Improving application timing predictability and caching performance on multi-core systems

Xiao, J.

Publication date
2019

Document Version
Final published version

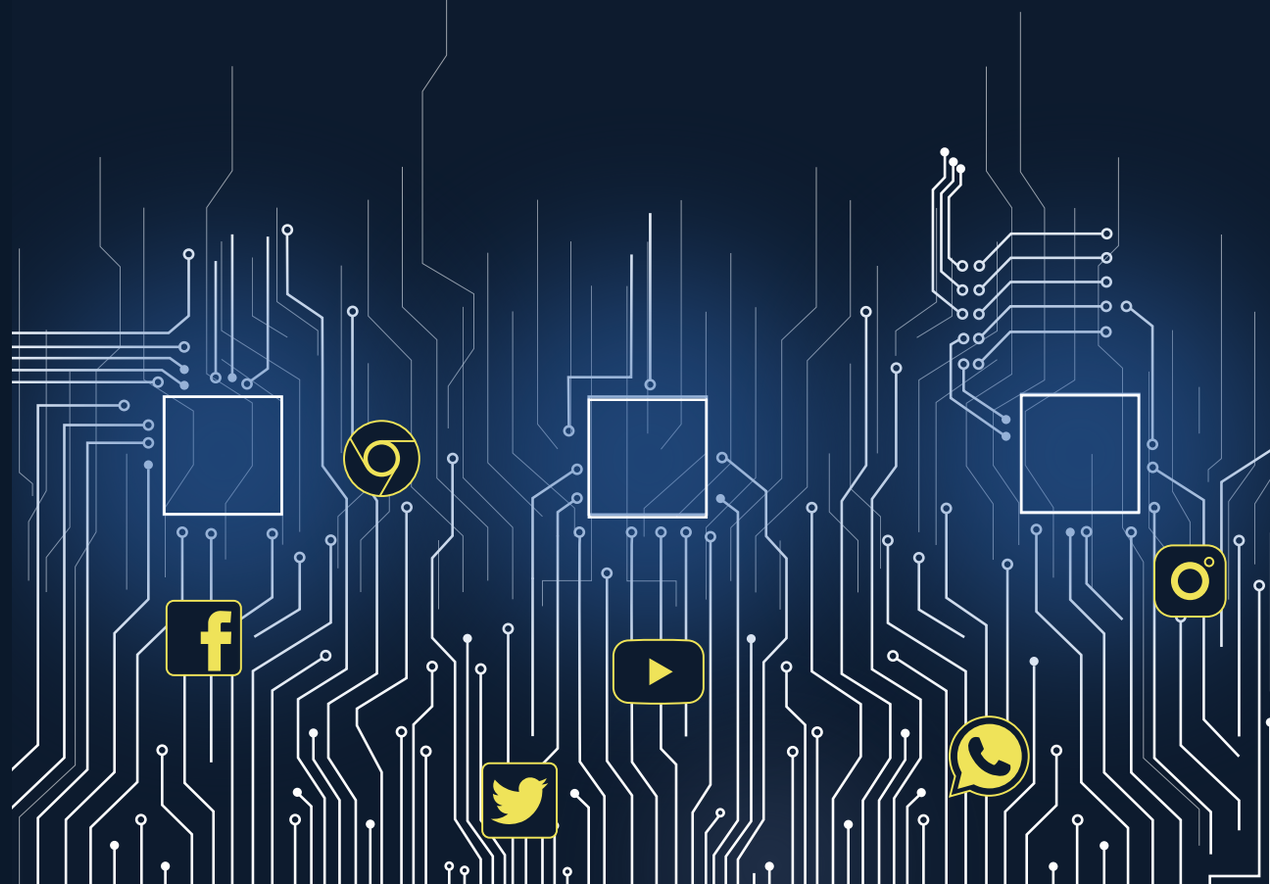
License
Other

Improving Application Timing Predictability and Caching Performance on Multi-core Systems

JUN XIAO

Improving Application Timing Predictability and Caching Performance on Multi-core Systems

• JUN XIAO •



Improving Application Timing Predictability and Caching Performance on Multi-core Systems

Jun Xiao

Improving Application Timing Predictability and Caching Performance on Multi-core Systems

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. ir. K.I.J. Maex
ten overstaan van een door het College voor Promoties ingestelde
commissie, in het openbaar te verdedigen in
de Aula
op 18 oktober 2019, te 11:00 uur

door

Jun Xiao

geboren te Jiangxi

Promotiecommissie

Promotor:		
	Prof. dr. ir. Cees T. A. M. de Laat	Universiteit van Amsterdam
	Dr. Andy D. Pimentel	Universiteit van Amsterdam
Overige leden:		
	Prof. dr. Sebastian Altmeyer	University of Augsburg
	Prof. dr. Giuseppe Lipari	University of Lille
	Dr. John Shalf	Lawrence Berkeley National Laboratory
	Prof. dr. Rob van Nieuwpoort	Universiteit van Amsterdam
	Prof. dr. Cees G.M. Snoek	Universiteit van Amsterdam
	Dr. Clemens Grelck	Universiteit van Amsterdam
	Dr. Ana Lucia Varbanescu	Universiteit van Amsterdam

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

The research was supported by NWO under project number 12696 (CPS-4).

Cover by Shichen Li
Printed by Ipskamp Printing, Enschede

ISBN: 978-94-028-1739-3

Acknowledgements

It is about four and half years since I started working towards my PhD in March, 2015. In those years, there were disappointments, worries, and depression, but there were more joys, cheers and hope. It has been surely a long, hard but enjoyable journey for me. As I finish this journey, my heart are filled with gratitude for a number of extraordinary people who made it possible. This dissertation could never have been completed without the help from them.

First and foremost, I would like to express my sincere gratitude to my daily supervisor Andy Pimentel, for being valuable sources of technical knowledge, tactical advice and scientific support. You have made selfless dedication to me: I can not count how much coffee you consumed before our regular meetings, but I remember that you corrected almost every sentences of the draft I wrote for the first paper of my PhD. Thank Andy for having faith in me and encouraging me when I got stuck with research and fell into darkness. I am especially grateful to you for letting me pursue my own research agenda as you provided the nurturing blend of trust, support, encouragement, patience, and funding. Your contribution to me as a person goes beyond our professional relationship: you showed me the art of communication and you taught me the power of respect. Your teachings and principles will continue to influence me, both as a researcher and as an individual. Thank you very much for all the help, Andy!

I am also greatly indebted to my promoter Prof. Cees de Laat, for always being available for discussions on my research and providing me with constructive comments and suggestions for the improvement of my work.

I was very fortunate to work with, and learn from, a number of (both formal and current) colleagues in and SNE research group: Benjamin Rouxel, Catalin Ciobanu, Dolly Sapra, Hao Zhu, Hongyun Liu, Hugo Meyer, Junchao Wang, Julius Roeder, Lu Zhang, Lukasz Makowski, Ralph Koning, Uraz Odyurt, Wei Quan, Xiaofeng Liao, Zeshun Shi, Zhiming Zhao and so on. Those incredible colleagues helped by brainstorming, exchanging ideas, providing feedback, and being exceptional friends. A warm thanks to Giulio Stramondo for your humor and pleasing words at the moment when we held a cup of wine and "Ganbei". A sincere thanks also goes to Simon Polstra, who provided technical support at the the early stages of my PhD and helped me translate this dissertation summary into Dutch. I also would like to give special thanks to Huan Zhou and Hu Yang for interesting talks and jokes we had during our daily coffee times.

I would like to thank the members of my examination committee: Prof. Sebastian Altmeyer, Prof. Giuseppe Lipari, Dr. John Shalf, Prof. Rob van Nieuwpoort, Prof. Cees G.M. Snoek, Dr. Clemens Grelck and Dr. Ana Lucia Varbanescu for reviewing this dissertation, providing me with invaluable comments and feedback and taking time to come to Amsterdam to discuss the various aspects of this work. Additionally, I wish to thank Giuseppe for your master course on design patterns in object-oriented programming, which have profoundly influenced the way I now develop software and your help in building the SysRT simulator. A Special thanks goes to Sebastian, for attending my practice talks, proof-reading papers, participating in discussions, listened, and argued like good friends. I will also not forget your help in preparing the rebuttal letter for the RTSS paper: we exchanged mails at the very dark night. A sincere thanks

goes to Clemens, for many discussions we had, in which I always benefit from your critical ideas and insights. I also learned a lot from you about the compilers. Thank Ana, for giving me the access to equipment for conducting experiments.

I met exceptional people and made formidable friendships throughout the various stages of my PhD. I would like to thank all my great and cheerful friends during my stay in the Netherlands: Biwen Wang, Hui Xiong, Jian Lin, Jinglan Wang, Linlin Zhang, Renjie Lv, Songyu Yang, Shunan He, Shuangshuang Hu, Shaojie Jiang, Si Wen, Wenyang Wu, Wei Du, Weiyu Li, Xiaolong Liu, Yumei Wang, Yipeng Song, Zijian Zhou, Ziming Li, Zenglin Shi and so on. Without them, I would never have had such a wonderful life in the past years. I also want to thank my friends who are far away from me but have provided support: Binfei Lin, Jiachang Chen, Lianhua Liu, Mao Nie, Michele Linardi, Shichen Li, Youcai Gao and so on.

I am very grateful to meet and talk with many brilliant researchers. I want to thank Prof. Lieven Eeckhout for hosting my short visit to your research group, and discussions we had about simulation and modeling of computer architecture. I desire to thank Prof. Per Stenstorm for interesting discussions we had about cache contention problems. I wish to thank Prof. Xu Liu for your guidance in programming with Intel's PEBS techniques and cache locality theory. I learned a lot from their expertise. I hope the cooperation with them will continue.

特别感谢我的妻子孙玮玮。三年前你放弃在意大利的学习生活，选择来荷兰陪我。感谢你这些年陪伴、支持与理解。在我工作一筹莫展的时侯，你总会做一桌好菜，讲述有趣之事来排解我的失落沮丧。在我为第二天的演讲焦虑的时候，你总是不厌其烦地听完我的准备练习。在我企图放弃学术之时，及时提醒我还有比金钱更重要的东西。感谢你为了实现我的梦想所作出的牺牲！感谢岳父岳母的信任，把您们心爱的女儿交给我。

感谢我可爱漂亮的女儿，肖千淮，欢迎你在我博士最后阶段的到来。你的微笑，是我忙时最好的放松。

衷心感谢爸爸妈妈和外婆，对我每一次抉择和人生每一步的无条件的支持和鼓励！您们的教育和引导是我攻读博士最坚实的基础。感谢您们的理解，我深知，七年多的欧洲留学生涯，于我是追梦，对于您们而言，更多的是日日夜夜的思念与牵挂。感谢在我二十多年成长道路中倾注的无私的爱，让我感受这个世界的美好，给我探索世界的勇气和自由。

最后，感谢生命中遇到的每一个人提供的帮助！在此，谨以此文献给所有我爱的和爱我的你们，以及为未来不懈奋斗的自己。

Jun Xiao 肖俊
Almere, 18th September 2019

Contents

1 Introduction

Both Moore's Law (the number of transistors in a processor will double every two years) and Dennard Scaling (power density remains constant because of transistor scaling) have allowed us to make large improvements over the last decades of microprocessor design. While Dennard Scaling is seen as now coming to an end, the resulting inability to increase clock frequencies significantly has fueled the move to the multi-core processors, which allows to continue to scale the performance of computing platforms through the use of multiple (and many) efficient cores.

The high performance achieved by the modern multi-core processors has been accomplished by using new architectural mechanisms. An important class of those mechanisms aims to overcome the performance gap between the processors and memory, referred to as the *memory wall*. For example, a hierarchy of cache memory levels, which rely on the principle of memory access locality, and hardware prefetching, which predicts future memory accesses and issues requests for the corresponding memory blocks in advance of the explicit accesses, are deployed in nearly every modern multi-core processors to hide memory access latency.

Although there are many benefits to moving from single-core processors to multi-core processors, architects must address disadvantages and associated risks such as the contention on the shared hardware resources. Cores on the same processor share both processor-internal resources like L3 cache, system bus, memory controller, I/O controllers and interconnects and processor-external resources like main memory, I/O devices and networks. Multiple applications executing concurrently on a multi-core system can interfere with each other at those shared resources. Such inter-application interference, if uncontrolled, could lead to unpredictable execution delay for individual applications and severe performance degradation for the whole system.

In this dissertation, we investigate two issues raised by the increasing complexity of the underlying hardware and software for multi-core systems: timing predictability for embedded computing and caching performance for high performance computing.

Challenges to build timing predictable multi-core embedded systems

In many embedded systems, a high performance is useless if we can not provide guarantees on the timing performance of the applications when designing the system.

One example of such systems is real-time systems, where the computing system must interact with its environment in a timely manner. Violating timing constraints is fatal to such a system, which may lead to catastrophic consequences such as loss of human life. For example, an air-bag controller, which must inflate the air-bag in time before the driver's head hits the steering wheel. A flight control system must correct turbulence before the airplane becomes unstable.

A significant trend in embedded computing hardware is the paradigm shift from uniprocessor to multi-core processors. This brings great benefits such as higher computation power at lower cost of energy consumption, while at the same time also poses new challenges for the timing analysis of embedded software executing on a multi-core processor.

The scheduling of applications on multi-core processors not only involves the time dimension, i.e., to decide when to execute a certain application, but also involves the spatial dimension as it also needs to decide where (i.e., on which core) to execute the application. Apart from the processing cores, different applications also contend on many shared hardware resources in multi-core processors, such as caches, buses and main memory. Interleaving of concurrent accesses to these shared hardware resources results in execution delays for individual applications and creates a tremendous state space of the system behavior, making its timing analysis extremely difficult.

Challenges to manage shared resources for high performance computing systems

Different from real-time embedded systems, the primary goal of high performance computing is to achieve the best system performance, i.e., to increase system throughput or to process the computing jobs as fast as possible. Here, the major challenges are to mitigate the inter-application interference and to efficiently manage the shared hardware resources in multi-core processor for high performance computing.

Applications running simultaneously on different cores utilize a plethora of memory components including a hierarchy of caches, prefetchers and memory controllers. Inter-application interference makes it difficult to the predict the performance degradation for individual applications, as some applications may be slowed down significantly, others may not

Furthermore, the interaction between these various components can be fairly complicated and it is challenging to study the impact of the interaction on application performance at run time due to the current limited transparency and monitoring capabilities for hardware behaviors. Since recent commodity CPUs have provided hardware support for control over hardware resources such as caches and memory bandwidth, a large amount of research attention is given to the management of shared hardware resources. However, mitigating the impact of the hardware interaction and resource contention and allocating the shared resources among the co-running applications to maximize system performance, remain challenges in high performance computing systems.

Next, we elaborate more in details on the research questions that we address in the thesis.

1.1 Research Outline and Questions

The work in this thesis focuses on developing tools, analyses and algorithmic methods for addressing the challenges raised in the two general research themes described above: time predictability for real time embedded computing and shared resource management for high performance computing. More specifically, in the first research line, we deal with two subtopics related to simulation and analytical approaches for the timing analysis: system-level modeling and simulation of real time systems for design space exploration and schedulability analysis of (global and partitioned) real-time scheduling for multi-core systems with shared caches. In the second research line, we study the interaction between the hardware prefetching and shared cache management and we exploit the opportunity to improve caching performance in the presence of hardware prefetching.

Modeling and simulation of real time embedded system

In today's embedded systems, together with the increasing multi-core hardware platform complexity, the software complexity has also been growing dramatically. Modern embedded systems increasingly execute several applications of different types concurrently on the underlying computing platform. These applications can have different execution requirements. For example, control applications typically are hard real-time applications and thus have stringent timing constraints, while best-effort applications prefer a short task response time. These systems are usually managed by a Real-Time Operating System (RTOS).

The complexities of the multi-processor system-on-chip (MPSoC) design space have made traditional cycle- or instruction-accurate simulators inefficient. Raising the level of abstraction is generally considered as a solution to address the design complexity, thus reducing time-to-market. To help in the design space exploration (DSE) at the early stages of design [68], various system-level design languages (SLDL) such as SystemC [93] and SpecC [85] have been proposed to provide a simulation environment. Originally, SLDLs primarily focused on hardware modeling and did not properly address the modeling of software aspects.

The modeling and simulation of RTOS with SLDL have received widespread attention from many researchers [40, 50, 116, 117]. Those simulators are built by a quantum-granularity based simulation approach, in which the modeled scheduler is invoked every simulation quantum, similar to the way a real OS scheduler behaves. This therefore introduces large overheads, resulting in low simulation speeds. Later efforts [48, 89] were made to trade-off speed for accuracy. [73] and [78] rely on the prediction of preemption points to speedup simulation while maintaining accuracy. However, predictions of preemption points are difficult if the simulation uses more complex task models like Directed Acyclic Graphs (DAGs) and resource sharing models.

Therefore, our research questions in the first study are the following, which are referred as **RQ1**:

RQ1 How to provide fast simulation of real-time embedded systems for design space exploration at the early stages of system design? How to accurately capture

the timing behaviour of embedded software? How to efficiently implement the simulator to provide support for easy plug-in of new task models, new schedulers and new resource sharing protocols?

Schedulability analysis of real-time multi-core systems

In single-core systems, timing behaviour is typically verified via a two-step process [61]. In the first timing analysis step, the Worst-Case Execution Time (WCET) of each task is derived. The WCET is an upper bound on the execution time, assuming the task runs in full isolation on the platform, i.e. without preemption, nor any co-runners. The WCET is then integrated into the second step, schedulability analysis. Schedulability analysis involves considering the worst-case pattern of task execution under a scheduling policy. Schedulability analysis determines the Worst-Case Response Time (WCRT) of each task, by which the timing constraint of each task can be verified.

The clear separation between the two steps can not applied to the timing verification of multi-core systems where the interference on shared hardware resources can depend heavily on the behaviour of co-runners executing concurrently on other cores. When a task executes alone on a multi-core processor platform, the timing behaviour of the system is defined by that task alone, the same as executing the task on a uniprocessor platform. However, when multiple tasks run simultaneously on different cores, the interplay between the tasks on shared hardware resources may results in unpredictable execution delays. Therefore, using the WCET of tasks executing in isolation on a multi-core platform without considering the co-runner interference can potentially lead to incorrect WCRT values.

With a multi-core system, the WCRTs are strongly dependent on the amount of inter-core interference on shared hardware resources such as main memory, shared caches and interconnects. In this dissertation, we shall only focus on the shared cache interference.

The schedulability analysis of global multiprocessor scheduling has been intensively studied [8, 14, 22, 51, 57, 118], of which comprehensive surveys can be found in [26, 82]. Most multi-core scheduling approaches assume that the WCETs are estimated in an offline and isolated manner and that WCET values are fixed. A few works address schedulability analysis for multi-core systems with shared caches [35, 113], but these works assume that so-called cache space isolation is deployed, which requires explicit hardware support.

In this thesis, we consider multi-core systems in which cache isolation techniques are not deployed, i.e. the last level cache is shared by cores. We study the schedulability analysis of global scheduling (Earliest Deadline First and Fixed Priority) for hard real-time tasks that exhibit shared cache interferences. Thus, we ask the following research questions, which are referred as **RQ2**:

RQ2 Is it possible to derive an upper bound on shared cache interference between two tasks running simultaneously on a multi-core system? Given a real-time taskset globally scheduled by EDF or FP, how to obtain an upper bound on the shared cache interference exhibited by each task in the taskset? How to derive

a schedulable condition for the globally scheduled taskset, accounting for the shared cache interference?

Besides the global scheduling, the partitioned (semi-partitioned) scheduling is another paradigm that are widely used for scheduling real-time tasks. In partitioned scheduling, tasks are statically allocated to processor cores, i.e., each task is assigned to a core and is always executed on that particular core. Although the partitioned approaches cannot exploit all unused processing capacity since a bin-packing-like problem needs to be solved to assign tasks to cores, it offers lower runtime overheads and provides consistently good empirical performance at high utilizations [11].

Furthermore, taking the shared cache interference into account, partitioned scheduling may achieve better schedulability than global scheduling, which will be shown in Chapter 5.

Therefore, it is interesting to extend the answer to the previous question, which is developed for real-time global scheduling, to the partitioned scheduling. We then ask the following questions, referred as **RQ3**:

RQ3 How to develop a cache interference aware partitioned scheduling for real-time multi-core systems? Is the partitioned scheduling better than global scheduling in terms of schedulability performance?

Prefetch-aware cache partitioning for high performance caching

Hardware cache prefetching is a popular technique that is deployed in modern multi-core processors to reduce memory latencies, addressing the memory wall problem [105]. However, it tends to increase the Last Level Cache (LLC) contention among applications executing on multi-core system, leading to a performance degradation for the overall system.

Shared cache management has attracted a lot of research attention in the past decades. Heracles [59] and Dirigent [120] control the amount of shared hardware resources, including the LLC, used by latency sensitive applications to improve Quality of Service and utilization. Selfa et. al. [79] cluster applications using the k-means algorithm and distributes cache ways between the groups to improve system fairness. Pons et. al. [69] assigns more cache space to critical applications to improve system turnaround time. [106] proposes a framework that dynamically monitors and predicts a workload's cache demand and reallocates the LLC given a performance target. KPart [30] leverages online profiling to obtain miss ratio curves for clustering applications and assigns each cluster of applications to a cache partition to improve system throughput.Park et. al. [66] proposed a coordinated partitioning of the LLC and memory bandwidth to improve the fairness of workloads on commodity servers. All these works have been implemented on existing processors, however, those works do not study the impact of hardware prefetching on cache performance and do not explicitly reveal the interaction between the hardware prefetching and LLC management.

In a real system, cache references by hardware prefetching also contributes to last level shared cache (LLC) interference [103]. However, there is little understanding about the interaction between the hardware prefetching and the shared caches. In this

research line, we focus on the LLC management to improve system performance in the presence of hardware prefetching.

Our questions in this study, then, are the following , refered as **RQ4**:

RQ4 How does hardware prefetching affect the caching performance? How to manage shared caches to improve system performance in the presence of hardware prefetching?

1.2 Main Contributions

In this section, we summarize the main contributions presented in this thesis.

Modeling and simulation of RTOS. We developed SysRT, a simulator of RTOS in SystemC that allows developers and researchers to easily explore and validate embedded RTOS design alternatives. Compared with quantum-granularity based simulators and prediction-based simulators, SysRT has two main advantages: (i) it has been developed to be generic and modular to support for easy plug-in of new schedulers as well as new resource sharing protocols. Thus, it is more flexible to simulate various real-time scheduling algorithms; (ii) it typically achieves higher simulation speeds via an event-driven simulation approach while obtaining identical accuracy results.

A Method to derive the upper bound on shared cache interference. We construct an integer programming formulation to calculate the upper bound on the cache interference exhibited by a task within a given execution window. We then present an iterative algorithm to obtain the upper bound on inter-core cache interference a task may exhibit during its job executions.

The above approach is extended to compute the upper bound on the cache interference for tasks under partitioned scheduling.

Schedulability analysis for real-time multi-core systems with shared caches. A schedulability condition is derived by integrating the calculated upper bound on inter-core cache interference into the schedulability analysis for global scheduling algorithms (EDF and FP).

We also propose a novel cache interference aware task partition algorithm: CA-TPAR. We conduct schedulability analysis of CA-TPAR and formally prove the correctness of CA-TPAR.

Evaluation of schedulability performance for global and partitioned scheduling. We perform a range of experiments to investigate how the schedulability of global (EDF and FP) and partitioned (CA-TPAR) scheduling are degraded by shared cache interference. We also compare the schedulability performance of EDF, FP scheduling and CA-TPAR over randomly generated tasksets.

Study of the interaction between hardware prefetching and cache management. We study the interaction between hardware prefetching and LLC management in a real system instead of in a simulator. We evaluate the variation of application

performance when varying the effective LLC space in the presence and absence of hardware prefetching. We observed that hardware prefetching can compensate the application performance loss due to the reduced effective cache space. Based on this observation, we classify applications into two categories, prefetching sensitive (*PS*) and non prefetching sensitive (*NPS*) applications, by the performance benefit they experience from hardware prefetchers.

A prefetch-aware cache partitioning approach. We propose CP_{pf}, a prefetch-aware LLC partitioning approach for improving LLC management. CP_{pf} consists of a method using Precise Event-Based Sampling (PEBS) techniques for online classification of PS and NPS applications and a LLC partitioning scheme using Cache Allocation technology (CAT) for PS and NPS applications. We have implemented the prototype of CP_{pf} as a user-level runtime system on Linux.

1.3 Thesis Overview

This thesis is organized in 7 chapters. After a background chapter, we present four research chapters containing our core contributions plus a concluding chapter:

Chapter 3 answers the research question RQ1. We present SysRT, a generic, modular and high-level RTOS simulator that is highly suited for early design space exploration. The simulator contains different types of application models and a modular RTOS kernel model, all developed in SystemC. Efficient and precise modeling of preemptive scheduling is achieved via an event-driven simulation approach, allowing simulations to be performed much faster than cycle-accurate simulations. We compare SysRT with state-of-art simulators to show the advantage of SysRT in both simulation speeds and accuracy. We also demonstrate the flexibility of SysRT and its benefits for early DSE using experiments with a mixed workload executing on multiprocessor platforms with different numbers of cores.

Chapter 4 addresses the research question RQ2. We develop a new schedulability analysis for real-time multicore systems with shared caches, globally scheduled by EDF and FP algorithms. We construct an integer programming formulation, which can be transformed to an integer linear programming formulation, to calculate an upper bound on cache interference exhibited by a task within a given execution window. Using the integer programming formulation, an iterative algorithm is then presented to obtain the upper bound on cache interference a task may exhibit during one job execution. The upper bound on cache interference is subsequently integrated into the schedulability analysis to derive a new schedulability condition. A range of experiments is performed to investigate how the schedulability is degraded by shared cache interference. We also evaluate the schedulability performance of EDF against FP scheduling over randomly generated tasksets.

Chapter 5 answers the research question RQ3. We propose a novel cache interference aware task partitioning algorithm, called CA-TPAR. We extended the approach to calculating the upper bound on cache interference for tasks that are globally scheduled, presented in the previous chapter, to bound the shared cache interference for tasks under partitioned scheduling. We conduct schedulability analysis of CA-TPAR and

formally prove its correctness. A set of experiments is performed to show CA-TPAR outperforms global EDF scheduling in terms of schedulability performance over the randomly generated tasksets.

Chapter 6 answers the research question RQ4. We propose CP$_{pf}$, a prefetch aware LLC partitioning approach for high performance caching. We first study the interaction between hardware prefetching and LLC cache management by analyzing the variation of application performance when varying the effective LLC space in the presence and absence of hardware prefetching. We observe that hardware prefetching can compensate the application performance loss due to the reduced effective cache space. Motivated by this observation, we then classify applications into two categories, prefetching sensitive (*PS*) and non prefetching sensitive (*NPS*) applications, by the performance benefit they experience from hardware prefetchers. After that, we propose CP$_{pf}$. CP$_{pf}$ first classifies PS and NPS applications at run time and then partitions the LLC among PS and NPS applications. Finally, we show the system performance improvement achieved by CP_{pf}, compared with the baseline configuration, in which the LLC is unpartitioned and is fully shared among all applications.

Chapter 7 draws the Conclusions. We summarize our main findings and discuss directions for future research.

1.4 Origins

For each research chapter, we list on which publication(s) it is based, and we briefly discuss the role of the co-authors.

Chapter 3 is based on **J. Xiao, A. D. Pimentel and G. Lipari [109], SysRT: A modular multiprocessor RTOS simulator for early design space exploration**, *proceedings of the 17th International Conference on Embedded Computer Systems: Architectures,Modeling, and Simulation (SAMOS),2017*. I am the principal author of this paper. I proposed the ideas, built the simulator, and was the lead writer of the paper. All the co-authors contributed to the discussions and paper writing.

Chapter 4 is based on **J. Xiao, S. Altmeyer and A. D. Pimentel [108], Schedulability analysis of non-preemptive real-time scheduling for multicore processors with shared caches**, *proceedings of IEEE Real-Time Systems Symposium (RTSS), 2017*, and its extension as a journal version: **J. Xiao, S. Altmeyer and A. D. Pimentel [110], Schedulability analysis of global scheduling for multicore systems with shared caches**, *submited to IEEE Transactions on Computers*. I am the principal author of the two papers. I proposed the ideas, proved the results, conducted the experiments, and was the lead writer of the two papers. All the co-authors contributed to the discussions and paper writing.

Chapter 5 is based on **J. Xiao and A. D. Pimentel [46], Partitioned non-preemptive scheduling for real-time multi-core systems with shared caches**, *submitted to Design, Automation and Test in Europe Conference 2020 (DATE2020)*. I am the principal author of this paper. I proposed the ideas, proved the results, conducted the experiments, and was the lead writer of the paper. All the co-authors contributed to the discussions and paper writing.

Chapter 6 is based on **J. Xiao, A. D. Pimentel and X. Liu [111], CP$_{pf}$: a prefetch**

aware LLC partitioning approach, *proceedings of the International Conference on Parallel Processing, 2019 (ICPP'19)* . I am the principal author of this paper. I proposed the ideas, conducted the experiments and analyses, and was the lead writer of the paper. All the co-authors contributed to the discussions and paper writing.

Work on other publications also contributed to the thesis, albeit indirectly. We mention the following paper:

J. Xiao and G. Buttazzo [107], Adaptive embedded control for a ball and plate system, *proceedings of the 8th International Conference on Adaptive and Self-Adaptive Systems and Applications, 2016*. Buttazzo proposed the project, I did the implementation, conducted the experiments and analyses, and was the lead writer of the paper. All the co-authors contributed to the discussions and paper writing.

Paper not related to the thesis but published during the PhD:

- **J. Xiao, S. Chiaradonna, F. Di Giandomenico, and A. Pimentel [47], Improving voltage control in mv smart grids**, *2016 IEEE International Conference on Smart Grid Communications (SmartGridComm)*.
- **S. Chiaradonna, F. Di Giandomenico, and J. Xiao [20], Quantification of the effectiveness of medium voltage control policies in smart grids**, *proceedings of the 17th International Symposium on High Assurance Systems Engineering (HASE), 2016*.

2 Background

In this chapter, we provide the concepts and background needed in later chapters in this thesis. We start with a brief introduction to computer architecture in Section 2.1, then we briefly describe the real-time scheduling theory in Section 2.2.

2.1 Computer architecture

We begin with a discussion of relevant computer architecture fundamentals. Given the breadth of the topic, a comprehensive review of computer architecture is beyond the scope of this dissertation. Instead, we focus on the parts of a computing platform: multi-core processors, caches, hardware prefetching and hardware performance monitoring unit.

2.1.1 Multi-core processors

During the last decades, the performance of uniprocessor systems has been increasing by several magnitudes. The high performance has been achieved by using a high processor clock frequency. While Dennard Scaling is seen as now coming to an end, the resulting inability to increase clock frequencies has fueled the move from uniprocessor systems to the multi-core processors, which allows to continue to boost the performance of processors through scaling up the number of cores in a processor. By doing so, the software architect is able to process in parallel, thereby significantly improving performance.

In this dissertation, a multi-core processor is considered to be a computer system with multiple (two or more) central processing units (CPUs) that share full access to a main memory and peripherals. We do not distinguish between multi-core processors and multiprocessors, thus multi-core processor is used as a synonym for multiprocessor.

Depending on the memory organization and interconnect, multiprocessors can be divided into two shared-memory model categories: *symmetric shared-memory multiprocessors* (SMPs) and *distributed shared memory multiprocessors* (DSMs) [41]. In SMPs, the processors share a single centralized memory and a bus is typically used to interconnect the processors and memory. As all processors have a uniform access latency to the memory, this type of architectures are also called *uniform memory access* (UMA) multiprocessors. By contrast, in DSMs, memory is distributed among the

processors but forms a single shared address space. A processor can access its local memory faster than accessing remote memories. Therefore, a DSM multiprocessor is also referred to as a *nonuniform memory access* (NUMA) multiprocessor. In this dissertation, we restrict our focus to SMP architectures.

2.1.2 Processor caches

To hide high off-chip memory latencies, a hierarchy of fast cache memories that contain recently accessed instructions and data is employed, taking the benefits of the principal of locality and cost-performance of memory technologies.

The *principal of locality* is the tendency of programs to access the same set of instructions or data repetitively over a short period of time. There are two types of locality: temporal and spatial locality [41].

Temporal locality: if an item is referenced, it tends to be referenced again in the near future.

Spatial locality: if an item is referenced, items whose addresses are close by tend to be referenced in the near future.

Temporal and spatial locality in programs arise from natural program structures. For example, most programs contain loops, instructions and data tend to be accessed repeatedly, experiencing high degrees of temporal locality. It is also common that instructions and elements of an array or a record are accessed sequentially, showing a high amounts of spatial locality.

Cache access. Each access to the cache results in either a *cache hit* or a *cache miss*. Cache hits occur when an application accesses data (or instructions) and finds that data (or instructions) in the cache. A cache miss happens when accessed data is not present in the cache.

Cache organization. Data is transferred between memory and cache in blocks of fixed size, referred to as *cache lines*. A cache line usually contains multiple data elements. An access to one data element causes the whole cache line to be loaded into the cache. As a result, a following access to another element in the same cache line also results in a cache hit.

Caches are typically organized as a hierarchy of several cache levels. The fastest and smallest caches are denoted level-1 (L1) caches, with deeper caches (L2, L3, etc.) being successively larger but slower. A cache contains either instructions or data, and can also contain both if it is unified. In multiprocessors, caches can be either private or shared. Private caches serve only one core. By contrast, shared caches can be accessed by multiple cores. Usually lower level caches are private while the last level caches are shared. A typical design of cache hierarchy is shown in Figure 2.1, where each core has a private L1 and L2 cache and four cores share an L3 cache.

The size of an L1 cache is about several tens of KB and has an access latency of less than 5 cycles. If a memory access misses in the L1 cache, the L2 cache is queried. The capacity of L2 caches may range from hundreds of KB to several MB, with an access latency of around 10 cycles. In some high performance multi-core processors, an L3 cache with the size of several tens of MB is deployed to further expand cache capacity. The access latency of an L3 cache ranges from 40 to 80 cycles. Misses in

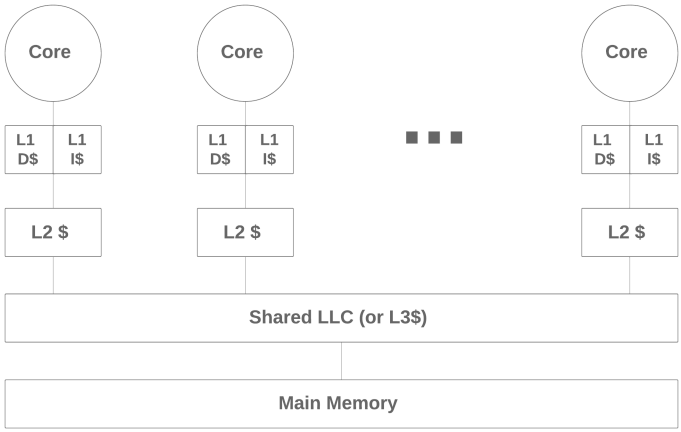


Figure 2.1: A typical design of cache hierarchy in a multi-core processor.

the last level cache trigger accesses to the main memory via the off-chip memory bus, causing a delay in the order of hundreds of cycles.

Cache mapping. Another issue in cache design is to decide where lines should be stored, i.e. if a cache line is fetched from main memory, where should it be placed? The answer depends on the cache mapping. At one extreme is a *fully associative cache*, in which a newly fetched memory block can be placed at any location in the cache. At the other extreme is a *direct mapped cache*, in which each memory block is mapped directly to exactly one location in the cache. Intermediate schemes are *n-way set associative caches*. In these schemes, every cache set has a fixed number of *ways*, each of which is a single cache line. The total number of ways within a cache set is called *associativity*. To load a memory block, the processor first determines which cache set the block maps to and then selects one of the n different ways in the cache set for the data placement.

Cache replacement. When a cache miss occurs in a direct-mapped cache, as the requested memory block can only be loaded to exactly one position, and the block occupying that position must be replaced. When a memory block can be mapped to several different locations in a cache like in a fully associative and n-way set associative cache, it is necessary to decide which cache line should be replaced. In a fully associative cache, all cache lines are candidates for replacement. In an n-way set associative cache, a way within the selected cache set has to be chosen for placing the requested block. A *cache replacement policy* is responsible for deciding which cache line is replaced when a cache miss occurs. The most commonly used scheme is *least recently used* (LRU). In an LRU scheme, the cache line that has been unused for the longest time will be chosen for the replacement. First In First Out (FIFO), and Pseudo-LRU (PLRU) are alternative cache replacement algorithms currently used by multi-core processors.

Three C's model. Cache misses are classified into one of three categories in the **three C's** model, by the source of misses in a cache [43]:

- *Compulsory misses:* these are cache misses caused by the first access to a memory

block that has never been brought into the cache.

- *Capacity misses:* these are cache misses caused when the cache cannot contain all the memory blocks accessed by a program. Capacity misses occur because of blocks being replaced in the cache and later on requested again by the CPU.

- *Conflict misses:* these are cache misses that occur when multiple memory blocks map to and compete for the same cache set. These cache misses are also called *collision misses*.

A special Cache: TLB. Processors with virtual memory using *memory management units* (MMU) usually have a *translation look-aside buffer* (TLB) [64]. A MMU translates virtual memory addresses into physical memory addresses. Since performing such a translation is relatively slow, the TLB, a special address translation cache, is deployed to store previously resolved virtual-to-physical address mappings. Thanks to the principle of locality (if the accesses have locality, the address translations for the accesses will also have locality), the TLB ensures that the MMU does not have to perform a translation on every memory reference.

2.1.3 Shared cache interference

When multiple applications run concurrently on a multi-core processor, they compete among each other for cache space. The execution time of a task in a multi-core processor can be affected by two types of cache interference: intra-core cache interference and inter-core cache interference.

Intra-core cache interference intra-core interference occurs within a core, specifically, when a task is preempted and its data is evicted from the cache by the preempting tasks. As a result, the preempted task may experience an extra execution delay due to the increased data access time as soon as it is rescheduled. The severity of the experienced delay depends on the particular cache replacement policy, the length of the preemption and the data access pattern of the preempting task [49, 74].

Inter-core cache interference inter-core interference may happen when tasks executing on different cores access the shared cache simultaneously [49]. If data in the different addressing spaces of the running tasks are loaded to the same cache line, memory (i.e. cache) accesses from different tasks can evict each other in cache, leading to complex timing interactions. Since this type of interference is suffered from tasks that run in parallel, an exact analysis requires analyzing all the possible interleavings of task executions, which is intractable. Therefore, it is extremely difficult to integrate the inter-core interference into a static timing analysis framework.

2.1.4 Cache Partitioning

Cache partitioning, i.e., dividing cache space between applications (or cores), is a promising approach to mitigate the negative impact of cache sharing. Cache partitioning has been widely used to improve system performance, fairness and QoS (quality-of-service) guarantees. We now present three common techniques (hardware, software and hybrid techniques) for partitioning shared caches in multi-core processors.

Hardware Techniques. Hardware techniques modify the cache to support partitioning. Way-partitioning [3, 21, 76], the most common technique, restricts insertions from each partition to its assigned subset of ways. However, simple, way-partitioning has significant limitations: it supports only coarsely-sized partitions, which is multiples of the way size, and the number of partitions is proportional to the number of ways. Prior work has proposed alternative hardware cache partition techniques. For example, in [12, 72, 96], the cache is partitioned by sets instead of ways by configuring the indexing function. [63, 77, 97, 102, 112] modify the cache insertion and replacement policies.

Software Techniques. The most common software-based cache partitioning technique is page coloring[94]. Page coloring exploits the virtual to physical page address translations present in virtual memory systems at OS-level. Each partition is allowed to use its own assigned physical pages that are mapped to specific cache sets. By restricting the physical pages used by each partition, the overlap of cache spaces can be avoided. Page coloring has the advantage of no need for hardware support and does not sacrifice associativity. However, it has several drawbacks. First, page coloring requires heavy modifications to the OS's virtual memory subsystem and precludes the use of other beneficial features, such as superpages. Second, partitions are coarsely sized, which is in multiples of page size$\times$cache ways, resulting in a limited number of partitions. Third, repartitioning incurs large overheads due to the costly process of recoloring memory pages.

Hybrid Techniques. As a hybrid cache partition technique, SWAP [98] combines both set- and way-partitioning to achieve finer-granularity partitions. By cooperatively managing cache ways and sets, SWAP can successfully provide hundreds of fine-grained cache partitions for the manycore era. SWAP requires no additional hardware beyond way partitioning. In fact, SWAP is readily implemented in existing commercial servers whose processors provide support for hardware way-partitioning. However, SWAP leverages page coloring, thus inherits the limitations of page coloring.

2.1.5 Cache Allocation technology

Recent Intel processors have proposed the so-called *cache allocation technology* (CAT), as hardware support for Way-partitioning [42]. CAT provides software-programmable control over the amount of cache space that can be used by a given application.

Processors that support CAT have a predefined number of classes of service (CLOS), for example, 11 in the Intel Xeon Gold 6148 processor and 20 in the Intel Xeon E5 2658 processor. Each CLOS is associated with a capacity bit mask (CBM) that controls the accessibility of cache lines at cache-way granularity. Each bit CBM grants write access to the corresponding way in the cache set. Cores (or threads) can be configured to belong to a CLOS. CBMs can overlap at some cache ways, which means that parts of cache ways can be shared by different CLOSs. One requirement of configuring a CBM is that all the bits set in a CBM must be consecutive, i.e. a CLOS uses consecutive cache ways in the cache. Each application is assigned a CLOS and an application can only access the cache ways defined by the CBM for that CLOS.

One can use `Intel-cat-cmt`, which is a library [24] developed by Intel, to configure CAT. By default, all cores (and applications) are grouped into to CLOS #0.

Figure 2.2 shows an example of a possible cache partitioning scheme. Each of the four possible classes of service (CLOS #0 to CLOS #3) has assigned a subset of the 20 ways of the LLC, and each core is mapped to a CLOS. Each CLOS is identified by a color which marks both the applications that belong to the CLOS and the cache ways they can access. For instance, core 0 is assigned to CLOS #0 and core 1 to CLOS #1. Note that all the CBMs are contiguous and core 1 and core 2 share cache ways 10 and 11.

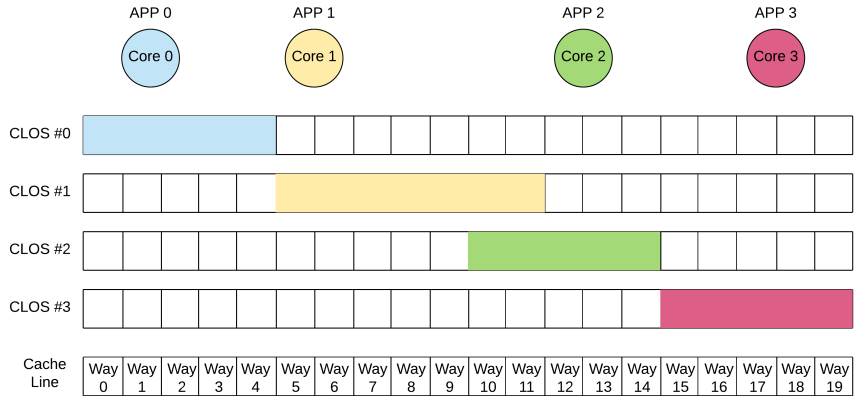


Figure 2.2: An cache partitioning scheme using cache allocation technology.

2.1.6 Hardware prefetching

Hardware prefetching is another optimization technique that is commonly employed to reduce the observed memory access time and the performance gap between processors and memory. Prefetching predicts the memory addresses a program will access in the near future and issues memory requests to those addresses in advance of explicit accesses. By doing so, prefetching can hide the latency of a memory access since the processor either does not experience a cache miss for that data access or incurs a cache miss that is satisfied before the processor needs that data. There have been a myriad of proposed prefetching techniques, and nearly every modern processor includes some hardware prefetching mechanisms targeting simple and regular memory access patterns.

For example, there are five distinct hardware prefetchers on Intel Xeon platforms. Two prefetchers are associated with the L1-data caches: a Data Cache Unit (DCU) IP prefetcher and a DCU streamer prefetcher per core. The DCU IP prefetcher keeps track of individual load instructions. It uses sequential load history to determine whether to prefetch additional lines. The DCU streamer prefetcher is triggered accesses to very recently loaded data. It fetches the next cache line into L1-D cache.

Two prefetchers are associated with the L2 caches: a Mid-Level Cache (MLC) spatial prefetcher and a MLC streaming prefetcher. The spatial prefetcher strives to complete every cache line fetched to the L2 cache with the pair line that comprises a 128-byte aligned chunk. The streamer prefetcher monitors read requests from the L1

cache for ascending (and descending) sequences of addresses. Monitored read requests include L1 data cache requests initiated by load and store operations, and L1 instruction cache requests for fetching code. When a forward or backward stream of requests is detected, the anticipated cache lines are prefetched into the L2 cache. Prefetched cache lines must be in the same 4K virtual memory page.

Xeon processors support a special L2 streaming prefetcher, which prefetches data only into the L3. It is also known as LLC prefetch (or L3 prefetch) though it is still initiated by L2.

We can activate or deactivate these hardware prefetchers by setting the corresponding machine state register (MSR) bits [25].

2.1.7 Hardware PMU

To provide realtime micro-architectural information about the processes currently executing on the chip, a rich set of *Hardware Performance Monitoring Units* (PMUs) is implemented in today's processor micro-architectures. PMUs are a set of special-purpose registers to store the counts of hardware-related activities within computer systems such as cpu cycles, instructions executed, cache statistics, etc. PMUs also support advanced event sampling, a mechanism that collects event samples at a predefined sampling period. The event based sampling is realized by Intel's Precise Event-Based Sampling (PEBS) [36] and AMD's Instruction Based Sampling (IBS) [28].

To use the PEBS mechanism, a counter is configured to overflow after it has counted a preset number of events. After the counter overflows, the processor copies the current state of the general-purpose registers and instruction pointer in the records buffer. The processor then resets the performance counters and restarts the event counter.

As illustrated in Figure 2.3, the event `MEM_LOAD_UOPS_RETIRED:L3_MISS` is configured to drive PMU sampling. It precisely monitors cache misses at the LLC. If the sampling period is set to n, the PMU samples one data address that causes an LLC miss every n LLC misses.

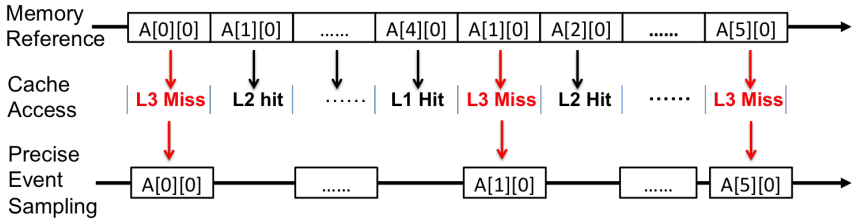


Figure 2.3: PMU data address sampling.

Linux' `perf_event` is a standard programming interface to set up performance monitoring through PMUs. More specifically, `perf_event_open` [27] can set the PMUs in sampling mode, and the overflow event can be enabled via `ioctl()` calls. The Linux kernel can deliver a signal to the threads whose PMU event counter overflows. The user code can `mmap` a circular buffer into which the kernel keeps appending the PMU data on each sample. The user can also read those circular buffers.

2.2 Real-time systems

Some embedded systems, which are referred to as real time systems, must react to events in the environment with precise time constraints. A real-time system is a computer system whose behavior depends not only on the functional correctness of the computation, but also on the time at which results are produced [18]. Violating timing constraints of a real time system such as chemical and nuclear plant control, railway switching systems, flight control systems, may lead to catastrophic consequences.

Rather than being computationally fast, a real-time computing system must be predictable. To achieve predictability, it is necessary to apply methodologies at every stage of the development of the system, from design to testing. Over the last decades, a number of methodologies and analysis techniques have been proposed in the literature to improve the predictability of real-time systems. In the following, we briefly review the real-time task models, real-time scheduling algorithms and schedulability analysis.

2.2.1 Real-time task models

A *task* is a computational activity that is executed by the processor in a sequential fashion. Particularly well-studied real-time task models are the *periodic task model* and the *sporadic task model*. In both models, a task is a infinite sequence of jobs. In the periodic task model, the jobs of a task are released periodically, separated by a fixed time interval. In the sporadic task model, two consecutive jobs are separated by a minimum inter-arrival time. Each task $\tau_k = (C_k, D_k, T_k)$ is characterized by a worst-case computation time C_k, a period or minimum inter-arrival time T_k, and a relative deadline D_k.

Three levels of constraint on task deadlines are studied in the literature: (1) Implicit deadlines, in which task deadlines are equal to their periods ($D_k = T_k$), (2) Constrained deadlines, in which task deadlines are less than or equal to their periods ($D_k \leq T_k$), (3) Arbitrary deadlines, in which task deadlines can be less than, equal to, or greater than their periods. In this dissertation, we restrict our focus to constrained deadlines.

As illustrated in Figure 2.4, a task τ_k is a sequence of jobs. Let J_k^j denote the jth instance of task τ_k. The arrival time of J_k^j, i.e. the time instant when a job becomes available for execution, is denoted by r_k^j. Once a task is ready for execution, it may not get executed immediately. The time instant at which J_k^j starts to execute is denoted by s_k^j and J_k^j completes its execution at f_k^j. The absolute deadline of J_k^j is $d_k^j = r_k^j + D_k$.

J_k^j's response time, denoted by R_k^j, is the time interval from the arrival time to the time when the job is terminated, i.e. $R_k^j = f_k^j - r_k^j$. The goal of a real-time scheduling algorithm is to guarantee that each job will complete before its absolute deadline: $f_k^j \leq d_k^j$.

2.2.2 Scheduling algorithms

From the perspective of real-time scheduling, shared-memory multiprocessors can be further classified into three categories based on the capabilities of their constituent processors: *identical multiprocessors*, *uniform multiprocessors* and *heterogeneous*

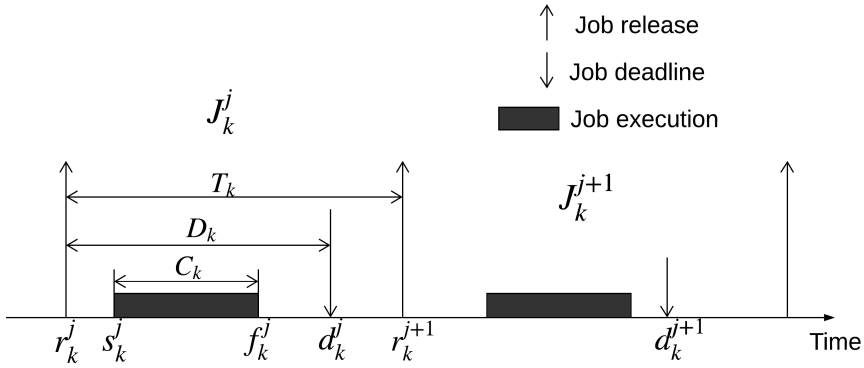


Figure 2.4: Periodic (sporadic) task model and task parameters.

multiprocessors. In *identical multiprocessors*, each processor has the same computation power, hence the amount of work completed by executing a task for a fixed duration of time on a processor is the same, regardless of which task is being executed, and on which processor. In *uniform multiprocessors*, each processor is characterized by its own computing capacity hence the execution rate of a task depends on which processor it executes on. Finally, in *heterogeneous multiprocessors*, each processor may have special capabilities such as application-specific co-processors hence the amount of work completed by executing a job for a fixed duration of time upon a processor depends on the identities of both the job and the processor. We restrict our focus to identical multiprocessors in this dissertation.

In multitasking systems, the processor(s) are assigned to the various tasks according to a predefined criterion, referred as a *scheduling policy*. The set of rules that determines the order in which tasks are executed is called a *scheduling algorithm*. Real-time scheduling problems can be divided into two categories by the number of processors in the computation platform: uniprocessor real-time scheduling and multiprocessor real-time scheduling.

Uniprocessor real-time scheduling

Research on uniprocessor real-time scheduling started in the late 1960s and significant research efforts were made in the 1980s and 1990s. [5] and [82] provide historical accounts of the most important achievements in the field of uniprocessor scheduling during those decades. The uniprocessor real-time scheduling theory is reasonably mature, as a large amount of research results are documented in the textbooks such as [17, 18], and some of those results are successfully applied to industrial practice. The two well-known uniprocessor scheduling policies are *fixed-priority* (FP) and *earliest-deadline first* (EDF) scheduling.

Under FP scheduling, each task is statically assigned a unique priority prior to execution. At runtime, competing jobs are then scheduled in order of decreasing task priority. One example of FP scheduling is *rate monotonic* (RM) scheduling. RM

scheduling assigns priorities to tasks according to their request rates. Specifically, tasks with higher request rates, i.e. shorter periods, get higher priorities. Since task periods are constant, the priority assigned to the task does not change over time.

EDF is a dynamic priority scheduling algorithm that selects tasks according to their absolute deadlines. Specifically, tasks with earlier deadlines are assigned with higher priorities. Since each job's absolute deadline changes over time, the priority of a task changes dynamically. In a classic result, EDF is optimal for uniprocessor real-time scheduling with HRT constraints [56].

Multiprocessor real-time scheduling

Multiprocessor real-time scheduling theory also traces its origins back to the late 1960s. As noted in [55], multiprocessor real-time scheduling is intrinsically a much more difficult problem than uniprocessor scheduling. Few of the results obtained for a uniprocessor generalize directly to the multiprocessor case. Unlike uniprocessor scheduling, in which the scheduling of tasks only involves the dimension of time, i.e., to decide when to execute a certain task, multiprocessor scheduling also involves the dimension of space as it also needs to decide where (i.e., on which core) to execute a task.

There are two fundamental classes of multiprocessor schedulers: global and partitioned. Under global scheduling, all processors serve a single queue of tasks ready to execute and jobs may migrate among processors. In contrast, under partitioned scheduling, tasks are statically assigned to processors during an offline phase and no task migration is permitted. Each processor then is scheduled individually using a uniprocessor policy such as EDF or FP.

In some systems, the running task can be interrupted at any time. If a task with higher priority becomes ready to execute and all processors are occupied by some other tasks, the running task with lowest priority is suspended, leaving the processor for the execution of the ready task with higher priority. The operation of suspending a running task is called *preemption*.

Scheduling algorithms can be further classified into three categories with respect to whether preemption is allowed or not. (1) Preemptive scheduling. The running task can be preempted at any time, giving the core to another ready task. (2) Non-preemptive scheduling. Once a task starts executing, it will not be preempted and will therefore occupy the core until the completion of its execution. (3) Cooperative. Tasks can only be preempted at defined scheduling points within their execution. We restrict our focus to preemptive and non-preemptive scheduling in this dissertation.

We now show some examples of preemptive and non-preemptive real-time scheduling by considering a taskset τ consisting of 4 real-time tasks: $T_1 = (2, 4, 4)$, $T_2 = (2, 5, 5)$, $T_3 = (4, 9, 10)$, $T_4 = (5, 20, 20)$ to be scheduled on a processor with 2 cores. Note that the 3-tuple task model is explained in Section 2.2.1.

Preemptive scheduling

The scheduling of τ under preemptive RM scheduling is depicted in Figure 2.5. RM scheduling assigns a fixed priority P_i to each task τ_i ($i = 1, 2, 3, 4$), such that $P_1 >$

$P_2 > P_3 > P_4$.

At $t = 0$, although all tasks are ready to execute, only τ_1 and τ_2 execute as only two core are available. At $t = 2$, both τ_1 and τ_2 finish execution, so τ_3 and τ_4 start their execution. At $t = 4$, τ_1 becomes ready again. Since τ_1 has the highest priority, the executing task with the lowest priority, which is τ_4, is preempted, leaving one core for the execution of τ_1. Similarly, at $t = 5$, τ_3 is preempted for the execution of τ_2.

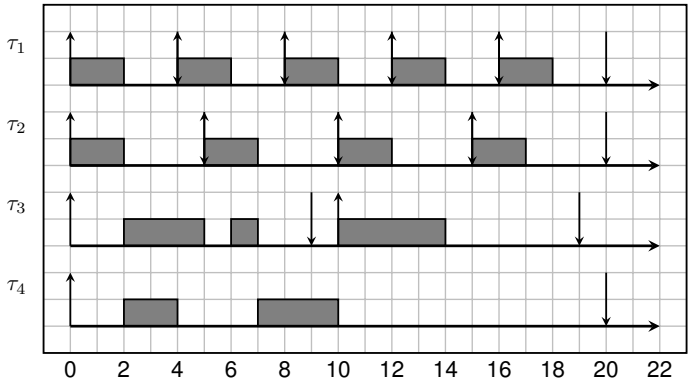


Figure 2.5: The scheduling of taskset τ under preemptive RM.

Figure 2.6 shows the scheduling of the same taskset τ under preemptive EDF. Different from preemptive RM scheduling, at $t = 5$, τ_2 can not preempt τ_3 since the absolute deadline of the job from τ_3 (i.e. $t = 9$) is earlier than the absolute deadline of τ_2's ready job (i.e. $t = 10$). Thus, τ_2 starts its execution when τ_3 finishes.

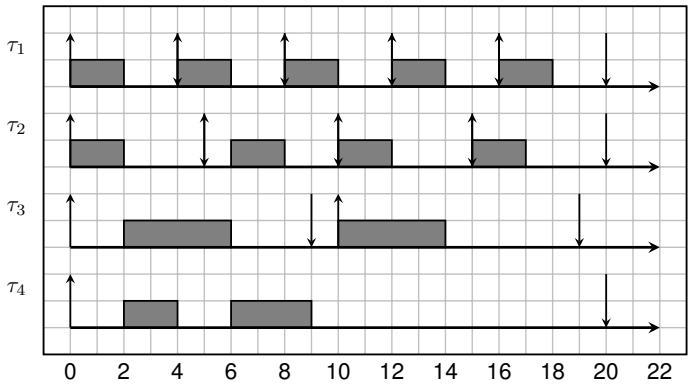


Figure 2.6: The scheduling of taskset τ under preemptive EDF.

Non-preemptive scheduling

The scheduling of τ under non-preemptive RM is illustrated in Figure 2.7. At $t = 2$, τ_3 and τ_4 start their job execution. Once a job gets executed, it is assigned with the highest

priority. Even though τ_1 and τ_2 have a higher task priority than τ_3 and τ_4, they can not preempt τ_3's and τ_4's executing jobs. τ_1's job starts its execution when the job of τ_3 finishes. Similarly, the processing of τ_2's job begins when the job of τ_4 completes.

The scheduling of taskset τ under non-preemptive EDF is same as shown in Figure 2.7.

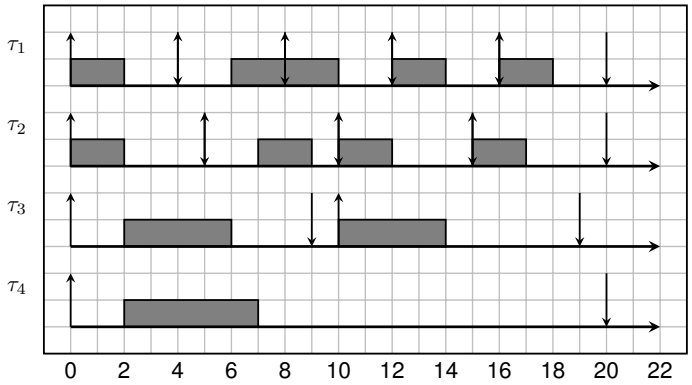


Figure 2.7: The scheduling of taskset τ under non-preemptive RM.

Preemptive scheduling typically allows higher efficiency, in the sense that it allows to schedule a real-time task set with higher processor utilization. However, preemption destroys program locality and consequently may introduce cache preemption related delay that inflates the execution time of tasks.

2.2.3 Schedulability analysis

A task is *schedulable* according to a given scheduling algorithm if all of its released jobs can be guaranteed to complete their executions before their deadlines. A taskset is schedulable according to a given scheduling algorithm if all of its tasks are schedulable.

The fundamental problem in embedded real-time system design is to analyze and verify the schedulability of the taskset under the scheduling algorithm, which is referred to as *schedulability analysis*.

The schedulability analysis of global multiprocessor scheduling has been intensively studied, of which comprehensive surveys can be found in [26, 82]. Details on the analytical schedulability analysis can be found in textbooks like [58] and [18]. As one of the fundamental methodologies for schedulability analysis, the Response Time Analysis (RTA) [18], which employs an iterative procedure to compute a task's worst-case response time, has been wildly applied to the timing verification of real-time systems.

Most multi-core scheduling approaches assume that the WCETs are estimated in an offline and isolated manner and that WCET values are fixed. when two or more tasks are executed in parallel on different cores. However, the interplay between the tasks on shared caches may lead to unpredictable delays [95].For example, useful cache blocks that were loaded by one task can be evicted by another task executing simultaneously

on a different core. Therefore, using the WCET of tasks executing in isolation on a multi-core platform without considering co-runner interference can potentially lead to incorrect WCRT values in the context of the complete multi-core system, which invalidates the traditional analysis framework with independent program-level and system-level timing analysis. This problem is a major obstacle to use multi-core processors for real-time systems [61].

Now we have provided the necessary background for the reminder of this dissertation, we move to the first research chapter: SysRT: A Modular Multiprocessor RTOS Simulator for Early Design Space Exploration.

Timing predictability for embedded multi-core computing

3

SysRT: A Modular Multiprocessor RTOS Simulator for Early Design Space Exploration

In the previous chapter we have described the background knowledge for this thesis. Starting from this chapter, we begin to present our research and answer the research questions we listed in Chapter 1. This chapter addresses RQ1, which is concerned with modeling and simulation of real time embedded systems.

In the past years, the design of systems-on-chip (SoCs) has become increasingly complex. Hardware architectures are migrating from simple single-core based systems to more complex multi-core architectures. In the embedded systems domain, together with the increasing hardware complexity, the software complexity has also been growing dramatically. Modern embedded systems increasingly execute several applications of different types concurrently on the underlying computing platform. These applications can have different execution requirements. For example, control applications typically are hard real-time applications and thus have stringent timing constraints, while best-effort applications prefer a short task response time. These systems are usually managed by a Real-Time Operating System (RTOS).

Raising the level of abstraction is generally considered as a solution to address the design complexity, thus reducing time-to-market. To provide a simulation environment and to help in the design space exploration (DSE) at the early stages of design, various system-level design languages (SLDL) and methodologies have been proposed, such as SystemC [93] and SpecC [85]. Originally, SLDLs primarily focused on hardware modeling and did not properly address the modeling of software aspects. Later efforts introduced methods to model timing behavior of software in SLDLs. But most solutions still lack direct support for simulating the real-time behavior of concurrent applications, such as preemption or scheduling within the RTOS. To verify that the timing requirements posed by applications are met during the early stages of design, a fast system-level simulator, capturing both the modeling of software and hardware, is needed.

The modeling and simulation of RTOS with SLDL have received widespread attention from many researchers, [50, 116, 117]. In [40], the modeling capability of SystemC

This chapter was published as [109].

has been extended by RTOS services to provide more realistic software modeling features. However, to realize features such as preemption and scheduling, a scheduler model is invoked every simulation quantum, similar to the way a real OS scheduler behaves. This quantum-granularity based simulation approach therefore introduces large overheads, resulting in low simulation speeds. Later efforts such as [48, 89] focused on improving the accuracy of high-level simulation while maintaining high performance. However, these works still trade-off speed for accuracy.

In [73], a host-compiled multi-core system simulator is presented for early real-time performance evaluation. They present an integrated approach for automatic timing granularity adjustment to optimally navigate simulation speed versus accuracy. This approach switches between prediction mode and fallback mode. In prediction mode, a prediction of the next scheduling points is performed based on the simulation parameters and states of periodic tasks. Schirner et al. [78] introduce preemptive scheduling in abstract RTOS models using Result Oriented Modeling (ROM). To speed up simulation, ROM optimistically predicts the finish time of a process already at the start time by a ”run to finish” assumption. ROM records any possible preemption that may alter the predicted outcome. While time passes, it validates the prediction and takes corrective measures to ensure accuracy. However, predictions of preemption points are difficult if the simulation uses more complex task models like Directed Acyclic Graphs (DAGs) and resource sharing models.

Therefore, we address the following research questions:

RQ1 How to provide fast simulation of real-time embedded systems for design space exploration at the early stages of system design? How to accurately capture the timing behaviour of embedded software? How to efficiently implement the simulator to provide support for easy plug-in of new task models, new schedulers and new resource sharing protocols?

To answer this question, we develop SysRT, a generic and high-level RTOS simulator that is highly suited for early design space exploration (DSE). SysRT contains different types of application models and a modular RTOS kernel model, all developed in SystemC. Efficient and precise modeling of preemptive scheduling is achieved via an event-driven simulation approach, which utilizes scheduling events associated with task states and interrupts, allowing simulations to be performed much faster than cycle-accurate simulations. At the same time, the kernel model is developed to be generic and modular to support for easy plug-in of new schedulers as well as new resource sharing protocols. Comparing SysRT to state-of-art simulators, it achieves faster simulation speeds with an identically small simulation error. We demonstrate the flexibility of SysRT and its benefits for early DSE using experiments with a mixed workload executing on multiprocessor platforms with different numbers of cores.

The rest of the chapter is organized as follows. The overall RTOS simulation framework is described in Section 3.1. Section 3.2 describes the application models. In Section 3.3, the kernel model is detailed, and Section 3.4 presents a range of experimental results. Section 3.5 concludes this chapter.

3.1 Modeling Framework

SysRT consists of three layers, as shown in Figure 3.1: the *application layer*, the *kernel layer*, and the *architecture layer*. In the application layer, the user can model a set of *processes*. A process can be a single job instance (named ST in Figure 3.1), a *Periodic Task* (PT) of which job instances are invoked periodically, or a process with execution precedences modeled by a DAG, as will be explained in Section 3.2.

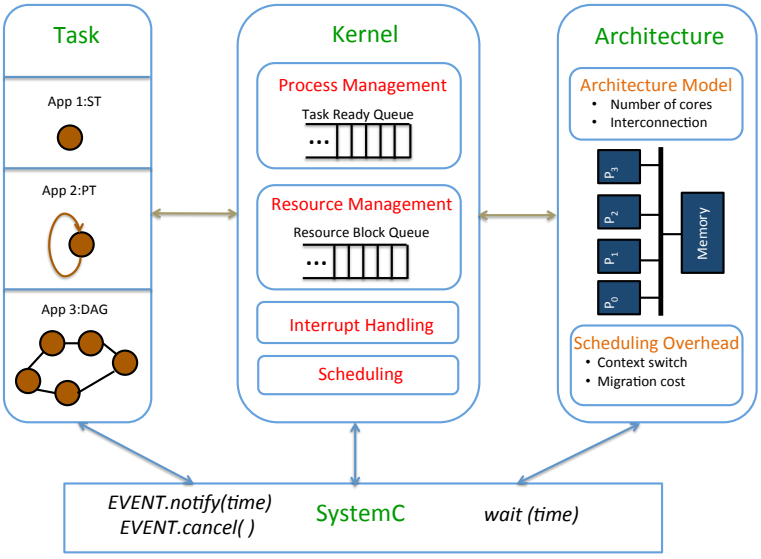


Figure 3.1: Simulation framework of SysRT.

The application layer interacts with the RTOS kernel layer. The application informs the kernel of its execution states, while the kernel model returns task scheduling decisions. We model four functionalities of the OS kernel, namely process management, resource management, interrupt handling and real-time scheduling. A queue in the OS kernel is used to order the tasks that become ready for execution. The OS kernel further has a resource manager sub-module that controls access to resources shared between tasks. The resource block queues store tasks waiting to get access to a particular resource due to mutual exclusion. Moreover, interrupt service routines are defined in the OS kernel model. When an interrupt is generated, either from software or hardware, the OS kernel schedules the corresponding interrupt handler depending on the handler priority. Different real-time (preemptive) schedulers are implemented in the scheduling module of the OS kernel model. The architecture layer models the hardware computing platform. It specifies the number of cores in the SoC platform, the interconnection between the cores, and the hardware interrupt interfaces. The current architecture model mainly accounts for the scheduling overhead including migration and context switching overhead after a scheduling decision is made by the OS kernel. The implementation details of the architecture model are beyond the scope of this chapter.

Application layer, OS kernel layer and architecture layer are implemented on top of the basic classes and primitives provided by SystemC. We use event-driven simulation, where events are modeled by the *sc_event* class. This class allows explicit triggering of events by means of a notification method. The *Event.notify(sc_time t)* method notifies or posts an event after time *t*. If a simulation process is set to be sensitive to an event, then this process acts as the corresponding event handler. When an event occurs, the corresponding event handler is invoked and scheduled by the SystemC simulation kernel. Scheduled events may be canceled with the *event.cancel()* method.

Modelling preemption is always a challenging topic for a RTOS simulator. Most RTOS simulators that are built on top of SystemC use *wait(sc_time time)* to model task execution latency. If a task is preempted for some time, then the preemption time is counted as extra task execution latency, resulting in another execution of *wait(sc_time time)* for that task. However, this approach comes with a speed penalty due to the frequent computations of the preemption time and the frequent executions of *wait(sc_time time)*. Unlike this approach, SysRT adopts an event-driven approach that uses only *sc_event* to model preemption. Events are extracted from the task execution states, which will be discussed soon. Once a task is preempted, the only work to do is to cancel the task finishing event. When this task is scheduled again, a new task finishing event is posted after the remaining execution time. compared with the *wait(sc_time time)* method, this event-driven approach introduces less simulation overhead.

3.2 Application model

The *Application* is a program that contains a set of coordinated tasks modeled by the user through the *Task* module. In this work, the actual task functionality is abstracted away, and only the timing of task execution is simulated. Here, we assume that timing information of task execution latencies are estimated or known a-priori.

3.2.1 Task Model

In the task model, three kinds of constraints specified on real-time tasks are considered: timing constraints, precedence relations, and access control on shared resources. Timing constraints, such as execution times and job deadlines, are specified at the creation of a real-time task object. Precedence constraints are realized by a DAG task model [75]. Contention on shared resources is simulated by adding *wait/signal* instructions in the task execution routine, as will be explained below.

A task module contains a list of high-level instructions that are executed in sequence. *Instruction* sub-modules are added to a *task* module by the *InsertCode* method. For example, consider a task T_1 that computes for 500 milliseconds, then tries to get access to a shared variable R_1 after which it occupies the resource for 50 milliseconds once the access is granted, and after releasing the shared resource the task finishes its current job by computing for another 300 milliseconds. This can be modeled by: T_1.*InsertCode*("*execute(500)*; *wait(*R_1*)*; *execute(50)*; *signal(*R_1*)*; *execute(300)*"). Details about the instruction module will be described in Section 3.2.2.

The simulation is driven by events generated by the first job of each task. The typical

events generated for a task are illustrated in Figure 3.2. A *job_arrival* event is posted at the activation offset (start time) ϕ_i by the *start_of_simulation()* method in the *Task* module which is called at the beginning of the simulation. A *job_arrival* event is notified every time when the task becomes ready to execute. Between the job arrival time and finish time, a job may miss its relative deadline. For such cases, a *deadline_miss* event is posted at time ϕ_i+D_i, where D_i is the relative deadline of task i. The action of the *deadline_miss* event handler is specified by the user. Possible actions are to kill the job instance, to ignore the deadline miss or even to stop the simulation. Once a job starts its execution, a *job_end* event is posted at time ϕ_i+C_i, where C_i is the execution latency of task i. The responsibility of the *job_end* event handler is to cancel the pending *deadline_miss* event and to call the kernel interface to inform it to schedule another task. A schedule event is posted by the OS kernel to a specific task if it was selected to be scheduled. The schedule event handler *schedule()* then schedules the instructions of the task. A *deschedule* event is generated if a task is preempted by another task with a higher priority. The *deschedule* event handler *deschedule*() cancels the pending *job_end* event, records the current time stamp and computes the executed job length. When the task is re-scheduled, a new *job_end* event is posted for the job's remaining execution time.

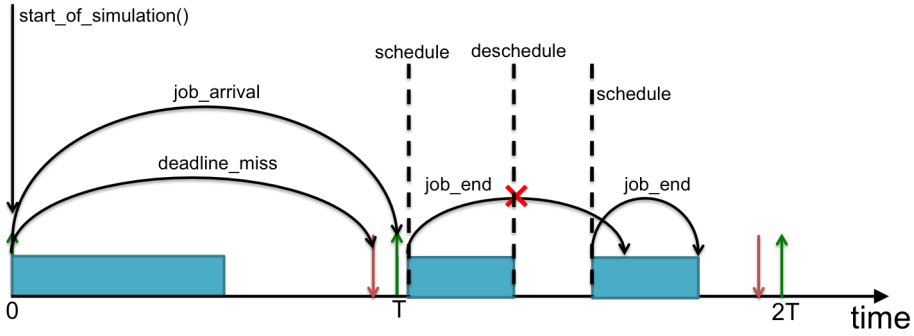


Figure 3.2: Task events.

The UML class diagram of task modules is shown in Figure 3.3(a). *AbsTask* defines the interface that must be implemented by a general task. It includes an *activate()* method, which activates the task, as well as *schedule()/deschedule()* methods, which modify the task state and related variables when a task is scheduled/descheduled. *AbsRTTask* defines the interface that should be provided by a real-time task and contains methods for getting the absolute and relative deadline of a task.

Periodic Task Model: Periodic tasks consist of a number of instances or jobs that are regularly activated at each period. Periodic tasks are reactivated by the *job_arrival* event handler, which posts a new *job_arrival* event at the next period.

DAG Task Model: A DAG is a graph of real-time subtasks (also called nodes) that captures their execution precedences. The subtasks share the same deadline and period but differ in their WCET. The *DagNode* module is used to construct a DAG application model in SysRT.

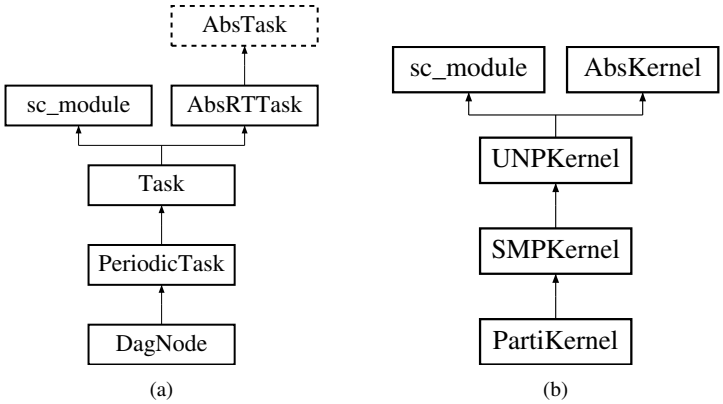


Figure 3.3: (a) Task module and (b) Kernel module.

3.2.2 Instruction Model

Instructions inside tasks are modeled using the *Instruction* class. There are two kinds of instructions. First, *execute(sc_time time)* is used to model the execution time required to execute a real code segment in an application. It can be described by a random variable, making it possible to model a portion of code with an arbitrarily distributed random execution time. The other instruction type is *wait(Resource res)*/*signal(Resource res)*, which models the request or release of a shared resource. A task executes all the instructions in sequence. A job instance is completed only after its last instruction was executed. If a task is activated again (i.e. firing a new job), then the instruction pointer is reset to the first instruction.

The *schedule/deschedule* event propagates from a task to its instructions. If a task is selected to execute at time *t*, the task calls its instruction interface and notifies a *schedule* event in the *Instruction* module. Suppose that the execution duration of the instruction is *instr_time*, the *schedule* event handler in the *Instruction* module will post an *end_instr* event at time t+$instr\_time$. The *end_instr* event handler increments the instruction pointer to the next instruction in the task and posts a new *end_instr* event for the next instruction. If there are no more instructions to execute, the interface of the task module is invoked and a *job_end* event is posted. During instruction execution, a task may be preempted and rescheduled. A similar event propagation mechanism between a task and its instructions applies to the *deschedule* event.

Based on the assumption that the actual requesting and releasing of a resource takes zero time, the *end_instr* event is notified immediately if the current scheduled instruction is *wait* or *signal*. The *end_instr* event handler for the *wait* instruction communicates with operating system kernel by calling the interface *request_resource(Kernel, Resource, resource_quantity)*. As a result, the task gets the resource if a sufficient quantity of that resource is available. Otherwise, the task is blocked by the operating system kernel. For the *signal* instruction, the *end_instr* event handler invokes the interface *release_resource(Kernel, Resource, resource_quantity)* in the operating system kernel module. The task releases the resource quantity used.

3.3 RTOS Kernel Model

Figure 3.3(b) shows the UML class diagram of the OS kernel module. The AbsKernel class is an abstract class that defines the minimal functionality of a kernel. The *UNPKernel* and *SMPKernel* classes are implemented to model an OS kernel running on a uniprocessor system (UNP) or a symmetric multiprocessor system (SMP), respectively. Traditional real-time multiprocessor schedulers can be classified in two categories: global and partitioned schedulers. Global Earliest-Deadline-First (G-EDF) and Partitioned-EDF (P-EDF) are examples of each category. The *SMPKernel* class models a general OS kernel with a global scheduler, whereas the *PartiKernel* class models an OS kernel with partitioned schedulers.

In this work, we mainly consider services of process management, resource management, interrupt handling and real-time scheduling provided by the OS kernel. We have developed the modules of the OS kernel model with the aim to provide a flexible and extendable framework to facilitate implementation, testing and evaluation of different real-time schedulers with various resource sharing protocols.

3.3.1 UNPKernel Model

The *UNPKernel* module is developed to model a real-time OS kernel running on a uniprocessor. It contains sub-components such as the *Scheduler* module and the *ResManager* module that is responsible for performing resource access related operations. These sub-components are set through methods *set_sched (Scheduler* s)* and *set_resmanager (ResManager* rm)*.

At initialization, a *CPU* pointer, which points to the modeled architecture, is created in the *UNPKernel* module to get information of the architecture platform. Since at most one task is allowed to execute at a time in a uniprocessor system, one pointer *cur_exe* is enough to track the current executing task.

For the communication with tasks, the *UNPKernel* module provides several functions. These include the functions *Arrival(AbsRTTask* t)* and *End(AbsRTTask* t)*. The function *Arrival(AbsRTTask* t)* is called by the task arrival event handler. This method inserts the task in the ready queue, followed by a function call to make a schedule decision. *End(AbsRTTask* t)* is invoked by a task when the task completes its execution. This function removes the task from the ready queue and sets the *cur_exe* pointer to null. To suspend a task, the UNPKernel class implements a *Suspend(AbsRTTask* t)* function. This function removes the task from the ready queue. If the task was executing, then it will first be descheduled. When a task is resumed (from suspension by the OS or from being blocked on a resource), the kernel reactivates the task by calling *Activate(AbsRTTask* t)* which simply inserts the task in the ready queue and changes the task's state to ready.

The operation of allocating the CPU for task execution is referred to as dispatching. The dispatching activity is simulated by the *dispatch()* function. Any circumstance that may change the current executing task should invoke *dispatch()* to make a scheduling decision:

- when a new task becomes ready;

- when a task finishes its current job;
- when a task is blocked;
- when an interrupt arrives, activating its corresponding interrupt handler.

On uniprocessor systems, just one execution flow can progress at a time. Therefore, *dispatch()* is simple in *UNPKernel* as compared with its implementation in other kernel modules. It simply compares the executing task with the first task in the ready queue. If they are different, it forces a context switch, which involves the participation of architecture model to simulate the context switch overhead. When the context switch has finished, the kernel schedules the newly dispatched task. Important to realize is that the *dispatch()* function *has been decoupled from the scheduler* that actually determines the order of the tasks in the ready queue, according to the implemented scheduling algorithm.

3.3.2 SMPKernel Model

The *SMPKernel* is a module modeling a real-time kernel with a global scheduler for (SMP) multiprocessor systems. On multiprocessor systems, multiple tasks are allowed to run concurrently. The *SMPKernel* module keeps track of the status of each individual processor, storing information about which task is executing on which processor, which tasks are about to be dispatched to which processor, and whether or not processors are in the process of performing a context switch.

The functions provided to the *Task* module and methods related to process management in the *SMPKernel* module are similar to those in the *UNPKernel* module. However, the function to make a scheduling decision, *dispatch()*, is more complicated. Pseudocode 3.1 shows the procedure of the *dispatch()* method in *SMPKernel*.

Pseudocode 3.1: The procedure of $dispatch()$ with a system architecture with m processors

```
1:  while newtasks > 0 do
2:     t^new ← first non-executing task in ready queue that needs to be scheduled (i.e.,
       among the first m entries)
3:     c ← find next free core {return NULL if no more free cores}
4:     if c == NULL then
5:        t_remove ← first executing task in ready queue not part of the first m entries ;
6:        c ← get the index of core executing task t_remove
7:     end if
8:     dispatch_to_proc(t^new, c)
9:     newtasks ← newtasks - 1
10: end while
```

In this code, the variable *newtasks* denotes the number of tasks that are not executing but need to be scheduled. Assuming a simulated architecture with *m* processors, *newtasks* therefore equals to the number of tasks that are among the first *m* tasks in the task ready queue that are not yet executing or being dispatched. Newly scheduled tasks

are dispatched to free processors if there are any available. If all processors are busy, then task preemption will take place.

The *dispatch()* method decides on the index of the selected cores for task dispatch. By calling *dispatch_to_proc(Task * newtask, CPU *c)*, the OS kernel also deschedules any task currently executing on processor c and computes the scheduling overhead including the context switch and task migration costs. The computed scheduling overhead is passed from the kernel layer to the architecture layer, which subsequently simulates this overhead. Hereafter, a newly dispatched task is selected to start execution on processor c. The procedure of *dispatch_to_proc(Task * newtask, CPU *c)* is shown in Pseudocode 3.2.

Pseudocode 3.2: The procedure of $dispatch\_to\_proc(Task * newtask, CPU * c)$

```
1:  AbsRTTask current_task ← the task currently executing on core c
2:  if current_task ≠ NULL then
3:      deschedule current_task
4:  end if
5:  if newtask == NULL then
6:      RETURN
7:  else
8:      prepare newtask to execute on core c
9:  end if
10: Compute the scheduling overhead
11: Send the overhead to architecture model
```

3.3.3 PartiKernel Model

The structure of a partition-based scheduler is shown Figure 3.4. In a partitioned scheduler, ready tasks are first inserted in a global ready queue. Through this global scheduler, ready tasks are then dispatched to a specific local task queue according to the task's affinity. Each processor has its own local queue in which the order depends on the local scheduler. Each processor may use a different scheduler.

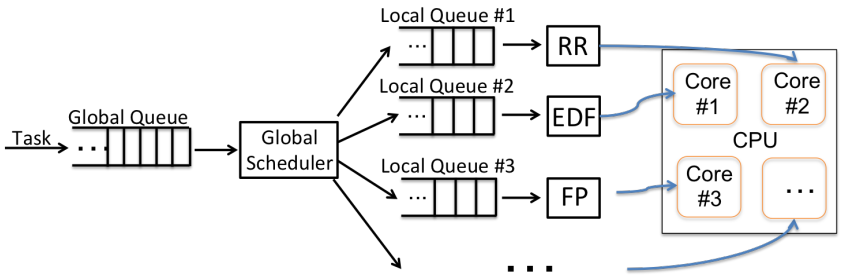


Figure 3.4: Structure of a partitioned-based scheduler.

Since the structure of such a partitioned scheduler is different from the global scheduler, a different kernel module, *PartiKernel*, has been implemented to facilitate the development of partitioned schedulers.

The interface provided to the *Task* module and functions related to process management in the *PartiKernel* module are slightly different than those in *SMPKernel* due to task affinity. However, the *dispatch()* method has been completely re-implemented. If a task is inserted to or is removed from a local queue, instead of calling *dispatch()*, *PartiKernel* invokes a *dispatch(CPU *cpu)* function that passes the task affinity as a parameter to make a local rescheduling decision for the processor in question. Changes on a local queue have no effect on the ordering of other local queues. In this sense, the *dispatch(CPU *cpu)* function is similar to *dispatch()* in *UNPKernel*.

3.3.4 Scheduler Model

When a task becomes ready to execute, it is inserted to the ready queue managed by the scheduler, which is a sub-component of a kernel module. The ready queue is ordered by task priority assigned by the scheduling algorithm. At a scheduling point, the scheduler (i.e. dispatcher) is responsible for selecting the task(s) at the front of the ready queue to execute. In SysRT, the following schedulers have currently been implemented:

- Global Earliest Deadline First [56] (G-EDF)
- First Come First Out (FIFO)
- Fixed Priority Scheduler (FPS)
- Rate Monotonic Scheduler (RMS)
- Round Robin (RR).
- Proportional Fairness [10] (P-FAIR)
- Partitioned-based Scheduler (PS) including P-EDF
- Non-Preemptive EDF (NP-EDF)

3.3.5 Resource Management Model

The *Resource* module models a resource shared by two or more tasks. It provides an interface to the OS kernel module to, for example, perform locking operations for providing access to these shared resources. The resource availability is checked by the method *IsAvailable(int amount)*. It returns false if the quantity of a certain resource is not sufficient. Every task uses resources through a critical section surrounded by *wait* and *signal* instructions. If the executing task requests/releases a certain resource quantity, the resource manager in the OS kernel invokes the interface of the resource, *lock(int amount)/unlock(int amount)*, to decrease/increase resource availability for that particular resource.

The *ResManager* module models a resource manager that implements the resource accessing protocol. It contains multiple block queues, each associated with a particular

resource to store tasks blocked on that resource. These block queues are ordered by task priority. Different resource sharing protocols can be implemented by the *ResManager* module.

Taking the Priority Inheritance Protocol [81] as an example, requesting a resource is implemented by first checking the availability of the requested resource. If there are not enough available resources, the resource manager calls the kernel interface to suspend the task that is requesting the resource. Furthermore, the priority of the resource owner is changed to the maximum priority of those tasks that are blocked for the resource. If the requested resources are available, the resource manager invokes the unlock interface of the resource and grants the resources to the task. Releasing a resource unlocks the resources and changes the priority of the releasing task back to its original priority, after which it checks if the resource block queue is empty. If the queue is not empty, the resource manager removes the first task from the block queue, and activates the task through the kernel interface and locks the resource for the new owner.

3.4 Experimental Results

In this section, we evaluate the accuracy and simulation performance of SysRT, and demonstrate its flexibility and benefit in DSE. All experiments were conducted on a 3.4GHZ Intel Core I5. The default time unit of the task parameters in the following experiments is the simulation resolution set by SystemC.

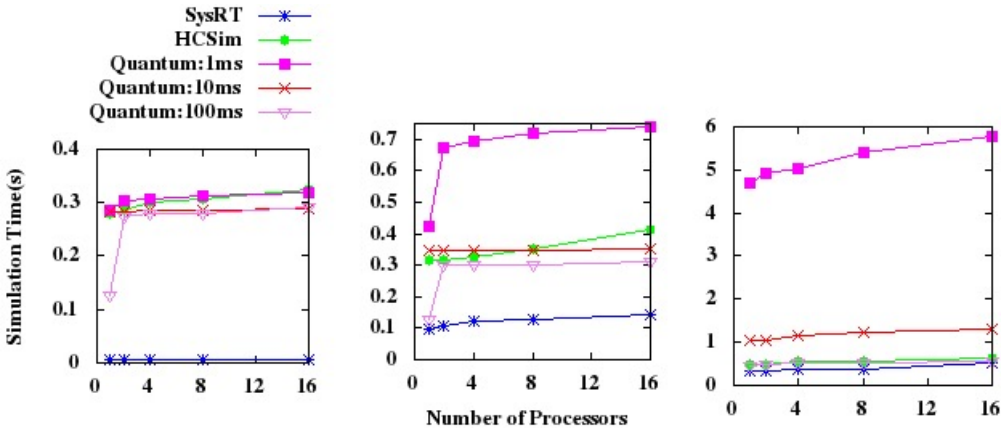
3.4.1 Simulation performance and accuracy

The first experiment is to evaluate the accuracy and simulation performance of SysRT by comparing it with four other simulators: the state-of-art (prediction-based) HCSim simulator [73] and three conventional quantum-granularity based simulators (also described in [73]) with a simulation quantum of 1ms, 10ms and 100ms, respectively. All simulators model a Partitioned-Fixed Priority scheduler, where tasks have been uniformly partitioned over the simulated processors. Task execution costs and periods, priorities are randomly distributed over the intervals [50ms, 150ms], [100ms,10s] and [1, 100], respectively. The simulated time is 10 minutes. Note that all these tasks are not necessarily real-time tasks.

Figures 3.5 (a), (b) and (c) show the simulation times taken by each simulator simulating a different number of processors, ranging from 1 to 16, where the number of tasks is 16, 100 and 1000. Figure 3.5 clearly shows that SysRT achieves the fastest simulation speed in these experiments. Both SysRT and HCSim are scalable with respect to the number of processors and the number of tasks.

The simulation speed of the conventional simulator with largest simulation quantum is similar to that of HCSim and SysRT. However, it suffers from a lower accuracy, as will be discussed later on. Conventional simulators get much slower if the simulation quantum size decreases.

To derive a reference for the task response times, we have also performed the experiment with the same task sets on a real Linux-based RTOS, i.e. Litmus [19], varying the number of active processors from 1 to 4. For each task, we calculate the



(a) 16 tasks (b) 100 tasks (c) 1000 tasks

Figure 3.5: Simulation time of five simulators.

relative errors between the response times obtained from simulators and the actual response times from Litmus. The accuracy is measured by the average error of all tasks in the testing task set.

Table 3.1 is the average simulation error of those tests. The number of active processors and the number of tasks in different testing sets is not reported since it turns out that these factors have little effect on the relative error of each individual task. SysRT, HCSim and conventional simulation with the smallest simulation quantum yield high accuracy, whereas conventional simulators with a larger simulation quantum suffer from degraded accuracy.

Table 3.1: Average Simulation Error of Five Simulators

HCSim	SysRT	Quantum:1ms	Quantum:10ms	Quantum:100ms
0.166%	0.166%	0.166%	4.182%	>100%

Note that, although SysRT and HCSim are supposed to be theoretically accurate, several factors in Litmus such as context switches and kernel tasks with high priorities could lead to small simulation errors. Fortunately, both SysRT and HCSim provide support to model the scheduling overhead to improve accuracy.

3.4.2 Flexibility of SysRT

As most prediction-based RTOS simulators do not support simulating real-time resource access protocols due to difficulties in predicting preemption points, we show the flexibility of SysRT by simulating a set of four periodic tasks T_1, ..., T_4 that exclusively access two shared resources R_1 and R_2. Task parameters are listed in Table 3.2. P_i is the task activation period and C_i the execution time. Variable $\xi_{j,i}$ denotes the duration of the

critical section that T_i occupies R_j. The value 0 for $\xi_{j,i}$ means that T_i does not use R_j. Tasks are scheduled on an uniprocessor by a RM scheduler with priority inheritance as resource sharing protocol.

Table 3.2: Task Parameters and Theoretical WCRT.

Tasks	P_i	C_i	$\xi_{1,i}$	$\xi_{2,i}$	$WCRT_i$
T_1	100	5	0	0	5
T_2	110	16	3	3	71
T_3	200	70	20	0	142
T_4	350	102	0	30	310

The analytically calculated Worst Case Response Time (WCRT) for each task is given in the last column of Table 3.2. We have run the simulation for 80000 time units. The simulated response time of the first 200 jobs of each task are shown in Figure 3.6. As can be seen from Figure 3.6, the response times obtained from simulation are consistently lower than the theoretical WCRTs. Thanks to the modular and flexible implementation of SysRT, the resource sharing protocol is correctly simulated.

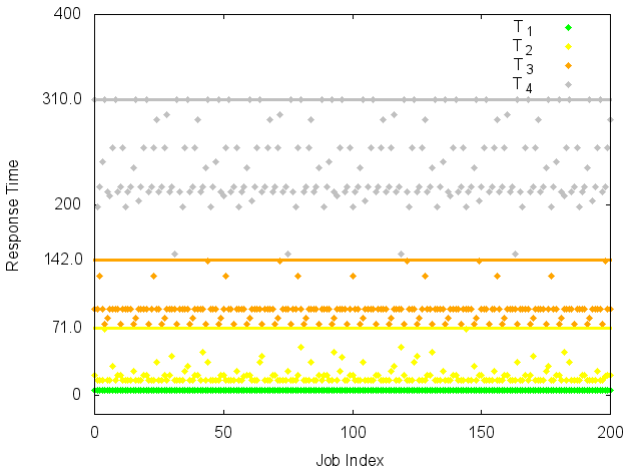


Figure 3.6: Response time of jobs in tasks.

3.4.3 Benefit of SysRT in DSE

The second experiment demonstrates the flexibility of SysRT and its benefits for early DSE. An embedded system with a mixed application workload is simulated. The task set is composed of three Hard Real-Time (HRT) tasks, five Soft Real-Time (SRT) tasks and three Best-Effort (BE) tasks. Task types, parameters and utilization (P_i divided by C_i) are listed in Table 3.3. If an interval [a, b] is assigned to P_i (or C_i), then P_i (or

C_i) is a random variable uniformly distributed in that interval. This models workload variations.

Table 3.3: Task Type and Parameters.

Tasks	Type	P_i	C_i	U_i
T_1	HRT	50	20	0.4
T_2	HRT	90	30	0.333
T_3	HRT	140	50	0.357
T_4	SRT	190	30	0.157
T_5	SRT	350	80	0.228
T_6	SRT	500	170	0.34
T_7	SRT	1000	[200, 700]	[0.2, 0.7]
T_8	SRT	1300	[500, 900]	[0.385, 0.692]
T_9	BE	[1000, 5000]	200	[0.04, 0.2]
T_{10}	BE	[3000, 9000]	500	[0.056, 0.167]
T_{11}	BE	[5000, 15000]	1500	[0.1, 0.3]

The application requirement for hard real-time tasks is to guarantee that deadlines are always met. SRT tasks are allowed to miss deadlines, thus their performance is measured by the deadline miss ratio. For best-effort tasks, the performance is calculated by their average response time. We have run simulations with three kinds of schedulers on different architecture models. EDF and FPS schedulers are tested with systems containing 2 to 8 processors, and a partitioned-based scheduler (PS) has been tested for systems with 3 to 5 cores. For the latter, Table 3.4 lists the local scheduling policies and scheduled task(s) on each processor. The simulation is aborted if a HRT task misses a deadline.

Table 3.4: Patitioned-based Scheduler Configuration.

# Processors	Processor	Local Scheduler	Tasks
3	1	FPS	T_1,T_2,T_9,T_{11}
	2	EDF	T_3,T_4,T_6
	3	RR	T_5,T_7,T_8,T_{10}
4	1	P-FAIR	T_1,T_2,
	2	FPS	T_3,T_4,T_6
	3	EDF	T_5,T_7,T_8
	4	RR	T_9,T_{10},T_{11}
5	1	P-FAIR	T_1,T_2,T_5
	2	FPS	T_3
	3	NP-EDF	T_4,T_7
	4	EDF	T_6,T_8
	5	RR	T_9,T_{10},T_{11}

The average deadline miss ratio of the five SRT tasks is shown in Figure 3.7(a). The deadline miss ratio decreases as the number of processors increases and becomes 0 for

five processors. HRT tasks are not schedulable under EDF if the number of processors is less than four, thus no results are plotted for EDF for 2 and 3 processors.

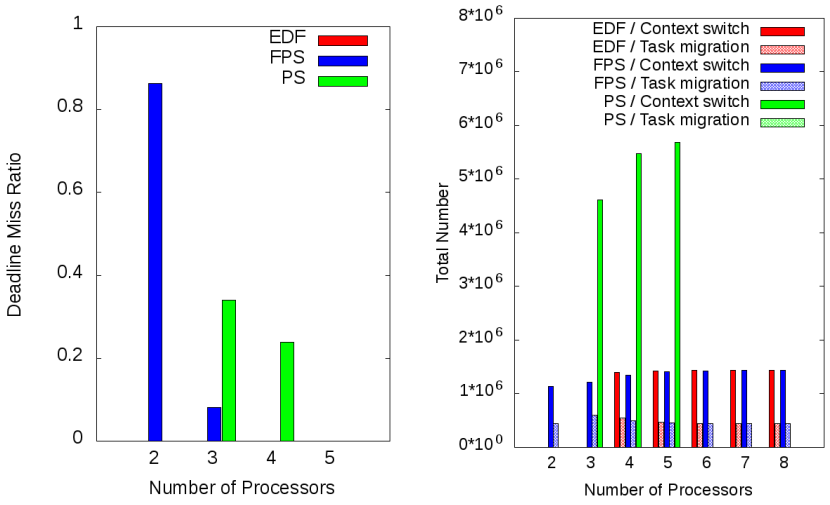


Figure 3.7: (a) Average deadline miss ratio (b) Scheduling overhead.

Figure 3.7(b) shows the scheduling overhead including the total number of context switches and task migrations. It is interesting to observe that partitioned schedulers have no task migration but suffer from a large number of context switches incurred by P-FAIR, which serves as a local scheduler.

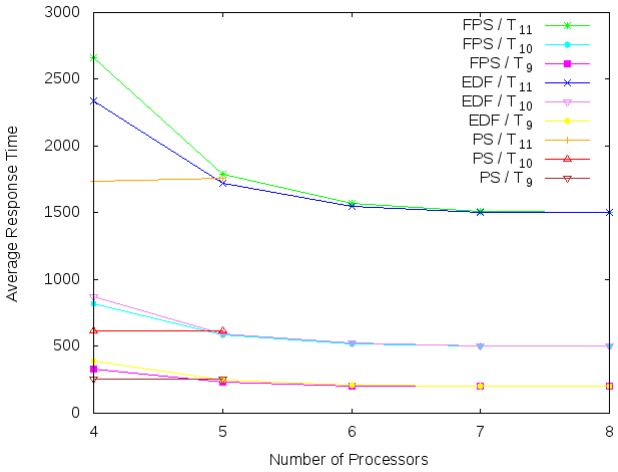


Figure 3.8: Response time of BE tasks.

Figure 3.8 illustrates the average response times of the BE tasks. As the number of processors increases, the average response time becomes smaller. The response times are very large if the number of processors is less than 4, thus they are not plotted. Evidently, such system performance estimates as obtained by SysRT are helpful to make design decisions at the very early system design stages.

3.5 Conclusion

In this chapter, we presented SysRT, a generic and high-level SystemC-based multiprocessor RTOS simulator. It provides the unique and novel combination of being highly accurate, efficient and easy to extend to facilitate early DSE. To this end, it contains different types of application models and a modular RTOS kernel model. Efficient and precise modeling of preemptive scheduling is achieved via an event-driven simulation approach. Its modular design allows for easy plug-in of new schedulers as well as new resource sharing protocols. Comparing SysRT with state-of-art simulators, it achieves faster simulation speeds with the same small simulation error. We demonstrated the flexibility of SysRT by experiments with a mixed workload executing on multiprocessor platforms with different numbers of cores.

4

Schedulability Analysis of Global Scheduling for Multicore Systems with Shared Caches

In the previous chapter, we studied the simulation and modeling of real time embedded system. In this chapter, we continue our research on the topic of improving timing predictability for multicore system by an analytic approach. This chapter addresses RQ2 listed in chapter 1, which is concerned with schedulability analysis of global scheduling for real time multicore systems with shared caches.

Multicore architectures are increasingly used in both the desktop and the embedded markets. Modern multicore processors incorporate shared resources between cores to improve performance and efficiency. Shared caches are among the most critical shared resources on multicore systems as they can efficiently bridge the performance gap between memory and processor speeds by backing up small private caches. However, this brings major difficulties in providing guarantees on real-time properties of embedded software due to the interaction and the resulting contention in a shared cache.

In a multicore processor with shared caches, a real-time task may suffer from two different kinds of cache interferences [49], which severely degrade the timing predictability of multicore systems. The first is called intra-core cache interference, which occurs within a core, when a task is preempted and its data is evicted from the cache by other real-time tasks. The second is inter-core cache interference, which happens when tasks executing on different cores access the shared cache simultaneously. Inter-core cache interference may cause several types of cache misses including capacity misses, conflict misses and so on [13].

It is challenging to design real-time applications executing on multicore platforms with shared caches, which cannot afford to miss deadlines and hence demand timing predictability. Any schedulability analysis requires knowledge about the Worst-Case Execution Time (WCET) of real-time tasks. With a multicore system, the WCETs are strongly dependent on the amount of inter-core interference on shared hardware resources such as main memory, shared caches and interconnects. In this chapter, we restrict our focus on the shared cache interferences.

A major obstacle is to predict the cache behavior to accurately obtain the WCET of

This chapter was published as [108] and [110] (submitted)

a real-time task considering inter-core cache interference since different cache behaviors (cache hit or miss) will result in different execution times of each instruction. In [91], it was even pointed out that "it will be extremely difficult, if not impossible, to develop analysis methods that can accurately capture the contention among multiple cores in a shared cache". In this chapter, we assume that a task's WCET itself does not account for shared cache interference but, instead, we determine this interference explicitly (as will be explained later on). [38] presents such an approach to derive a task's WCET without considering shared cache interference.

In this work, we consider non-preemptive task systems, which implies that intra-core cache interference is avoided since no preemption is possible during task execution. We therefore focus on inter-core cache interference and study the schedulability analysis problem for hard real-time tasks that exhibit shared cache interferences. Therefore, we address the following research questions listed in Chapter 1:

RQ2 Is it possible to derive an upper bound on shared cache interference between two tasks running simultaneously on a multi-core system? Given a real-time taskset globally scheduled by EDF or FP, how to obtain an upper bound on the shared cache interference exhibited by each task in the taskset? How to derive a schedulable condition for the globally scheduled taskset, accounting for the shared cache interference?

To answer this research question, we proposes a more general framework for the schedulability analysis of global scheduling, accounting for shared cache interference. The main contributions in this chapter are as follows:

- An integer programming formulation, which can be transformed to an integer linear programming formulation, is constructed to calculate an upper bound on cache interference exhibited by a task within a given execution window.
- An iterative algorithm is presented to obtain the upper bound on cache interference a task may exhibit during one job execution.
- A new schedulability condition is derived by integrating the upper bound on inter-core cache interference into the schedulability analysis.
- A range of experiments is performed to investigate how the schedulability is degraded by shared cache interference for a range of different tasksets. We also evaluated the schedulability performance of EDF and FP scheduling over randomly generated tasksets.

The rest of the chapter is organized as follows. Section 4.1 gives an overview of the related work. The system model is described in Section 4.2. Section 4.3 describes the proposed schedulability analysis, where we also detail the computation of processor-contention and inter-core cache interferences applied in the analysis. Section 4.4 presents an iterative computation to obtain the upper bound of inter-core cache interferences. Section 4.5 presents the experimental results, after which Section 4.6 concludes the chapter.

4.1 Related work

WCET estimation. For hard real-time systems, it is essential to obtain each real-time task's WCET, which provides the basis for the schedulability analysis. WCET analysis has been actively investigated in the last two decades, of which an excellent overview can be found in [100]. There are well-developed techniques to estimate real-time tasks' WCET for single processor systems. Unfortunately, the existing techniques for single processor platforms are not applicable to multicores with shared caches. Only a few methods have been developed to estimate task WCETs for multicore systems with shared caches [39, 52, 119]. In almost all those works, due to the assumption that cache interferences can occur at any program point, WCET analysis will be extremely pessimistic, especially when the system contains many cores and tasks. An overestimated WCET is not useful as it degrades system schedulability.

Shared cache interference. Since shared caches considerably complicate the task of accurately estimating the WCET, many researchers in the real-time systems community have recognized and studied the problem of cache interference in order to use shared caches in a predictable manner. Cache partitioning, which isolates application workloads that interfere with each other by assigning separate shared cache partitions to individual tasks, is a successful and widely-used approach to address contention for shared caches in (real-time) multicore applications. There are two cache partitioning methods: software-based and hardware-based techniques [33]. The most common software-based cache partitioning technique is page coloring [53, 62, 99]. By exploiting the virtual to physical page address translations present in virtual memory systems at OS-level, page addresses are mapped to pre-defined cache regions to avoid the overlap of cache spaces. [114] presented vCAT for dynamic shared cache management on multicore virtualization platforms based on Intel's Cache Allocation Technology. Hardware-based cache partitioning is achieved using a cache locking mechanism [62, 83, 91], which prevents cache lines from being evicted during program execution. The main drawback of cache locking is that it requires specific hardware support that is not available in many commercial processors. With shared cache partitioning techniques, one can apply existing analyses to derive the upper bounds of a task's WCET assuming that no cache interference can occur between tasks simultaneously running on different cores. In that case, it is safe to use the derived WCETs in the schedulability analysis.

Real-time Scheduling. The schedulability analysis of global multiprocessor scheduling has been intensively studied [8, 14, 22, 51, 57, 118], of which comprehensive surveys can be found in [26, 82]. Most multi-core scheduling approaches assume that the WCETs are estimated in an offline and isolated manner and that WCET values are fixed.

A few works address schedulability analysis for multi-core systems with shared caches [35, 113], but these works assume that cache space isolation is deployed. These solutions are not applicable to our problem since we consider systems in which cache isolation techniques are not deployed. An ongoing work in [37] describes that caches should be taken into account when performing task partitioning. They formulated a problem of finding a system partitioning such that the real-time constraints of all tasks are met while the sum of inter-core cache interference is minimized. [67] proposed a Predictable Execution Model (PREM), that co-schedules among CPU tasks executions and memory accesses, to reduce the low-level contention for shared resource such as

caches, memories, and buses.

Our work also differs from other approaches to the timing verification of multicore systems [4] in that all other sources of interferences are assumed to be included within the WCET. We analyze the effect of shared cache interference on the schedulability. To the best of our knowledge, this is among the first works that integrates inter-core cache interferences into schedulability analysis.

4.2 System Model

4.2.1 Task Model

We consider a set τ of n periodic or sporadic real-time tasks $\{\tau_1, \tau_2, ... \tau_n\}$ to be scheduled on a multicore processor. Each task $\tau_k = (C_k, D_k, T_k) \in \tau$ is characterized by a worst-case computation time C_k, a period or minimum inter-arrival time T_k, and a relative deadline D_k. All tasks are considered to be deadline constrained, i.e. the task relative deadline is less or equal to the task period: $D_k \leq T_k$.

We further assume that all those tasks are independent, i.e. they have no shared variables, no precedence constraints, and so on. Moreover, jobs of any task cannot be executed at the same time on more than one core. A task τ_k is a sequence of jobs J_k^j, where j is the job index. We denote the arrival time, starting time, finishing time and absolute deadline of a job j as r_k^j, s_k^j, f_k^j and d_k^j, respectively. Note that the goal of a real-time scheduling algorithm is to guarantee that each job will complete before its absolute deadline: $f_k^j \leq d_k^j = r_k^j + D_k$.

As explained, it is difficult to accurately estimate C_k considering cache interference of other tasks executing concurrently. It should be pointed out that C_k in this chapter refers to the WCET of task k, assuming task k is the only task executing on the multicore processor platform, i.e. any cache interference delays are not included in C_k.

Since time measurement cannot be more precise than one tick of the system clock, all timing parameters and variables in this chapter are assumed to be non-negative integer values.

4.2.2 Architecture Model

Our system architecture consists of a multicore processor with m identical cores onto which the individual tasks are scheduled. Most multicore processors have instruction and data caches. Caches are organized as a hierarchy of multiple cache levels to address the tradeoff between cache latency and hit rate. The low level caches ($L1$) in our considered multicore processor are assumed to be private, while the last level caches (*LLC*) are shared between all cores. Furthermore, we assume that the *LLC* cache is noninclusive with respect to the private caches ($L1$), and that *LLC* caches are direct-mapped caches.

In this work, we only consider instruction caches since we adopt the approach in [38], which only accounts for instruction caches, to derive WCET.

4.2.3 Global Schedulers

In this chapter, we focus on non-preemptive global scheduling. Once a task instance starts execution, any preemption during the execution is not allowed, so it must run to completion. So we do not have to consider intra-core cache interference. If not explicitly stated, cache interference will therefore refer to inter-core cache interference in the following discussion. We consider two well-known global scheduling algorithms: Non-Preemptive Earliest Deadline First (EDF_{np}) and Non-Preemptive Fixed Priority (FP_{np}).

EDF_{np} assigns a priority to a job according to the absolute deadline of that job. A job with an earlier absolute deadline has higher priority than others with a later absolute deadline. Since each job's absolute deadline changes over time, the priority of a task changes dynamically.

For FP_{np} scheduling, a fixed priority P_k is assigned to each task τ_k ($k = 1, 2, ...n$). As each task has a unique priority, we use $hp(k)$ to denote the set of tasks with higher priorities than τ_k, and $hep(k) = hp(k) \cup \{\tau_k\}$ the set of tasks whose priorities are not lower than τ_k. Similarly, $lp(k)$ is the set of tasks with lower priorities than τ_k and $lep(k) = lp(k) \cup \{\tau_k\}$ the set of tasks whose priorities are not higher than τ_k.

The EDF_{np} and FP_{np} scheduling algorithms are work-conserving, according to the following definition.

Definition 4.1. A scheduling algorithm is *work-conserving* if there are no idle cores when a ready task is waiting for execution.

4.3 Schedulability Analysis

In this section, we give an overview of the new schedulability analysis that accounts for cache interference. We also present the approaches to derive the upper bound on the parameters used in the schedulability condition.

4.3.1 Overview

We first analyze the execution of one job J_k^j of a task τ_k. Let o_k^j denote the latest time-instant no later than r_k^j ($o_k^j \leq r_k^j$) at which at least one processor is idle and let $A_k = r_k^j - o_k^j$. As all processors are idle when the system starts, there always exists such a o_k^j. The time interval $[o_k^j, d_k^j]$ is named a problem window. This problem window can be divided into two parts $[o_k^j, s_k^j]$ and $[s_k^j, d_k^j]$.

As shown in Figure 4.1, a job J_k^j of task τ_k exhibits two kinds of interferences during the problem window. The first interference is called processor-contention interference, denoted by I_k^{pre}. It is the cumulative length of all intervals over $[o_k^j, s_k^j]$ in which all the processing cores are busy executing jobs other than J_k^j. We define the interference $I_{i,k}^{pre}$ of a task τ_i on a task τ_k over the interval $[o_k^j, s_k^j)$ as the cumulative length of all intervals in which τ_i is executing. The second type of interference is the cumulative length of all extra execution delays caused by shared cache interference from all other tasks running concurrently on other cores, denoted as I_k^{sc}. We also define the interference $I_{i,k}^{sc}$ as the

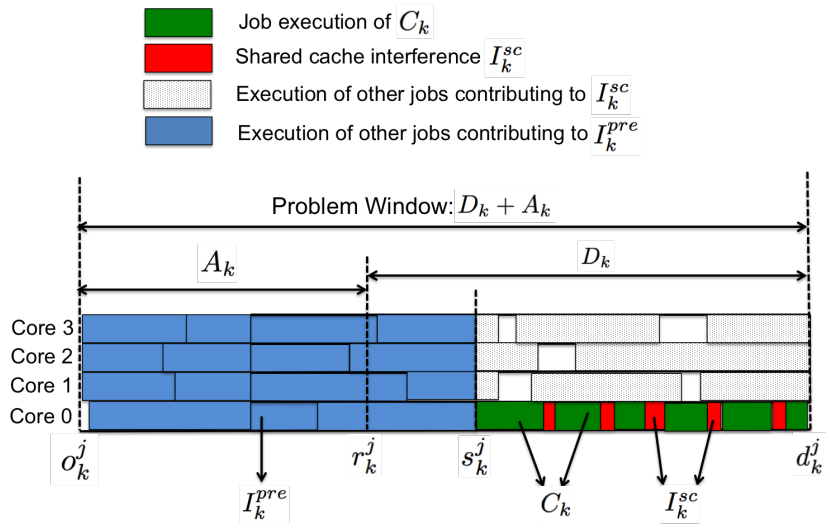


Figure 4.1: Overview of the schedulability analysis that accounts for cache interference.

cumulative length of all extra execution delays of τ_k caused by shared cache accesses between task τ_i and task τ_k.

Furthermore, we define the upper bound on processor-contention interference as $\bar{I}_k^{pre}$ and similarly the upper bound on shared cache interference as $\bar{I}_k^{sc}$.

Note that the processor-contention interference I_k^{pre} occurs during $[o_k^j, s_k^j]$, so I_k^{pre} depends on A_k and the length of $[r_k^j, s_k^j]$. While the shared cached interference I_k^{sc} occurs only during τ_k's execution. We will present the derivation of $\bar{I}_k^{sc}$ in the next section and it can be shown that $\bar{I}_k^{sc}$ does not depend on A_k and the length of $[r_k^j, s_k^j]$. Let us now assume $\bar{I}_k^{sc}$ is known.

We can compute the latest start time of job J_k^j from task τ_k: $l_k^j = d_k^j - C_k - \bar{I}_k^{sc}$ i.e., if J_k^j starts its execution before l_k^j, it must be able to finish execution before deadline d_k^j. The length of $[r_k^j, l_k^j]$ is $S_k = D_k - C_k - \bar{I}_k^{sc}$. Obviously, if $S_k \leq 0$, J_k^j will miss its deadline. We assume $S_k > 0$ in the following description.

As the processor-contention interference only occurs before the start of the τ_k's execution, we restrict I_k^{pre}, $I_{i,k}^{pre}$ and $\bar{I}_k^{pre}$ to the time interval $[o_k^j, l_k^j]$.

By construction, we have the first schedulability test for τ.

Theorem 4.1. *A task set τ is schedulable with a EDF_{np} or FP_{np} scheduling policy on a multicore processor composed of m identical cores with shared caches if for each task $\tau_k \in \tau$ and all $A_k \geq 0$:*

$$\bar{I}_k^{pre} + C_k + \bar{I}_k^{sc} < D_k + A_k.$$

4.3.2 Computation of $\bar{I}_k^{pre}$

The workload $W_{i,k}$ of a task τ_i is the time task τ_i executes during time interval $[o_k^j, l_k^j)$ of length $A_k + S_k$, according to a given scheduling policy.

Lemma 4.2. *The processor-contention interference that a task τ_i causes on a task τ_k in $[o_k^j, l_k^j)$ is never greater than the workload of τ_i in $[o_k^j, l_k^j)$,*

$$\forall i, k, j \quad I_{i,k}^{pre} \leq W_{i,k}.$$

Lemma 4.2 is obvious, since $W_{i,k}$ is an upper bound on the execution of τ_i in $[o_k^j, l_k^j)$.

Note that τ_i may execute more than C_i due to the shared cache interference. That is, the actual execution time of τ_i's job is bounded by $C_i^* = C_i + \bar{I}_i^{sc}$. In the following discussion, we use C_i^* as the upper bound on the workload contribution from a single job of τ_i.

As the number of τ_i's jobs released in $[o_k^j, l_k^j)$ is at most $\left\lceil \frac{A_k+S_k}{T_i} \right\rceil$, $W_{i,k}$ can be roughly bounded by $\left\lceil \frac{A_k+S_k}{T_i} \right\rceil \times C_i^*$. However, a tighter upper bound on the worst-case workload can be calculated by categorizing each job of τ_i in $[o_k^j, l_k^j]$ into one of the three types [6]:

carry-in job: a job with its release time earlier than o_k^j but with its deadline earlier than l_k^j;

body job: a job with both its release time and its deadline in $[o_k^j, l_k^j]$;

carry-out job: a job with its release time in $[o_k^j, l_k^j]$, but with its deadline later than l_k^j.

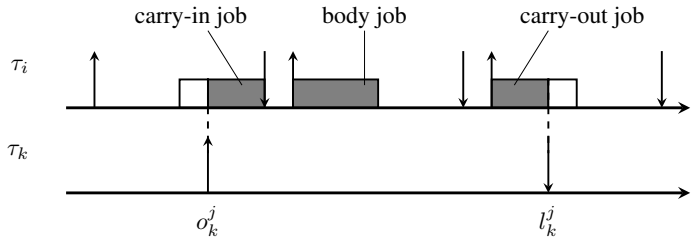


Figure 4.2: Three types of contribution jobs and problem window.

As shown in Figure 4.2, the worst-case workload of τ_i occurs when a carry-in job (if τ_i has a carry-in job) finishes execution as late as possible and a carry-out job starts its execution as early as possible. We use $W_{i,k}^n$ to denote an upper bound of τ_i's workload in $[o_k^j, l_k^j]$ if τ_i has no carry-in job, and use $W_{i,k}^c$ to denote an upper bound of τ_i's workload if τ_i has a carry-in job.

Following the approach in [34], we derive a tighter upper bound on W_i^n and W_i^c for the EDF_{np} and FP_{np} scheduling policies, separately. We omit the proof due to space limitations. Interested readers can refer to [34] for a detailed explanation.

Upper bound on $W_{i,k}^n$ for EDF_{np}.

EDF_{np} assigns a priority to a job by the absolute deadline of that job. We have the following lemma.

Lemma 4.3. *For EDF_{np}, if $D_i > D_k$, the necessary condition for J_i^j to cause interference to J_k^j is $r_i^j < r_k^j$, i.e.,J_i^j must be released earlier than J_k^j; if $D_i \leq D_k$, the necessary condition for J_i^j to cause interference to J_k^j is $d_i \leq d_k$, i.e., J_i's absolute deadline must be no later than that of J_k.*

Since τ_i has no carry-in jobs in this case, the worst case of $W_{i,k}^n$ occurs when the first job of τ_i is released at time o_k^j. The next jobs of τ_i are then released periodically every T_i time units.Thus, $W_{i,k}^n$ is computed by three cases: (1) $i = k$, (2) $D_i \leq D_k$, (3) $D_i > D_k$.

(1) $i = k$. As shown in Figure 4.3, only body jobs in $[o_k^j, r_k^j]$ contribute to processor-contention interference and the number of τ_i's body instances is $\left\lfloor \frac{A_k}{T_k} \right\rfloor$. So we have

$$W_{i,k}^{n_1} = \left\lfloor \frac{A_k}{T_k} \right\rfloor C_k^* \tag{4.1}$$

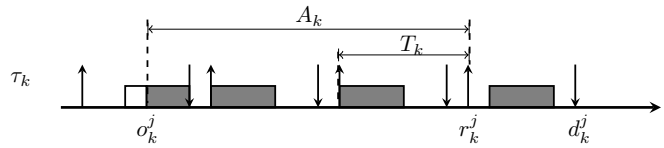


Figure 4.3: The densest possible packing of jobs of τ_i without carry-in job, if $i = k$.

(2) $D_i \leq D_k$. Figure 4.4 shows the worst case of $W_{i,k}^n$ for $D_i \leq D_k$. The number of body jobs of τ_i is $\left\lfloor \frac{A_k+S_k}{T_i} \right\rfloor$. We use α to denote the distance between o_k^j and the deadline of τ_i's carry-out job, $\alpha = \left\lfloor \frac{A_k+S_k}{T_i} \right\rfloor T_i + D_i$. The deadline of τ_i's carry-out job is $o_k^j + \alpha$.

(2.A) If $\alpha \leq A_k + D_k$, as shown in case (a) in Figure 5.1, the contribution of the carry-out job is bounded by $\min(C_i^*, (A_k + S_k) \mod T_i)$. In this case, we have:

$$W_{i,k}^{n_2} = \left\lfloor \frac{A_k + S_k}{T_i} \right\rfloor C_i^* + \min(C_i^*, (A_k + S_k) \mod T_i) \tag{4.2}$$

(2.B) If $\alpha > A_k + D_k$, shown as case (b) in Figure 5.1, the contribution of the carry-out job is 0, we have

$$W_{i,k}^{n_3} = \left\lfloor \frac{A_k + S_k}{T_i} \right\rfloor C_i^* \tag{4.3}$$

(3) $D_i > D_k$. Figure 4.5 shows the worst case of $W_{i,k}^n$ for $D_i > D_k$. The number of body jobs of τ_i is $\left\lfloor \frac{A_k+S_k}{T_i} \right\rfloor$. By Lemma 4.3, a job of τ_i can interfere with J_k^j only if

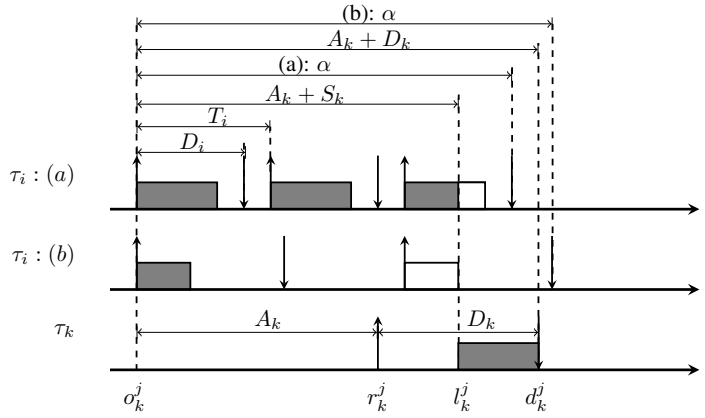


Figure 4.4: The densest possible packing of jobs of τ_i without carry-in job and $D_i \leq D_k$. Case (a): $\alpha \leq A_k + D_k$, Case (b): $\alpha > A_k + D_k$.

its release time is earlier than r_k^j. We use β to denote the distance between o_k^j and the release time of τ_i's carry-out job, $\beta = \left\lfloor \frac{A_k + S_k}{T_i} \right\rfloor T_i$.

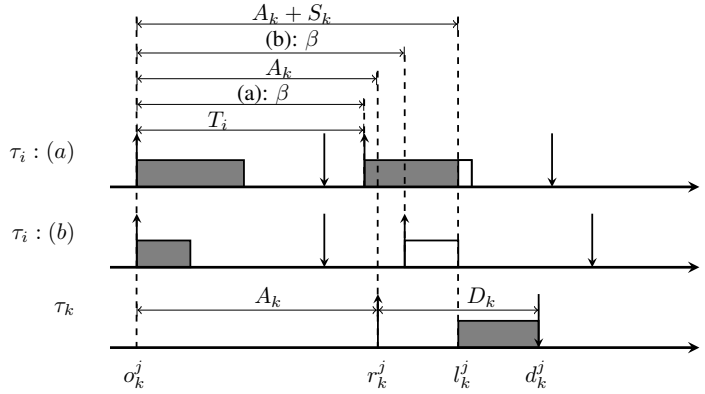


Figure 4.5: The densest possible packing of jobs of τ_i without carry-in job and $D_i > D_k$. Case (a): $\beta < A_k$, Case (b): $\beta \geq A_k$.

(3.A) If $A_k = 0$, then $o_k^j = r_k^j$. Since $D_i > D_k$, any task instance released no earlier than o_k^j has a deadline later than d_k^j, so, $W_{i,k}^n = 0$.

(3.B) If $\beta < A_k$, shown as case (a) in Figure 4.5. The contribution of τ_i's carry-out job is bounded by $\min(C_i^*, (A_k + S_k) \mod T_i)$. $W_{i,k}^n$ is computed by Equation (4.2).

(3.C) If $\beta \geq A_k > 0$, as shown in Figure 4.5 case (b), the contribution of τ_i's carry-out job is 0, and $W_{i,k}^n$ is computed by Equation (4.3).

By the discussions above, we can compute $W^n_{i,k}$ for EDF_{np} by:

$$W^n_{i,k} = \begin{cases} 0 & D_i > D_k \wedge A_k = 0 \\ W^{n_1}_{i,k} & i = k \\ W^{n_2}_{i,k} & (i \neq k \wedge D_i \leq D_k \wedge \alpha \leq A_k + D_k) \\ & \vee (D_i > D_k \wedge \beta < A_k) \\ W^{n_3}_{i,k} & otherwise \end{cases} \tag{4.4}$$

where $W^{n_1}_{i,k}$, $W^{n_2}_{i,k}$, $W^{n_3}_{i,k}$ are defined in Equations (4.1), (4.2) and (4.3) respectively.

Upper bound on $W^c_{i,k}$ for EDF_{np}.

We now compute the upper bound on $W^c_{i,k}$ by four cases: (1) $i = k$, (2) $D_i \leq D_k$ and $S_i > C^*_k$ (3) $D_i > D_k$ and $S_i \geq C^*_k$ (4) the remaining cases.

(1) $i = k$. The number of body jobs of τ_k is $\left\lfloor \frac{A_k}{T_k} \right\rfloor$. The contribution of the carry-in job is bounded by $\min(C^*_k, \max(0, (A_k \mod T_k) - T_k + D_k))$. So in this case, we have:

$$W^{c_1}_{i,k} = \left\lfloor \frac{A_k}{T_k} \right\rfloor C^*_k + \min(C^*_k, \max(0, (A_k \mod T_k) - T_k + D_k)) \tag{4.5}$$

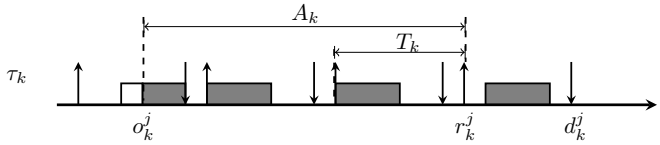


Figure 4.6: The densest possible packing of jobs of τ_i with carry-in job, if $i = k$.

(2) $D_i \leq D_k \wedge S_i > C^*_k$. Shown as case (a) in Figure 4.7, the worst case of $W^c_{i,k}$ occurs when τ_i's last released instance has its deadline at d^j_k. The number of τ_i's body jobs is $\left\lfloor \frac{A_k + D_k}{T_i} \right\rfloor$. The contribution of the carry-in job is bounded by $\min(C^*_i, (A_k + D_k) \mod T_i)$. So, we have:

$$W^{c_2}_{i,k} = \left\lfloor \frac{A_k + D_k}{T_i} \right\rfloor C^*_i + \min(C^*_i, (A_k + D_k) \mod T_i) \tag{4.6}$$

(3) $D_i > D_k \wedge S_i \geq C^*_k$. Case (b) in Figure 4.7 shows the worst case of $W^c_{i,k}$. By Lemma 4.3, τ_i's job can interfere with J^j_k only if its release time is earlier than r^j_k. So, the worst case of $W^c_{i,k}$ occurs when one of τ_i's instances is released at $r^j_k - 1$.

(3.A) If $A_k > 0$, the number of τ_i's body instances is $\left\lfloor \frac{A_k - 1}{T_i} \right\rfloor$, the carry-out is C^*_i, the carry-in is bounded by $\mu = \min(C^*_i, \max(0, (A_k - 1) \mod T_i - (T_i - D_i)))$.

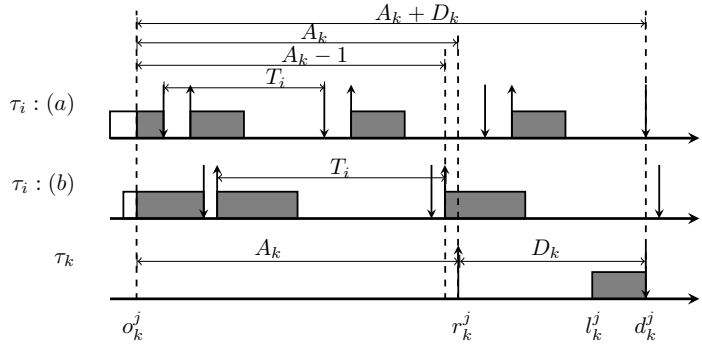


Figure 4.7: The densest possible packing of jobs of τ_i with carry-in job. Case (a): $D_i \leq D_k \wedge S_i > C_k^*$, Case (b):$D_i > D_k \wedge S_i \geq C_k^*$.

(3.B) If $A_k = 0$, only the carry-out job contributes at most $C_i^* - 1$. So, we have

$$W_{i,k}^{c_3} = \begin{cases} C_i^* - 1 & A_k = 0 \\ (\left\lfloor \frac{A_k - 1}{T_i} \right\rfloor + 1)C_i^* + \mu & A_k > 0 \end{cases} \tag{4.7}$$

(4) For the remaining cases, i.e. $(D_i \leq D_k \wedge S_i \leq C_k^*) \vee (D_i > D_k \wedge S_i < C_k^*)$, the worst case of $W_{i,k}^c$ occurs when one of τ_i's instances is released at $l_k^j - C_i^*$, as shown in Figure 4.8.

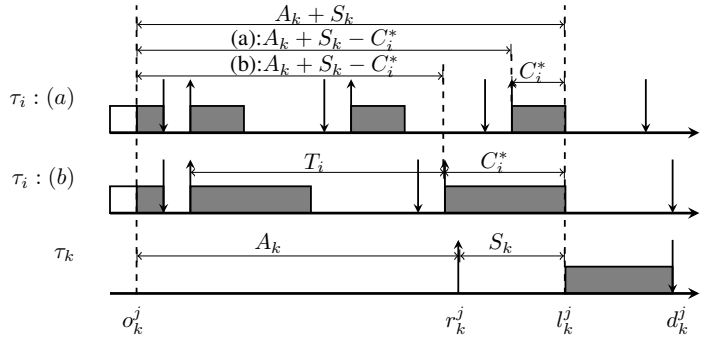


Figure 4.8: The densest possible packing of jobs of τ_i with carry-in job. Case (a): $D_i \leq D_k \wedge S_i \leq C_k^*$, case (b):$D_i > D_k \wedge S_i < C_k^*$.

(4.A) If $A_k + S_k \leq C_i^*$, then $W_{i,k}^c = A_k + S_k$.

(4.B) If $A_k + S_k > C_i^*$, the number of τ_i's body job is $\left\lfloor \frac{A_k + S_k - C_i^*}{T_i} \right\rfloor$, the contribution of the carry-out job is C_i^*; carry-in is bounded by $\nu = \min(C_i^*, \max(0, (A_k + S_k - C_i^*) \mod T_i - (T_i - D_i)))$.

$$W_{i,k}^{c_4} = \begin{cases} A_k + S_k & A_k + S_k \leq C_i^* \\ (\left\lfloor \frac{A_k + S_k - C_i^*}{T_i} \right\rfloor + 1)C_i^* + \nu & A_k + S_k > C_i^* \end{cases} \tag{4.8}$$

By the discussion above, we can compute $W^c_{i,k}$ for EDF_{np} by:

$$W^c_{i,k} = \begin{cases} W^{c_1}_{i,k} & i = k \\ W^{c_2}_{i,k} & i \neq k \wedge D_i \leq D_k \wedge S_i > C^*_k \\ W^{c_3}_{i,k} & D_i > D_k \wedge S_k \geq C^*_i \\ W^{c_4}_{i,k} & otherwise \end{cases} \tag{4.9}$$

where $W^{c_1}_{i,k}$, $W^{c_2}_{i,k}$, $W^{c_3}_{i,k}$ and $W^{c_4}_{i,k}$ are defined in Equation (4.5), (4.6), (4.7) and (4.8) respectively.

Upper bound on $W^n_{i,k}$ for FP_{np}.

The following lemma describes the condition of processor-contention interference on τ_k caused by lower-priority tasks in $lp(k)$ for FP_{np}.

Lemma 4.4. *For FP_{np}, a task instance J^j_i of $\tau_i \in lp(k)$ can interfere with J^j_k only if J^j_i is released before r^j_k.*

We compute the upper bound on $W^n_{i,j}$ by three cases: (1)$i = k$, (2)$\tau_i \in hp(k)$, (3)$\tau_i \in lp(k)$.

(1) $i = k$. The worst case workload is the same as in the case of EDF_{np}, thus $W^n_{i,j}$ can be computed by Equation (4.1).

(2) $\tau_i \in hp(k)$. The worst-case workload of task τ_i occurs when a job of τ_i arrives at o^j_k, as shown in case (a) in Figure 5.1. $W^n_{i,j}$ can be computed using Equation (4.2).

(3) $\tau_i \in lp(k)$. The worse case of $W^n_{i,k}$ occurs when one of τ_i's instances is released at o^j_k. The number of body jobs of τ_i is $\left\lfloor \frac{A_k+S_k}{T_i} \right\rfloor$. Let γ be the distance between o^j_k and the release time of τ_i's last instance. So $\gamma = \left\lfloor \frac{A_k+S_k}{T_i} \right\rfloor$.

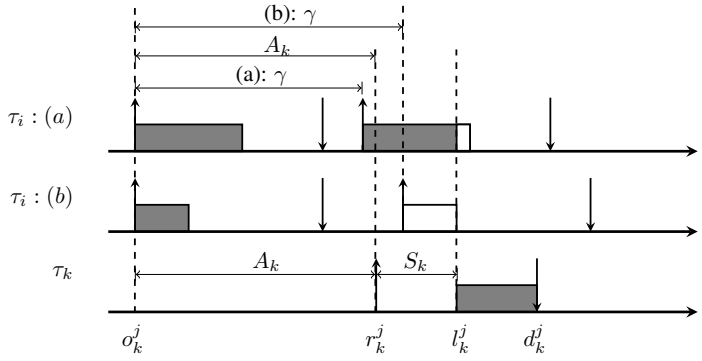


Figure 4.9: The densest possible packing of jobs of τ_i with carry-in job. Case (a): $\gamma < A_k$, Case (b):$\gamma \geq A_k > 0$.

(3.A) If $A_k = 0$, then $o^j_k = r^j_k$, according to Lemma 4.4, $W^n_{i,k} = 0$.

(3.B) If $\gamma < A_k$, τ_i's last job is released earlier than r_k^j, as shown in Figure 4.9 case (a), its contribution is bounded by $\min(A_k + S_k \mod T_i, C_i^*)$. In this case, $W_{i,k}^n$ is computed by Equation (4.2).

(3.C) If $\gamma \geq A_k > 0$, as shown case (b) in Figure 4.9, the contribution of the last released job of τ_i is 0. In this case, $W_{i,k}^n$ can be computed by Equation (4.3).

By the above discussion, we can compute $W_{i,k}^n$ by:

$$W_{i,k}^n = \begin{cases} 0 & \tau_i \in lp(k) \wedge A_k = 0 \\ W_{i,k}^{n_1} & i = k \\ W_{i,k}^{n_2} & \tau_i \in hp(k) \vee (\tau_i \in lp(k) \wedge \gamma < A_k) \\ W_{i,k}^{n_3} & otherwise \end{cases} \tag{4.10}$$

where $W_{i,k}^{n_1}$, $W_{i,k}^{n_2}$, $W_{i,k}^{n_3}$ are defined in Equations (4.1), (4.2) and (4.3) respectively.

Upper bound on $W_{i,k}^c$ for FP_{np}.

We compute the upper bound on $W_{i,k}^c$ by three cases: (1)$i = k$, (2) $\tau_i \in lp(k) \wedge S_k \geq C_i^*$, (3) the remaining cases.

(1) $i = k$. The worst case of $W_{i,k}^c$ occurs as it does for EDF_{np}, and therefore $W_{i,k}^c$ is computed by Equation (4.5).

(2) $\tau_i \in lp(k) \wedge S_k \geq C_i^*$. The worst case of $W_{i,j}^c$ occurs when one of τ_i's job is released at $r_k^j - 1$, as shown case (b) in Figure 4.7. We can compute $W_{i,j}^c$ by Equation (4.7).

(3) The remaining cases, i.e. $\tau_i \in hp(k)$ or $\tau_i \in lp(k) \wedge C_i^* > S_k$. The worst-case workload of τ_i is generated when one of τ_i's instances is released at time instance $s_k^j - C_i^*$. Such a situation is depicted in Figure 4.8. In this case, we can compute $W_{i,j}^c$ by Equation (4.8).

By the above discussion, we compute $W_{i,j}^c$ by:

$$W_{i,k}^c = \begin{cases} W_{i,k}^{c_1} & i = k \\ W_{i,k}^{c_3} & \tau_i \in lp(k) \wedge S_k \geq C_i^* \\ W_{i,k}^{c_4} & otherwise \end{cases} \tag{4.11}$$

where $W_{i,k}^{c_1}$, $W_{i,k}^{c_3}$ and $W_{i,k}^{c_4}$ are defined in Equation (4.5), (4.7) and (4.8) respectively.

Upper bound on I_k^{pre}.

By the definition of o_k^j, at least one core is idle at o_k^j, therefore at most $m - 1$ tasks have carry-in jobs. The task set τ can be partitioned into two subsets τ^c and τ^n that include tasks with carry-in jobs and tasks without carry-in jobs, respectively. Now we define Ω_k as the maximal value of the sum of all tasks' workloads in $[o_k^j, l_k^j]$ among all possible cases:

$$\begin{aligned} \Omega_k &= \max \sum_{\tau_i \in \tau} W_{i,k} \\ &= \max_{(\tau^n, \tau^c) \in \tau} \left(\sum_{\tau_i \in \tau^n} W_{i,k}^n + \sum_{\tau_i \in \tau^c} W_{i,k}^c \right) \end{aligned} \tag{4.12}$$

where τ^n and τ^c satisfy $\tau^n \cup \tau^c = \tau$, $\tau^n \cap \tau^c = \varnothing$ and $|\tau^c| \leq m-1$.

By taking the maximum over the task set, Ω_k describes an upper bound on the total worst-case workload in $[o_k^j, l_k^j]$. The complexity to compute Ω_k is $\mathcal{O}(n)$, as explained in [8].

Since both EDF_{np} and FP_{np} are work-conserving, the processor-contention interference exhibited by τ_k can be bounded by $\frac{\Omega_k}{m}$. So, we have the following Lemma.

Lemma 4.5. *If tasks are scheduled with a EDF_{np} or FP_{np} scheduling policy on a multicore processor composed of m identical cores with shared cache,*

$$I_k^{pre} \leq \frac{\Omega_k}{m}.$$

4.3.3 Computation of $\bar{I}_k^{sc}$

We first identify the maximum cache interference between two tasks and then we construct an integer programming formulation to calculate the upper bound on the shared cache interference exhibited by a task within an execution window.

Cache interference between two tasks

We first analyze the cache interference during one job execution between τ_k and τ_i. Let τ_k be the interfered and τ_i be the interfering task.

Following the approach in [38], we can obtain the WCET of a task by performing a Cache Access Classification (*CAC*) and Cache Hit/Miss Classification (*CHMC*) analysis for each instruction memory access at the private caches and the shared LLC cache separately. The *CAC* determines the possibility that an instruction being fetched from memory will access a certain cache level, and the access to a certain cache level can be *Always* (A), *Uncertain* (U) or *Never* (N). *CHMC* assigns a cache lookup result to each memory reference according to the cache states. As a result, a reference to a memory block of instructions can be classified as *Always Hit* (AH), *Always Miss* (AM) or *Uncertain* (U).

Since we consider noninclusive caches, accesses to the private caches cannot be affected by tasks executing on other cores. Accesses classified as AM or U at the shared LLC cache will also not be affected by shared cache interferences, since they are already counted as misses in the WCET analysis.

We start the cache interference analysis by defining two concepts for cache blocks.

Definition 4.2. A *Hit Block* (HB) is a memory block whose access is classified as AH at the shared LLC cache.

Definition 4.3. A *Conflicting Block* (CB) is a memory block whose access is classified as A or U at the shared LLC cache.

HB and CB can be identified by the approach proposed in [38].

We use $HB_k = \{m_{k,1}, m_{k,2}, ..., m_{k,p}\}$ to represent the set of HB for task τ_k and use $n_{k,x}$ $(x = 1, 2, ..., p)$ to denote the number of $m_{k,x}$'s accesses that are classified as a AH at the LLC cache. Similarly, we define $CB_i = \{m_{i,1}, m_{i,2}, ..., m_{i,q}\}$ as the set

of CB for task τ_i and denote $n_{i,x}$ as the number of $m_{i,x}$'s accesses that are classified as a A or U at the LLC cache. Note that HB_k and CB_i includes the memory blocks that meet the requirement in every program path that may be taken by the task.

In our system architecture, cache interference occurs only at the shared LLC cache when a cache line used by τ_k is evicted by τ_i and consequently causing reload overhead for τ_k. A cache line that may cause cache interference for τ_k needs to satisfy at least two conditions:

(i) access to that cache line will result in a cache hit at the LLC cache in WCET analysis of τ_k,

(ii) the cache line may be used by τ_i.

From the above two conditions, we can analyze memory block accessing that may cause interference. The first condition implies that only accessing to HB_k may cause cache interference for τ_k, while the second condition indicates that accessing to CB_i by τ_i may interfere with τ_k. Furthermore, cache interference occurs only if τ_k accesses memory blocks in HB_k and τ_i accesses memory blocks in CB_i concurrently, and those memory blocks have the same cache index.

We use $I_{i,k}^{sc}$ to represent the upper bound on the shared cache interference imposed on τ_k by only one job execution of τ_i.

Suppose the indexes of the LLC cache range from 0 to $N-1$, we can derive N subsets of HB_k according to the mapping function idx that maps a memory address to the cache line index at the LLC cache as follows,

$$\hat{m}_{k,u} = \{m_{k,x} \in HB_k | idx(m_{k,x}) = u\}, (0 \leq u < N, u \in \mathbb{N}).$$

We define the characteristic function of a set A which indicates membership of an element x in A as:

$$\chi_A(x) = \begin{cases} 1 & x \in A \\ 0 & otherwise \end{cases}.$$

Let $N_{k,u}$ represent the number of hit accesses to the u-th cache line by τ_k without cache interference. $N_{k,u}$ equals to the total number of access to the HBs mapping to the k-th cache line,

$$N_{k,u} = \sum_{x=1}^{p} n_{k,x} \chi_{\hat{m}_{k,u}}(m_{k,x}).$$

Similarly, we divide CB_i into N subsets by

$$\hat{e}_{i,u} = \{m_{i,x} \in CB_i | idx(m_{i,x}) = u\}, (0 \leq u < N, u \in \mathbb{N}).$$

The number of accesses to the k-th cache line by τ_i is bounded by

$$N_{i,u} = \sum_{x=1}^{q} n_{i,x} \chi_{\hat{e}_{i,u}}(m_{i,x}),$$

Cache interference can only happen among memory blocks that are in the same subset that maps to the same cache line. For the u-th cache line, τ_k can be interfered at

most $N_{k,u}$ times and τ_i can interfere at most $N_{i,u}$ times. The following formula gives an upper bound on the number of cache miss by accessing the HBs for task τ_k.

$$S(\tau_i, \tau_k) = \sum_{u=0}^{N-1} min(N_{i,u}, N_{k,u})$$

Suppose the penalty for an LLC cache miss is a constant, C_{miss}, then $I_{i,k}^{sc}$ satisfies:

$$I_{i,k}^{sc} = S(\tau_i, \tau_k) C_{miss}.$$

The computation only takes the memory accesses of τ_k and τ_i as input, so $I_{i,k}^{sc}$ only depends on memory accesses of τ_k and τ_i.

Lemma 4.6. $I_{i,k}^{sc} = S(\tau_i, \tau_k) C_{miss}$.

Proof. The lemma holds as discussed above. □

Given a taskset, $I_{i,k}^{sc}$ can be computed, as shown in the proof of Lemma 4.6. In the following discussion, we assume $I_{i,k}^{sc}$ is known.

Lemma 4.6 gives an upper bound on cache interference for τ_k imposed by only one job of τ_i. It is possible that more than one job of τ_i interfere with τ_k. We denote the number of jobs of τ_i that interfere with τ_k as $N_{i,k}$.

Lemma 4.7. *The total cache interference τ_k exhibited from $N_{i,k}$ jobs of τ_i is bounded by $N_{i,k} I_{i,k}^{sc}$.*

Proof. For $N_{i,k}$ jobs of τ_i, the total number of accesses to each memory block $m_{i,x}$ is bounded by $N_{i,k} n_{i,x}$. Thus, the execution of $N_{i,k}$ jobs of τ_i accesses the k-th cache line also at most $N_{i,k} N_{i,u}$ times. From the proof of Lemma 4.6, the upper bound of the total cache interference exhibited by τ_k from $N_{i,k}$ jobs of τ_i is $\sum_{u=0}^{N-1} min(N_{i,k} N_{i,u}, N_{k,u}) C_{miss}$.

$$\begin{aligned} N_{i,k} I_{i,k}^{sc} &= N_{i,k} \sum_{u=0}^{N-1} min(N_{i,u}, N_{k,u}) C_{miss} \\ &= \sum_{u=0}^{N-1} min(N_{i,k} N_{i,u}, N_{i,k} N_{k,u}) C_{miss} \\ &\geq \sum_{u=0}^{N-1} min(N_{i,k} N_{i,u}, N_{k,u}) C_{miss} \end{aligned}$$

□

IP formulation

We can compute an upper bound of the maximum cache interference a task may exhibit during an execution window by introducing an Integer Programming (IP) formulation, which can be transformed to an integer linear programming formulation.

It is necessary to check the schedulability of the task-set without considering cache interference. If the task-set does not pass the initial schedulability test, there is no need to calculate the cache interference. Only if all tasks (including τ_i) pass the schedulability test (without considering cache interference), the IP is solved to compute the upper bound on cache interference. Therefore, the IP formulation is based on the assumption that τ_i is schedulable without cache interference.

If $N_{i,k}$ jobs of τ_i executing concurrently with τ_k, the cache interference that τ_i causes on τ_k is bounded by $N_{i,k}I^{sc}_{i,k}$ according to Lemma 4.7. As a task may exhibit cache interference from more than one task during a job execution, the total cache interference for one job execution of τ_k is bounded by the sum of the contributions of all other tasks $\tau_i (i \neq k)$ in the task set τ. Thus, the objective function of the IP formulation is:

$$\max \sum_{i \neq k} N_{i,k} I^{sc}_{i,k}. \tag{4.13}$$

The IP formulation will have an unbounded solution without further constraints to the variable $N_{i,k}$. To get a bounded solution, we analyze the constraints on $N_{i,k}$. First, we define the concept of the execution window of a job.

Definition 4.4. The Execution Window (EW) of the j-th job of τ_k (J^j_k) is time interval $[s^j_k, f^j_k]$ from the staring time to the finishing time of J^j_k.

Note that the length of an execution window may be larger than C_k, since the EW includes the cache interference. We use C'_k as the length of the EW because of the iterative computation which will be described later on.

$N_{i,k}$ reaches its minimal value when a job of τ_i starts to execute as soon as it is released and the execution finishes just before the start of the EW, as shown the case (a) in Figure 4.10. Denoting C^{min}_i as the smallest execution time of τ_i, often called Best-Case Execution Time (BCET), we have the following constraint:

$$\forall i \neq k, \left\lfloor \frac{\max(0, C'_k - T_i + C^{min}_i)}{T_i} \right\rfloor + \xi_i \leq N_{i,k} \tag{4.14}$$

where $\xi_i = \begin{cases} 1 & ((C'_k + C^{min}_i) \mod T_i) - D_i + C^{min}_i > 0 \\ 0 & otherwise \end{cases}$.

The term ξ_i indicates whether the last job of τ_i released within the EW that interferes with τ_k since the last released job should start its execution C^{min}_i before its relative deadline if the task is schedulable.

The maximum value of $N_{i,k}$ is taken when the first interfering job of τ_i finishes just after the start of the EW and the last interfering job of τ_i starts to execute at the time when it is released. Such a situation is depicted as case (b) in Figure 4.10. Thus, we have the second constraint on $N_{i,k}$:

$$\forall i \neq k,\ N_{i,k} \leq 1 + \left\lceil \frac{\max(0, C'_k - T_i + D_i)}{T_i} \right\rceil \tag{4.15}$$

If $N_{i,k} > 2$, the first and last interfering jobs of τ_i may occupy almost 0 computation capacity in the EW. Let J^j_i be such a job among the remaining $N_{i,k} - 2$ interfering

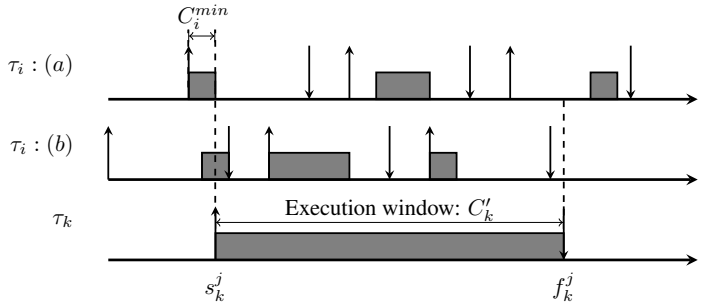


Figure 4.10: Situations where τ_i interferes τ_k with the most and least number of jobs.

jobs of τ_i between the first and the last ones. Both release time r_i^j and deadline d_i^j of J_i^j are within the EW of τ_k.

Lemma 4.8. *If τ_i is schedulable without considering cache interference, C_i computation capacity of the processing core is reserved for the execution of J_i^j during $[r_i^j, d_i^j]$. If J_i^j executes for $C_i^{act} < C_i$, the processing core will be accumulatively idle (executing nothing, simply wasting the processing capacity for τ_i) for at least $C_i - C_i^{act}$ during $[r_i^j, d_i^j]$.*

Proof. If τ_i satisfies the schedulability condition without considering cache interference: $\frac{\Omega_i(C)}{m} + C_i < D_i$, the core on which J_i^j is executed spends at most $D_i - C_i$ in total for the execution of other interfering tasks during $[r_i^j, d_i^j]$. J_i^j is guaranteed to have C_i computation capacity during $[r_i^j, d_i^j]$. □

The remaining computation capacity of a multicore processor with m cores is $(m-1)C'_k$ since one core is dedicated to the execution of τ_k. Due to the limited computation capacity of the processor, the total execution of the tasks that may interfere with τ_k within the EW can not exceed $(m-1)C'_k$. Hence, we have the third constraint:

$$\sum_{i \neq k} \max(0, N_{i,k} - 2) C_i \leq (m-1)\, C'_k. \tag{4.16}$$

The objective function (4.13) together with three constraints on $N_{i,k}$ i.e. inequalities (4.14), (4.15) and (4.16) form our IP problem. Since C_i^{min} is a relatively small number, we take the extreme case: $C_i^{min} = 0$. As task parameters such as C_i, D_i, T_i is known, the optimal solution of the IP only depends on the length of EW. Thus, we use $I^{sc}(C'_k)$ to denote the optimal value of the IP problem if C'_k is used as the length of the EW in the IP.

Note that Inequalities (4.14) and (4.16) are based on the assumption that τ_i is schedulable. Thus, before solving the IP, we have to check the schedulability of the taskset assuming no cache interference between tasks, i.e. $\bar{I}_i^{sc} = 0$.

Computation complexity of the IP. The original IP can be easily transformed to an Integer Linear Programming (ILP) problem by introducing a new integer variable $y_{i,j}$ for each $N_{i,j}$ with two additional constraints: $y_{i,j} \geq 0$ and $y_{i,j} \geq N_{i,k} - 2$.

Inequality (4.16) can be replaced by $\sum_{i \neq k} y_{i,k} C_i \leq (m-1)\, C'_k$. In the transformed ILP problem, we have totally $2(n-1)$ variables and $4(n-1)+1$ constraints. The complexity of the IP is the same as the complexity of solving the transformed ILP problem, which is $\mathcal{O}(n64^n \ln 4n)$ [23].

4.4 Iterative Computation

Due to the presence of cache interference, a job may execute longer than C_k on a multicore platform with shared caches. However, a larger execution time may introduce more cache interference, as illustrated in Figure 4.11.

In Figure 4.11 (a), if the job of τ_k executes for C'_k, only one job of τ_i interferes with τ_k. In Figure 4.11(b), if the job of τ_k executes for a larger execution time, say $C'_k + I^{sc}(C'_k)$, two jobs of τ_i could possibly interfere with τ_k, which potentially may increase the cache interference exhibited by τ_k. This example suggests an iterative method to find an upper bound on the cache interference.

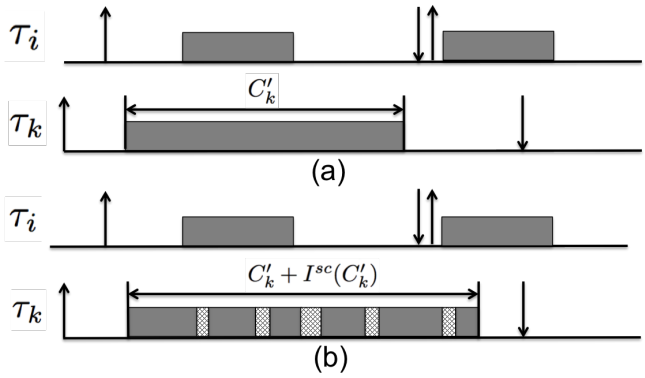


Figure 4.11: More cache interference if τ_k executes for a longer time.

Lemma 4.9. *$I^{sc}(C'_k)$ is non-decreasing with respect to C'_k.*

Lemma 4.9 is explained by the above example.

We give a sufficient condition for a certain value that can be used as an upper bound on cache interference.

Lemma 4.10. *if $\exists\, C^*_k \geq C_k$ such that $C^*_k = C_k + I^{sc}(C^*_k)$, then $\bar{I}^{sc}_k = I^{sc}(C^*_k)$.*

Proof. If $C^*_k = C_k + I^{sc}(C^*_k)$, then $I^{sc}(C^*_k) = I^{sc}(C_k + I^{sc}(C^*_k))$. According to Lemma 4.9, given an execution window of τ_k that is no more than $C_k + I^{sc}(C^*_k)$, the cache interference exhibited by τ_k is not larger than $I^{sc}(C^*_k)$. Therefore, $I^{sc}(C^*_k)$ is the upper bound on cache interference for τ_k. By definition, $\bar{I}^{sc}_k = I^{sc}(C^*_k)$. □

We now derive the iterative algorithm, called $CacheInterference(\tau)$ which takes taskset τ as input, to compute an upper bound on cache interference for each task $\tau_k \in \tau$:

- Since the constraints of IP assume the taskset is schedulable, we first check the schedulability of the taskset assuming no cache interference between each task. Only if all tasks pass schedulability test, the following steps will be taken.
- C'_k is initialized with C_k and an upper bound value on the cache interference $I^{sc}(C'_k)$ is created which is initially set to zero
- By solving the IP, we compute a new upper bound of the cache interference $I^{sc}(C'_k)$.
- If the new upper bound of cache interference is the same as the old upper bound, the $I^{sc}(C'_k)$ is the final upper bound of τ_k. Otherwise, another round of computing the upper bound on cache interference is performed using the upper bound derived at the previous iteration. The iteration for τ_k stops either if no update on $I^{sc}(C'_k)$ is possible anymore or if the computed $I^{sc}(C'_k)$ is large enough to make τ_k unschedulable.
- The previous steps are repeated for every task in τ.

A more formal version of the $CacheInterference(\tau, m)$ algorithm is given by Pseudocode 4.1. The algorithm returns I^* which includes the upper bounds on cache interference $I^{sc}(C^*_k)$ for each task τ_k and C^* which includes the upper bounds on the execution length C^*_k for each τ_k. If I^* and C^* are empty, the taskset is not schedulable.

Since the solution of the IP is non-decreasing with respect to C'_k according to Lemma 4.9 and one termination condition is $C'_k \geq D_k$, the termination of the iterative algorithm is guaranteed.

Before presenting the final theorem to check the schedulability of the task set, we define the following notations.

We denote $U(\tau_i)$ as task τ_i' s utilization taking shared cache interference into account, $U(\tau_i)$ is defined by:

$$U_i = \frac{C^*_i}{T_i}.$$

The utilization of taskset τ, denoted by $U(\tau)$, is defined by:

$$U(\tau) = \sum_{\tau_i \in \tau} U_i = \sum_{\tau_i \in \tau} \frac{C^*_i}{T_i}.$$

We sort all C^*_i in a non-increasing order, and use $\Delta^{m-1}_{C^*_i}$ to denote the sum of the first $(m-1)$ elements in this list, so

$$\Delta^{m-1}_{C^*_i} = \sum_{the\ (m-1)\ largest} C^*_i$$

For task τ_k, we also define a constant L_k by:

$$L_k = \frac{\sum_{\tau_i \in \tau} C^*_i + \Delta^{m-1}_{C^*_i}}{m - U(\tau)} - S_k.$$

We propose the following Theorem to check the schedulability of the task set.

Pseudocode 4.1: CacheInterference(τ, m)

```
1: Input: Task parameters, number of cores: m
2: I* ← empty list, used to store I^sc(C*_k) for each task
3: C* ← empty list, used to store C*_k for each task
4: pass ← true
5: for all τ_k ∈ τ do
6:    for all A_k ∈ [0, K] do
7:       Ω_k(C) ← calculation of Equation (4.12) using C
8:       if Ω_k(C)/m + C_k ≥ D_k + A_k then
9:          pass ← false
10:         break
11:      end if
12:   end for
13: end for
14: if pass then
15:   for all τ_k ∈ τ do
16:      update ← true, I^old_k ← 0, I^new_k ← 0
17:      C'_k ← C_k
18:      while update do
19:         I^old_k ← I^new_k
20:         I^new_k ← Solution of IP with C'_k as the EW
21:         C'_k = C_k + I^new_k
22:         if I^new_k == I^old_k or C'_k ≥ D_k then
23:            update ← false
24:         end if
25:      end while
26:      Add I^new_k to I*
27:      Add C'_k to C*
28:   end for
29: end if
30: return I*, C*
```

Theorem 4.11. *A task set τ is schedulable with the EDF_{np} or FP_{np} scheduling policy on a multicore platform composed of m identical cores with shared caches if for each task $\tau_k \in \tau$ and $0 \leq A_k \leq L_k$,*

(1) $\exists\, C_k^* \geq C_k$ such that $C_k^* = C_k + I^{sc}(C_k^*)$,

(2) $\frac{\Omega_k}{m} + C_k^* < D_k + A_k$.

Proof. From (1), $\bar{I}_k^{sc}$ is bounded and $\bar{I}_k^{sc} = I^{sc}(C_k^*)$ according to Lemma 5.3.

From Lemma 4.5, $\bar{I}_k^{pre} = \frac{\Omega_k(C^*)}{m}$.

$\forall A_k \geq 0$, if $\frac{\Omega_k}{m} + C_k^* = \frac{\Omega_k}{m} + C_k + I^{sc}(C_k^*) < A_k + D_k$ then $\bar{I}_k^{pre} + C_k + \bar{I}_k^{sc} < A_k + D_k$. Theorem 4.11 follows from Theorem 4.1.

We further prove that if condition (2) is to be violated for any A_k, then it must also be violated for some $A_k \leq L_k$.

$W^n_{i,k}$ can be bounded by considering the number of body jobs to be $\left\lfloor \frac{A_k+S_k}{T_i} \right\rfloor$ and the contribution of the carry-out to be C^*_i, so

$$W^n_{i,k} \leq \left\lfloor \frac{A_k+S_k}{T_i} \right\rfloor C^*_i + C^*_i \leq \frac{A_k+S_k}{T_i} C^*_i + C^*_i$$
$$= (A_k+S_k)U_i + C^*_i$$

Similarly, $W^c_{i,k}$ can be bounded by considering the contribution of both the carry-in and the carry-out are C^*_i:

$$W^c_{i,k} \leq \left\lfloor \frac{A_k+S_k}{T_i} \right\rfloor C^*_i + 2C^*_i \leq (A_k+S_k)U_i + 2C^*_i$$

From Equation (4.12)

$$\Omega_k = \max_{(\tau^n,\tau^c)\in\tau} \Big(\sum_{\tau_i\in\tau^n} W^n_{i,k} + \sum_{\tau_i\in\tau^c} W^c_{i,k} \Big)$$
$$\leq (A_k+S_k) \sum_{\tau_i\in\tau} U_i + \sum_{\tau_i\in\tau} C^*_i + \Delta^{m-1}_{C^*_i}$$
$$= (A_k+S_k)U(\tau) + \sum_{\tau_i\in\tau} C^*_i + \Delta^{m-1}_{C^*_i}$$

If condition (2) is to be violated for any A_k, then $\exists A_k$, $\frac{\Omega_k}{m} + C^*_k \geq D_k + A_k$,

$$\Longrightarrow \Omega_k \geq m(D_k + A_k - C^*_k)$$
$$\Longrightarrow (A_k+S_k)U(\tau) + \sum_{\tau_i\in\tau} C^*_i + \Delta^{m-1}_{C^*_i} \geq m(S_k+A_k).$$

Solve the above inequality for A_k, we have:

$$A_k \leq \frac{\sum_{\tau_i\in\tau} C^*_i + \Delta^{m-1}_{C^*_i}}{m - U(\tau)} - S_k = L_k.$$

This tells us the range of A_k that should be tested.

□

Finally, we give the procedure $CheckSchedulability(\tau, m)$ to perform the schedulability test, as illustrated by Pseudocode 4.2.

Computational complexity: Let n represent the number of tasks in the task-set. For τ_k, let I^{min}_k be the smallest difference between cache interference caused by one job of τ_i and τ_j, i.e. $I^{min}_k = \min_{i,j}(I^{sc}_{i,k} - I^{sc}_{j,k})$, the iterative algorithm takes at most $\eta = \max_k \frac{(D_k - C_k)}{I^{min}_k}$ iterations to terminate since C'_k either stays the same or increases at least with I^{min}_k in each iteration. Thus, the complexity of the iterative algorithm to compute the upper bound on cache interference is $\mathcal{O}(\eta n^2 64^n ln4n)$. The complexity of computing L_k, Ω_k is polynomial. Therefore, the complexity to perform the schedulability test is $\mathcal{O}(\eta n^2 64^n ln4n)$.

Pseudocode 4.2: CheckSchedulability(τ, m)

```
1: Input: Task parameters, number of cores: m
2: I*, C* ← CacheInterference(τ, m)
3: if I* == null then
4:    return Unschedulable
5: else
6:    for all τ_k ∈ τ do
7:       for all A_k ∈ [0, K] do
8:          Ω_k(C*) ← calculation of Equation (4.12) using C*
9:          if Ω_k(C*)/m + C*_k ≥ D_k + A_k then
10:            return Unschedulable
11:         end if
12:      end for
13:   end for
14: end if
15: return Schedulable
```

4.5 Experiments

In this section, we evaluate the performance of the proposed schedulability test for EDF_{np} and FP_{np} in terms of acceptance ratio. More specifically, we will quantify the effects of cache interference on the schedulability of the generated tasksets. We will also compare the schedulability performance of EDF_{np} against FP_{np} over randomly generated tasksets.

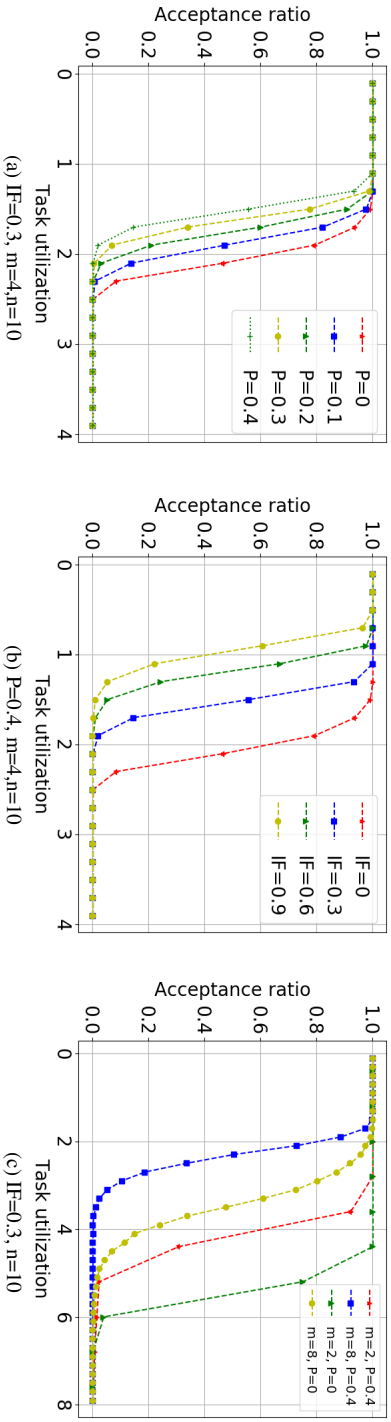
The experiments have been performed varying i) the probability of two tasks having cache interference on each other: P ($P = 0.1,\ 0.2,\ 0.3$ or 0.4), ii) the cache interference factor IF ($IF = 0,\ 0.3,\ 0.6$ or 0.9), iii) the number of cores m ($m = 2,\ 4$ or 8), iv) total task utilization U_{tot} (U_{tot} from 0.1 to $m - 0.1$ with steps of 0.2). Given those four parameters, we have generated 20000 tasksets in each experiment. As the task generation policies may significantly affect experimental results, we give the policies used in the experiments as follows.

Task utilization generation policy. We use Randfixedsum [88] to generate vectors that consist of N elements and whose components sum to the U_{tot}. Each element in the vector is assigned an individual task utilization U_k in the taskset.

Task period and WCET generation policy. For each task τ_k, T_k is uniformly distributed over the interval $[100, 200]$. The WCET of τ_k is derived by $C_k = T_k \times U_k$. We consider an implicit deadline task system, which implies that $D_i = T_i$.

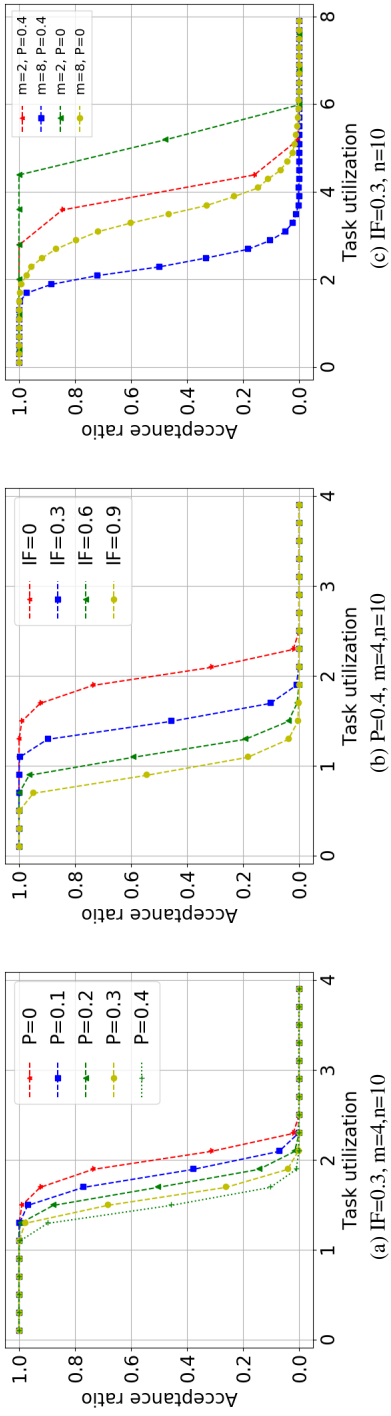
Cache interference generation policy. The probability of two task having cache interference is P. If two tasks τ_k and τ_i interfere with each other, $I^{sc}_{i,k}$ is generated as $I^{sc}_{i,k} = IF \times min(0.5C_i, 0.5C_k)$.

In each experiment, we measure the number of schedulable tasksets that pass the proposed schedulability test. The acceptance ratio, which is the number of schedulable tasksets devided by the total number of tasksets (20000), are shown in Figure 4.12 and Figure 4.13 for EDF_{np} and FP_{np}, respectively.



(a) IF=0.3, m=4,n=10 (b) P=0.4, m=4,n=10 (c) IF=0.3, n=10

Figure 4.12: Acceptance ratio of EDF_{np} scheduler when varying IF, P and m.



(a) IF=0.3, m=4,n=10 (b) P=0.4, m=4,n=10 (c) IF=0.3, n=10

Figure 4.13: Acceptance ratio of FP_{np} scheduler when varying IF, P and m.

Fixing $m = 4,\ n = 10, IF = 0.3$, Figure 4.12a and Figure 4.13a illustrate the acceptance ratio with different P for EDF_{np} and FP_{np}, respectively. With the same U_{tot}, the acceptance ratio for both EDF_{np} and FP_{np} decreases as P increases because a larger P indicates more tasks in the taskset could interfere with each other, which may potentially increase the upper bound on cache interference for each task. Fixing P, it can be observed that the acceptance ratio of EDF_{np} is slightly higher than FP_{np} when $U_{tot} \in [1.1, 2.5]$.

Figure 4.12b and Figure 4.13b show the acceptance ratio achieved by EDF_{np} and FP_{np}, respectively, for the cases $IF = 0,\ 0.3,\ 0.6,\ 0.9$, fixing $m = 4,\ n = 10,\ P = 0.4$. The red line with $IF = 0$ represents the acceptance ratio when tasks have no cache interference. Evidently, the acceptance ratios with a lower IF are better than those with a larger IF. As we increase IF with the same amount, the average acceptance ratio decreases in a slower fashion. However, it does not indicate that a lower bound on the average acceptance ratio is possible since the cache interference gets larger as IF increases, eventually making the interfered tasks unschedulable. Fixing IF, it is also clearly that the acceptance ratio achieved by EDF_{np} is slightly better than FP_{np} when $U_{tot} \in [0.7, 2.5]$.

Figure 4.12c and Figure 4.13c illustrate the acceptance ratio with respect to the number of cores for EDF_{np} and FP_{np}, respectively. In the two figures, the acceptance ratio for task having no cache interference are also plotted. Instead of using U_{tot} as horizontal axis, we scale the horizontal axis with $\frac{U_{tot} \times 8}{m}$ for $m = 2,\ 4$. It is worth noting that an execution platform with fewer cores is more efficient in terms of acceptance ratio than those with more cores. However, for processors with different cores scheduled by EDF_{np} (or FP_{np}), the difference in the acceptance ratio of scheduling between the baseline (tasks having no cache interference, $IF = 0$) and tasks having cache interference is almost similar.

We measured the execution time of running the proposed schedulability test with different task-set scales. The executions are conducted on an Intel Xeon processor using only one core running at 2.4GHz. On average, it takes 0.13 seconds to check the schedulability of the task-set consisting of 10 tasks, 0.27 seconds for task-set with 20 tasks, while 0.56 seconds for task-set with 30 tasks.

4.6 Conclusions

In this chapter, we developed a new schedulability analysis of global scheduling (EDF_{np} and FP_{np}) for real-time multicore systems with shared caches. We constructed an integer programming formulation that can be transformed to an integer linear programming formulation to calculate the upper bound on cache interference exhibited by a task during a given execution window. Using this integer formulation, we subsequently proposed an iterative algorithm to obtain an upper bound on the shared cache interference a task may exhibit during one job execution. We derived a new schedulability condition by integrating the upper bound on the cache interference into the schedulability analysis. A set of experiments has been performed using our proposed schedulability analysis to demonstrate the effects of cache interference for a range of different tasksets. We also compared the schedulability performance of EDF_{np} against

FP_{np} in the presence of cache interference. Our empirical evaluations showed that EDF_{np} is slightly better than FP_{np} in terms of task sets deemed schedulable.

5

Partitioned Scheduling for Real-time Systems with Shared Caches

In the previous chapter, we studied the schedulability analysis of global scheduling for real time multicore systems with shared caches. In this chapter, we extend the schedulabilty analysis to another real time scheduling paradigm: partitioned scheduling. This chapter addresses RQ3 listed in chapter 1, which is concerned with cache interference aware partitioned scheduling for real-time multicore systems.

On multi-core systems, two paradigms are widely used for scheduling real-time tasks: global and partitioned (semi-partitioned) scheduling. For global scheduling, a job is allowed to execute on any core. In partitioned scheduling, on the other hand, tasks are statically allocated to processor cores, i.e., each task is assigned to a core and is always executed on that particular core. Although the partitioned approaches cannot exploit all unused processing capacity since a bin-packing-like problem needs to be solved to assign tasks to cores, it offers lower runtime overheads and provides consistently good empirical performance at high utilizations [11].

Furthermore, taking the shared cache interference into account, partitioned approaches can achieve better schedulability than global scheduling. We provide a simple example to illustrate this. Consider three tasks τ_1, τ_2 and τ_3 with the same period and relative deadline of 7, the WCETs of τ_1, τ_2 and τ_3 are 3, 3 and 2, respectively. The execution platform is a processor with 2 cores including a last-level shared cache. If τ_1 and τ_2 run concurrently, the maximum cache interference exhibited by τ_1 and τ_2 is 3. We assume that τ_3 has no cache interference with τ_1 and τ_2.

It is impossible to conclude that this taskset is schedulable under global scheduling. Figure 5.1 shows a case where τ_3 misses its deadline. At time $t = 0$, tasks τ_1 and τ_2 are scheduled to execute on the two cores. In the figure, the black area of a cumulative length of 3 denotes the $WCET$, and the hatched area of a cumulative length of 3 represents the extra execution time due to the cache interference. At $t = 6$, τ_1 and τ_2 both finish their executions, after which τ_3 starts its execution. At $t = 7$, τ_3 misses its deadline. Similarly, consider another case: at $t = 0$, τ_3 and τ_1 (or τ_2) are scheduled, at $t = 2$, τ_3 finishes and τ_2 (or τ_1) starts its execution. Since cache interference is counted per job [108], in the worst case, the cache interference exhibited by τ_2 (or τ_1) can still be 3 even though the duration of co-running τ_2 (or τ_1) and τ_1 (or τ_2) is less than in the

This chapter was published as [46](submitted)

previous case. Due to the cache interference, τ_2 (or τ_1) could finish its execution at $t = 8$, leading to a deadline miss for τ_2 (or τ_1).

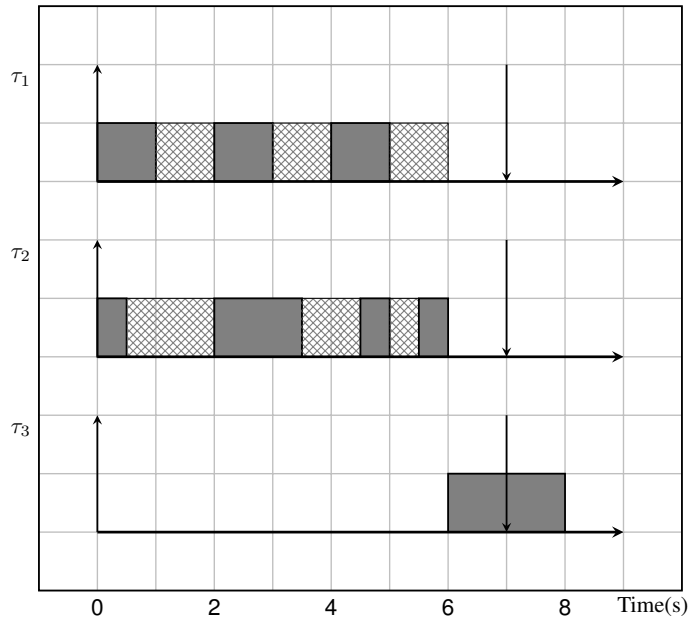


Figure 5.1: Case where τ_3 misses its deadline if τ_1, τ_2 and τ_3 are scheduled globally.

However, the taskset is schedulable under the partitioned scheduling. Consider, e.g., the partitioning scheme in which τ_1 and τ_2 are assigned to core 1, and task τ_3 is assigned to core 2. Since τ_1 and τ_2 are assigned to the same core, they cannot run simultaneously. As no cache interference can occur during task execution, it can be verified that every task meets its deadline.

Motivated by the above example, we address the following main research questions listed in Chapter 1:

RQ3 How to develop a cache interference aware partitioned scheduling for real-time multi-core systems? Is the partitioned scheduling better than global scheduling in terms of schedulability performance?

To answer this question, we propose a novel cache-interference aware task partitioning algorithm, called CA-TPAR. To the best of our knowledge, this is the first work on partitioned scheduling for real-time multi-core systems, accounting for shared cache interference. In chapter 4, we presented an approach to calculating the upper bound on cache interference for tasks that are globally scheduled. In this chapter, we will extend the approach to deriving the upper bound on cache interference for partitioned scheduling, which is required by CA-TPAR. We conduct schedulability analysis of CA-TPAR and formally prove its correctness. A set of experiments is performed to evaluate the schedulability performance of CA-TPAR against global EDF scheduling

over randomly generated tasksets. Our empirical evaluations show that CA-TPAR outperforms global EDF scheduling in terms of tasksets deemed schedulable.

As most commodity processors in the embedded domain does not provide support for cache partitioning, it is worth to note that, in this work, we do not deploy any cache partitioning techniques to mitigate the inter-core cache interference. Instead, we address the problem of task partitioning in the presence of shared cache interference.

The rest of the chapter is organized as follows. The system model and some other prerequisites for this chapter are described in Section 5.1. Section 5.2 describes the proposed CA-TPAR, where we also detail the computation of the inter-core cache interference and schedulability analysis of CA-TPAR. Section 5.3 presents the experimental results, after which Section 5.4 concludes the chapter.

5.1 System Model and Prerequisites

5.1.1 System Model

Task Model and Architecture Model

We continue to use the same task and architecture model discussed in section 4.2 in chapter 4. We repeat the most important details and notations that will be used in this chapter.

A taskset τ comprises n periodic or sporadic real-time tasks τ_1, τ_2, ... τ_n. Each task $\tau_k = (C_k, D_k, T_k) \in \tau$ is characterized by a worst-case computation time C_k, a period or minimum inter-arrival time T_k, and a relative deadline D_k. As explained in chapter 4, it is difficult to accurately estimate C_k considering cache interference of other tasks executing concurrently. In our task model, C_k is derived, assuming task k is the only task executing on the multi-core processor platform, i.e. any cache interference delays are not included in C_k.

A task τ_k is a sequence of jobs J_k^j, where j is the job index. We denote the arrival time, starting time, finishing time and absolute deadline of a job j as r_k^j, s_k^j, f_k^j and d_k^j, respectively.

Our system architecture consists of a multi-core processor with m identical cores onto which the individual tasks are scheduled. The last-level caches (*LLC*) are shared between all cores.

Partitioned Non-preemtive Schedulers

In this paper, we focus on non-preemptive partitioned scheduling. Once a task instance starts execution, any preemption during the execution is not allowed, so it must run to completion. So we do not have to consider intra-core cache interference. If not explicitly stated, cache interference will therefore refer to inter-core cache interference in the following discussion.

Since partitioning tasks among a multi-core processor reduces the multi-core processor scheduling problem to a series of single-core scheduling problems (one for each core), the optimality without idle inserted time [32, 45] of non-preemptive EDF (EDF_{np}) makes it a reasonable algorithm to use as the run-time scheduler on each core.

Therefore, we make the assumption that each core, and the tasks assigned to it by the partitioning algorithm, are scheduled at run time according to an EDF_{np} scheduler.

EDF_{np} assigns a priority to a job according to the absolute deadline of that job. A job with an earlier absolute deadline has higher priority than others with a later absolute deadline. EDF_{np} scheduling is work-conserving: using EDF_{np}, there are no idle cores when a ready task is waiting for execution.

5.1.2 The Demand-Bound Function

A successful approach to analyzing the schedulability of real-time tasks is to use a demand bound function [9]. The demand bound function $DBF(\tau_i, t)$ is the largest possible cumulative execution demand of all jobs that can be generated by τ_i to have both their arrival times and their deadlines within any time interval of length t. Let t_0 be the starting time of a time interval of length t, the cumulative execution demand of τ_i's jobs over $[t_0, t_0 + t]$ is maximized if one job arrives at t_0 and subsequent jobs arrive as soon as permitted i.e., at instants $t_0 + T_i, t_0 + 2T_i, t_0 + 3T_i$,... Therefore, $DBF(\tau_i, t)$ can be computed by Equation (5.1),

$$DBF(\tau_i, t) = max(0, (\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1) \times C_i). \tag{5.1}$$

[2] proposed a technique for approximating the $DBF(\tau_i, t)$. The approximated demand bound function $DBF^*(\tau_i, t)$ is given by the following equation:

$$DBF^*(\tau_i, t) = \begin{cases} 0 & t < D_i \\ C_i + U_i \times (t - D_i) & otherwise \end{cases} \tag{5.2}$$

where $U_i = \frac{C_i}{T_i}$.

Observe that the following inequality holds for all τ_i and all $0 \leq t$:

$$DBF^*(\tau_i, t) \geq DBF(\tau_i, t) \tag{5.3}$$

5.1.3 Uniprocessor Schedulability

The schedulabity analysis of uniprocessor scheduling is well studied. [7] presented a necessary and sufficient condition for the feasibility test of a sporadic task system τ scheduled by EDF_{np} on a uniprocessor platform.

Theorem 5.1. *A taskset τ is schedulable under EDF_{np} on a uniprocessor platform if and only if*

$$\forall t, \sum_{i=1}^{n} DBF(\tau_i, t) \leq t \tag{5.4}$$

and for all $\tau_j \in \tau$:

$$\forall t : C_j \leq t \leq D_j : C_j + \sum_{i=1; i \neq j}^{n} DBF(\tau_i, t) \leq t. \tag{5.5}$$

Note that the computation of $DBF(\tau_i, t)$ and $DBF^*(\tau_i, t)$ by Equation (5.1) and (5.2) and the two schedulability test conditions (5.4) and (5.5) do not account for shared cache interference. We will extend the computation of $DBF(\tau_i, t)$ and $DBF^*(\tau_i, t)$ and the two schedulability conditions to the cases where shared cache interference is considered.

5.1.4 Cache Interference

As an LLC is shared by multiple cores, it allows running tasks to compete among each other for shared cache space, which is governed by a cache replacement policy. As a consequence, the tasks replace blocks that belong to other tasks, causing shared cache interference.

Let τ_k be the interfered and τ_i be the interfering task. We use $I^c_{i,k}$ to represent the upper bound on the shared cache interference imposed on τ_k by only one job execution of τ_i. From Lemma 4.6 in chapter 4, $I^c_{i,k}$ can be bounded. The proof of Lemma 4.6 in chapter 4 also shows the method to compute $I^c_{i,k}$. In the following discussion, we assume $I^c_{i,k}$ is known.

It is however possible that multiple jobs of τ_i interfere with τ_k. We denote the number of jobs of τ_i that interfere with τ_k as $N_{i,k}$.

As stated in Lemma 4.7 in 4, the total cache interference τ_k exhibits from $N_{i,k}$ jobs of τ_i is bounded by $N_{i,k} \cdot I^c_{i,k}$.

5.2 Cache interference aware task partitioning : CA-TPAR

Given a taskset τ comprised of n periodic or sporadic tasks and a processing platform π with m identical cores $\pi = \{\pi_1, \pi_2, ..., \pi_m\}$, a partitioning algorithm decides how to assign tasks to cores to avoid task deadline misses. The problem of assigning a set of tasks to a set of cores is analogous to the bin-packing problem. In this case, the tasks are the objects to pack and the bins are cores. The bin-packing problem is known to be NP-hard in the strong sense. Thus, searching for an optimal task assignment is not practical.

[60] and [31] studied several bin-packing heuristics for the preemptive and non-preemptive task model. Typically, each of the bin-packing heuristics follows the following pattern: tasks of the task system are first sorted by some criterion, after which the tasks are assigned in order to a core that satisfies a sufficient condition.

Let $\tau(\pi_x)$ denote the set of tasks assigned to processor core π_x where $1 \leq x \leq m$. $\tau_i \in \tau(\pi_x)$ means τ_i is assigned to core π_x. If taskset τ can be scheduled by a partitioned algorithm, the outcome of running a partitioning algorithm is a task partition such that:

- All tasks are assigned to processor cores:

$$\cup_{1 \leq x \leq m} \tau(\pi_x) = \tau$$

- Each task is assigned to only one core:

$$\forall y \neq x, 1 \leq y \leq m, 1 \leq x \leq m, \ \ \tau(\pi_y) \cap \tau(\pi_x) = \emptyset$$

In Section 5.2.1, we describe our cache interference aware task partitioning : CA-TPAR. Section 5.2.2 derives the calculation of the upper bound on the shared cache interference. Section 5.2.3 conducts the schedulability analysis for CA-TPAR.

Before describing CA-TPAR, we first extend the DBF to account for shared cache interference. Due to the extra execution delay caused by shared cache interference, a task τ_i may execute longer than C_i. Given a task partitioning scheme, one can compute the upper bound on cache interference exhibited by task τ_i, denoted as $\bar{I}_i^c$. We will show the method to compute this $\bar{I}_i^c$ later. In multiprogrammed environment, the actual execution time including cache interference of τ_i can be bounded by $C_i + \bar{I}_i^c$. We denote $DBF^c(\tau_i, t)$ as the demand bound function which accounts for cache interference. $DBF^c(\tau_i, t)$ can be computed by extending Equation (5.1):

$$DBF^c(\tau_i, t) = max(0, (\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1) \times (C_i + \bar{I}_i^c)). \tag{5.6}$$

Similarly, the approximated demand bound function $DBF^{c*}(\tau_i, t)$ is given by the following equation by extending Equation (5.2):

$$DBF^{c*}(\tau_i, t) = \begin{cases} 0 & t < D_i \\ C_i + \bar{I}_i^c + U_i^c \times (t - D_i) & otherwise \end{cases} \tag{5.7}$$

where $U_i^c = \frac{C_i + \bar{I}_i^c}{T_i}$.

It can also be observed that:

$$DBF^{c*}(\tau_i, t) \geq DBF^c(\tau_i, t) \tag{5.8}$$

5.2.1 The Task Partitioning Algorithm: CA-TPAR

We now propose CA-TPAR, a task partitioning algorithm taking shared cache interference into account.

We assume the tasks are sorted in non-decreasing order by means of a certain criterion. For example, if a task's relative deadline is chosen as criterion, then $D_i \leq D_{i+1}$ for $1 \leq i \leq n$. More criteria for sorting the tasks will be discussed in Section 5.3.

CA-TPAR performs the following steps:

step 1: for each task $\tau_i \in \tau$:

1. **Attempt** to assign τ_i to π_x,
2. Calculate the upper bound on cache interference $\bar{I}_k^c$ for $\tau_k \in \tau(\pi_x) \cup \{\tau_i\}$, i.e. tasks that are already assigned to π_x and τ_i, assuming τ_i is assigned to π_x. We will show the calculation procedure in the next subsection.
3. Check if the following condition holds for each $\tau_k \in \tau(\pi_x) \cup \{\tau_i\}$

$$D_k \geq \sum_{\substack{\tau_j \in \tau(\pi_x) \cup \{\tau_i\} \\ D_j \leq D_k}} DBF^{c*}(\tau_j, D_k) + \max_{\substack{\tau_j \in \tau(\pi_x) \cup \{\tau_i\} \\ D_j > D_k}} C_j + \bar{I}_j^c. \tag{5.9}$$

(a) If no τ_k violates condition (5.9), the attempt is **admitted** and τ_i is added to $\tau(\pi_x)$.

(b) If condition (5.9) is violated by at least one τ_k, the attempt is **rejected**. We attempt to assign τ_i to the next core π_{x+1} and repeat steps (2) and (3). If no core can be assigned to τ_i, then τ_i is added to the temporarily non-allocable taskset, denoted as τ^{tna}.

step 2: after performing step 1, the resulting τ^{tna} is either an empty set or non-empty.

(a) If $\tau^{tna} = \emptyset$, which means all tasks have been allocated to cores, CA-TPAR returns $Success$,

(b) Otherwise, we perform step 1 to each $\tau_t \in \tau^{tna}$. τ_t is removed from τ^{tna} if it can be assigned to a core. We repeatedly perform step 1 to $\tau_t \in \tau^{tna}$ until τ^{tna} becomes empty or no more tasks in τ^{tna} could be allocated to cores. If $\tau^{tna} = \emptyset$ at the end, CA-TPAR returns $Success$, otherwise CA-TPAR returns $Fail$: it is unable to determine if scheduling τ is feasible on the multi-core platform.

We briefly explain the rationale behind condition (5.9). Given a task τ_k, the execution demand of tasks (including τ_k) with a relative deadline no larger than D_k is calculated by the first part (left-hand side) of the sum in condition (5.9). Since we consider a non-preemptive task system, the second part of the sum accounts for the blocking time due to the execution of a task with a larger relative deadline than τ_k at the time a job of τ_k arrives. If the sum of the execution demand and the blocking time is smaller than D_k, the task τ_k will not miss its deadline. We will prove this in Section 5.2.3.

A more formal version of the task partitioning algorithm CA-TPAR is given by Pseudocode 5.1.

Pseudocode 5.1: *CATPAR*(τ, π)

```
1: sort τ in non-decreasing order by a selected criterion
2: τ^tna ← τ, taskAssigned ← true, τ(π_1), τ(π_2), ..., τ(π_m) ← ∅
3: τ(π) = (τ(π_1), τ(π_2), ..., τ(π_m))
4: while τ^tna ≠ ∅ and taskAssigned == true do
5:    τ^tna, taskAssigned, τ(π)=TaskPartition(τ^tna, π, τ(π))
6: end while
7: if τ^tna == ∅ then
8:    return Success
9: else
10:   return Failed
11: end if
```

The input to procedure *CATPAR* is the taskset τ to be partitioned and the execution platform π consisting of m cores. *CATPAR* repeatedly invokes the procedure *TaskPartition*, illustrated by Pseudocode 5.2, to perform step 1 of the CA-TPAR algorithm. The input to *TaskPartition* is the temporarily non-allocable taskset τ^{tna}, π, and existing task assignment $\tau(\pi) = (\tau(\pi_1), \tau(\pi_2), ..., \tau(\pi_m)$. τ^{tna} is initialized as τ. Every time when

TaskPartition finishes, some tasks in the taskset τ^{tna} can be assigned to cores, and thus τ^{tna} and $\tau(\pi)$ are updated.

Pseudocode 5.2: *TaskPartition*(τ, π, $\tau(\pi)$)

```
1:  taskAssigned ← false, τ^tna ← ∅
2:  for all τ_i ∈ τ do
3:     assignTo ← NULL, coreSuccess ← true
4:     for all π_x ∈ π do
5:        for all τ_k ∈ τ(π_x) ∪ {τ_i} do
6:           calculate Ī_k^c
7:        end for
8:        for all τ_k ∈ τ(π_x) ∪ {τ_i} do
9:           if condition ∼ (5.9) violates for τ_k then
10:             coreSuccess ← false
11:             break;
12:          end if
13:       end for
14:       if coreSuccess then
15:          τ(π_x) ← τ(π_x) ∪ {τ_i}
16:          assignTo ← π_x, taskAssigned ← true
17:          break;
18:       end if
19:    end for
20:    if assignTo == NULL then
21:       τ^tna ← τ^tna ∪ {τ_i}
22:    end if
23: end for
24: return τ^tna, taskAssigned, τ(π)
```

Lines $5-7$ in the procedure of *TaskPartition* perform step 1.(2) of CA-TPAR to compute the upper bound on cache interference for tasks. When CA-TPAR attempts to assign τ_i to π_x, the upper bound on cache interference caused by $\tau_k \in \tau(\pi_x)$, i.e. tasks that are already assigned to π_x, is recomputed. This is because a tighter bound can be possibly obtained by the recalculation, as will be shown soon. Considering τ_i is more likely to be assigned to π_x if the upper bound on the cache interference caused by $\tau_k \in \tau(\pi_x)$ is smaller, the recalculation makes CA-TPAR less pessimistic.

5.2.2 Calculation of The Upper Bound on Cache Interference: $\bar{I}_k^c$

The CA-TPAR algorithm requires to calculate the upper bound on cache interference before it assigns a new task to a core. We now describe such a procedure for the calculation of $\bar{I}_k^c$.

In last chapter, we have presented an approach to calculating the upper bound on cache interference for tasks that are globally scheduled. By extending the approach, we compute the upper bound on cache interference for partitioned scheduling. This

is done by two steps. First, given the existing task assignment represented by $\tau(\pi) = (\tau(\pi_1), \tau(\pi_2), ..., \tau(\pi_m)$ and τ^{na} as the taskset consisting of the tasks that have not been assigned, we construct an integer programming (IP) formulation to calculate the upper bound on the cache interference exhibited by a task within an execution window. Then, we use an iterative algorithm to obtain the upper bound on cache interference a task may exhibit during its job executions.

IP formulation

In the following discussion, we compute the upper bound on cache interference exhibited by τ_k, assuming τ_i is the interfering task and τ_k is assigned to π_x.

The Execution Window (EW) of the j-th job of τ_k (J_k^j) is defined as the time interval $[s_k^j, f_k^j]$ from the staring time to the finishing time of J_k^j. We use C'_k as the length of the EW because of the iterative computation which will be described later on.

The objective function of the IP formulation is to maximize the the total cache interference exhibited by task τ_k. The total cache interference for one job execution of τ_k is bounded by the sum of the contributions of all tasks τ_i in the taskset τ. So the objective function is:

$$\max \sum N_{i,k} \cdot I_{i,k}^c. \tag{5.10}$$

To get a bounded solution, we analyze the constraints on $N_{i,k}$.

If tasks τ_i and τ_k are assigned to the same core π_x, at each time instant, at most one task of τ_i and τ_k executes on core π_x. No jobs from τ_i could interfere with τ_k. Therefore, we have the following:

$$\forall \tau_i \in \tau(\pi_x), N_{i,k} = 0. \tag{5.11}$$

$N_{i,k}$ reaches its minimal value when a job of τ_i starts to execute as soon as it is released and the execution finishes just before the start of the EW, as shown in case (a) of Figure 4.10 in chapter 4. Taking the smallest execution time of τ_i, C_i^{min}, as 0, we have the following constraint:

$$\forall \tau_i \notin \tau(\pi_x), \left\lfloor \frac{\max(0, C'_k - T_i)}{T_i} \right\rfloor + \xi_i \leq N_{i,k} \tag{5.12}$$

where $\xi_i = \begin{cases} 1 & (C'_k \mod T_i) - D_i > 0 \\ 0 & otherwise \end{cases}$.

The term ξ_i indicates whether or not the last job of τ_i released within the EW interferes with τ_k.

The maximum value of $N_{i,k}$ is taken when the first interfering job of τ_i finishes just after the start of the EW and the last interfering job of τ_i starts to execute at the time when it is released. Such a situation is depicted as case (b) in Figure 4.10 in chapter 4. Thus, we have the second constraint on $N_{i,k}$:

$$\forall \tau_i \notin \tau(\pi_x),\ N_{i,k} \leq 1 + \left\lceil \frac{\max(0, C'_k - T_i + D_i)}{T_i} \right\rceil. \tag{5.13}$$

If $N_{i,k} > 2$, the first and last interfering jobs of τ_i may occupy almost 0 computation capacity in the EW. Let J_i^j be a job among the remaining $N_{i,k} - 2$ interfering jobs of τ_i between the first and the last ones. Both release time r_i^j and deadline d_i^j of J_i^j are within the EW of τ_k.

If τ_i is (or will be) successfully assigned to core π_y , at least C_i computation capacity of the processing core is reserved for the execution of J_i^j during $[r_i^j, d_i^j]$. The total execution of interfering tasks τ_i on each processor y (with $y \neq x$) cannot exceed C'_k. Since we do not know the core assignment for tasks in τ^{na}, those tasks are allowed to execute on any core. Thus, we have the following inequality (5.14),

$$\forall y \neq x, \sum_{\tau_i \in \tau(\pi_y) \cup \tau^{na}} \max(0, N_{i,k} - 2)C_i \leq C'_k. \tag{5.14}$$

The objective function (5.10) together with constraints on $N_{i,k}$ i.e. inequalities (5.11), (5.12), (5.13) and (5.14) form our IP problem. As task parameters such as C_i, D_i, T_i are known, the input of the IP formulation is the length of EW: C'_k, existing task assignment: $\tau(\pi) = (\tau(\pi_1), \tau(\pi_2), ..., \tau(\pi_m)$, and remaining tasks that need to be assigned: τ^{na}. Thus, we use $IP(C'_k, \tau(\pi), \tau^{na})$ to denote the IP problem and use $I^c(C'_k, \tau(\pi), \tau^{na})$ to denote the optimal solution.

When CA-TPAR attempts to assign a task τ_i to a core π_x, the upper bound on cache interference caused by $\tau_k \in \tau(\pi_x)$, i.e. tasks that are already assigned to π_x, is recomputed. We now show that a tighter upper bound for task τ_k can be possibly obtained by the re-computation.

Given a task τ_k and an execution window of length C'_k, let us suppose the IP formulation in the previous computation of cache interference is $IP(C'_k, \tau_p(\pi), \tau_p^{na})$, and the IP formulation for the re-computation is $IP(C'_k, \tau_q(\pi), \tau_q^{na})$.

Between the two computations for the same task τ_k, CA-TPAR may assign some tasks to cores. If a task τ_i is assigned to a core π_x, τ_i is removed from τ_p^{na} and is added to $\tau_q(\pi_x)$. Obviously, we have $\tau_q^{na} \subseteq \tau_p^{na}$ and $\forall 1 \leq x \leq m, \tau_p(\pi_x) \subseteq \tau_q(\pi_x)$.

Lemma 5.2. *Given τ_k and C'_k,*

$$I^c(C'_k, \tau_q(\pi), \tau_q^{na}) \leq I^c(C'_k, \tau_p(\pi), \tau_p^{na}).$$

Proof Sketch: We show the proof sketch.

From condition 5.9, one can prove the following: if $\tau_i \in \tau(\pi_x)$ and $\tau_k \in \tau(\pi_x)$, then $C_k + \bar{I}_k^c \leq D_i$.

By the above statement and the constraints of the IP problem, we can prove that any solution of $IP(C'_k, \tau_q(\pi), \tau_q^{na})$ is also feasible for $IP(C'_k, \tau_p(\pi), \tau_p^{na})$. Thus,

$$I^c(C'_k, \tau_q(\pi), \tau_q^{na}) \leq I^c(C'_k, \tau_p(\pi), \tau_p^{na}).$$

Lemma 5.2 is the reason CA-TPAR forces the recalculation of upper bound on cache interference caused by tasks that are already assigned to cores by CA-TPAR.

Iterative Computation

Due to the presence of cache interference, a job may execute longer than C_k on a multi-core platform with shared caches. However, a larger execution time may introduce more cache interference.

We give a sufficient condition for a certain value that can be used as an upper bound on cache interference exhibited by τ_k, denoted by $\bar{I}_k^c$.

Lemma 5.3. *Given $\tau(\pi)$ and τ^{na}, if $\exists\, C_k^* \geq C_k$ such that $C_k^* = C_k + I^c(C_k^*, \tau(\pi), \tau^{na})$, then $\bar{I}_k^c = I^c(C_k^*, \tau(\pi), \tau^{na})$.*

The equation can be solved by means of fixed point iteration: the iteration starts with an initial value for the length of EW and upper bound on cache interference, i.e. $C'_k = C_k$ and $I^c(C'_k) = 0$. By solving the IP, we compute a new upper bound of the cache interference $I^c(C'_k, \tau(\pi), \tau^{na})$ and a new corresponding length of EW, $C'_k = C_k + I^c(C'_k, \tau(\pi), \tau^{na})$. The iterative computation for τ_k stops either if no update on $I^c(C'_k, \tau(\pi), \tau^{na})$ is possible anymore or if the computed $I^c(C'_k, \tau(\pi), \tau^{na})$ is large enough to make τ_k unschedulable i.e. $I^c(C'_k, \tau(\pi), \tau^{na}) + C'_k > D_k$.

Computational complexity: The original IP can be easily transformed to an Integer Linear Programming (ILP) problem. In the transformed ILP problem, we have totally $2n$ variables and $4n + m - 1$ constraints. The complexity of the IP is the same as the complexity of solving the transformed ILP problem, which is $\mathcal{O}((4n + m)64^n \ln(4n + m))$ [23].

From chapter 4, the complexity to compute the upper bound on cache interference exhibited by each task is $\mathcal{O}(\eta(4n + m)64^n \ln(4n + m))$ (η is defined in chapter 4). In *TaskPartition*, at most n tasks in τ are checked for at most m cores, thus, the complexity of *TaskPartition* is $\mathcal{O}(\eta nm(4n+m)64^n \ln(4n+m))$. Since the while loop in *CATPAR* executes at most n times, the complexity of CA-TPAR is $\mathcal{O}(\eta n^3 m64^n \ln(4n + m))$. Although it is exponential complexity, current implementations of LP solver are very efficient to get a solution.

5.2.3 Schedulability Analysis

Uniprocessor feasibility

Task partitioning reduces the problem of multi-core processor scheduling into a set of single-core processor scheduling problems (one for each core). Following Theorem 5.1, we first propose a schedulability condition, as stated in Theorem 5.4, for uniprocessor scheduling, taking shared cache interference into consideration. Note that the condition in Theorem 5.4 is sufficient and not necessary as $\bar{I}_j^c$ is the calculated upper bound on the shared interference exhibited by τ_j, the actual cache interference can be smaller than $\bar{I}_j^c$.

Theorem 5.4. *A taskset $\tau(\pi_x)$ is schedulable under EDF_{np} on a uniprocessor platform if*

$$\forall t, \sum_{\tau_i \in \tau(\pi_x)} DBF^c(\tau_i, t) \leq t \tag{5.15}$$

and for all $\tau_j \in \tau(\pi_x)$*:*

$$\forall t : C_j + \bar{I}_j^c \leq t \leq D_j : C_j + \bar{I}_j^c + \sum_{\substack{\tau_i \in \tau(\pi_x) \\ i \neq j}} DBF^c(\tau_i, t) \leq t. \tag{5.16}$$

Schedulability analysis of CA-TPAR

We first derive one property that must be satisfied for tasks assigned to the same core by CA-TPAR. This is useful for the proof of the feasibility analysis conducted later for CA-TPAR.

Lemma 5.5. *If tasks are assigned to cores by CA-TPAR,*

$$\forall \pi_x \in \pi, \sum_{\tau_i \in \tau(\pi_x)} U_i^c \leq 1. \tag{5.17}$$

Proof. Let τ_u be the task with the largest relative deadline among tasks in $\tau(\pi_x)$, so, $D_u = max\{D_i | \tau_i \in \tau(\pi_x)\}$. Obviously,

$$\tau_i \in \tau(\pi_x) \Longrightarrow D_i \leq D_u.$$

Since τ_u satisfies Inequality (5.9), we have

$$D_u \geq \sum_{\tau_i \in \tau(\pi_x)} DBF^{c*}(\tau_i, D_u). \tag{5.18}$$

From Equation (5.7), $DBF^{c*}(\tau_i, D_u)$ is computed by:

$$DBF^{c*}(\tau_i, D_u) = U_i^c \times (D_u - D_i + T_i) \geq U_i^c \times D_u.$$

Replacing $DBF^{c*}(\tau_i, D_u)$ in Inequality (5.18),

$$D_u \geq \sum_{\tau_i \in \tau(\pi_x)} U_i^c \times D_u \Longrightarrow \sum_{\tau_i \in \tau(\pi_x)} U_i^c \leq 1.$$

This is Inequality (5.17). □

On each core $\pi_x \in \pi$, tasks in $\tau(\pi_x)$ are scheduled under EDF_{np}. The next lemma shows the feasibility of $\tau(\pi_x)$.

Lemma 5.6. *If the tasks are assigned to cores by CA-TPAR,* $\forall \pi_x \in \pi$*,* $\tau(\pi_x)$ *is feasible on core* π_x *by* EDF_{np}*.*

Proof. For the sake of contradiction, assume that each task in $\tau(\pi_x)$ satisfies condition (5.9), but that a task's deadline is missed when scheduling the tasks in $\tau(\pi_x)$ on core π_x. Let t_f be the time that a task misses a deadline on core π_x.

By Theorem 5.4, either

$$\sum_{\tau_i \in \tau(\pi_x)} DBF^c(\tau_i, t_f) > t_f, \tag{5.19}$$

or $\exists \tau_p, \tau_p \in \tau(\pi_x)$ and $\exists t_f, C_p + \bar{I}_p^c \leq t_f \leq D_p$, such that

$$C_p + \bar{I}_p^c + \sum_{\substack{\tau_i \in \tau(\pi_x) \\ i \neq p}} DBF^c(\tau_i, t_f) > t_f. \tag{5.20}$$

It will be shown that if either Inequality (5.19) or (5.20) holds, then a contradiction is reached.

We first prove the existence of $\tau_i \in \tau(\pi_x)$ that satisfies $D_i \leq t_f$. Assuming $\forall \tau_i \in \tau(\pi_x)$, $D_i > t_f$, from Equation (5.7),

$$\sum_{\tau_i \in \tau(\pi_x)} DBF^{c*}(\tau_i, t_f) = 0.$$

By the assumption, neither Inequality (5.19) nor (5.20) will hold. So the assumption is false.

Therefore, we can always find $\tau_i \in \tau(\pi_x)$ that satisfies $D_i \leq t_f$. Let τ_s be the task with the largest relative deadline, i.e. $D_s = max\{D_i | \tau_i \in \tau(\pi_x) \wedge D_i \leq t_f\}$

(A) we first prove that if Inequality (5.19) holds, it would lead to contradiction.

From Inequality (5.8) and (5.19),

$$\sum_{\tau_i \in \tau(\pi_x)} DBF^{c*}(\tau_i, t_f) > t_f. \tag{5.21}$$

By the definition of $DBF^{c*}(\tau_i, t_f)$, we have

$$\sum_{\substack{\tau_i \in \tau(\pi_x) \\ D_i > D_s}} DBF^{c*}(\tau_i, t_f) = 0.$$

$$\begin{aligned}
&\sum_{\tau_i \in \tau(\pi_x)} DBF^{c*}(\tau_i, t_f) \\
&= \sum_{\substack{\tau_i \in \tau(\pi_x) \\ D_i \leq D_s}} DBF^{c*}(\tau_i, t_f) + \sum_{\substack{\tau_i \in \tau(\pi_x) \\ D_i > D_s}} DBF^{c*}(\tau_i, t_f) \\
&= \sum_{\substack{\tau_i \in \tau(\pi_x) \\ D_i \leq D_s}} C_i + \bar{I}_i^c + U_i^c \times (t_f - D_i) \\
&= \sum_{\substack{\tau_i \in \tau(\pi_x) \\ D_i \leq D_s}} C_i + \bar{I}_i^c + U_i^c \times (t_f - D_s + D_s - D_i) \\
&= \sum_{\substack{\tau_i \in \tau(\pi_x) \\ D_i \leq D_s}} DBF^{c*}(\tau_i, D_s) + U_i^c \times (t_f - D_s).
\end{aligned} \tag{5.22}$$

τ_s satisfies condition (5.9):

$$D_s \geq \sum_{\substack{\tau_i \in \tau(\pi_x) \\ D_i \leq D_s}} DBF^{c*}(\tau_i, D_s).$$

From Equation (5.22) and Inequality (5.21), we have

$$D_s + \sum_{\substack{\tau_i \in \tau(\pi_x) \\ D_i \leq D_s}} U_i^c \times (t_f - D_s) > t_f \tag{5.23}$$

$$\implies \sum_{\substack{\tau_i \in \tau(\pi_x) \\ D_i \leq D_s}} U_i^c > 1 \implies \sum_{\tau_i \in \tau(\pi_x)} U_i^c > 1.$$

This contradicts to Lemma 5.5.

(B) we now prove that if Inequality (5.20) holds, it would also lead to contradiction.

We know that $\exists \tau_s, \tau_p$ such that $D_s \leq t_f \leq D_p$. We consider two cases (B1): $D_s = D_p$ and (B2): $D_s < D_p$.

(B1) if $D_s = D_p$, then $t_f = D_p$

$$DBF^{c*}(\tau_p, t_f) = C_p + \bar{I}_p^c$$

From Inequality (5.20),

$$\sum_{\tau_i \in \tau(\pi_x)} DBF^c(\tau_i, t_f) > t_f.$$

This leads to contradiction as proved in case (A).

(B2) if $D_s < D_p$, we have

$$C_p + I_p^c \leq \max_{\substack{\tau_j \in \tau(\pi_x) \\ D_j > D_s}} C_j + \bar{I}_j^c,$$

and

$$\sum_{\substack{\tau_i \in \tau(\pi_x) \\ i \neq p}} DBF^c(\tau_i, t_f) \leq \sum_{\tau_i \in \tau(\pi_x)} DBF^c(\tau_i, t_f).$$

From Inequality (5.20), we have

$$\max_{\substack{\tau_j \in \tau(\pi_x) \\ D_j > D_s}} C_j + \bar{I}_j^c + \sum_{\tau_i \in \tau(\pi_x)} DBF^{c*}(\tau_i, t_f) > t_f.$$

Replacing $\sum_{\tau_i \in \tau(\pi_x)} DBF^{c*}(\tau_i, t_f)$ in the above inequality using equation (5.22), we have

$$\max_{\substack{\tau_j \in \tau(\pi_x) \\ D_j > D_s}} C_j + \bar{I}_j^c + \sum_{\substack{\tau_i \in \tau(\pi_x) \\ D_i \leq D_s}} DBF^{c*}(\tau_i, D_s) + U_i^c \times (t_f - D_s) > t_f. \tag{5.24}$$

Since τ_s satisfies condition (5.9),

$$D_s \geq \sum_{\substack{\tau_i \in \tau(\pi_x) \\ D_i \leq D_s}} DBF^{c*}(\tau_i, D_s) + \max_{\substack{\tau_i \in \tau(\pi_x) \\ D_i > D_s}} C_i + \bar{I}_i^c. \tag{5.25}$$

From Inequality (5.24) and (5.25),

$$\sum_{\tau_i \in \tau(\pi_x)} U_i^c > 1.$$

This also contradicts to Lemma 5.5. □

The correctness of Algorithm CA-TPAR follows, by application of Lemma 5.6:

Theorem 5.7. *If the task partitioning algorithm CA-TPAR returns $Success$ on taskset τ, then the resulting partitioning is schedulable by EDF_{np} on each core.*

5.3 Experiments

In this section, we compare the performance of CA-TPAR using different sorting criteria with Global EDF (GEDF) in terms of acceptance ratio, that is, the number of tasksets that are deemed schedulable divided by the number of tasksets tested. We also quantify the effects of cache interference on the feasibility of the generated tasksets.

As mentioned in the beginning of Section 5.2.1, the CA-TPAR algorithm first sorts tasks in non-decreasing order using some criterion and then assigns tasks to the processor cores according to Equations (5.9).

We consider the following five sorting criteria: the reciprocal of a task's WCET $\frac{1}{C_i}$, a task's period T_i, the reciprocal of a task's utilization $\frac{1}{U_i} = \frac{T_i}{C_i}$, a task's slack $S_i = T_i - C_i$ and $random$ order.

The schedulability condition for GEDF, taking inter-core cache interference into consideration, is described in chapter 4.

5.3.1 Experimental Setup

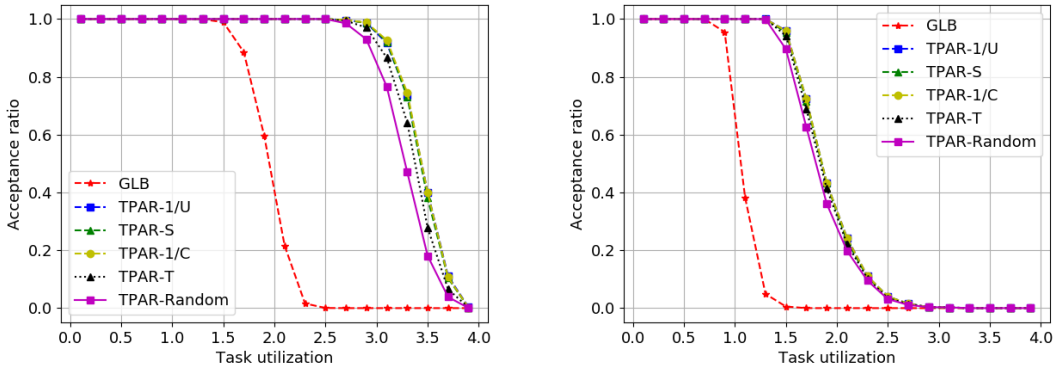
The experiments have been performed varying i) the number of tasks n ($n = 10,\ 20$) in the taskset, ii) total task utilization U_{tot} (U_{tot} from 0.1 to $m - 0.1$ with steps of 0.2), iii) the cache interference factor IF ($IF = 0.2$ or 0.8), and iv) the probability of two tasks having cache interference on each other: P ($P = 0.1$ or 0.4). Given those four parameters, we have generated 20000 tasksets in each experiment.

We use the same task utilization generation policy, task period and WCET generation policy and cache interference generation policy, described in chapter 4.

In each experiment, we measure the number of tasksets that can be successfully partitioned by CA-TPAR with different sorting criteria and the number of tasksets that can be scheduled by $GEDF$. The acceptance ratio is the number of schedulable tasksets divided by the total number of tasksets.

5.3.2 Results

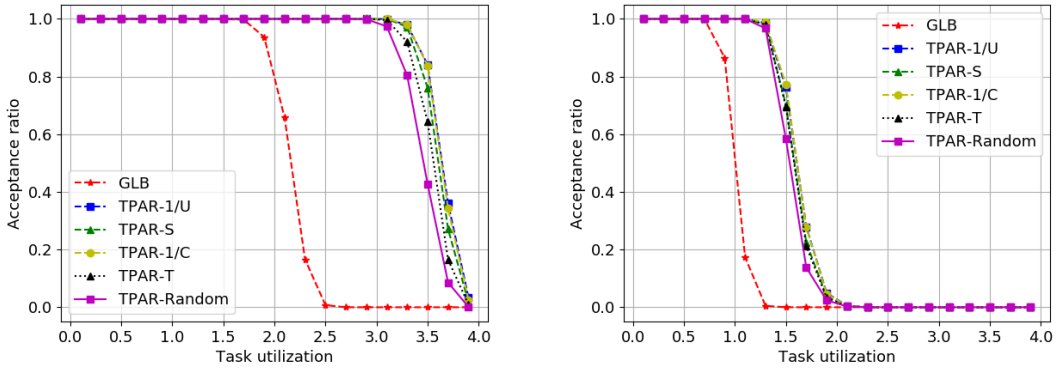
We report the major trends characterizing the experimental results, illustrated in Figures 5.2 and 5.3. In the figures, TPAR<*criterion*> represents a variant of CA-TPAR using <*criterion*> for sorting tasks, GLB stands for the GEDF scheduler.



(a) IF=0.2, P=0.1.

(b) IF=0.4, P=0.8.

Figure 5.2: Acceptance ratio with different IF and P when $m = 4$, $n = 10$.



(a) IF=0.2, P=0.1.

(b) IF=0.4, P=0.8.

Figure 5.3: Acceptance ratio with different IF and P when $m = 4$, $n = 20$.

CA-TPAR outperforms global EDF. Our results clearly show that CA-TPAR outperforms global EDF in all the test cases. It is also evident that CA-TPAR is highly effective for multi-core real-time systems, accounting for cache interference.

As shown in Figure 5.2a, when $IF = 0.2, P = 0.1$, all the generated tasksets can be successfully partitioned by all variants of CA-TPAR if $U_{tot} < 2.5$. while the global EDF achieves the full acceptance ratio when $U_{tot} < 1.5$. CA-TPAR is able to partition tasksets with the highest tested total utilization, i.e. $U_{tot} = 3.9$. Global EDF can only schedule tasksets with a total utilization of up to $U_{tot} = 2.5$.

It is important to observe that the gap of acceptance ratio between all variants of CA-TPAR and global scheduling is large when $U_{tot} \in [2, 3.5]$. Such a schedulability

performance gap also exists for different degrees of cache interference and different numbers of tasks in the taskset, as shown in Figure 5.2b, Figure 5.3a and Figure 5.3b.

We have also compared the schedulability performance of CA-TPAR and GEDF using heterogeneous task periods i.e. $T_i \in [100, 300]$ or $T_i \in [100, 500]$ (of which the results are omitted due to space limitations). In those tests, CA-TPAR still outperforms GEDF.

Performance gap among different variants of CA-TPAR is small. As is depicted in Figures 5.2a and 5.3a, when the cache interference is small ($IF = 0.2, P = 0.1$), TPAR-T and TPAR-$random$ performed worse than the TPAR-$1/C$, TPAR-S and TPAR-$1/U$ when $U_{tot} > 3$. while as the degree of cache interference increases, the schedulability performance gap becomes smaller, as shown in Figure 5.2b and Figure 5.3b. One reason could be that even though tasks are sorted by different criteria, all variants of CA-TPAR force recalculation of the upper bound on cache interference to obtain an upper bound that is as small as possible. The cache interference obtained by all variants of CA-TPAR thus is likely to be similar. Therefore, if cache interference dominates the schedulability result, the gap of schedulability performance among different variants of CA-TPAR is small.

Cache interference degrades schedulability performance. Figure 5.2a and Figure 5.2b compare the acceptance ratio with different P and IF for tasksets consisting of 10 tasks. With the same U_{tot}, the acceptance ratio achieved by all variants of CA-TPAR and global EDF decrease as P and IF increase. This is because a larger P and IF indicate more tasks in the taskset having larger cache interference with each other, which can potentially increase the upper bound on cache interference, eventually making the interfered tasks unschedulable. Similar observation can be made from Figure 5.3a and Figure 5.3b for tasksets consisting of 20 tasks.

5.3.3 Average Execution Time

We measured the execution time of CA-TPAR with different taskset sizes. The executions are conducted on an Intel Xeon processor using only one core running at $2.4GHz$. On average, it takes 0.85 seconds to run CA-TPAR for assignment of the taskset consisting of 10 tasks to a processor with 4 cores, while it takes 2.3 seconds for tasksets with 20 tasks.

5.4 Conclusions

Shared caches in multi-core processors introduce serious difficulties in providing guarantees on the real-time properties of embedded software. In this chapter, we addressed the problem of task partitioning in the presence of cache interference. To achieve this, CA-TPAR, a cache-interference aware task partitioning algorithm was proposed. To the best of our knowledge, this is the first work on partitioned scheduling for real-time multi-core systems, accounting for shared cache interference. An integer programming formulation was constructed to calculate the upper bound on cache interference exhibited by a task, which is required by CA-TPAR. We conducted schedulability analysis of CA-TPAR and formally proved the correctness of CA-TPAR. A set of experiments

was performed to evaluate the schedulability performance of CA-TPAR against global EDF scheduling over randomly generated tasksets. Our empirical evaluations show that CA-TPAR outperforms global EDF scheduling in terms of tasksets deemed schedulable.

Caching performance for high performance computing

6

CP_{pf}: a prefetch aware LLC partitioning approach

In the previous three chapters, we focused on research about improving timing predictability of multicore systems. In this chapter, we change our research angle to high performance caching. This chapter addresses RQ4 listed in chapter 1, which is concerned with improving caching performance in the presence of hardware prefetching.

Modern multicore processors implement a large Last Level Cache (LLC) to hide the long memory access latencies. Such a LLC is usually shared by multiple cores to allow high cache utilization. However, cache sharing also causes inter-application cache interference, which occurs when concurrently running applications compete among each other for shared cache space, governed by a cache replacement policy.

Hardware prefetching is another optimization technique that is commonly employed to reduce memory latencies. Although hardware prefetching can improve the applications' performance by fetching useful data in advance, it tends to increase the LLC contention among applications running concurrently on different cores. Taking the hardware prefetching into account, inter-application cache interference becomes more complicated.

Inter-application cache interference and the shared cache management has attracted a lot of research attention in the past decades.

UCP [71] and ASM [90] designed additional hardware components to modify the eviction and insertion policies to partition the cache, but these have not been implemented in existing processors.

Recent Intel processors support hardware-base cache partitioning, called cache allocation technology(CAT). CAT is used in the following work to improve system throughput, fairness and (or) average slowdown. Heracles [59] and Dirigent [120] control the amount of shared hardware resources, including the LLC, used by latency sensitive applications to improve Quality of Service and utilization.Pons et.al. [79] clusters applications using the k-means algorithm and distributes cache ways between the groups to improve system fairness. [69] assigns more cache space to critical applications to improve system turnaround time. Xiang et.al. [106] proposes a framework that dynamically monitors and predicts a workload's cache demand and reallocates the LLC given a performance target. KPart [30] leverages online profiling to obtain miss

This chapter was published as [111]

ratio curves for clustering applications and assigns each cluster of applications to a cache partition to improve system throughput.Park et.al. [66] proposed a coordinated partitioning of the LLC and memory bandwidth to improve the fairness of workloads on commodity servers.

A significant amount of work has been devoted to software-based cache partitioning approaches [16, 54, 94, 115]. These efforts are based on the classic technique of OS page-coloring, which is used to control where the physical page required by the target application is located in the cache.

Most of the previous works involving both hardware-based and software-based cache partitioning have been implemented and evaluated the performance of their cache partitioning policies on real machines. However, those works do not study the impact of hardware prefetching on cache performance nor do they explicitly reveal the interaction between the hardware prefetching and LLC management.

Some work has also been done to improve the cache management policy in the presence of hardware prefetching. [104] proposed a prefetching-aware cache replacement policy that treats prefetch and demand requests identically. [87] estimates prefetcher accuracy and prefetch-related cache pollution to adjust the aggressiveness of the hardware prefetcher dynamically. In [121, 122], a number of hardware-based prefetch pollution filtering mechanisms is proposed to differentiate good and bad prefetches dynamically to reduce the ineffective prefetches. [80] proposed a self-tuning prefetch accuracy predictor to predict if a prefetch is accurate or inaccurate to mitigate prefetch-related cache pollution. [29] proposed mechanisms that manage the shared resources on a multicore chip to obtain high performance and fairness. However, those approaches require additional hardware components that are not available in existing processors.

In a real system, many factors such as cache references by the operating system and hardware prefetching contribute to LLC interference [103]. In this study, we focus on the LLC management in the presence of hardware prefetching for multiprogrammed workloads.Therefore, we address the following research questions listed in Chapter 1:

RQ4 How does hardware prefetching affect the caching performance? How to manage shared caches to improve system performance in the presence of hardware prefetching?

To answer these questions, we first analyze the variation of application performance when varying the effective LLC space in the presence and absence of hardware prefetching, in order to study the interaction between hardware prefetching and LLC cache management. We show that hardware prefetching can compensate the application performance loss due to the reduced effective cache space. Motivated by this observation, we then classify applications into two categories, prefetching sensitive (*PS*) and non prefetching sensitive (*NPS*) applications, by the performance benefit they experience from hardware prefetchers. To address the cache contention and to also mitigate the potential prefetch-related cache interference, we propose CP_{pf}, a prefetch aware LLC partitioning approach for improving LLC management. CP_{pf} consists of a method using Precise Event-Based Sampling (PEBS) techniques for online classification of PS and NPS applications and a LLC partitioning scheme using Cache Allocation technology (CAT) for PS and NPS applications. Compared with a non-partitioning approach, CP$_{pf}$ achieves performance improvements of up to 1.20, 1.08 and 1.06 for

workloads with, respectively, 2, 4, and 8 applications and achieves speedups of up to 1.21 and 1.11 for workloads composed of two applications with 4 threads and 8 threads, respectively.

The prefetch aware cache partitioning approach presented in this work is a software-only solution by using hardware features like PEBS and CAT, which are readily available in existing multicore processors.

The rest of the chapter is organized as follows. Section 6.1 presents the motivation of this work. Section 6.2 provides the definition of PS and NPS applications. Section 6.3 describes CP$_{pf}$, where we also detail the online classification of *PS* and *NPS* applications and the LLC cache partitioning scheme. Section 6.4 presents the performance evaluation of CP$_{pf}$. Section 6.5 concludes the chapter.

6.1 Motivation

6.1.1 The impact of hardware prefetching on cache performance

Hardware prefetching implemented in today's high performance systems significantly influences memory sub-system performance. To understand the effects of hardware prefetching on the LLC performance for a single application, we evaluate the variation of application performance when varying the number of assigned LLC cache-ways in the presence and absence of hardware prefetching.

All the experiments in this work are conducted on a 20-core Intel Xeon commodity processor, of which the specifications are summarized in Table 6.1. There are five distinct hardware prefetchers on the Xeon platforms. Two prefetchers are associated with the L1-data caches: a Data Cache Unit (DCU) IP prefetcher and a DCU streamer prefetcher per core. Two prefetchers associated with the L2 caches: a Mid-Level cache (MLC) spatial prefetcher and a MLC streaming prefetcher. Finally, there is one LLC prefetcher. We can activate or deactivate these hardware prefetchers by setting the corresponding machine state register (MSR) bits [25].

Table 6.1: System Configuration

Component	Description
Processor	Intel Xeon Gold 6148 CPU @ 3.50GHz
L1 I-cache	Private, 32KB
L1 D-cache	Private, 32KB
L2 cache	Private, 1MB
L3 cache	Shared, 27.5MB, 11 ways
Memory	376G
OS	CentOS 7, Linux Kernel 4.17

Given the number of assigned LLC cache-ways, we run an application in isolation and measure its execution time for two cases: (1) hardware prefetchers are disabled, (2) hardware prefetchers are enabled. Figure 6.1 compares the slowdown for applications in

the SPEC CPU2017 [84], NPB [65] and Polybench [70] benchmark suites when varying the number of assigned cache-ways for the two cases. Due to space limitations, we only show the comparison for six representative applications, each application is identified by its index (in SPEC CPU2017) or abbreviation for its name (in NPB and Polybench). In Figure 6.1(a), the slowdown of an application is calculated by taking the execution time when it utilizes all the cache ways (11, in our experimental platform) and hardware prefetchers are disabled as the baseline, while in Figure 6.1(b), the baseline is execution time when the application fully utilizes all cache-cache ways and hardware prefetchers are enabled. Note that the baselines in Figure 6.1(a) and Figure 6.1(b) are thus different.

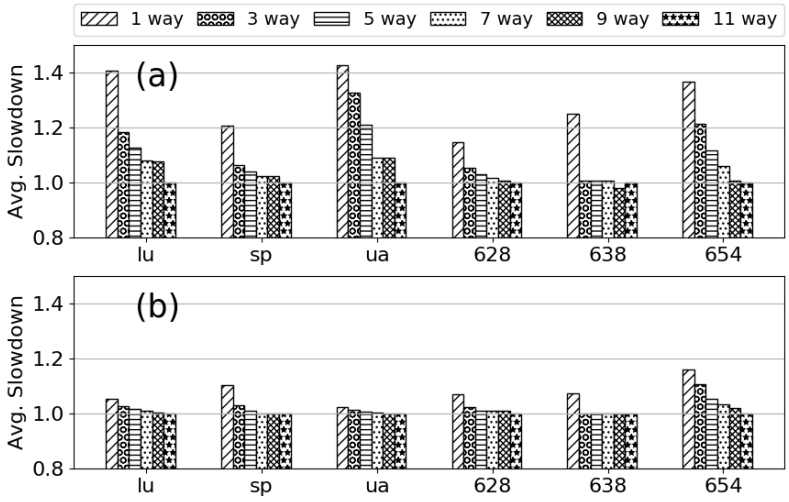


Figure 6.1: Comparison of application slowdown when varying the number of cache-ways allocated: (a) prefetching is disabled, (b) prefetching is enabled.

As illustrated in Figure 6.1, some applications, which originally experience significant performance degradation from a smaller LLC space in the absence of prefetching, encounter less performance degradation when the hardware prefetchers are enabled. For example, when hardware prefetching is disabled, the worst slowdowns for applications lu and ua are 1.41 and 1.42, respectively. However, the worst slowdowns are improved to 1.05 and 1.02 for lu and ua, if hardware perfetching is enabled. Thus, we make the following observation:

Observation 6.1. Hardware prefetching can compensate the application performance loss due to a reduced effective LLC space.

This can be explained by the fact that a prefetch-enabled LLC cache-controller will prefetch data from main memory before the actual references take place in order to try to avoid memory access latencies. Even though the effective LLC size for an application is decreased, the demanded data can often still be directly and timely serviced by the hardware prefetchers.

6.1.2 Inter-core prefetch-related cache pollution

The prefetched data for one application are placed in the shared LLC, competing for the available cache resources with its co-runners (i.e., other, simultaneously running applications). Therefore, one major drawback of hardware prefetching is the prefetch-related cache pollution which occurs when prefetched blocks of one application evict useful blocks of another application from the LLC. In this work, we assume that hardware prefetching taking place on behalf of an application itself has a more positive than negative influence on its performance. Thus, we neglect cache interference caused by self-prefetching and only consider inter-core prefetch-related LLC interference.

In a multicore system, inter-core prefetch-related cache pollution impacts the performance of applications in a non-uniform fashion. Some applications can be slowed down severely as a large number of their useful blocks are replaced by prefetched blocks, while others may not. Hardware prefetching can interact poorly with LLC management, which unnecessarily reduces the overall system performance. This leaves a significant opportunity to improve LLC management by means of prefetch-aware cache partitioning.

6.2 *PS* and *NPS* applications

In this section, we first classify applications into two categories: prefetching sensitive (PS) and non prefetching sensitive (NPS) applications by the performance benefit they experience from hardware prefetchers. We then study the performance sensitiveness to the available cache space for *PS* and *NPS* applications.

6.2.1 Definition of *PS* and *NPS* applications

We measure the execution time of an application in the presence and absence of hardware prefetching, respectively. We calculate the speedup of an application i by $SpeedUp_i = \frac{ET_{i,nopf}}{ET_{i,enpf}}$, where $ET_{i,nopf}$ is the execution time of application i when prefetchers are disabled and $ET_{i,enpf}$ is the execution time when hardware prefetchers are enabled.

We define applications whose performance is significantly improved by hardware prefetching as prefetching sensitive (PS) applications. In this work, application i is considered a *PS* application if $SpeedUp_i > 20\%$. An application that is not a *PS* application is considered to be an *NPS* application. By this definition, we classify the applications in the SPEC CPU2017 [84], NPB [65] and Polybench [70] benchmark suites into *PS* and *NPS* applications. The classification is shown in Table 6.2.

Table 6.2: Classification of *PS* and *NPS* applications.

Type	Applications
PS applications	619,654,628,638,603,mg,cg,sp, is,bt,ft,fdtd2d,jacobi2d,heat3d
NPS applications	602,605,607,631,623,627,600,641, 644,648,657,620,ua,lu,dc,ep,adi

6.2.2 Cache sensitivity of *PS* and *NPS* applications

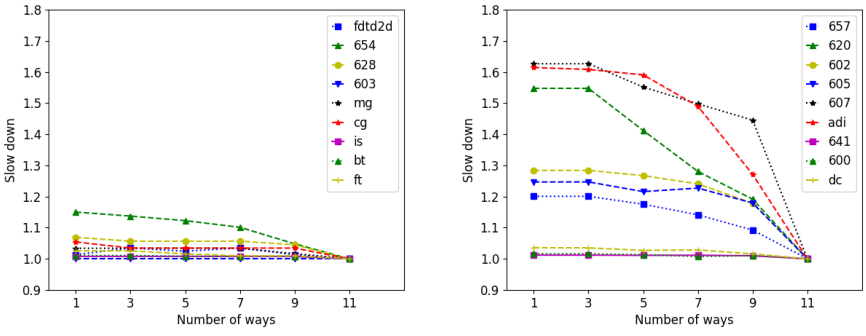
In a multiprogramming environment, the shared cache interference caused by co-runners (i.e., simultaneously running applications) reduces the effective number of cache-ways that an application can use. To study the impact of available cache-ways on the performance of *PS* and *NPS* applications, we conduct several experiments in which we use CAT to adjust the number of LLC ways available to the application from 1 to 11 (i.e., the total cache space in our experimental platform). In the experiments, all hardware prefetchers are enabled. Using this approach, we model the reduction in the available LLC space due to cache interference caused by co-runners.

Figure 6.2a and Figure 6.2b show the slowdown for 8 representative *PS* and *NPS* applications, respectively. The slowdown is calculated by taking the execution time when an application runs in isolation and utilizes all the cache ways as the baseline.

As can be seen, compared with *NPS* applications, the effective LLC size has, on average, a relatively small influence on the performance of *PS* applications. The performance of most *PS* application is sightly degraded if the effective LLC size decreases. The average maximum slowdown (obtained when an application runs with one cache-way) for *PS* applications is 1.05 with a worst-case slowdown of 1.15 for SPEC CPU2017 benchmark 654. For *NPS* applications, however, the average slowdown is 1.18 with a worst case of 1.62, experienced by SPEC CPU2017 benchmark 607. Thus, we make the following observation:

Observation 6.2. If hardware prefetchers are enabled, on average, the effective LLC size has a relatively small influence on the performance of PS applications, while the performance of NPS applications can be significantly affected by the effective LLC size.

The much smaller influence of the LLC size on the performance of *PS* applications can be explained by:



(a) *PS* applications (b) *NPS* applications

Figure 6.2: Application slowdown when varying the number of available ways with respect to a 11-way cache, if hardware prefethers are enabled.

1. PS applications may have a low reuse of data cached in the LLC because of timely prefetched data in the smaller, upper-levels of the cache hierarchy (L1/L2 caches) [80]. Subsequent data requests are directly serviced by the prefetched cache lines inserted into the L1/L2 caches, and rarely reach the LLC.

2. PS applications can more easily cope with higher LLC miss rates caused by the reduction of effective LLC space as a majority of demanded data elements can still be directly and timely serviced by the hardware prefetchers.

6.3 Prefetch aware LLC Partitioning

To exploit Observation 6.2, this section presents the prefetch aware LLC partitioning approach CP$_{pf}$. The general idea is to classify *PS* and *NPS* applications at run time and then divide the LLC into two partitions: one for *PS* applications and the other for *NPS* applications. Section 6.3.1 describes the online classification of *PS* and *NPS* applications, and Section 6.3.2 presents the LLC partitioning approach.

6.3.1 Online classification of applications

A classification criterion: cache miss distribution

The definition of PS and NPS application cannot be used directly for the online classification of *PS* and *NPS* applications. Due to the uncontrollable and unclear nature of hardware prefetching mechanisms implemented in modern commodity processors, we developed a non-trivial solution for the online classification of *PS* and *NPS* applications, which is based on the distribution of cache misses over the cache sets. The idea comes from the fact that prefetchers do not prefetch across virtual page boundaries. As indicated in [25], prefetched data will always be within the same 4K bytes memory page as the load instruction that triggered the prefetching.

The first (several) references to a data element in a new virtual page usually cannot be prefetched. Therefore, accesses to those data elements always result in LLC misses. After these first accesses, the hardware prefetchers start to recognize the data access patterns and start to predict and prefetch the data that is expected to be referenced in the near future. As a consequence, later data references inside the same virtual page do not necessarily cause LLC misses, as the hardware prefetchers may have inserted those data elements into the LLC before referencing them.

By using the PMU sampling mechanism, one can obtain the virtual addresses that were missed by a process in the LLC. Given a missed virtual address, one can determine the associated LLC set that the virtual address maps to. We will show the method to determine the missed cache set soon. By sampling the LLC misses over a short execution period (for instance, 1 second) for a process, we can obtain the cache miss distribution over the cache sets for the sampled process.

We use histograms to represent the distribution of LLC misses over the LLC sets. Figure 6.3 illustrates the histogram of missed cache sets when hardware prefetchers are disabled. Due to space limitations, we only show the histograms for four representative

applications. As can be seen, when all hardware prefetchers are disabled, cache misses are mostly uniformly distributed over all the cache sets.

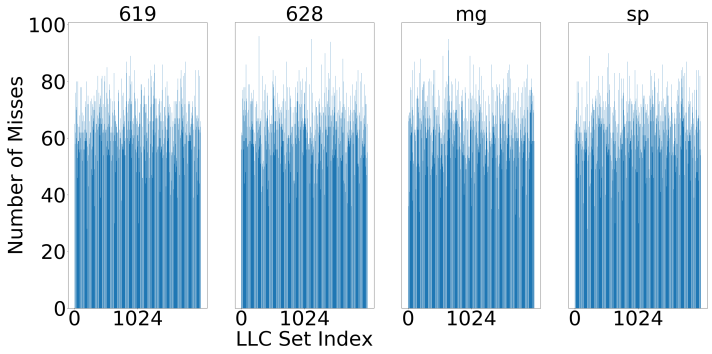


Figure 6.3: Histogram of missed LLC sets when hardware prefetchers are disabled.

Although we only show cache miss distributions of *PS* applications for later comparison, the uniform distribution is observed also for *NPS* applications. Observation 6.3 follows:

Observation 6.3. When hardware prefetchers are disabled, cache misses are mostly uniformly distributed over all the LLC sets for both *PS* and *NPS* applications.

Observation 6.3 verifies the assumption that a program block has a uniform probability of being present in any of the cache sets in the works on analytic cache models [1].

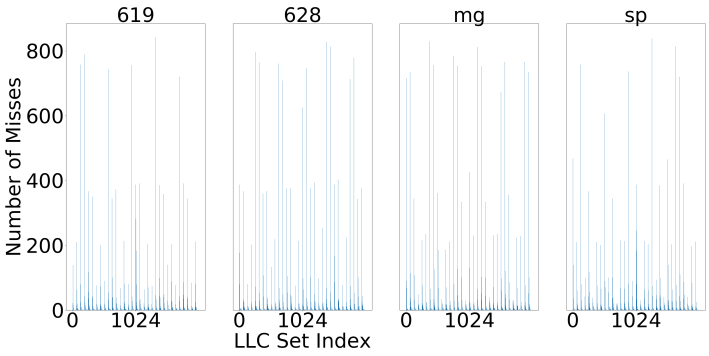
However, if hardware prefetchers are enabled, we obtain different cache miss distributions for *PS* and *NPS* applications, as illustrated in Figure 6.4. Note that the scale of the y-axes in Figure 6.4a and Figure 6.4b are different.

As shown in Figure 6.4a, the cache miss distributions over cache sets are non-uniform for *PS* applications. It is clear that cache sets associated with spikes exhibit many more (more than $10\times$) cache misses than other sets. In most cases, the index of those sets is $64p$ with $p = 1, 2, 3, ...$, where the beginning of a new virtual page is mapped to. From this, we infer that cache misses at those sets are caused by the first references to the data in a new virtual memory page.

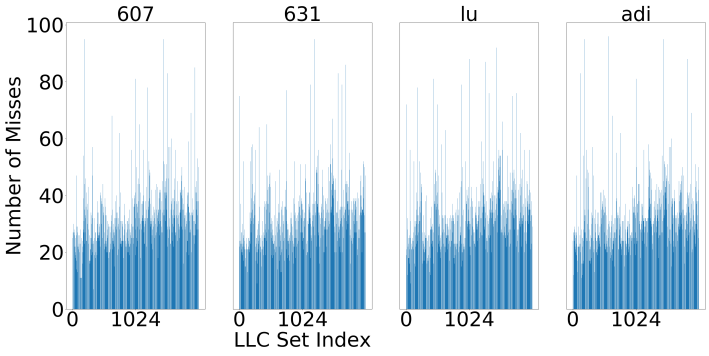
Figure 6.4b depicts the distributions of missed cache sets for *NPS* applications when hardware prefetchers are enabled. Although there exist a few cache sets with spikes, the gap between the spikes and the average number of misses over a cache set is much smaller. Overall, the cache misses are still uniformly distributed over all the cache sets. Thus, we make the following observation:

Observation 6.4. When hardware prefetchers are enabled, cache miss distributions over cache sets are non-uniform for PS applications, while the distributions are mostly uniform for NPS applications.

Based on the difference in cache miss distributions between *PS* and *NPS* applications when hardware prefetchers are enabled, we propose a ratio between the maximum value



(a) *PS* applications



(b) *NPS* applications

Figure 6.4: Histogram of missed cache sets when hardware prefetchers are enabled.

and the median value of the frequency of LLC misses exhibited by one cache set to determine whether an application is *PS* or not. To reduce the complexity, the median value is approximately computed as the average of LLC misses exhibited by 30 randomly selected cache sets. We skip selecting the cache sets that exhibit misses more than 70% of the maximum value. When the ratio is larger than a threshold (10, in this work), the application is classified as a *PS* application. Otherwise, it is considered to be an *NPS* application.

Obtaining cache miss distribution

As described, the cache miss distribution over the cache sets can be obtained by following these steps: virtual addresses that missed in the LLC can be obtained by using the PMU sampling mechanism, after which each obtained virtual address needs to be translated to the corresponding physical data address to determine the missed LLC cache set. By sampling the LLC misses over a short execution period, one can obtain the cache miss distribution. We describe those steps in details below.

PMU sampling. Intel PEBS supports address sampling, a type of event-based sampling that allows associating sampled performance events with instruction pointers (IP) and effective data addresses. Moreover, PEBS address sampling in recent Intel processors (i.e., Haswell and its successors) allows precisely monitoring cache misses at memory level. In this work, we choose the event `MEM_LOAD_UOPS_RETIRED:L3_MISS` to drive PMU sampling. After experimenting with different sampling periods ranging from 5 (i.e., every 5th miss) to 1000, we decided to use a sampling period of 10, as it incurs a small overhead while still providing enough samples for the later analysis. In this configuration, the PMU therefore samples one per ten data addresses that missed in LLC. Note that the sampled data addresses are virtual addresses.

Virtual-to-physical address translation. As LLCs are physically indexed and physically tagged (PIPT), a virtual address obtained from a PMU sample does not suffice to get the information about the missed LLC set. Therefore, a virtual-to-physical address translation is required. This translation can be done by using `Pagemap`, a set of interfaces in the Linux kernel that allow user space programs to examine the page tables and related information.

Since the default page size of most Linux systems in the virtual address space is 4K bytes, during the virtual-to-physical address translation, bits $0 - 11$ of the virtual address that encode the page offset are preserved. Bits 12 and above of the virtual address, which encode the page number in the virtual address space, are replaced by the physical page frame number. The mapping from the virtual page to the physical page frame can be found in `/proc/self/pagemap`, a component in `Pagemap`.

LLC addressing. The LLC in a modern multicore processor is usually organized into as many slices as the number of cores with the purpose of reducing the bandwidth bottleneck when more than one core attempts to retrieve data from the LLC at the same time.

Typically, the LLC is set-associative, with a total of k cache sets in each cache slice and m ways. A cache line with a size of c bytes occupies a single way of a cache set. The slice and cache set to which a physical memory address maps is determined by its address bits, as shown in Figure 6.5.

As indicated in [44], the least significant $\log_2 c$ bits of the physical address are used to address a byte or word within a cache line. The next $\log_2 k$ bits select the set that the cache line belongs to. Bits $\log_2 k + \log_2 c$ and above are utilized as a tag for comparison when looking for data in the cache. The Intel processors use an undocumented hash function of higher bits (bits $\log_2 k + \log_2 c$ and above) of a physical address to decide the cache slice.

In the absence of knowledge about the hash function used for mapping, a given cache line can be present in any of the slices. As cache miss behavior in different cache slices is very similar, in this work, we do not distinguish the cache lines in different cache slices.

Histogram of missed cache sets. The histogram of missed cache sets can be derived by sampling the LLC misses for a short execution period and calculating the missed cache set that corresponds to each sampled miss. We have set the sampling period to 1 second in this work.

The proposed detection approach is accurate and able to detect all the *PS* applications in the benchmarks used in this study, even when they co-run with 10 other applications.

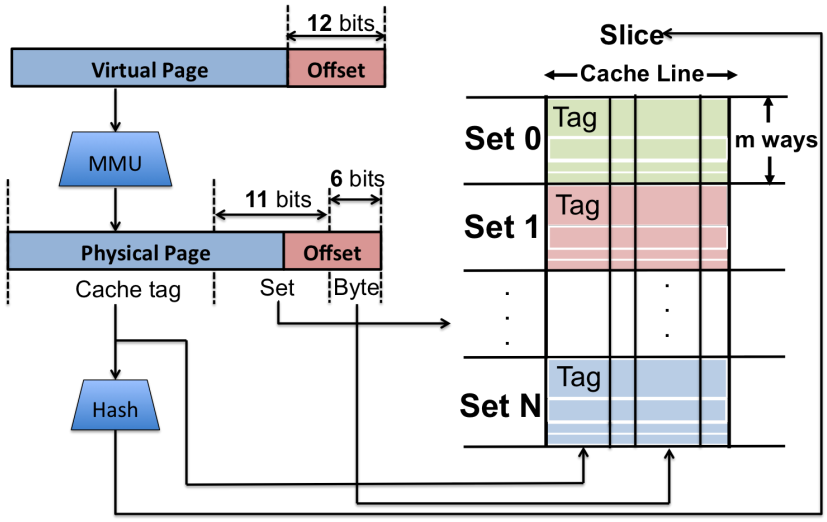


Figure 6.5: LLC addressing. A virtual data address is translated to a physical data address by the memory management unit (MMU). For a typical caching system with $k = 2048$, $c = 64$, the lowest 6 bits (bits $0 - 5$) are used to determine the offset within a cache line and bits $6 - 16$ select the cache set. Higher bits (bits 17 and above) are used as tag and input to a hash function to decide the cache slice.

6.3.2 LLC partitioning for *PS* and *NPS* applications

Most of the *PS* applications are memory-intensive. When *PS* applications run simultaneously with *NPS* applications fully sharing the LLC, we observe that *PS* applications often occupy more LLC space than the *NPS* applications, leading to significant performance degradation of *NPS* applications. We will show such a scenario in the next section.

One of the reasons *PS* applications can occupy more LLC space is that PS applications can generate a large number of prefetching requests. As observed in [86], applications that gain more benefit from hardware prefetching tend to generate more prefetch requests.

When the hardware prefetchers are enabled, we observed in Section 6.2 that the effective LLC size has only very limited effect on the performance of *PS* applications. The aim of the cache partitioning in this work is therefore to limit the LLC size occupied by *PS* applications and reserve more LLC space for *NPS* applications. By doing so, the potential prefetch-related cache pollution for *NPS* applications is also mitigated.

Our cache partitioning scheme is simple: it initially allocates one exclusive cache-way to each newly classified *PS* application as it does not benefit greatly from a larger LLC size. It then allocates the remainder of the cache-ways to the *NPS* applications. When a *PS* application finishes its execution, the exclusive cache-way that was previously owned by that application is assigned to the *NPS* applications.

We also observed that the performance of *PS* applications degrades only slightly

even when multiple of such applications share a single way of the LLC. If multiple *PS* applications are present, we randomly select two among these applications to share the same way for a short time interval (0.1 second, in this work), We repeat the dynamic adjusting of one shared way for two randomly selected *PS* applications for up to 10 times, each time measuring the IPC of all co-running applications. When the repetition finishes, we keep the best CAT configuration with the maximum sum of IPC of all co-running applications.

Note that, in this work we only focus on LLC partitioning between *PS* and *NPS* applications. Further improvement can be achieved by LLC partitioning among *NPS* applications, as has already been done in [30, 79, 106].

6.4 Experiments

The prototype of CP$_{pf}$ is implemented as a user-level runtime system on Linux. This section evaluates the performance of CP$_{pf}$. The experiment platform is described in Section 6.1.1. It has 376GB of main memory and the maximum memory bandwidth is 119.21 GB/s, so the memory contention will be small. Hyperthreading is disabled to avoid intra-core interference. All of the hardware prefetchers are kept enabled during the experiments.

Single-threaded workload mixes: The experiments have been conducted with more than 200 workload mixes from the SPEC 2017 [84], NPB [65] and Polybench [70] benchmark suites. We select three representative sets of 50 multiprogram mixes. The first set contains ten 2-application workloads with index $W0 - W9$, the second set twenty 4-application workloads with index $W10 - W29$ and the third set has twenty 8-application workloads with index $W30 - W49$. Though we would have liked to go beyond 8-application workloads, CAT in our tested platform can only support at most 11 CLOSs.

In each set, the workload mixes were randomly generated by varying the ratio of *PS* applications (25%, 50% and 75%). The proportions of *PS* applications in each workload mix are listed in Table 6.3. For each workload mix, performance is measured by executing each application until all the applications have completed the same number of instructions they execute when running alone for 20 seconds. The applications are pinned to cores to facilitate the performance monitoring and cache partitioning.

Table 6.3: Composition of workload mixes.

PS applications (%)	Workloads Index
25%	$W10 - W15$, $W30 - W36$
50%	$W0 - W9$, $W16 - W22$, $W37 - W41$
75%	$W23 - W29$, $W42 - W49$

Metrics: We measure system performance using the average speedup, calculated as

follows for the workload with a mix of N applications:

$$AverageSpeedup = \frac{1}{N}\sum_{i=1}^{N}\frac{IPC_{i,CP_{pf}}}{IPC_{i,FullShare}}$$

where $IPC_{i,FullShare}$ is the IPC of program i measured in the baseline configuration, in which the LLC is unpartitioned and is fully shared among all the application; $IPC_{i,CP_{pf}}$ is the IPC of program i obtained when CP$_{pf}$ is applied.

6.4.1 CP$_{pf}$ performance gain

Figure 6.6 summarizes the performance gained by CP$_{pf}$ for the workload mixes composed of single-threaded applications. Note that, in Figure 6.6, workload mixes having the same number of applications and same proportions of *PS* applications are sorted by their speedups. Compared with the baseline performance, CP$_{pf}$ improves the performance for 45 out of 50 workload mixes.

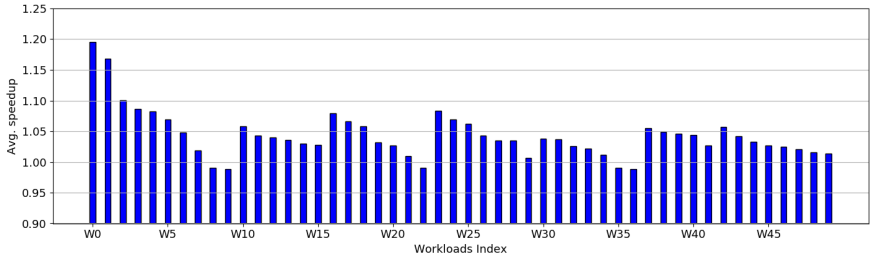


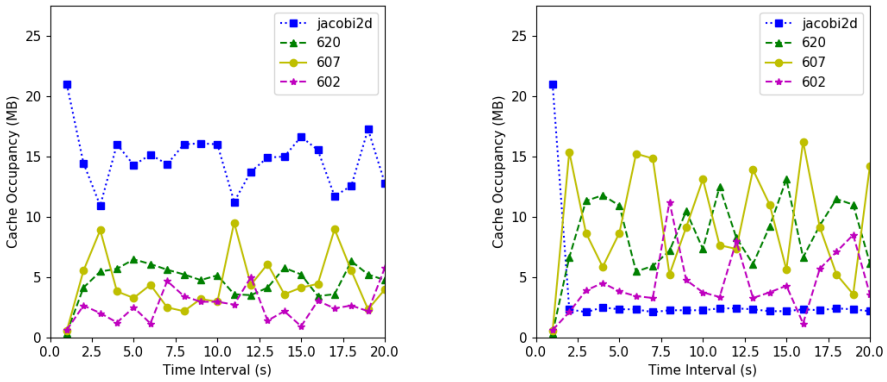
Figure 6.6: Average speedup for each single-threaded workload mix.

CP$_{pf}$ achieves a speedup of 1.08 on average for workloads with 2 applications, with a best case speedup of 1.20. The average speedup for workloads with 4 applications is 1.04 with a best case of 1.08. Finally, for workloads with 8 applications, the average speedup is 1.03, with a best case of 1.06.

6.4.2 Cases study of CP$_{pf}$

We now take a closer look at representative workload mix $W11$ consisting of four applications (i.e. $jacobi2d, 620, 607, 602$) to better understand how CP$_{pf}$ can improve the overall system performance.

Figure 6.7a and Figure 6.7b illustrate the run-time cache occupancy of the 4 applications in case the LLC is fully shared and CP$_{pf}$ is applied, respectively, during a 20 seconds time interval. When the LLC is fully shared (Figure 6.7a), the *PS* application $jacobi2d$ occupies more than half of the LLC space for most of the time. As a result, *NPS* applications 620, 607, 602 get less LLC space. This situation is improved by CP$_{pf}$. Once CP$_{pf}$ has identified $jacobi2d$ as the only *PS* application, it allocates only one way to $jacobi2d$, leaving the rest of the LLC shared by the *NPS* applications 620, 607, 602, as depicted in Figure 6.7b. In this case, CP$_{pf}$ achieves a 1.10, 1.02 and 1.06 speedup



(a) when LLC is shared

(b) CP$_{pf}$ is applied

Figure 6.7: Dynamic cache occupancy by applications in workload $W11$.

for 620, 607 and 602, respectively, while the speedup of $jacobi$ is 0.99. At the cost of a small slowdown of *PS* applications, CP$_{pf}$ yields a higher speedup for *NPS* applications.

We also take a look at one the the workload mixes that exhibits a performance degradation under CP$_{pf}$: $W9$. $W9$ is composed of cg (the *PS* application) and 641 (the *NPS* application). The performance of 641 cannot be improved enough by getting more cache space, in this case, the speedup of 641 is 1.001. The performance of cg is degraded by 0.975 as CP$_{pf}$ allocates a one-way cache space to cg. However, no significant performance losses are observed as the lowest speedup (i.e., slowdown) is 0.988.

6.4.3 CP$_{pf}$ with multithreaded workloads

CP$_{pf}$ also supports multithreaded workloads. For multithreaded workloads, cache miss distributions are obtained per thread, and the LLC is partitioned per thread.

We generate two sets of totally 30 multithreaded workload mixes. Each workload mix consists of two multithreaded applications, one randomly selected from PARSEC [15] or SPLASH [101] as an *NPS* application, and the other from NPB [65] or an OpenMP version of Polybench [70] as a *PS* application (we skip applications from SPEC CPU2017 [84] as it provides multithreaded implementations for a very limited number of applications). The first set contains fifty 4-threaded workloads with index $W50-W64$ and the second set has fifty 8-threaded workloads with index $W65-W79$.

Figure 6.8 presents the average speedups for the multithreaded workload sets. Compared with the baseline performance where caches are fully shared among all the threads, CP$_{pf}$ achieves a speedup of 1.05 on average for workloads with 4 threads, with a best case speedup of 1.22. The average speedup for workloads with 8 threads is 1.04 with a best case of 1.11.

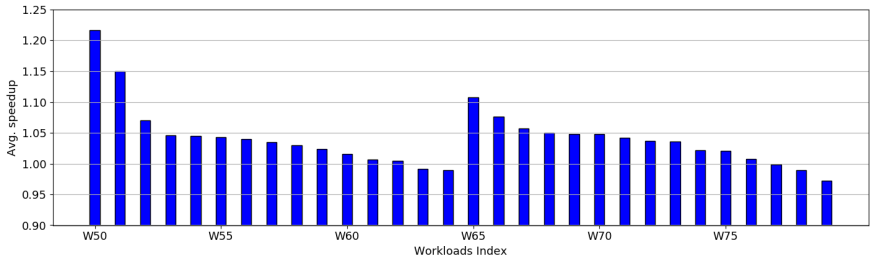


Figure 6.8: Average speedup for each multithreaded workload mix.

6.4.4 Sensitivity Analysis

We now analyze CP$_{pf}$'s sensitivity to the characteristics of the workload mix, particularly the ratio between *PS* and *NPS* applications and the workload mix size.

The effect of workload distribution. CP$_{pf}$ achieves average speedups of 1.03 , 1.06 and 1.04 for workloads with 25%, 50% and 75% of *PS* applications.

When the workload mixes are dominated by *PS* applications, the performance improvement due to an increased LLC space allocated for *NPS* applications by CP$_{pf}$ will be limited by the small number of *NPS* applications in the workload mixes.

When workload mixes are dominated by *NPS* applications, the benefits of CP$_{pf}$ also become more muted. This is because, as indicated in Section 6.3.2, CP$_{pf}$ does not partition the cache among the *NPS* applications. Even though the cache space occupied by *PS* applications is limited, most of the rest of the cache can be occupied by *NPS* applications whose performance will not be improved greatly by getting more cache space. CP$_{pf}$ cannot guarantee that those *NPS* applications whose performance significantly improves from a larger effective cache size will always occupy more cache space than other applications.

The effect of workload size. We compare the performance gained by CP_{pf} under different sizes of workload mixes (ranging from 2 to 8 applications per workload mix). CP_{pf} gains less performance when the number of co-executing applications increases. This is inevitable because cache contention for both LLC space and cache set associativity is increased as more applications share the LLC.

6.4.5 Overhead

In order to obtain the actual performance degradation that CP$_{pf}$ results in, we compare the execution times of the applications in SPEC CPU2017, NPB and Polybench benchmarks with two settings, CP$_{pf}$ off and CP$_{pf}$ on. The results show that CP$_{pf}$ causes 0.56% slowdown on average with a worst case of 1.72%.

The overhead of CP$_{pf}$ mainly comes from the online classification of *PS* and *NPS* applications at run time. For an execution phase which typically lasts more than 30 seconds, the PMU samples LLC misses for only 1 second, during which on average 30% of the time is dedicated to PMU sampling and virtual-physical address translation. As PMU can sample up to 200000 data addresses in 1 second, it takes up to 100

milliseconds to obtain the miss distribution over cache sets.

6.5 Conclusion

Hardware prefetching can interact poorly with LLC management, leading to performance degradation. To study the interaction between hardware prefetching and LLC cache management, we analyzed the variation of application performance when varying the effective LLC space in the presence and absence of hardware prefetching. We observed that hardware prefetching can compensate the application performance loss due to the reduced effective cache space. Motivated by this observation, we classified applications into two categories, prefetching sensitive (*PS*) and non prefetching sensitive (*NPS*) applications, by the degree of performance benefit they experience from hardware prefetchers. To address the cache contention and to also mitigate the potential prefetch-related cache interference, we proposed CP$_{pf}$, a prefetch aware cache partitioning approach for improving the LLC management in the presence of hardware prefetching. CP$_{pf}$ consists of a method using PEBS techniques for the online classification of *PS* and *NPS* applications and a LLC partitioning scheme via CAT to distribute the cache space among *PS* and *NPS* applications. We have implemented CP$_{pf}$ as a user-level runtime system on Linux. Compared with a non-partitioning approach, CP$_{pf}$ achieves speedups of up to 1.20 (1.08 on average), 1.08 (1.04 on average) and 1.06 (1.03 on average) for workloads with 2, 4 and 8 single-threaded applications, respectively. Moreover, it achieves speedups of up to 1.22 (1.05 on average) and 1.11 (1.04 on average) for workload mixes composed of two applications with 4 threads and 8 threads, respectively.

7 Conclusions

The research presented in this dissertation revolves around two research themes: improving timing predictability and caching performance for multi-core systems. The four research chapters of this thesis address the challenges as follows. First, in Chapter 3, we focused on how to improve the timing predictability and evaluate the timing performance for real-time multi-core system by a simulation and modeling approach. In particular, we developed SysRT, a generic and modular high-level RTOS simulator that is highly suited for early DSE to study RTOS design alternatives. Second, in Chapter 4, we study the timing predictability of multi-core systems by an analytical approach. We developed a novel schedulability analysis of real-time global scheduling for multi-core systems with shared caches. We proposed an approach to calculating the upper bound on shared cache interference for tasks that are globally scheduled. By extending the approach to the partitioned scheduling paradigm, in Chapter 5, we developed a cache-interference aware task partitioning algorithm. Finally, in Chapter 6, we focused on high performance caching. Motivated by the study on the interaction between hardware prefetching and LLC management, we proposed CP_{pf}, a prefetch aware cache partitioning approach for improving the LLC management in the presence of hardware prefetching.

Below, we provide a more detailed summary of the contributions and results of our research, and answer the research questions set out in Chapter 1 of the dissertation. We conclude with an outlook on future research directions.

7.1 Main findings

Modeling and simulation of real-time embedded systems

We addressed the research problems related to the modeling and simulation of real-time embedded systems. Specifically, our research questions in this study are:

RQ1 How to provide fast simulation of real-time embedded systems for design space exploration at the early stages of system design? How to accurately capture the timing behaviour of embedded software? How to efficiently implement the simulator to provide support for easy plug-in of new task models, new schedulers and new resource sharing protocols?

To answer these questions, we developed SysRT [92], a generic and modular high-level RTOS simulator that is highly suited for early DSE to study embedded RTOS design alternatives. SysRT contains different types of application models, an RTOS kernel model and an abstract architecture model. The kernel model is developed to be generic and modular to support for easy plug-in of new schedulers as well as new resource sharing protocols.

To precisely model the timing in RTOS, it is necessary to model preemption. SysRT adopts an event-driven simulation approach that utilizes scheduling events associated with task states and interrupts, to achieve efficient and precise modeling of preemptive scheduling. Events are extracted from the task execution states, for example, a *job_arrival* and a *job_end* event are posted at the time instance when a task becomes ready to execute and when a task is supposed to finish, respectively. *Schedule* and *deschedule* are the scheduling events. Once a task is preempted, job_end event is cancelled. When this task is scheduled again, a new task finishing event is posted after the remaining execution time. compared with quantum-granularity based and prediction-based simulation approaches, this event-driven simulation approach introduces less simulation overhead.

SysRT implements a set of interfaces to model the services of process management, resource management, interrupt handling and real-time scheduling provided by the OS kernel. For example, *Arrival(AbsRTTask* t)*, *Suspend(AbsRTTask* t)* and *Activate(AbsRTTask* t)* are invoked in the kernel module to schedule, suspend and reactivate a task, respectively. Those generic interfaces can be reused in the different implementation of specific schedulers.

The kernel module is developed to be modular. The *UNPKernel* module is developed to model a real-time OS kernel running on a uniprocessor. The *SMPKernel* is a module modeling a real-time kernel with a global scheduler for (SMP) multiprocessor systems. Since the structure of such a partitioned scheduler is different from the global scheduler, a different kernel module, *PartiKernel*, has been implemented to facilitate the development of partitioned schedulers. All the kernel modules contain a *Scheduler* module in which different schedulers are implemented. In the current version of SysRT, a number of schedulers have currently been implemented including First Come First Out, Round Robin, Fixed Priority Scheduler, Proportional Fairness, global and partitioned Earliest Deadline First and so on.

The kernel also contains the *ResManager* module that models the management of shared software resources such as shared variables. In the *ResManager* module, a set of interfaces are implemented to perform access to shared software resources. For example, *lock(int amount)/unlock(int amount)* is used to decrease/increase resource availability for a particular resource. Based on those generic interfaces, a number of resource accessing protocols such as Non-Preemptive Protocol and Priority Inheritance Protocol are implemented.

A set of experiments are performed to evaluate the accuracy and simulation performance of SysRT by comparing it with four the state-of-art simulators. It has been shown that SysRT typically achieves higher simulation speeds while obtaining identical accuracy results.

The flexibility of SysRT has been demonstrated by simulating real-time resource access protocols. We also showed the benefits of SysRT for early design space exploration

by simulating an embedded system with a mixed application workload consisting of hard real-time tasks, soft real-time tasks and best-effort tasks using different schedulers.

Schedulability analysis of real-time multi-core systems

We performed research on the schedulability analysis of real-time scheduling for multi-core systems with shared caches. In particular, we conducted analysis on both global and partitioned scheduling. Regarding global scheduling, our research questions are:

RQ2 Is it possible to derive an upper bound on shared cache interference between two tasks running simultaneously on a multi-core system? Given a real-time taskset globally scheduled by EDF or FP, how to obtain an upper bound on the shared cache interference exhibited by each task in the taskset? How to derive a schedulable condition for the globally scheduled taskset, accounting for the shared cache interference?

Shared caches in multi-core processors introduce serious difficulties in providing guarantees on the real-time properties of embedded software due to the interaction and the resulting contention in the shared caches. To answer this question, we develop a new schedulability analysis for real-time multi-core systems with shared caches, globally scheduled by Earliest Deadline First (EDF) and Fixed Priority (FP) algorithms.

In the new schedulability analysis, we defined the problem window for the interfered task. In the problem window, a job of task exhibits two kinds of interference. The first interference is processor-contention interference which is the cumulative length of all intervals over in which all the processing cores are busy executing interfering jobs other than interfered jobs. The second type of interference is shared cache interference which is the cumulative length of all extra execution delays caused by cache sharing.

Based on the previous work, we computed an upper bound on the processor-contention interference.

We came up with an approach to derive an upper bound on shared cache interference a task may exhibit during an job execution. To do so, we first analyze the cache interference during one job execution between two tasks by performing a Cache Access Classification and Cache Hit/Miss Classification analysis for each instruction memory access of the two tasks. Let τ_i be the interfering task and τ_k be the interfered task, Lemma 4.6 in Chapter 4 gives an upper bound on cache interference for τ_k imposed by only one job of τ_i.

We compute an upper bound of the maximum cache interference a task may exhibit during an execution window by introducing an Integer Programming formulation, which can be transformed to an integer linear programming formulation. The objective function of the IP formulation is to maximize the sum of the contributions of all interfering tasks τ_i in the task set τ. Three constraints on the number of jobs from each interfering task that can interfere with τ_k are derived. The optimal solution is the upper bound on the cache interference the task τ_i may exhibit during an execution window.

Due to the presence of cache interference, a job may execute longer than its worst-case execution time on a multi-core platform with shared caches. Based on the observation that a larger execution time may introduce more cache interference, we derive an

iterative algorithm to refine the bound until a fixed-point is reached. The fixed point is an upper bound on cache interference during a job execution.

The upper bound on shared cache interference, together with the upper bound on processor contention are subsequently integrated into the schedulability analysis to derive a new schedulability condition for the globally scheduled taskset, which is stated in Theorem 4.11 in Chapter 4.

Using our proposed schedulability analysis, a set of experiments has been performed to investigate how the schedulability is degraded by shared cache interference, varying various factors such as the probability of two tasks having cache interference and the amount of cache interference between two tasks.

We also compared the schedulability performance of non-preemptive EDF (EDF_{np}) against non-preemptive FP (FP_{np}) in the presence of cache interference. Our empirical evaluations showed that EDF_{np} is better than FP_{np} in terms of tasksets deemed schedulable.

Regarding partitioned scheduling, our research questions are:

RQ3 How to develop a cache interference aware partitioned scheduling for real-time multi-core systems? Is the partitioned scheduling better than global scheduling in terms of schedulability performance?

As most commodity processors in the embedded domain does not provide support for cache partitioning, we do not deploy any cache partitioning techniques to mitigate the inter-core cache interference. Instead, we address the problem of task partitioning in the presence of shared cache interference.

To answer these questions, we proposed CA-TPAR, a cache-interference aware task partitioning algorithm. Tasks are sorted by means of a certain criterion, for example, by tasks' relative deadline. CA-TPAR then assigns the tasks to cores. Each core, and the tasks assigned to it, are scheduled at run time by an non-preemptive EDF scheduler.

We extended the approach presented in Chapter 4 to calculate the upper bound on cache interference exhibited by a task for the partitioned scheduling. We proved that the bound on cache interference exhibited by a task under paritioned scheduling can not be larger than the bound for that task under global scheduling. We conducted schedulability analysis of CA-TPAR using demand bound function and formally proved its correctness.

A set of experiments was performed to evaluate the schedulability performance of CA-TPAR against global EDF scheduling over randomly generated tasksets. Our empirical evaluations show that CA-TPAR outperforms global EDF scheduling in terms of task sets deemed schedulable.

Prefetch-aware cache partitioning for high performance caching

We exploited the opportunity to improve caching performance by a prefetching-aware cache partitioning approach. Our research questions are:

RQ4 How does hardware prefetching affect the caching performance? How to manage shared caches to improve system performance in the presence of hardware prefetching?

To answer these questions, firstly we analyzed the variation of application performance when varying the effective LLC space in the presence and absence of of hardware prefetching. We observed that hardware prefetching can compensate the application performance loss due to the reduced effective cache space. Motivated by this observation, we classified applications into two categories, prefetching sensitive (*PS*) and non prefetching sensitive (*NPS*) applications, by the degree of performance benefit they experience from hardware prefetchers. We define applications whose performance is significantly improved by hardware prefetching as prefetching sensitive (PS) applications.

We studied the cache sensitivity of *PS* and *NPS* applications by conducting several experiments in which we used CAT to adjust the number of LLC ways available to the application. We observed that if hardware prefetchers are enabled, on average,the effective LLC size has a relatively small influence on the performance of PS applications, while the performance of NPS applications can be significantly affected by the effective LLC size.

To address the cache contention among *PS* and *NPS* applications and to also mitigate the potential prefetch-related cache interference, we proposed CP_{pf}, a prefetch aware cache partitioning approach for improving the LLC management in the presence of hardware prefetching. CP_{pf} consists of a method using PEBS techniques for the online classification of *PS* and *NPS* applications and a LLC partitioning scheme via CAT to distribute the cache space among *PS* and *NPS* applications. The proposed prefetch aware cache partitioning approach is a software-only solution by using hardware features like PEBS and CAT, which are readily available in existing multi-core processors.

The non-trivial solution for the online classification of *PS* and *NPS* applications is based on the distribution of cache misses over the cache sets. The idea comes from the fact that prefetched data will always be within the same 4K bytes memory page as the load instruction that triggered the prefetching and prefetchers do not prefetch across virtual page boundaries.

A set of experiments are performed to obtain the cache miss distribution for both *PS* and *NPS* applications. As observed, when hardware prefetchers are disabled, cache misses are mostly uniformly distributed over all the LLC sets for both *PS* and *NPS* applications. While when hardware prefetchers are enabled, cache miss distributions over cache sets are non-uniform for PS applications, while the distributions are mostly uniform for NPS applications. One can distinguish *PS* and *NPS* applications by their differences in cache miss distributions when hardware prefetchers are enabled.

Our LLC partitioning scheme is simple: it allocates one exclusive cache-way to each newly classified *PS* application as it does not benefit greatly from a larger LLC size. It then allocates the remainder of the cache-ways to the *NPS* applications.

We have implemented CP_{pf} as a user-level runtime system on Linux. Compared with a non-partitioning approach, CP_{pf} achieves speedups of up to 1.20 (1.08 on average), 1.08 (1.04 on average) and 1.06 (1.03 on average) for workloads with 2, 4 and 8 single-threaded applications, respectively. Moreover, it achieves speedups of up to 1.22 (1.05 on average) and 1.11 (1.04 on average) for workloads mixes composed of two applications with 4 threads and 8 threads, respectively.

7.2 Future work

As described in the previous four chapters, the research presented in this dissertation has addressed four research problems in two different domains: timing predictability of embedded computing and caching performance for high performance computing. In this section we lay out some future research directions.

In Chapter 4, we proposed an approach to calculating the upper bound on cache interference a task may exhibit during a job execution. We plan to perform evaluations on real benchmarks to show the pessimism of upper bounding the shared cache interference using the proposed method.

In Chapter 4 and 5, we addressed the schedulability analysis of real-time scheduling for multi-core systems with shared caches. Cache partitioning is a widely used technique to isolate the accesses to shared caches to prevent cache interference. However, if cache partitioning is deployed, tasks' WCET can be larger due to the reduced available caches during tasks' execution. Therefore, it would be very interesting to compare the schedulability performance of real time scheduling in which a cache is fully shared among all tasks against the case where the cache is partitioned to tasks.

In Chapter 6, we presented CP_{pf}, a prefetch aware cache partitioning approach to improve the average speedup for all applications in the system. We plan to apply CP_{pf} for other purposes such as to improve fairness and to provide quality-of-service guarantees. For example, we plan to extend CP_{pf} to the cache management in data centers where latency critical applications are co-located with throughput oriented applications. The goal of cache management is to provide guarantees with respect to the latency while also maximizing the throughput of the other applications.

Finally, cache-related problems have attracted a lot of attention from different research communities. This dissertation deals with the cache issues in real time computing and high performance computing. It is interesting to investigate the possibilities to extend the presented tools, analyses, observations to improving the design of secure caching in the computer security domain.

Bibliography

[1] A. Agarwal, J. Hennessy, and M. Horowitz. An analytical cache model. *ACM Transactions on Computer Systems (TOCS)*, 7(2):184–215, 1989. (Cited on page 98.)

[2] K. Albers and F. Slomka. An event stream driven approximation for the analysis of real-time systems. In *Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004.*, pages 187–195. IEEE, 2004. (Cited on page 74.)

[3] D. H. Albonesi. Selective cache ways: On-demand cache resource allocation. In *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 248–259. IEEE, 1999. (Cited on page 15.)

[4] 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 *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, pages 129–138. ACM, 2015. (Cited on page 46.)

[5] N. C. Audsley, A. Burns, R. I. Davis, K. W. Tindell, and A. J. Wellings. Fixed priority pre-emptive scheduling: An historical perspective. *Real-Time Systems*, 8(2-3):173–198, 1995. (Cited on page 19.)

[6] T. P. Baker. Multiprocessor edf and deadline monotonic schedulability analysis. In *RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003*, pages 120–129. IEEE, 2003. (Cited on page 49.)

[7] S. Baruah. The limited-preemption uniprocessor scheduling of sporadic task systems. In *17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, pages 137–144. IEEE, 2005. (Cited on page 74.)

[8] S. Baruah. Techniques for multiprocessor global schedulability analysis. In *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pages 119–128. IEEE, 2007. (Cited on pages 4, 45, and 56.)

[9] S. K. Baruah, A. K. Mok, and L. E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *In Proceedings of the 11th Real-Time Systems Symposium*, pages 182–190. IEEE Computer Society Press, 1990. (Cited on page 74.)

[10] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *ALGORITHMICA*, 15:600–625, 1996. doi: 10.1.1.123.2848. (Cited on page 36.)

[11] A. Bastoni, B. B. Brandenburg, and J. H. Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers. In *2010 31st IEEE Real-Time Systems Symposium*, pages 14–24. IEEE, 2010. (Cited on pages 5 and 71.)

[12] N. Beckmann and D. Sanchez. Jigsaw: Scalable software-defined caches. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 213–224. IEEE, 2013. (Cited on page 15.)

[13] E. Berg, H. Zeffer, and E. Hagersten. A statistical multiprocessor cache model. In *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 89–99. IEEE, 2006. (Cited on page 43.)

[14] M. Bertogna, M. Cirinei, and G. Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Transactions on parallel and distributed systems*, 20(4):553–566, 2008. (Cited on pages 4 and 45.)

[15] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008. (Cited on page 104.)

[16] J. Brock, C. Ye, C. Ding, Y. Li, X. Wang, and Y. Luo. Optimal cache partition-sharing. In *2015 44th International Conference on Parallel Processing*, pages 749–758. IEEE, 2015. (Cited on page 92.)

[17] A. Burns and A. J. Wellings. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Pearson Education, 2001. (Cited on page 19.)

[18] G. C. Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011. (Cited on pages 18, 19, and 22.)

[19] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. Litmusrt: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proc. of the 27th IEEE Real-Time Systems Symposium*, pages 111–123, 2006. doi: 10.1.1.92.3483. (Cited on page 37.)

[20] S. Chiaradonna, F. D. Giandomenico, and J. Xiao. Quantification of the effectiveness of medium voltage control policies in smart grids. In *2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE)*, pages 284–291, Jan 2016. doi: 10.1109/HASE.2016.42. (Cited on page 9.)

[21] D. Chiou, P. Jain, L. Rudolph, and S. Devadas. Application-specific memory management for

embedded systems using software-controlled caches. In *Proceedings of the 37th Annual Design Automation Conference*, pages 416–419. ACM, 2000. (Cited on page 15.)

[22] H. Cho, B. Ravindran, and E. D. Jensen. An optimal real-time scheduling algorithm for multiprocessors. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 101–110. IEEE, 2006. (Cited on pages 4 and 45.)

[23] K. L. Clarkson. Las vegas algorithms for linear and integer programming when the dimension is small. *Journal of the ACM (JACM)*, 42(2):488–499, 1995. (Cited on pages 61 and 81.)

[24] I. Corporation. User space software for intel(r) resource director technology. *Available: https://github.com/intel/intel-cmt-cat*. (Cited on page 15.)

[25] I. Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 2018. (Cited on pages 17, 93, and 97.)

[26] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM computing surveys (CSUR)*, 43(4):35, 2011. (Cited on pages 4, 22, and 45.)

[27] P. developers. *perf_event_open - Linux man page*. URL `https://linux.die.net/man/2/perf_event_open`. (Cited on page 17.)

[28] A. M. Devices. *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 30h-3Fh Processors*, February, 2015. (Cited on page 17.)

[29] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Prefetch-aware shared resource management for multi-core systems. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 141–152. ACM, 2011. (Cited on page 92.)

[30] N. El-Sayed, A. Mukkara, P.-A. Tsai, H. Kasture, X. Ma, and D. Sanchez. Kpart: A hybrid cache partitioning-sharing technique for commodity multicores. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 104–117. IEEE, 2018. (Cited on pages 5, 91, and 102.)

[31] N. Fisher and S. Baruah. The partitioned multiprocessor scheduling of non-preemptive sporadic task systems. In *14th International conference on real-time and network systems*, 2006. (Cited on page 75.)

[32] L. George, P. Muhlethaler, and N. Rivierre. *Optimality and non-preemptive real-time scheduling revisited*. PhD thesis, INRIA, 1995. (Cited on page 73.)

[33] G. Gracioli and A. A. Fröhlich. An experimental evaluation of the cache partitioning impact on multicore real-time schedulers. In *2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 72–81. IEEE, 2013. (Cited on page 45.)

[34] N. Guan, W. Yi, Z. Gu, Q. Deng, and G. Yu. New schedulability test conditions for non-preemptive scheduling on multiprocessor platforms. In *2008 Real-Time Systems Symposium*, pages 137–146. IEEE, 2008. (Cited on page 49.)

[35] 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*, pages 245–254. ACM, 2009. (Cited on pages 4 and 45.)

[36] P. Guide. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part 2*, 2, 2011. (Cited on page 17.)

[37] Z. Guo, Y. Zhang, L. Wang, and Z. Zhang. Work-in-progress: Cache-aware partitioned edf scheduling for multi-core real-time systems. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, volume 1, pages 384–386, Dec 2017. (Cited on page 45.)

[38] D. Hardy and I. Puaut. Wcet analysis of multi-level non-inclusive set-associative instruction caches. In *2008 Real-Time Systems Symposium*, pages 456–466. IEEE, 2008. (Cited on pages 44, 46, and 56.)

[39] D. Hardy, T. Piquet, and I. Puaut. Using bypass to tighten wcet estimates for multi-core processors with shared instruction caches. In *2009 30th IEEE Real-Time Systems Symposium*, pages 68–77. IEEE, 2009. (Cited on page 45.)

[40] P. Hastono, S. Klaus, and S. A. Huss. Real-time operating system services for realistic systemc simulation models of embedded systems. In *FDL*, pages 380–392, 2004. (Cited on pages 3 and 27.)

[41] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011. (Cited on pages 11 and 12.)

[42] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer. Cache qos: From concept to reality in the intel® xeon® processor e5-2600 v3 product family. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 657–668. IEEE, 2016. (Cited on page 15.)

[43] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Transactions on Computers*, 38(12):1612–1630, 1989. (Cited on page 13.)

[44] G. Irazoqui, T. Eisenbarth, and B. Sunar. Systematic reverse engineering of cache slice selection in

intel processors. In *2015 Euromicro Conference on Digital System Design*, pages 629–636. IEEE, 2015. (Cited on page 100.)

[45] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of period and sporadic tasks. In *[1991] Proceedings Twelfth Real-Time Systems Symposium*, pages 129–139. IEEE, 1991. (Cited on page 73.)

[46] X. Jun and A. D. Pimentel. In *Design, Automation and Test in Europe Conference 2020, under review*. (Cited on pages 8 and 71.)

[47] Jun Xiao, S. Chiaradonna, F. Di Giandomenico, and A. Pimentel. Improving voltage control in mv smart grids. In *2016 IEEE International Conference on Smart Grid Communications (SmartGridComm)*, pages 382–387, Nov 2016. doi: 10.1109/SmartGridComm.2016.7778791. (Cited on page 9.)

[48] R. S. Khaligh and M. Radetzki. Modeling constructs and kernel for parallel simulation of accuracy adaptive tlms. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 1183–1188. IEEE, 2010. (Cited on pages 3 and 28.)

[49] H. Kim, A. Kandhalu, and R. Rajkumar. A coordinated approach for practical os-level cache management in multi-core real-time systems. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 80–89. IEEE, 2013. (Cited on pages 14 and 43.)

[50] R. Le Moigne, O. Pasquier, and J.-P. Calvez. A generic rtos model for real-time systems simulation with systemc. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, volume 3, pages 82–87. IEEE, 2004. (Cited on pages 3 and 27.)

[51] J. Lee, K. G. Shin, I. Shin, and A. Easwaran. Composition of schedulability analyses for real-time multiprocessor systems. *IEEE Transactions on Computers*, 64(4):941–954, 2014. (Cited on pages 4 and 45.)

[52] Y. Liang, H. Ding, T. Mitra, A. Roychoudhury, Y. Li, and V. Suhendra. Timing analysis of concurrent programs running on shared cache multi-cores. *Real-Time Systems*, 48(6):638–680, 2012. (Cited on page 45.)

[53] J. Liedtke, H. Hartig, and M. Hohmuth. Os-controlled cache predictability for real-time systems. In *Proceedings Third IEEE Real-Time Technology and Applications Symposium*, pages 213–224. IEEE, 1997. (Cited on page 45.)

[54] 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 *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 367–378. IEEE, 2008. (Cited on page 92.)

[55] C. L. Liu. Scheduling algorithms for multiprocessors in a hard real-time environment. *JPL Space Programs Summary, 1969*, 1969. (Cited on page 20.)

[56] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, Jan. 1973. ISSN 0004-5411. doi: 10.1145/321738.321743. URL `http://doi.acm.org/10.1145/321738.321743`. (Cited on pages 20 and 36.)

[57] D. Liu, N. Guan, J. Spasic, G. Chen, S. Liu, T. Stefanov, and W. Yi. Scheduling analysis of imprecise mixed-criticality real-time tasks. *IEEE Transactions on Computers*, 67(7):975–991, 2018. (Cited on pages 4 and 45.)

[58] F. Liu, A. Narayanan, and Q. Bai. Real-time systems. 2000. (Cited on page 22.)

[59] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving resource efficiency at scale. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 450–462. ACM, 2015. (Cited on pages 5 and 91.)

[60] J. M. López, J. L. Díaz, and D. F. García. Utilization bounds for edf scheduling on real-time multiprocessor systems. *Real-Time Systems*, 28(1):39–68, 2004. (Cited on page 75.)

[61] C. Maiza, H. Rihani, R. Concepcion, J. Maria, J. Goossens, S. Altmeyer, and D. Robert. A survey of timing verification techniques for multi-core real-time systems. Technical report, Verimag Research Report TR-2018-9 (Technical Report), 2018. (Cited on pages 4 and 23.)

[62] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 45–54. IEEE, 2013. (Cited on page 45.)

[63] R. Manikantan, K. Rajan, and R. Govindarajan. Probabilistic shared cache management (prism). In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 428–439. IEEE, 2012. (Cited on page 15.)

[64] S. Mittal. A survey of techniques for architecting tlbs. *Concurrency and Computation: Practice and Experience*, 29(10):e4061, 2017. (Cited on page 14.)

[65] NASA. *NAS Parallel Benchmarks 3.3*. URL `https://www.nas.nasa.gov/assets/npb/`.

(Cited on pages 94, 95, 102, and 104.)

[66] J. Park, S. Park, and W. Baek. Copart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers. In *Proceedings of the Fourteenth EuroSys Conference 2019*, page 10. ACM, 2019. (Cited on pages 5 and 92.)

[67] 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 *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279. IEEE, 2011. (Cited on page 45.)

[68] A. D. Pimentel. Exploring exploration: A tutorial introduction to embedded systems design space exploration. *IEEE Design & Test*, 34(1):77–90, 2016. (Cited on page 3.)

[69] L. Pons, V. Selfa, J. Sahuquillo, S. Petit, and J. Pons. Improving system turnaround time with intel cat by identifying llc critical applications. In *European Conference on Parallel Processing*, pages 603–615. Springer, 2018. (Cited on pages 5 and 91.)

[70] L. Pouchet and T. Yuki. Polybench/c 4.1. *https://sourceforge.net/projects/polybench*, 2015. (Cited on pages 94, 95, 102, and 104.)

[71] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 423–432. IEEE, 2006. (Cited on page 91.)

[72] P. Ranganathan, S. Adve, and N. P. Jouppi. *Reconfigurable caches and their application to media processing*, volume 28. ACM, 2000. (Cited on page 15.)

[73] P. Razaghi and A. Gerstlauer. Host-compiled multicore system simulation for early real-time performance evaluation. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(5s):166, 2014. (Cited on pages 3, 28, and 37.)

[74] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, 2007. (Cited on page 14.)

[75] A. Saifullah, K. Agrawal, C. Lu, and C. Gill. Multi-core real-time scheduling for generalized parallel task models. In *Proceedings of the 2011 IEEE 32Nd Real-Time Systems Symposium*, RTSS '11, pages 217–226, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4591-2. doi: 10.1109/RTSS.2011.27. URL `http://dx.doi.org/10.1109/RTSS.2011.27`. (Cited on page 30.)

[76] D. Sanchez and C. Kozyrakis. The zcache: Decoupling ways and associativity. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 187–198. IEEE, 2010. (Cited on page 15.)

[77] D. Sanchez and C. Kozyrakis. Vantage: scalable and efficient fine-grain cache partitioning. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 57–68. ACM, 2011. (Cited on page 15.)

[78] G. Schirner and R. Dömer. Introducing preemptive scheduling in abstract rtos models using result oriented modeling. In *Proc. of DATE'08*, pages 122–127, New York, NY, USA, 2008. ISBN 978-3-9810801-3-1. doi: 10.1145/1403375.1403408. (Cited on pages 3 and 28.)

[79] V. Selfa, J. Sahuquillo, L. Eeckhout, S. Petit, and M. E. Gómez. Application clustering policies to address system fairness with intel's cache allocation technology. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 194–205. IEEE, 2017. (Cited on pages 5, 91, and 102.)

[80] V. Seshadri, S. Yedkar, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Mitigating prefetcher-caused pollution using informed caching policies for prefetched blocks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):51, 2015. (Cited on pages 92 and 97.)

[81] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, 39, 1990. (Cited on page 37.)

[82] L. Sha, T. Abdelzaher, K.-E. Årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real time scheduling theory: A historical perspective. *Real-time systems*, 28(2-3):101–155, 2004. (Cited on pages 4, 19, 22, and 45.)

[83] M. Shekhar, A. Sarkar, H. Ramaprasad, and F. Mueller. Semi-partitioned hard-real-time scheduling under locked cache migration in multicore systems. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 331–340. IEEE, 2012. (Cited on page 45.)

[84] SPEC. *SPEC CPU Benchmarks*. URL `https://www.spec.org/cpu2017/`. (Cited on pages 94, 95, 102, and 104.)

[85] SpecC. http://www.cecs.uci.edu/ specc/. (Cited on pages 3 and 27.)

[86] A. Sridharan, B. Panda, and A. Seznec. Band-pass prefetching: An effective prefetch management mechanism using prefetch-fraction metric in multi-core systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(2):19, 2017. (Cited on page 101.)

[87] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 63–74. IEEE, 2007. (Cited on page 92.)

[88] R. Stafford. Random vectors with fixed sum, 2006. URL `http://www.mathworks.com/matlabcentral/fileexchange/9700`. (Cited on page 65.)

[89] S. Stattelmann, O. Bringmann, and W. Rosenstiel. Fast and accurate resource conflict simulation for performance analysis of multi-core systems. In *2011 Design, Automation Test in Europe*, pages 1–6, March 2011. doi: 10.1109/DATE.2011.5763044. (Cited on pages 3 and 28.)

[90] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 62–75. ACM, 2015. (Cited on page 91.)

[91] V. Suhendra and T. Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In *2008 45th ACM/IEEE Design Automation Conference*, pages 300–303. IEEE, 2008. (Cited on pages 44 and 45.)

[92] SysRT. https://github.com/jxiao90/SysRT. (Cited on page 108.)

[93] SystemC. http://www.accellera.org. (Cited on pages 3 and 27.)

[94] D. Tam, R. Azimi, L. Soares, and M. Stumm. Managing shared l2 caches on multicore systems in software. In *Workshop on the Interaction between Operating Systems and Computer Architecture*, pages 26–33, 2007. (Cited on pages 15 and 92.)

[95] P. K. Valsan, H. Yun, and F. Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12. IEEE, 2016. (Cited on page 22.)

[96] K. Varadarajan, S. Nandy, V. Sharda, A. Bharadwaj, R. Iyer, S. Makineni, and D. Newell. Molecular caches: A caching structure for dynamic creation of application-specific heterogeneous cache regions. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 433–442. IEEE Computer Society, 2006. (Cited on page 15.)

[97] R. Wang and L. Chen. Futility scaling: High-associativity cache partitioning. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 356–367. IEEE Computer Society, 2014. (Cited on page 15.)

[98] X. Wang, S. Chen, J. Setter, and J. F. Martínez. Swap: Effective fine-grain management of shared last-level caches with minimum hardware support. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 121–132. IEEE, 2017. (Cited on page 15.)

[99] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson. Outstanding paper award: Making shared caches more predictable on multicore platforms. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 157–167. IEEE, 2013. (Cited on page 45.)

[100] 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)*, 7(3):36, 2008. (Cited on page 45.)

[101] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. *ACM SIGARCH computer architecture news*, 23(2):24–36, 1995. (Cited on page 104.)

[102] C.-J. Wu and M. Martonosi. A comparison of capacity management schemes for shared cmp caches. In *Proc. of the 7th Workshop on Duplicating, Deconstructing, and Debunking*, volume 15, pages 50–52. Citeseer, 2008. (Cited on page 15.)

[103] C.-J. Wu and M. Martonosi. Characterization and dynamic mitigation of intra-application cache interference. In *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, pages 2–11. IEEE, 2011. (Cited on pages 5 and 92.)

[104] C.-J. Wu, A. Jaleel, M. Martonosi, S. C. Steely Jr, and J. Emer. Pacman: prefetch-aware cache management for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 442–453. ACM, 2011. (Cited on page 92.)

[105] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995. (Cited on page 5.)

[106] Y. Xiang, X. Wang, Z. Huang, Z. Wang, Y. Luo, and Z. Wang. Dcaps: dynamic cache allocation with partial sharing. In *Proceedings of the Thirteenth EuroSys Conference*, page 13. ACM, 2018. (Cited on pages 5, 91, and 102.)

[107] J. Xiao and G. Buttazzo. Adaptive embedded control for a ball and plate system. In *The Eighth*

International Conference on Adaptive and Self-Adaptive Systems and Applications, 2016. (Cited on page 9.)

[108] 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)*, pages 199–208. IEEE, 2017. (Cited on pages 8, 43, and 71.)

[109] J. Xiao, A. Pimentel, and G. Lipari. Sysrt: A modular multiprocessor rtos simulator for early design space exploration. In *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 38–45. IEEE, 2017. (Cited on pages 8 and 27.)

[110] J. Xiao, S. Altmeyer, and A. D. Pimentel. Schedulability analysis of global scheduling for multicore systems with shared caches. *IEEE Transactions on Computers, under review*, 2019. (Cited on pages 8 and 43.)

[111] J. Xiao, A. D. Pimentel, and X. Liu. Cp_{pf}: A prefetch aware llc partitioning approach. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, pages 17:1–17:10, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6295-5. doi: 10.1145/3337821.3337895. URL `http://doi.acm.org/10.1145/3337821.3337895`. (Cited on pages 8 and 91.)

[112] Y. Xie and G. H. Loh. Pipp: promotion/insertion pseudo-partitioning of multi-core shared caches. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 174–183. ACM, 2009. (Cited on page 15.)

[113] M. Xu, L. T. X. Phan, H.-Y. Choi, and I. Lee. Analysis and implementation of global preemptive fixed-priority scheduling with dynamic cache allocation. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12. IEEE, 2016. (Cited on pages 4 and 45.)

[114] M. Xu, L. Thi, X. Phan, H.-Y. Choi, and I. Lee. vcat: Dynamic cache management using cat virtualization. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 211–222. IEEE, 2017. (Cited on page 45.)

[115] Y. Ye, R. West, Z. Cheng, and Y. Li. Coloris: a dynamic cache partitioning system using page coloring. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 381–392. IEEE, 2014. (Cited on page 92.)

[116] Y. Yi, D. Kim, and S. Ha. Fast and time-accurate cosimulation with os scheduler modeling. *Des. Autom. Embedded Syst.*, 8(2-3):211–228, June 2003. ISSN 0929-5585. doi: 10.1023/B:DAEM.0000003963.20442.29. URL `http://dx.doi.org/10.1023/B:DAEM.0000003963.20442.29`. (Cited on pages 3 and 27.)

[117] H. Zabel, W. Müller, and A. Gerstlauer. Accurate rtos modeling and analysis with systemc. In *Hardware-dependent Software*, pages 233–260. Springer, 2009. (Cited on pages 3 and 27.)

[118] F. Zhang and A. Burns. Schedulability analysis for real-time systems with edf scheduling. *IEEE Transactions on Computers*, 58(9):1250–1258, 2009. (Cited on pages 4 and 45.)

[119] W. Zhang and J. Yan. Accurately estimating worst-case execution time for multi-core processors with shared direct-mapped instruction caches. In *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 455–463. IEEE, 2009. (Cited on page 45.)

[120] H. Zhu and M. Erez. Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems. *ACM SIGARCH Computer Architecture News*, 44(2):33–47, 2016. (Cited on pages 5 and 91.)

[121] X. Zhuang and S. L. Hsien-Hsin. Reducing cache pollution via dynamic data prefetch filtering. *IEEE Transactions on Computers*, 56(1):18–31, 2006. (Cited on page 92.)

[122] X. Zhuang and H.-H. Lee. A hardware-based cache pollution filtering mechanism for aggressive prefetches. In *2003 International Conference on Parallel Processing, 2003. Proceedings.*, pages 286–293. IEEE, 2003. (Cited on page 92.)

Summary

Modern computing systems are constructed using commodity multi-core processors, on which part of the memory subsystem is shared by different cores on the same processor. Multiple applications executing simultaneously on a multi-core system contend for the shared memory resources, such as last level caches (LLC) and main memory, causing inter-application interference. Such inter-application interference, if uncontrolled, results in unpredictable execution delay for individual applications and severe system performance degradation. In this dissertation, we focus on shared cache interference and investigate two issues raised by the increasing complexity of underlying hardware and software for multi-core systems: timing predictability of real-time computing and caching performance for high performance computing.

In the first research line, we improve the timing predictability for embedded multi-core system by simulation and analytical approaches for the timing analysis. With regard to the simulation approach, we developed SysRT, a simulator of real-time operating systems (RTOS) that allows developers and researchers to easily explore and validate embedded RTOS design alternatives. The simulator contains different types of application models and a modular RTOS kernel model, all developed in SystemC. Efficient and precise modeling of preemptive scheduling is achieved via an event-driven simulation approach, allowing simulations to be performed much faster than cycle-accurate simulations. SysRT outperforms state-of-art simulators in both simulation speeds and accuracy. We also demonstrate the flexibility of SysRT and its benefits in evaluation of timing performance for software tasks .

With regards to the analytical approach, we first develop a new schedulability analysis of global scheduling for real-time multi-core systems with shared caches. We construct an integer programming formulation and an iterative algorithm to obtain the upper bound on shared cache interference a task may exhibit during one job execution. The upper bound on shared cache interference is subsequently integrated into the schedulability analysis to derive a new schedulability condition for global scheduling. Later, we extend the schedulability analysis for the partitioned scheduling. We propose a novel cache-interference aware task partitioning algorithm, called CA-TPAR. We conduct schedulability analysis of CA-TPAR and formally prove its correctness. A range of experiments is performed to investigate how the schedulability is degraded by shared cache interference. We also evaluate the schedulability performance of global scheduling (Earliest Deadline First and Fixed Priority) against CA-TPAR over randomly generated tasksets.

In the second research line, We propose CP$_{pf}$, a prefetch aware LLC partitioning approach to improving LLC management for high performance caching. We first study the interaction between hardware prefetching and LLC management by analyzing the variation of application performance when varying the effective LLC space in the presence and absence of hardware prefetching. Motivated by this study, we then classify applications into two categories, prefetching sensitive (PS) and non prefetching sensitive (NPS) applications, by the performance benefit they experience from hardware prefetchers. CP$_{pf}$ consists of a method using Precise Event-Based Sampling techniques for the online classification of PS and NPS applications and a cache partitioning scheme using Cache Allocation technology to distribute the cache space among PS and NPS

applications. The prototype of CP$_{pf}$ is implemented as a user-level runtime system on Linux. Finally, we show the system performance improvement achieved by CP$_{pf}$.

Samenvatting

Moderne computersystemen worden gebouwd met behulp van commodity multi-core processors, waarbij een gedeelte van het geheugensubsysteem gedeeld wordt door verschillende cores op dezelfde processor. Meerdere applicaties die gelijktijdig worden uitgevoerd op een multi-core systeem strijden om het gedeelde geheugen, zoals last level caches (LLC) en hoofdgeheugen, wat inter-applicatie interferentie veroorzaakt. Dergelijke inter-applicatie interferentie kan als deze niet goed beheerst wordt, resulteren in onvoorspelbare vertragingen van individuele toepassingen en ernstige verslechtering van de systeemprestaties. In dit proefschrift richten we ons op interferentie in gedeelde caches en onderzoeken we twee aspecten die worden beïnvloed door de toenemende complexiteit van onderliggende hardware en software voor multi-core systemen: timing voorspelbaarheid van real-time systemen en cachingprestaties bij high performance computing.

In de eerste onderzoekslijn verbeteren we de voorspelbaarheid van de timing van embedded multi-core systemen door simulatie en analytische benaderingen van de timing analyse. Met betrekking tot de simulatie benadering hebben we SysRT ontwikkeld, een simulator van real-time besturingssystemen (RTOS) waarmee ontwikkelaars en onderzoekers eenvoudig embedded RTOS kunnen onderzoeken en alternatieven ontwerpen kunnen valideren. De simulator bevat verschillende soorten applicatie modellen en een modulair RTOS-kernelmodel, allemaal ontwikkeld in SystemC.

EfficiÎnt en nauwkeurig modelleren van preemptive scheduling wordt bereikt via een event gestuurde simulatie benadering, waardoor simulaties veel sneller kunnen worden uitgevoerd dan bij klok-cyclusnauwkeurige simulaties. SysRT overtreft moderne simulatoren in zoveel simulaties snelheid als in nauwkeurigheid. We tonen ook de flexibiliteit van SysRT en zijn voordelen bij de evaluatie van timing prestaties voor software taken.

Met betrekking tot de analytische aanpak ontwikkelen we eerst een nieuwe schedulingsanalyse van de globale planning voor real-time multi-core systemen met gedeelde caches. We construeren een geheeltallig lineaire programmering formulering en een iteratief algoritme om de bovengrens te verkrijgen van de gedeelde cache-interferentie die een taak kan vertonen tijdens één taakuitvoering. De bovengrens van de gedeelde cache-interferentie wordt vervolgens geïntegreerd in de planningsanalyse om een nieuwe planningsvoorwaarde af te leiden voor een globale planning. Later breiden we de planningsanalyse uit voor de gepartitioneerde planning. We introduceren CA-TPAR, een nieuw partitioneringsalgoritme dat rekening houd met cache-interferentie. We voeren een planningsanalyse uit van CA-TPAR en geven een formeel bewijs van correctheid. Een scala aan experimenten is uitgevoerd om te onderzoeken hoe de planningsmogelijkheden worden verminderd door interferentie in de gedeelde cache. We vergelijken ook de planningsprestaties van globale planning methoden (vroegste deadline eerst en vaste prioriteit) met CA-TPAR voor willekeurig gegenereerde takensets.

In de tweede onderzoekslijn stellen we CP_{pf} voor, een prefetch bewust LLC partitioneringsbenadering om LLC-beheer te verbeteren voor hoge caching prestaties. We bestuderen eerst de interactie tussen hardware prefetching en LLC management door analyse van de variatie van applicatieprestaties bij het variëren van de effectieve LLC-ruimte in aanwezigheid en afwezigheid van hardware prefetching. Gemotiveerd door

deze studie, classificeren we toepassingen vervolgens in twee categorieën, prefetch-gevoelige (PS) en niet prefetch-gevoelige (NPS) applicaties, door het prestatievoordeel dat ze ervaren met hardware prefetching. CP_{pf} bestaat uit een methode die nauwkeurige event-gebaseerde sampling technieken gebruikt voor de online classificatie van PS en NPS-applicaties en een cache-partitioneringsschema dat met behulp van een cache allocatie technologie de cacheruimte verdeeld over de PS- en NPS-applicaties. Het prototype van CP_{pf} is geïmplementeerd als een runtime-systeem op gebruikersniveau op Linux. Tot slot laten we de systeem prestatieverbetering zien die is bereikt door CP_{pf}.