis in the state-of-the-art. Evaluations show that our proposed analysis significantly improve upon the existing approaches.

Finally, in the last part of thesis, i.e., Chapter 9, we evaluate how a sound estimate on the inter-task cache interference may impact the contention due to sharing of memory bus in multicore systems. We show that the number of bus/memory requests generated by a task executing on a multicore platform strongly depends on the number of cache misses suffered by that task which in turn depends on the inter-task cache interference experienced by the task during its execution. We built on the work done in Chapter 4, Chapter 7 and Chapter 8 to analyze memory bus contention in multicore architecture with single- and multi-level caches. Evaluations show that a tighter bound on the inter-task cache interference can significantly reduce memory bus contention suffered by tasks executing on multicore platforms and results in improving schedulability.

10.2 Limitations of Current Work and Future Directions

10.2.1 Cache Persistence Analysis for Multilevel Caches

In this thesis, we have shown that the notion of cache persistence between task instances can significantly improve schedulability for fixed-priority preemptive systems. However, the current cache persistence-aware analysis only support single-level, i.e., L1, caches. Considering that modern processors are equipped with multiple cache levels, it will be interesting to extend the notion of cache persistence to cache hierarchies. For example, in a processor architecture that support two-level caches, i.e., L1 and L2 caches, L2 cache is considerably larger than the L1 cache. Which means that L2 can hold more content than the L1 cache and tasks may have fewer conflicts in the L2 cache in comparison to the L1 cache. Consequently, in a two-level cache hierarchy more content can be “persistent" and re-used from the L2 cache which may lead to a significant reduction

in task's WCRT estimates. For example, consider the scenario shown in Figure 10.1, where task τ_1 has more memory blocks persistent in the cache when the cache has two levels, i.e., L1 and L2, in comparison to the case where there is only a single cache level, i.e., L1.

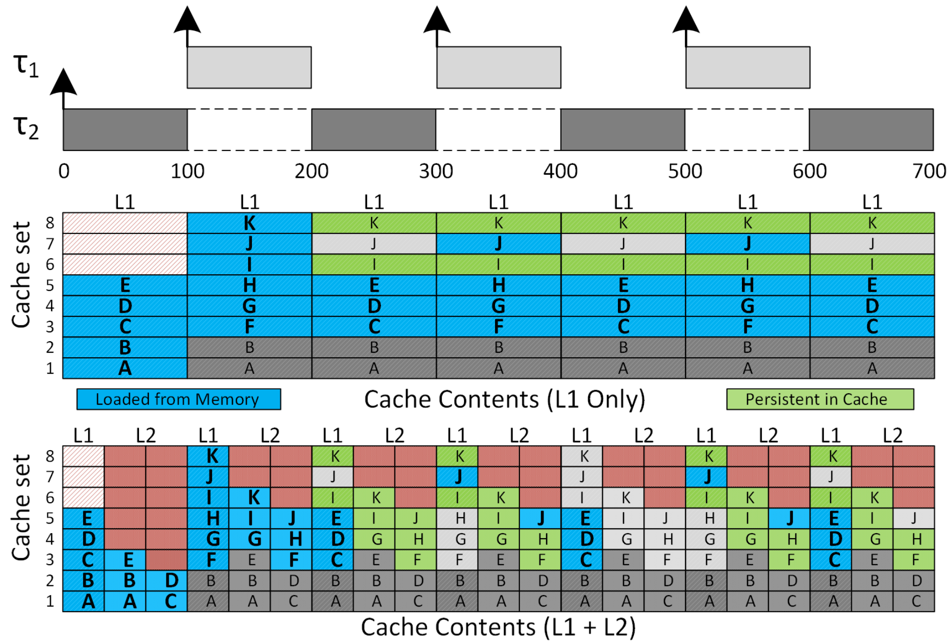


Figure 10.1: Cache persistence-aware analysis of multiple cache levels may lead more tighter WCRT bounds.

10.2.2 Inter-task Cache Interference Analysis for Last-level Shared Caches

The current analysis focus on inter-task cache interference, i.e., cache interference between tasks executing on the same processor/core, and assume that the cache(s) is/are private to the cores. However, in many modern processors last-level cache is usually shared among cores which may lead to *inter-core cache interference*, i.e., tasks running on different cores may concurrently access the last level cache and if two lines in the two addressing spaces of the running tasks map to the same cache line, said tasks can repeatedly evict each other in cache. Inter-core cache interference can occur between tasks that can run in parallel on different cores, therefore the exact interference analysis requires analyzing all the possible interleaving of task executions. This makes the analysis of inter-core cache interference much harder in comparison to the analysis of intra-core or inter-task cache interference. There exist few approaches in the state-of-the-art (Xiao et al., 2017, 2020) that focus on the computation of inter-core cache interference. However, these approaches focus on non-preemptive task systems and assume that the intra-core cache interference is already considered in the task's WCETs. Therefore, an interesting problem to solve is to bound inter-core cache interference for systems that allow preemptions. This requires to analyze intra- and inter-core cache interference simultaneously which is a challenging endeavor.

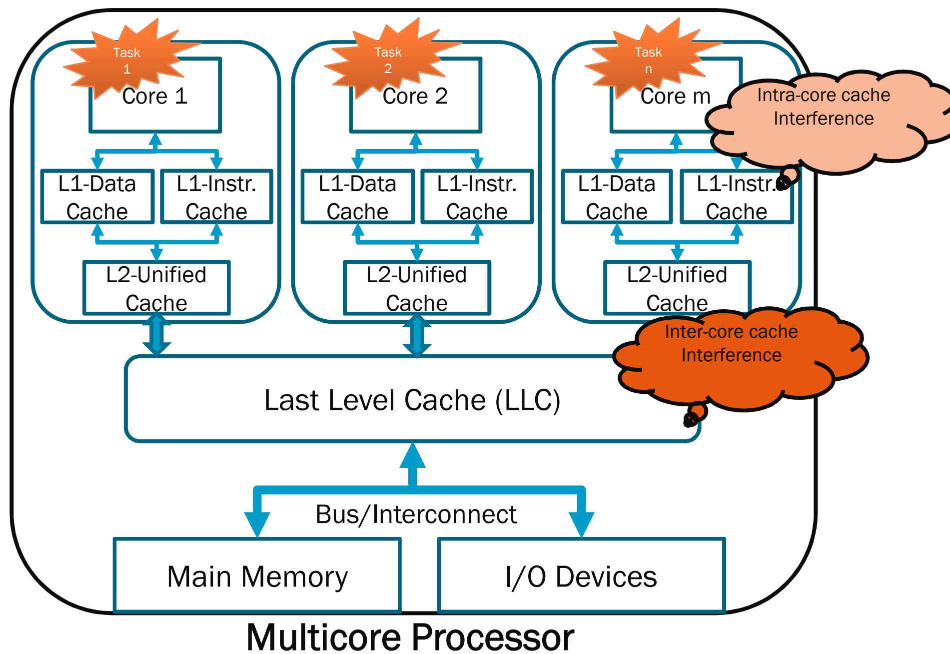


Figure 10.2: Under preemptive scheduling, simultaneous analysis of intra- and inter-core cache interference is a challenge.

10.2.3 Holistic Memory Contention Analysis for Preemptive Systems

Main memory is one of the hardware resources that is shared by different tasks executing on a multicore platform and the time needed to access a block from the main memory mainly depends on the number of requests generated by the executing tasks and the behavior of the memory controller. Several works have been proposed in literature that focus on bounding main memory contention by using techniques such as the DRAM bank partitioning (Reineke et al., 2011; Wu et al., 2013; Kim et al., 2014) and memory bandwidth reservation (Yun et al., 2012, 2013, 2014). However, these approaches may still result in pessimistic/optimistic estimates on the memory interference delay considering that these approaches usually do not consider the relationship between main memory and other resources, i.s., the bus and caches. As we have shown in Chapter 9, the number of main memory requests generated by a task during its execution strongly depends on the contention it may suffer on the cache or at the bus. Therefore, an interesting prospective is to provide a holistic memory contention analysis that considers the relationship between caches, bus and the main memory.

10.2.4 Cache Persistence-aware Inter-task Cache Interference Analysis considering Dynamic Priority Scheduling

In the current work, we focus on systems where tasks are scheduled under a fixed-priority assignment scheme, e.g., Rate or Deadline Monotonic (Liu and Layland, 1973), and show that a cache persistence-aware inter-task cache interference analysis can greatly improve system's schedulability. However, considering the benefits dynamic priority scheduling schemes such as the *Earliest*

Deadline First (EDF) can offer in comparison to fixed-priority schemes, it would be very interesting to adapt the proposed analysis to EDF. The work of Lunniss et al. (Lunniss et al., 2013, 2014) can be very helpful in this regard. It focus on integrating CRPDs into schedulability tests for EDF and compare the performance of FP and EDF scheduling algorithms in the presence of CRPD. Knowing that an inter-task cache interference analysis that accounts for both cache persistence and CRPDs dominates the analysis that only consider CRPDs, we expect EDF to offer significant performance gains over FP when cache persistence is considered.

10.3 Conclusions

The work done in this dissertation shows that inter-task interference due to contention for shared resources such as caches and memory bus can greatly affect the temporal behavior of tasks. A correct and sound computation of shared resource contention is therefore essential to improve the accuracy of schedulability analyses. The proposed analysis framework provides a holistic solution that considers the inter-dependency between the behavior of different shared resources thus providing deterministic bounds on the WCRT of tasks.

Bibliography

"Aramis project," https://www.projekt-aramis.de/.

"Frescor fp7 project," ftp://ftp.cordis.europa.eu/pub/ist/docs/dir_c/ems/frescor-v1_en.pdf.

"Multiprocessor execution platforms," http://download.tuxfamily.org/erika/webdownload/nios2/1422/FRESCOR_WP4_D-EP7v2.pdf.

"Single core equivalence," http://rtsl-edge.cs.illinois.edu/SCE/.

"Deadline scheduling for linux," https://en.wikipedia.org/wiki/SCHED_DEADLINE.

"ait wcet analyser," http://www.absint.com/ait.

"Front-side bus," May 2017. [Online]. Available: https://en.wikipedia.org/wiki/Front-side_bus

"Bus (computing)," May 2017. [Online]. Available: https://en.wikipedia.org/wiki/Bus_(computing)

"Rapitime," http://www.rapitasystems.com.

D. Adams, *The Hitchhiker's Guide to the Galaxy*. San Val, 1995. [Online]. Available: http://books.google.com/books?id=W-xMPgAACAAJ

B. Akesson and K. Goossens, "Architectures and modeling of predictable memory controllers for improved system integration," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*. IEEE, 2011, pp. 1–6.

B. Akesson, K. Goossens, and M. Ringhofer, "Predator: a predictable sdram memory controller," in *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. ACM, 2007, pp. 251–256.

S. Altmeyer, *Analysis of preemptively scheduled hard real-time systems*. epubli GmbH, 2013.

S. Altmeyer and C. M. Burguière, "Cache-related preemption delay via useful cache blocks: Survey and redefinition," *Journal of Systems Architecture*, vol. 57, no. 7, pp. 707–719, 2011.

S. Altmeyer and G. Gebhard, "Optimal task placement to improve cache performance," in *In EMSOFT*. Citeseer, 2007.

S. Altmeyer, C. Maiza, and J. Reineke, "Resilience analysis: tightening the crpd bound for set-associative caches," in *ACM Sigplan Notices*, vol. 45, no. 4. ACM, 2010, pp. 153–162.

S. Altmeyer, R. Davis, C. Maiza *et al.*, "Cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems," in *RTSS'11*. IEEE, 2011, pp. 261–271.

S. Altmeyer, R. I. Davis, and C. Maiza, "Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems," *Real-Time Systems*, vol. 48, no. 5, pp. 499–526, 2012.

S. Altmeyer, R. Douma, W. Lunniss, and R. I. Davis, "Outstanding paper: Evaluation of cache partitioning for hard real-time systems," in *2014 26th Euromicro Conference on Real-Time Systems*. IEEE, 2014, pp. 15–26.

S. Altmeyer, R. I. Davis, L. Indrusiak, C. Maiza, V. Nelis, and J. Reineke, "A generic and compositional framework for multicore response time analysis," in *RTNS'15*. ACM, 2015, pp. 129–138.

S. Altmeyer, R. Douma, W. Lunniss, and R. I. Davis, "On the effectiveness of cache partitioning in hard real-time systems," *Real-Time Systems*, vol. 52, no. 5, pp. 598–643, 2016.

R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical computer science*, vol. 126, no. 2, pp. 183–235, 1994.

B. Andersson and E. Tovar, "Multiprocessor scheduling with few preemptions," in *Embedded and Real-Time Computing Systems and Applications, 2006. Proceedings. 12th IEEE International Conference on*. IEEE, 2006, pp. 322–334.

B. Andersson, S. Baruah, and J. Jonsson, "Static-priority scheduling on multiprocessors," in *Real-Time Systems Symposium, 2001.(RTSS 2001). Proceedings. 22nd IEEE*. IEEE, 2001, pp. 193–202.

L. C. Aparicio, J. Segarra, C. Rodríguez, and V. Viñals, "Improving the wcet computation in the presence of a lockable instruction cache in multitasking real-time systems," *Journal of Systems Architecture*, vol. 57, no. 7, pp. 695–706, 2011.

N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, vol. 8, no. 5, pp. 284–292, 1993.

F. A. Authority, ""cast-32-a: Multi-core processors," 2016. [Online]. Available: https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/media/cast-32A.pdf

T. P. Baker, "An analysis of edf schedulability on a multiprocessor," *IEEE transactions on parallel and distributed systems*, vol. 16, no. 8, pp. 760–768, 2005.

A. Banerjee, S. Chattopadhyay, and A. Roychoudhury, "Precise micro-architectural modeling for wcet analysis via ai+ sat," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*. IEEE, 2013, pp. 87–96.

S. Baruah, "The limited-preemption uniprocessor scheduling of sporadic task systems," in *ECRTS 2005*. IEEE, 2005, pp. 137–144.

S. Baruah and J. Goossens, "Scheduling real-time tasks: Algorithms and complexity," *Handbook of scheduling: Algorithms, models, and performance analysis*, vol. 3, 2004.

S. Baruah, M. Bertogna, and G. Buttazzo, *Multiprocessor Scheduling for Real-Time Systems*. Springer, 2015.

S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, no. 6, pp. 600–625, 1996.

A. Bastoni, B. Brandenburg, and J. Anderson, "Cache-related preemption and migration delays: Empirical approximation and impact on schedulability," *Proceedings of OSPERT*, pp. 33–44, 2010.

S. Basumallick and K. Nilsen, "Cache issues in real-time systems," in *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, vol. 5. Citeseer, 1994.

P. Baufreton, V. Bregeon, K. Didier, G. Iooss, D. Potop-Butucaru, and J. Souyris, "Efficient fine-grain parallelism in shared memory for real-time avionics," in *ERTS 2020-10th European Congress Embedded Real Time Systems*, 2020.

F. Bellosa, "Process cruise control: Throttling memory access in a soft real-time environment," *University of Erlangen, Tech. Rep*, 1997.

M. Bertogna, O. Xhani, M. Marinoni, F. Esposito, and G. Buttazzo, "Optimal selection of preemption points to minimize preemption overhead," in *ECRTS'11*. IEEE, 2011, pp. 217–227.

E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1-2, pp. 129–154, 2005.

T. Blaß, S. Hahn, and J. Reineke, "Write-back caches in wcet analysis," in *LIPIcs-Leibniz International Proceedings in Informatics*, vol. 76. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

R. J. Bril, S. Altmeyer, M. M. van den Heuvel, R. I. Davis, and M. Behnam, "Integrating cache-related pre-emption delays into analysis of fixed priority scheduling with pre-emption thresholds," in *2014 IEEE Real-Time Systems Symposium*. IEEE, 2014, pp. 161–172.

B. D. Bui, M. Caccamo, L. Sha, and J. Martinez, "Impact of cache partitioning on multi-tasking real time embedded systems," in *2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE, 2008, pp. 101–110.

A. Burchard, J. Liebeherr, Y. Oh, and S. H. Son, "New strategies for assigning real-time tasks to multiprocessor systems," *IEEE transactions on computers*, vol. 44, no. 12, pp. 1429–1442, 1995.

P. Burgio, A. Marongiu, P. Valente, and M. Bertogna, "A memory-centric approach to enable timing-predictability within embedded many-core accelerators," in *Real-Time and Embedded Systems and Technologies (RTEST), 2015 CSI Symposium on*, Oct 2015, pp. 1–8.

C. Burguière, J. Reineke, and S. Altmeyer, "Cache-related preemption delay computation for set-associative caches-pitfalls and solutions," in *OASIcs-OpenAccess Series in Informatics*, vol. 10. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.

A. Burns, *Preemptive priority based scheduling: An appropriate engineering approach*. Citeseer, 1993.

A. Burns and R. I. Davis, "Adaptive mixed criticality scheduling with deferred preemption," in *2014 IEEE Real-Time Systems Symposium*. IEEE, 2014, pp. 21–30.

A. Burns, R. I. Davis, P. Wang, and F. Zhang, “Partitioned edf scheduling for multiprocessors using a c= d task splitting scheme,” *Real-Time Systems*, vol. 48, no. 1, pp. 3–33, 2012.

J. V. Busquets-Mataix, J. J. Serrano, R. Ors, P. Gil, and A. Wellings, “Adding instruction cache effect to schedulability analysis of preemptive real-time systems,” in *RTAS’96*. IEEE, 1996, pp. 204–212.

J. V. Busquets-Mataix, J. J. Serrano, and A. Wellings, “Hybrid instruction cache partitioning for preemptive real-time systems,” in *Real-Time Systems, Proceedings., Ninth Euromicro Workshop on*. IEEE, 1997, pp. 56–63.

J. V. Busquets-Mataix, D. Gil, P. Gil, and A. Wellings, “Techniques to increase the schedulable utilization of cache-based preemptive real-time systems,” *Journal of systems architecture*, vol. 46, no. 4, pp. 357–378, 2000.

G. Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications*. Springer Science & Business Media, 2011, vol. 24.

G. C. Buttazzo, M. Bertogna, and G. Yao, “Limited preemptive scheduling for real-time systems. a survey,” *Industrial Informatics, IEEE Transactions on*, vol. 9, no. 1, pp. 3–15, 2013.

M. Campoy, A. P. Ivars, and J. Busquets-Mataix, “Static use of locking caches in multitask preemptive real-time systems,” in *Proceedings of IEEE/IEE Real-Time Embedded Systems Workshop (Satellite of the IEEE Real-Time Systems Symposium)*. Citeseer, 2001, pp. 1–6.

J. Cavicchio, C. Tessler, and N. Fisher, “Minimizing cache overhead via loaded cache blocks and preemption placement,” in *2015 27th Euromicro Conference on Real-Time Systems*. IEEE, 2015, pp. 163–173.

N. Cecere, M. Tipaldi, R. Wenker, and U. Villano, “Measurement and analysis of schedulability of spacecraft on-board software,” in *2016 IEEE Metrology for Aerospace (MetroAeroSpace)*. IEEE, 2016, pp. 545–550.

F. Certification Authorities Software Team (CAST) Position Paper CAST-32, “Multi-core processors,” 2014.

S. Chattopadhyay and A. Roychoudhury, “Scalable and precise refinement of cache timing analysis via model checking,” in *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*. IEEE, 2011, pp. 193–203.

S. Chattopadhyay and A. Roychoudhury, “Cache-related preemption delay analysis for multilevel noninclusive caches,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 5s, pp. 1–29, 2014.

S. Chattopadhyay, A. Roychoudhury, and T. Mitra, “Modeling shared cache and bus in multi-cores for timing analysis,” in *Proceedings of the 13th international workshop on software & compilers for embedded systems*. ACM, 2010, p. 6.

S. Chattopadhyay, L. K. Chong, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk, “A unified wcet analysis framework for multicore platforms,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 4s, p. 124, 2014.

H. Cho, B. Ravindran, and E. D. Jensen, “An optimal real-time scheduling algorithm for multiprocessors,” in *Real-Time Systems Symposium, 2006. RTSS’06. 27th IEEE International*. IEEE, 2006, pp. 101–110.

A. Chousein and R. N. Mahapatra, “Fully associative cache partitioning with don’t care bits for real-time applications,” *ACM SIGBED Review*, vol. 2, no. 2, pp. 35–38, 2005.

E. M. Clarke, O. Grumberg, and D. Peled, *Model checking*. MIT press, 1999.

A. Colin and I. Puaut, “A modular and retargetable framework for tree-based wcet analysis,” in *Real-Time Systems, 13th Euromicro Conference on, 2001*. IEEE, 2001, pp. 37–44.

P. Cousot and R. Cousot, “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints,” in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 1977, pp. 238–252.

C. Cullmann, “Cache persistence analysis: Theory and practice,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 12, no. 1s, p. 40, 2013.

A. E. Dalsgaard, M. C. Olesen, M. Toft, R. R. Hansen, and K. G. Larsen, “Metamoc: Modular execution time analysis using model checking,” in *OASIcs-OpenAccess Series in Informatics*, vol. 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

D. Dasari, B. Andersson, V. Nelis, S. M. Petters, A. Easwaran, and J. Lee, “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*. IEEE, 2011, pp. 1068–1075.

D. Dasari, B. Andersson, V. Nelis, S. M. Petters, A. Easwaran, and J. Lee, “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*. IEEE, 2011, pp. 1068–1075.

D. Dasari, B. Akesson, V. Nelis, M. A. Awan, and S. M. Petters, “Identifying the sources of unpredictability in cots-based multicore systems,” in *2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*. IEEE, 2013, pp. 39–48.

D. Dasari, V. Nelis, and B. Akesson, “A framework for memory contention analysis in multi-core platforms,” *Real-Time Systems*, vol. 52, no. 3, pp. 272–322, 2016.

R. I. Davis and A. Burns, “A survey of hard real-time scheduling for multiprocessor systems,” *ACM Computing Surveys (CSUR)*, vol. 43, no. 4, p. 35, 2011.

R. I. Davis and A. Burns, “A survey of hard real-time scheduling for multiprocessor systems,” *ACM computing surveys (CSUR)*, vol. 43, no. 4, p. 35, 2011.

R. I. Davis, S. Altmeyer, and J. Reineke, “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*. ACM, 2016, pp. 309–318.

R. I. Davis, S. Altmeyer, L. S. Indrusiak, C. Maiza, V. Nelis, and J. Reineke, “An extensible framework for multicore response time analysis,” *Real-Time Systems*, vol. 54, no. 3, pp. 607–661, 2018.

R. I. Davis, I. Bate, G. Bernat, I. Broster, A. Burns, A. Colin, S. Hutchesson, and N. Tracey, "Transferring real-time systems research into industrial practice: Four impact case studies," in *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

M. Deck, "Software reliability and the" cleanroom" approach: a position paper," in *Reliability and Maintainability Symposium, 1998. Proceedings., Annual*. IEEE, 1998, pp. 218–223.

S. K. Dhall and C. L. Liu, "On a real-time scheduling problem," *Operations research*, vol. 26, no. 1, pp. 127–140, 1978.

M. Di Natale and A. L. Sangiovanni-Vincentelli, "Moving from federated to integrated architectures in automotive: The role of standards, methods and tools," *Proceedings of the IEEE*, vol. 98, no. 4, pp. 603–620, 2010.

H. Ding, Y. Liang, and T. Mitra, "Wcet-centric dynamic instruction cache locking," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*. IEEE, 2014, pp. 1–6.

A. Ermedahl, "A modular tool architecture for worst-case execution time analysis," Ph.D. dissertation, Acta Universitatis Upsaliensis, 2003.

M. S. Espinoza, J. Goncalves, P. Leitao, J. L. G. Sanchez, and A. Herreros, "Inverse kinematics of a 10 dof modular hyper-redundant robot resorting to exhaustive and error-optimization methods: A comparative study," in *Robotics Symposium and Latin American Robotics Symposium (SBR-LARS), 2012 Brazilian*. IEEE, 2012, pp. 125–130.

M. S. Espinoza, A. I. Pereira, and J. Gonçalves, "Optimization methods for hyper-redundant robots' inverse kinematics in biomedical applications," in *AIP Conference Proceedings*, vol. 1479, 2012, p. 818.

H. Falk and H. Kotthaus, "Wcet-driven cache-aware code positioning," in *2011 Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*. IEEE, 2011, pp. 145–154.

H. Falk, S. Plazar, and H. Theiling, "Compile-time decided instruction cache locking using worst-case execution paths," in *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. ACM, 2007, pp. 143–148.

C. Ferdinand and R. Wilhelm, "Efficient and precise cache behavior prediction for real-time systems," *Real-time systems*, vol. 17, no. 2, pp. 131–181, 1999.

C. Ferdinand, R. Heckmann, and B. Franzen, "Static memory and timing analysis of embedded systems code," in *Proceedings of VVSS2007-3rd European Symposium on Verification and Validation of Software Systems, 23rd of March*, 2007, pp. 07–04.

G. Fernandez, J. Jalle, J. Abella, E. Quiñones, T. Vardanega, and F. J. Cazorla, "Resource usage templates and signatures for cots multicore processors," in *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 2015, p. 155.

M. Fernández, R. Gioiosa, E. Quiñones, L. Fossati, M. Zulianello, and F. J. Cazorla, "Assessing the suitability of the ngmp multi-core processor in the space domain," in *Proceedings of the tenth ACM international conference on Embedded software*. ACM, 2012, pp. 175–184.

S. Funk, "Lre-tl: an optimal multiprocessor algorithm for sporadic task sets with unconstrained deadlines," *Real-Time Systems*, vol. 46, no. 3, pp. 332–359, 2010.

J. Goossens, S. Funk, and S. Baruah, "Priority-driven scheduling of periodic task systems on multiprocessors," *Real-time systems*, vol. 25, no. 2-3, pp. 187–205, 2003.

G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, "A survey on cache management mechanisms for real-time embedded systems," *ACM Computing Surveys (CSUR)*, vol. 48, no. 2, p. 32, 2015.

N. Guan, M. Stigge, W. Yi, and G. Yu, "Cache-aware scheduling and analysis for multicores," in *Proceedings of the seventh ACM international conference on Embedded software*. ACM, 2009, pp. 245–254.

N. Guan, X. Yang, M. Lv, and W. Yi, "Fifo cache analysis for wcet estimation: a quantitative approach," in *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 2013, pp. 296–301.

N. Guan, M. Lv, W. Yi, and G. Yu, "Wcet analysis with mru cache: challenging lru for predictability," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 4s, p. 123, 2014.

J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET benchmarks: Past, present and future," in *OASIcs-OpenAccess Series in Informatics*, vol. 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

A. Gustavsson, A. Ermedahl, B. Lisper, and P. Pettersson, "Towards wcet analysis of multicore architectures using uppaal," in *OASIcs-OpenAccess Series in Informatics*, vol. 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

S. Hahn, J. Reineke, and R. Wilhelm, "Towards compositionality in execution time analysis–definition and challenges," in *CRTS*, 2013.

S. Hahn, J. Reineke, and R. Wilhelm, "Towards compositionality in execution time analysis: definition and challenges," *ACM SIGBED Review*, vol. 12, no. 1, pp. 28–36, 2015.

D. Hardy and I. Puaut, "Wcet analysis of multi-level non-inclusive set-associative instruction caches," in *2008 Real-Time Systems Symposium*. IEEE, 2008, pp. 456–466.

D. Hardy and I. Puaut, "Wcet analysis of instruction cache hierarchies," *Journal of Systems Architecture*, vol. 57, no. 7, pp. 677–694, 2011.

D. Hardy, T. Piquet, and I. Puaut, "Using bypass to tighten wcet estimates for multi-core processors with shared instruction caches," in *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*. IEEE, 2009, pp. 68–77.

D. Hardy, B. Rouxel, and I. Puaut, "The heptane static worst-case execution time estimation tool," in *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

R. Hegde, "Optimizing application performance on intel core microarchitecture using hardware-implemented prefetchers," *Intel Software Network*, 2008.

F. e. a. Heiko, "TACLeBench: A benchmark collection to support worst-case execution time research," in *WCET 2016*, ser. OpenAccess Series in Informatics, M. Schoeberl, Ed., vol. 55. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016, pp. 2:1–2:10.

H. Heinecke, K.-P. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, J. Leflour, J.-L. Maté, K. Nishikawa, and T. Scharnhorst, "Automotive open system architecture-an industry-wide initiative to manage the complexity of emerging automotive e/e-architectures," *Convergence*, pp. 325–332, 2004.

N. Holsti and S. Saarinen, "Status of the bound-t wcet tool," *Space Systems Finland Ltd*, 2002.

W.-H. Huang, J.-J. Chen, and J. Reineke, "Mirror: symmetric timing analysis for real-time tasks on multicore platforms with shared resources," in *DAC*. ACM, 2016, p. 158.

B. K. Huynh, L. Ju, and A. Roychoudhury, "Scope-aware data cache analysis for wcet estimation," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*. IEEE, 2011, pp. 203–212.

B. Jacob, S. Ng, and D. Wang, *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

K. Jeffay, D. F. Stanat, and C. U. Martel, "On non-preemptive scheduling of period and sporadic tasks," in *RTSS'91*. IEEE, 1991, pp. 129–139.

M. Joseph and P. Pandya, "Finding response times in a real-time system," *The Computer Journal*, vol. 29, no. 5, pp. 390–395, 1986.

R. Kamal, *Embedded systems: architecture, programming and design*. Tata McGraw-Hill Education, 2011.

S. Kato and N. Yamasaki, "Portioned edf-based scheduling on multiprocessors," in *Proceedings of the 8th ACM international conference on Embedded software*. ACM, 2008, pp. 139–148.

S. Kato and N. Yamasaki, "Semi-partitioned fixed-priority scheduling on multiprocessors," in *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE*. IEEE, 2009, pp. 23–32.

T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury, "Bus-aware multicore wcet analysis through tdma offset bounds," in *2011 23rd Euromicro Conference on Real-Time Systems*. IEEE, 2011, pp. 3–12.

H. Kim, A. Kandhalu, and R. Rajkumar, "A coordinated approach for practical os-level cache management in multi-core real-time systems," in *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*. IEEE, 2013, pp. 80–89.

H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding memory interference delay in cots-based multi-core systems," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2014, pp. 145–154.

H. Kim, D. De Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding and reducing memory interference in cots-based multi-core systems," *Real-Time Systems*, vol. 52, no. 3, pp. 356–395, 2016.

D. B. Kirk and J. K. Strosnider, "Smart (strategic memory allocation for real-time) cache design using the mips r3000," in *Real-Time Systems Symposium, 1990. Proceedings., 11th*. IEEE, 1990, pp. 322–330.

S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983.

R. Kirner, P. Puschner, I. Wenzel *et al.*, *Measurement-based worst-case execution time analysis using automatic test-data generation.* na, 2004.

C. H. Koo and H. Kim, "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).* IEEE, 2018, pp. 234–235.

H. Kopetz, "An integrated architecture for dependable embedded systems," in *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on.* IEEE, 2004, pp. 160–161.

M. Kowarschik and C. Weiß, "An overview of cache optimization techniques and cache-aware numerical algorithms," in *Algorithms for memory hierarchies.* Springer, 2003, pp. 213–232.

K. Lakshmanan, R. Rajkumar, and J. Lehoczky, "Partitioned fixed-priority preemptive scheduling for multi-core processors," in *Real-Time Systems, 2009. ECRTS'09. 21st Euromicro Conference on.* IEEE, 2009, pp. 239–248.

C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim, "Analysis of cache-related preemption delay in fixed-priority preemptive scheduling," *Computers, IEEE Transactions on*, vol. 47, no. 6, pp. 700–713, 1998.

J. Y.-T. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks," *Performance evaluation*, vol. 2, no. 4, pp. 237–250, 1982.

X. Li, Y. Liang, T. Mitra, and A. Roychoudhury, "Chronos: A timing analyzer for embedded software," *Science of Computer Programming*, vol. 69, no. 1, pp. 56–67, 2007.

Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," in *LCTES '95: Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, R. Gerber and T. Marlowe, Eds., vol. 30, no. 11, New York, NY, USA, Nov. 1995, pp. 88–98.

Y.-T. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 12, pp. 1477–1487, 1997.

Y.-T. Li, S. Malik, and A. Wolfe, "Cache modeling for real-time software: Beyond direct mapped instruction caches," in *Real-Time Systems Symposium, 1996., 17th IEEE.* IEEE, 1996, pp. 254–263.

Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury, "Timing analysis of concurrent programs running on shared cache multi-cores," in *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE.* IEEE, 2009, pp. 57–67.

Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury, "Timing analysis of concurrent programs running on shared cache multi-cores," in *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE.* IEEE, 2009, pp. 57–67.

J. Liedtke, H. Hartig, and M. Hohmuth, "Os-controlled cache predictability for real-time systems," in *Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE.* IEEE, 1997, pp. 213–224.

J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems," in *HPCA*. IEEE, 2008, pp. 367–378.

C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *JACM*, vol. 20, no. 1, pp. 46–61, 1973.

F. Liu, A. Narayanan, and Q. Bai, "Real-time systems," 2000.

F. Liu and Y. Solihin, "Understanding the behavior and implications of context switch misses," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 7, no. 4, p. 21, 2010.

J. W. Liu, "Real-time systems. 2000."

T. Liu, M. Li, and C. J. Xue, "Instruction cache locking for real-time embedded systems with multi-tasks," in *Embedded and Real-Time Computing Systems and Applications, 2009. RTCSA'09. 15th IEEE International Conference on*. IEEE, 2009, pp. 494–499.

T. Liu, M. Li, and C. J. Xue, "Minimizing wcet for real-time embedded systems via static instruction cache locking," in *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE*. IEEE, 2009, pp. 35–44.

T. Liu, Y. Zhao, M. Li, and C. J. Xue, "Task assignment with cache partitioning and locking for wcet minimization on mpsoc," in *Parallel Processing (ICPP), 2010 39th International Conference on*. IEEE, 2010, pp. 573–582.

P. Lokuciejewski, H. Falk, and P. Marwedel, "Wcet-driven cache-based procedure positioning optimizations," in *2008 Euromicro Conference on Real-Time Systems*. IEEE, 2008, pp. 321–330.

T. Lundqvist, *A WCET analysis method for pipelined microprocessors with cache memories*. Citeseer, 2002.

W. Lunniss, S. Altmeyer, and R. I. Davis, "Optimising task layout to increase schedulability via reduced cache related pre-emption delays," in *Proceedings of the 20th International Conference on Real-Time and Network Systems*, 2012, pp. 161–170.

W. Lunniss, S. Altmeyer, C. Maiza, and R. I. Davis, "Integrating cache related pre-emption delay analysis into edf scheduling," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*. IEEE, 2013, pp. 75–84.

W. Lunniss, S. Altmeyer, and R. I. Davis, "A comparison between fixed priority and edf scheduling accounting for cache related pre-emption delays," *Leibniz Transactions on Embedded Systems*, vol. 1, no. 1, pp. 01–1, 2014.

M. Lv, W. Yi, N. Guan, and G. Yu, "Combining abstract interpretation with model checking for timing analysis of multicore software," in *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*. IEEE, 2010, pp. 339–349.

M. Lv, N. Guan, Q. Deng, G. Yu, and W. Yi, "Mcait–a timing analyzer for multicore real-time software," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2011, pp. 414–417.

M. Lv, N. GUAN, W. YI, J. REINEKE, and R. WILHELM, “A survey on cache analysis for real-time systems,” *ACM Computing Surveys*, p. 45, 2015.

C. Maiza, H. Rihani, J. M. Rivas, J. Goossens, S. Altmeyer, and R. I. Davis, “A survey of timing verification techniques for multi-core real-time systems,” *ACM Computing Surveys (CSUR)*, vol. 52, no. 3, pp. 1–38, 2019.

R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, “Real-time cache management framework for multi-core architectures,” in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*. IEEE, 2013, pp. 45–54.

R. Mancuso, R. Pellizzoni, M. Caccamo, L. Sha, and H. Yun, “Wcet (m) estimation in multi-core systems using single core equivalence,” in *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*. IEEE, 2015, pp. 174–183.

J. M. Marinho, V. Nélis, S. M. Petters, and I. Puaut, “An improved preemption delay upper bound for floating non-preemptive region,” in *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*. IEEE, 2012, pp. 57–66.

J. M. Marinho, V. Nélis, S. M. Petters, and I. Puaut, “Preemption delay analysis for floating non-preemptive region scheduling,” in *DATE'12*. IEEE, 2012, pp. 497–502.

F. Marković, “Preemption-delay aware schedulability analysis of real-time systems,” Ph.D. dissertation, Mälardalen University, 2020.

F. Markovic, J. Carlson, and R. Dobrin, “Tightening the bounds on cache-related preemption delay in fixed preemption point scheduling,” in *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

F. Marković, J. Carlson, and R. Dobrin, “Improved cache-related´ preemption delay estimation for fixed preemption point scheduling,” in *Ada-Europe International Conference on Reliable Software Technologies*. Springer, 2018, pp. 87–101.

F. Marković, J. Carlson, S. Altmeyer, and R. Dobrin, “Improving the accuracy of cache-aware response time analysis using preemption partitioning,” in *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

F. Marković, J. Carlson, and R. Dobrin, “Cache-aware response time analysis for real-time tasks with fixed preemption points,” in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2020, pp. 30–42.

M. A. Marsan, G. Balbo, G. Conte, and F. Gregoretti, “Modeling bus contention and memory interference in a multiprocessor system,” *IEEE Transactions on Computers*, no. 1, pp. 60–72, 1983.

F. Martin, M. Alt, R. Wilhelm, and C. Ferdinand, “Analysis of loops,” in *International Conference on Compiler Construction*. Springer, 1998, pp. 80–94.

E. Mezzetti and T. Vardanega, “A rapid cache-aware procedure positioning optimization to favor incremental development,” in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2013, pp. 107–116.

A. K. Mok, “Fundamental design problems of distributed systems for the hard-real-time environment,” 1983.

T. Moscibroda and O. Mutlu, “Memory performance attacks: Denial of memory service in multi-core systems,” in *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*. USENIX Association, 2007, p. 18.

F. Mueller, “Compiler support for software-based cache partitioning,” in *ACM Sigplan Notices*, vol. 30, no. 11. ACM, 1995, pp. 125–133.

F. Mueller, “Timing predictions for multi-level caches,” in *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*. Citeseer, 1997, pp. 29–36.

F. Mueller, “Timing analysis for instruction caches,” *Real-time systems*, vol. 18, no. 2, pp. 217–247, 2000.

D. Muench, M. Paulitsch, and A. Herkersdorf, “Iompu: Spatial separation for hardware-based i/o virtualization for mixed-criticality embedded real-time systems using non-transparent bridges,” in *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on*. IEEE, 2015, pp. 1037–1044.

K. Müller, G. Sigl, B. Triquet, and M. Paulitsch, “On mils i/o sharing targeting avionic systems,” in *Dependable Computing Conference (EDCC), 2014 Tenth European*. IEEE, 2014, pp. 182–193.

S. P. Muralidhara, M. Kandemir, and P. Raghavan, “Intra-application cache partitioning,” in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–12.

H. S. Negi, T. Mitra, and A. Roychoudhury, “Accurate estimation of cache-related preemption delay,” in *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2003, pp. 201–206.

V. Nélis, P. M. Yomsi, L. M. Pinho, J. Fonseca, M. Bertogna, E. Quiñones, R. Vargas, and A. Marongiu, “The challenge of time-predictability in modern many-core architectures,” in *14th International Workshop on Worst-Case Execution Time Analysis*, 2014.

F. Nemer, H. Casse, P. Sainrat, and J. Bahsoun, “Inter-task WCET computation for a-way instruction caches,” in *Industrial Embedded Systems, 2008. SIES 2008. International Symposium on*, June 2008, pp. 169–176.

F. Nemer, H. Cassé, P. Sainrat, and A. Awada, “Improving the worst-case execution time accuracy by inter-task instruction cache analysis,” in *Industrial Embedded Systems, 2007. SIES’07. International Symposium on*. IEEE, 2007, pp. 25–32.

J. Nowotsch and M. Paulitsch, “Quality of service capabilities for hard real-time applications on multi-core processors,” in *Proceedings of the 21st International Conference on Real-Time Networks and Systems*. ACM, 2013, pp. 151–160.

J. Nowotsch and M. Paulitsch, “Leveraging multi-core computing architectures in avionics,” in *Dependable Computing Conference (EDCC), 2012 Ninth European*. IEEE, 2012, pp. 132–143.

J. Nowotsch, M. Paulitsch, A. Henrichsen, W. Pongratz, and A. Schacht, “Monitoring and wcet analysis in cots multi-core-soc-based mixed-criticality systems,” in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2014, pp. 1–5.

I. NXP, Freescale Semiconductor, "Data sheet: "p4080/p4081 qoriq integrated processor hardware specifications";," March 2017. [Online]. Available: http://www.nxp.com/products/microcontrollers-and-processors/power-architecture-processors/qoriq-platforms/p-series/qoriq-p4080-p4040-p4081-multicore-communications-processors:P4080?&tab=Documentation_Tab&linkline=Data-Sheets

Y. Oh and S. H. Son, "Allocating fixed-priority periodic tasks on multiprocessor systems," *Real-Time Systems*, vol. 9, no. 3, pp. 207–239, 1995.

M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero, "Hardware support for wcet analysis of hard real-time multicore systems," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 57–68, 2009.

M. Paolieri, E. Quinones, F. J. Cazorla, and M. Valero, "An analyzable memory controller for hard real-time cmps," *IEEE Embedded Systems Letters*, vol. 1, no. 4, pp. 86–90, 2009.

M. Paulitsch, J. Nowotsch, D. Münch, and L. Girbinger, "Transparent software replication and hardware monitoring leveraging modern system-on-chip features," in *2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE, 2013, pp. 157–164.

M. Paulitsch, O. M. Duarte, H. Karray, K. Mueller, D. Muench, and J. Nowotsch, "Mixed-criticality embedded systems–a balance ensuring partitioning and performance," in *Digital System Design (DSD), 2015 Euromicro Conference on*. IEEE, 2015, pp. 453–461.

R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for COTS-based embedded systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*, April 2011, pp. 269–279.

R. Pellizzoni and M. Caccamo, "Toward the predictable integration of real-time cots based systems," in *RTSS'07*. IEEE, 2007, pp. 73–82.

R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele, "Worst case delay analysis for memory interference in multicore systems," in *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2010, pp. 741–746.

B. Peng, N. Fisher, and M. Bertogna, "Explicit preemption placement for real-time conditional code," in *2014 26th Euromicro Conference on Real-Time Systems*. IEEE, 2014, pp. 177–188.

G. Phavorin and P. Richard, "Cacherelated preemption delays and real-time scheduling: A survey for uniprocessor systems," Technical report, Laboratoire dInformatique et dAutomatique pour les Systemes, 2015. URL: http://www. lias-lab. fr/publications/19296/survey. pdf, Tech. Rep.

S. Plazar, P. Lokuciejewski, and P. Marwedel, "Wcet-aware software based cache partitioning for multi-task real-time systems," in *9th International Workshop on Worst-Case Execution Time Analysis (WCET'09)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.

P. J. Prisaznuk, "Integrated modular avionics," in *Aerospace and Electronics Conference, 1992. NAECON 1992., Proceedings of the IEEE 1992 National*. IEEE, 1992, pp. 39–45.

P. J. Prisaznuk, "Arinc 653 role in integrated modular avionics (ima)," in *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*. IEEE, 2008, pp. 1–E.

I. Puaut and A. Arnaud, "Dynamic instruction cache locking in hard real-time systems," in *Proc. of the 14th Int. Conference on Real-Time and Network Systems*, 2006.

I. Puaut and D. Decotigny, "Low-complexity algorithms for static cache locking in multitasking hard real-time systems," in *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE.* IEEE, 2002, pp. 114–123.

P. Puschner and A. Schedl, "Computing maximum task execution times with linear programming techniques," *Technische Universität Wien, Institut für Technische Informatik, Tech. Rep*, 1995.

H. Ramaprasad and F. Mueller, "Tightening the bounds on feasible preemption points," in *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06).* IEEE, 2006, pp. 212–224.

H. Ramaprasad and F. Mueller, "Bounding worst-case response time for tasks with non-preemptive regions," in *2008 IEEE Real-Time and Embedded Technology and Applications Symposium.* IEEE, 2008, pp. 58–67.

W. RapiTime, "tool homepage, 2006."

S. A. Rashid, G. Nelissen, and E. Tovar, "Cache persistence aware response time analysis for fixed priority preemptive systems," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS).* IEEE, 2016, p. 33.

J. Reineke, "The semantic foundations and a landscape of cache-persistence analyses," *Leibniz Transactions on Embedded Systems*, vol. 5, no. 1, pp. 03–1, 2018.

J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker, "A definition and classification of timing anomalies," in *OASIcs-OpenAccess Series in Informatics*, vol. 4. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2006.

J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee, "Pret dram controller: Bank privatization for predictability and temporal isolation," in *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2011 Proceedings of the 9th International Conference on.* IEEE, 2011, pp. 99–108.

S. Resch, A. Steininger, and C. Scherrer, "A composable real-time architecture for replicated railway applications," *Journal of Systems Architecture*, vol. 61, no. 9, pp. 472–485, 2015.

J. Rosen, A. Andrei, P. Eles, and Z. Peng, "Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip," in *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International.* IEEE, 2007, pp. 49–60.

J. Rosen, A. Andrei, P. Eles, and Z. Peng, "Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip," in *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International.* IEEE, 2007, pp. 49–60.

K. Rosvall and I. Sander, "A constraint-based design space exploration framework for real-time applications on mpsocs," in *Proceedings of the conference on Design, Automation & Test in Europe.* European Design and Automation Association, 2014, p. 326.

S. Schliecker and R. Ernst, "Real-time performance analysis of multiprocessor systems with shared memory," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 10, no. 2, p. 22, 2010.

S. Schliecker and R. Ernst, "Real-time performance analysis of multiprocessor systems with shared memory," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 10, no. 2, pp. 1–27, 2011.

S. Schliecker, J. Rox, M. Negrean, K. Richter, M. Jersak, and R. Ernst, "System level performance analysis for real-time automotive multicore and network architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 7, pp. 979–992, 2009.

S. Schliecker, M. Negrean, and R. Ernst, "Bounding the shared resource load for the performance analysis of multiprocessor systems," in *Proceedings of the conference on design, automation and test in Europe*. European Design and Automation Association, 2010, pp. 759–764.

J. Schneider, "Cache and pipeline sensitive fixed priority scheduling for preemptive real-time systems," in *Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE*. IEEE, 2000, pp. 195–204.

A. Schranzhofer, J.-J. Chen, and L. Thiele, "Timing analysis for tdma arbitration in resource sharing systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*. IEEE, 2010, pp. 215–224.

A. Schranzhofer, R. Pellizzoni, J.-J. Chen, L. Thiele, and M. Caccamo, "Timing analysis for resource access interference on adaptive resource arbiters," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*. IEEE, 2011, pp. 213–222.

J. Simonson and J. H. Patel, "Use of preferred preemption points in cache-based real-time systems," in *CPDS'95*. IEEE, 1995, pp. 316–325.

T. Sondag and H. Rajan, "A more precise abstract domain for multi-level caches for tighter wcet analysis," in *2010 31st IEEE Real-Time Systems Symposium*. IEEE, 2010, pp. 395–404.

D. S. Specification, "Jesd79," 2010.

A. Srinivasan and S. Baruah, "Deadline-based scheduling of periodic task systems on multiprocessors," *Information Processing Letters*, vol. 84, no. 2, pp. 93–98, 2002.

J. A. Stankovic, "Misconceptions about real-time computing: A serious problem for next-generation systems," *Computer*, vol. 21, no. 10, pp. 10–19, 1988.

J. Stärner and L. Asplund, "Measuring the cache interference cost in preemptive real-time systems," in *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, 2004, pp. 146–154.

J. Staschulat and R. Ernst, "Scalable precision cache analysis for real-time software," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 6, no. 4, pp. 25–es, 2007.

J. Staschulat, S. Schliecker, and R. Ernst, "Scheduling analysis of real-time systems with precise modeling of cache related preemption delay," in *ECRTS'05*. IEEE, 2005, pp. 41–48.

G. Stock, S. Hahn, and J. Reineke, "Cache persistence analysis: Finally exact," in *2019 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2019, pp. 481–494.

V. Suhendra and T. Mitra, "Exploring locking & partitioning for predictable shared caches on multi-cores," in *Proceedings of the 45th annual Design Automation Conference*. ACM, 2008, pp. 300–303.

Y. Tan and V. Mooney, "Timing analysis for preemptive multitasking real-time systems with caches," *ACM (TECS)*, vol. 6, no. 1, p. 7, 2007.

C. Tessler, "Bundle: Taming the cache and improving schedulability of multi-threaded hard real-time systems," Ph.D. dissertation, Wayne State University, 2019.

C. Tessler and N. Fisher, "Bundle: real-time multi-threaded scheduling to reduce cache contention," in *2016 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2016, pp. 279–290.

C. Tessler and N. Fisher, "Bundlep: Prioritizing conflict free regions in multi-threaded programs to improve cache reuse," in *2018 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2018, pp. 325–337.

C. Tessler and N. Fisher, "Npm-bundle: Non-preemptive multitask scheduling for jobs with bundle-based thread-level scheduling," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

M. Tham, "Writing research theses or dissertations," May 2001, university of Newcastle Upon Tyne. [Online]. Available: http://lorien.ncl.ac.uk/ming/dept/Tips/writing/thesis/thesis-intro.htm

H. Theiling, "Control flow graphs for real-time systems analysis," *Universität des Saarlandes, Diss*, 2002.

H. Theiling, C. Ferdinand, and R. Wilhelm, "Fast and precise WCET prediction by separated cache and path analyses," *Real-Time Systems*, vol. 18, no. 2-3, pp. 157–179, 2000.

S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand, "An abstract interpretation-based timing validation of hard real-time avionics software," in *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.* IEEE, 2003, pp. 625–632.

H. Tomiyama and N. D. Dutt, "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*. ACM, 2000, pp. 67–71.

H. Tomiyama and H. Yasuura, "Code placement techniques for cache miss rate reduction," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 2, no. 4, pp. 410–429, 1997.

T. Ungerer, F. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quinones, M. Gerdes, M. Paolieri, J. Wolf *et al.*, "Merasa: Multicore execution of hard real-time applications supporting analyzability," *IEEE Micro*, vol. 30, no. 5, pp. 66–75, 2010.

X. Vera, B. Lisper, and J. Xue, "Data cache locking for higher program predictability," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 1. ACM, 2003, pp. 272–282.

Y. Wang and M. Saksena, "Scheduling fixed-priority tasks with preemption threshold," in *Proceedings Sixth International Conference on Real-Time Computing Systems and Applications. RTCSA'99 (Cat. No. PR00306)*. IEEE, 1999, pp. 328–335.

B. C. Ward, A. Thekkilakattil, and J. H. Anderson, "Optimizing preemption-overhead accounting in multiprocessor real-time systems," in *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*. ACM, 2014, p. 235.

C. B. Watkins and R. Walter, "Transitioning from federated avionics architectures to integrated modular avionics," in *Digital Avionics Systems Conference, 2007. DASC'07. IEEE/AIAA 26th*. IEEE, 2007, pp. 2–A.

R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, p. 36, 2008.

R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, p. 36, 2008.

A. Wolfe, "Software-based cache partitioning for real-time applications," in *Third International Workshop on Responsive Computer Systems*, 1993.

L. Wu and W. Zhang, "A model checking based approach to bounding worst-case execution time for multicore processors," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 11, no. S2, p. 56, 2012.

Z. P. Wu, Y. Krish, and R. Pellizzoni, "Worst case analysis of dram latency in multi-requestor systems," in *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*. IEEE, 2013, pp. 372–383.

J. Xiao, S. Altmeyer, and A. Pimentel, "Schedulability analysis of non-preemptive real-time scheduling for multicore processors with shared caches," in *2017 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2017, pp. 199–208.

J. Xiao, S. Altmeyer, and A. D. Pimentel, "Schedulability analysis of global scheduling for multicore systems with shared caches," *IEEE Transactions on Computers*, 2020.

J. Yan and W. Zhang, "Wcet analysis for multi-core processors with shared l2 instruction caches," in *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS'08. IEEE*. IEEE, 2008, pp. 80–89.

J. Yan and W. Zhang, "Wcet analysis for multi-core processors with shared l2 instruction caches," in *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS'08. IEEE*. IEEE, 2008, pp. 80–89.

H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memory access control in multiprocessor for real-time systems with mixed criticality," in *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*. IEEE, 2012, pp. 299–308.

H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*. IEEE, 2013, pp. 55–64.

H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni, "Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*. IEEE, 2014, pp. 155–166.

X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in *Proceedings of the 4th ACM European conference on Computer systems*. ACM, 2009, pp. 89–102.

Z. Zhang and X. Koutsoukos, "Cache-related preemption delay analysis for multi-level inclusive caches," in *Proceedings of the 13th International Conference on Embedded Software*, 2016, pp. 1–10.

S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto, "Survey of scheduling techniques for addressing shared resources in multicore processors," *ACM Computing Surveys (CSUR)*, vol. 45, no. 1, p. 4, 2012.