REPORT DOCUMENTATION PAGE	Form Approved OMB NO. 0704-0188

1. REPORT DATE (DD-MM-YYYY) 26-01-2023	2. REPORT TYPE Thesis or Dissertation	3. DATES COVERED (From - To) -

4. TITLE AND SUBTITLE Real-Time Multicore Virtualization	5a. CONTRACT NUMBER W911NF-11-1-0403
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER 611102
6. AUTHORS Meng Xu	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES AND ADDRESSES University of Pennsylvania Office of Research Services 3451 Walnut Street, 5th Floor Philadelphia, PA 19104 -6205	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS (ES) U.S. Army Research Office P.O. Box 12211 Research Triangle Park, NC 27709-2211	10. SPONSOR/MONITOR'S ACRONYM(S) ARO 11. SPONSOR/MONITOR'S REPORT NUMBER(S) 59800-NC.21

12. DISTRIBUTION AVAILIBILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
The views, opinions and/or findings contained in this report are those of the author(s) and should not contrued as an official Department of the Army position, policy or decision, unless so designated by other documentation.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	15. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Insup Lee
UU	UU	UU	UU		19b. TELEPHONE NUMBER 215-898-3532

Standard Form 298 (Rev 8/98)
Prescribed by ANSI Std. Z39.18

REPORT DOCUMENTATION PAGE (SF298)
(Continuation Sheet)

Continuation for Block 13

Proposal/Report Number: 59800.21-NC

Report Title: Real-Time Multicore Virtualization
Report Type: Ph.D. Dissertation

Publication Type: Thesis or Dissertation
Institution: University of Pennsylvania
Date Received: 26-Jan-2023 Completion Date: 5/18/18 9:50PM
Title: Real-Time Multicore Virtualization
Authors: Meng Xu
Acknowledged Federal Support: **N**

CACHE-AWARE REAL-TIME VIRTUALIZATION

Meng Xu

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2018

Supervisor of Dissertation	Co-Supervisor of Dissertation
Insup Lee	Linh Thi Xuan Phan
Professor, Computer and Information Science	Assistant Professor, Computer and Information Science

Graduate Group Chairperson

Lyle Ungar, Professor, Computer and Information Science

Dissertation Committee:

Rahul Mangharam, Associate Professor, University of Pennsylvania

Oleg Sokolsky, Research Professor, University of Pennsylvania

Joseph Devietti, Assistant Professor, University of Pennsylvania

Chenyang Lu, Professor, Washington University in St. Louis

To Mom and Dad

Acknowledgement

Foremost, I would like to take advantage of this opportunity to express my earnest gratitude to my advisors, Professor Insup Lee and Professor Linh Thi Xuan Phan, for their tremendous support during my Ph.D. study. Insup introduced me to various aspects of cyber-physical systems, gave me the academic freedom to explore different projects, and provided me the opportunities to be trained to be a better researcher and thinker. His vision on research and his ability to observe things from a strategically advantageous perspective has helped me out of many dead-ends and has deeply influenced me in research and life. Linh introduced me to various aspects of real-time systems, taught me many research skills, closely guided me in various aspects in my Ph.D. study, and trained me to be a good researcher. Her vision and enormous passion on research, her persistent chase of scientific truth, her destructive aversion to low-hanging fruits, and her keen eyes on details has significantly impacted me in research and life. Her help on my research over years and the innumerable discussions we had over day and night have contributed immensely to the completion of the dissertation.

I would like to thank my dissertation committee for their valuable time, effort and suggestions on the dissertation and their help in my Ph.D. study. Thank Professor Rahul Mangharam for his help and support in our journey in the first F1/10 autonomous racing competition. I really had a lot of fun and definitely learned a lot in the experience. Thank Professor Joseph Devietti for his insights on my work. I learned a lot from Joseph in his advanced computer architecture classes. Thank

Professor Oleg Sokolsky for his insightful suggestions and comments on various aspects of my work. I enjoyed discussions with him in many meetings. Thank Professor Chenyang Lu for his help, feedback and advice in my Ph.D. study, ever since I joined the RT-Xen project in my master program. Working with him was a great pleasure and fortunate to me.

I would also like to thank the faculties and staffs in my Ph.D. life. Thank Professor Christopher Gill for his involvement and discussion in the RT-Xen project. Thank James Weimer for his help in the F1/10 competition. Thank Professor Boon Thau Loo for introducing me to the computer network world and providing various advices in Ph.D. life. I also want to thank Mike Felker and Wai Ping Ng for their help. They created the illustration to me that bureaucracy does not exist.

I want to thank my collaborators, Sisu Xi, Haoran Li and Chong Li from Washington University in St. Louis, Yuhan Lin from Northeastern University in China, and Jaewoo Lee, Hyon-Young Choi, Fanxin Kong and Dagaen Golomb from our PRECISE lab for their help and discussion in various research projects. I also want to thank Radoslav Ivanov, Nguyen Hung, Bipeen Acharya, Kyoungwon Kim and Sarvesh Patkar for our awesome teamwork in the F1/10 autonomous racing competition. Besides research, I would like to thank Dario Faggioli, Wei Liu and George Dunlap for their help in Xen community.

I would like to thank my friends: Mingchen Zhao, Zhuoyao Zhang, Yang Li, Sanjian Chen, Chen Chen, Ang Chen, Xujie Si, Min Wen, and Jing Li. Without them, my Ph.D. life would not be so colorful. A huge special thanks to Lu Chen for making my last two years at Penn the most wonderful time and pushing my dissertation to completion.

Finally and most importantly, I want to thank my parents for being my parents. Thank you mom and dad for raising me, understanding me, educating me and always supporting my ambition!

ABSTRACT

CACHE-AWARE REAL-TIME VIRTUALIZATION

Meng Xu

Insup Lee

Linh Thi Xuan Phan

Virtualization has been adopted in diverse computing environments, ranging from cloud computing to embedded systems. It enables the consolidation of multi-tenant legacy systems onto a multicore processor for Size, Weight, and Power (SWaP) benefits. In order to be adopted in timing-critical systems, virtualization must provide real-time guarantee for tasks and virtual machines (VMs). However, existing virtualization technologies cannot offer such timing guarantee. Tasks in VMs can interfere with each other through shared hardware components. CPU cache, in particular, is a major source of interference that is hard to analyze or manage.

In this work, we focus on challenges of the impact of cache-related interferences on the real-time guarantee of virtualization systems. We propose the *cache-aware real-time virtualization* that provides both system techniques and theoretical analysis for tackling the challenges. We start with the challenge of the private cache overhead and propose the private cache-aware compositional analysis. To tackle the challenge of the shared cache interference, we start with non-virtualization systems and propose a shared cache-aware scheduler for operating systems to co-allocate both CPU and cache resources to tasks and develop the analysis. We then investigate virtualization systems and propose a dynamic cache management framework that hierarchically allocates shared cache to tasks. After that, we further investigate the resource allocation and analysis technique that considers not only cache resource but also CPU and memory bandwidth resources. Our solutions are applicable to commodity hardware and are essential steps to advance virtualization technology into timing-critical systems.

Contents

List of Tables

List of Illustrations

Chapter 1

Introduction

Timing is critical for safety-critical systems, including automotive, avionics, manufacturing, and medical devices. These safety-critical systems directly interact with physical objects, their responses to physical events must strictly satisfy their timing requirements. For instance, the airbag must be deployed within 30μs for mitigating casualties in a car crash [8]. The correct timing behavior is a pre-requirement for a safety-critical system to be deployed in the real world, which is regulated by industrial standards, such as the automotive safety standard ISO-26262. The *worst-case* response time of tasks in these systems must be analyzed in design time and guaranteed at runtime for the correctness of their safety-critical functionalities.

Safety-critical systems are becoming increasingly complex and demanding. For example, in automotive systems, carmakers are now racing to bring more and more new features – including over-the-air update, advanced driver assistant systems (ADAS), and connectivities, into vehicles, for attracting more customers. These features are computation intensive and often dynamic. For example, ADAS understands the world around the car with image processing applications, which notoriously require lots of computation power to operate correctly and has dynamic resource demand in different driving scenarios: high in city streets and low in high way. In addition, these features are from different manufacturers, making the safety-critical systems multi-tenancy systems that require functional and timing isolation among different features. The automotive infotainment systems from manufacturer `A` should never affect the functional and timing properties of the engine control systems from manufacturer `B`, although both features are integrated to run on the same car. In the trend of increasing resource demand and complexity of multi-tenancy safety-critical systems, how can we satisfy the demand without violating the timing constraints?

Conventional approach that adds one-to-multiple computation units for each new feature is no longer an answer to the above question because it is not scalable. Adding a computation unit, which is called Electrical Control Unit (ECU) in automotive systems, introduces extra cost of wires, space, weight, and power consumption, increasing the cost of entire systems. More importantly, as the number of new features increases rapidly, it will eventually be impossible to add extra computation units, due to space and weight constraints.

Fortunately, the microprocessor industry is offering more computation power in the form of an exponentially growing number of cores on a single chip. Enabling an extra core on a chip introduce no extra cost of wires, weight or space because cores are embedded on and connected through an integrated circuit. Hence, it is becoming more and more common to run multiple system components on the same multicore platform, rather than to deploy them separately on different single-core processors. This shift towards shared multicore computing platforms enables system designers to reduce cost and increase performance; however, it also makes it significantly more challenging to achieve functional separation and to maintain correct timing behavior.

Virtualization has been widely adopted from data centers to embedded systems to integrate and consolidate multiple system components onto a shared (multicore) hardware without violating functional separation requirement. Virtualization is a promising technique to move safety-critical systems from single core platforms towards powerful multicore platforms. Multiple system components with different functionalities that were initially running on single-core processors can be deployed in virtual machines (VMs) on a shared multicore platform. These VMs provide a clean functional isolation between each other that one VM cannot view or modify the functionality of another VM. However, existing virtualization platforms are designed to provide good average performance – they are not designed to guarantee the worst-case performance of tasks or VMs. To make virtualization applicable for safety-critical systems, real-time virtualization that ensures tasks in each VM meet their worst-case performance requirements is required.

Real-time virtualization is challenging on multicore platforms due to the unpredictable interference from shared hardware components. Tasks within the same or different VMs can compete for the shared hardware resources, such as shared last level cache, incurring unpredictable interference to each other, increasing tasks' worst-case execution time (WCET), and potentially causing tasks' violating their timing requirements. Besides, a task may frequently migrate from one core to another, taking extra time to restore its per-core state, such as its content in private cache, from the old core to the new core at each migration. The task migration on a multicore platform also increases the task's WCET, putting the task's timing requirement at risk. Amongst shared resources on a multicore platform – such as CPU, and memory bus – cache, which is a fast and small memory between CPU and main memory that is designed to be shared and invisible to software, is the primary challenge in realizing real-time virtualization on multicore platforms.

The goal of this dissertation is to manage and analyze the cache effect on the timing guarantee of tasks in virtualization systems. In particular, this dissertation focuses on two questions fundamental to real-time virtualization on a cache-based multicore platform: (i) can the timing requirement be satisfied under the cache-related interferences; (ii) how to manage cache to mitigate the impact of cache interferences.

1.1 Real-time virtualization

Before we study the cache effect on real-time virtualization, we need to understand what real-time virtualization is. In this section, we describe and discuss the architecture of virtualization, the concepts of real-time systems, and the compositional analysis, based on which we define real-time virtualization.

1.1.1 Virtualization

Virtualization concept was initially introduced in the 1960s as a method of logically dividing system resources of mainframe computers, e.g., IBM System/370, to various applications. It has become a widely adopted technique to support multiple Operating Systems (OS) on the same hardware.

In virtualization, hypervisor, which is also called Virtual Machine Monitor (VMM), allows multiple VMs to execute on the same computer hardware concurrently. Depending on where the hypervisor runs, hypervisors can be classified into two types: (i) Type-1 hypervisors that run directly on the bare metal; and (ii) Type-2 hypervisors that run on a host OS. Type-1 hypervisors are more suitable for providing the predictable performance to VMs than Type-2 hypervisors do, because Type-1 hypervisors directly interact with the hardware and have full control on hardware resources allocated to VMs. In contrast, the extra host OS layer between the hardware and the hypervisor in Type-2 hypervisors can introduce unexpected and unpredictable delay to the hosted hypervisor and then to the VMs on the hypervisor. Among open-source Type-1 hypervisors, Xen [14] is the most popular one that is powering many commercial cloud computing platforms, including Amazon AWS. We use Xen as a prototype platform for our studies in this dissertation. Our solutions can also be extended to other hypervisors.

The Xen scheduling architecture is illustrated in Fig. 1.1. Each VM has tasks scheduled by the guest OS's scheduler on the Virtual CPUs (VCPUs) of the VM.

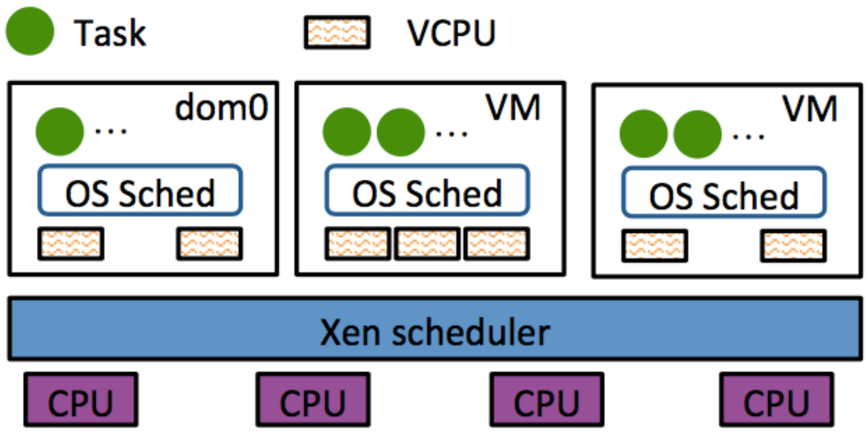


Figure 1.1: Xen scheduling architecture.

The Xen scheduler schedules all VCPUs of all domains on physical cores. Xen has an administration VM, called *dom0*, and multiple guest VMs, called *domU*.

Xen introduces a real-time scheduler, called Real-Time Deferrable Server (RTDS) scheduler, in Xen 4.5.0. The RTDS scheduler is built to provide guaranteed CPU capacity to guest VMs on symmetric multiprocessing (SMP) machines.

Each VCPU is implemented as a *deferrable server* in the RTDS scheduler. A VCPU is represented as $VP_i = (\Pi_i, \Theta_i)$, where Π_i is period and Θ_i is budget, indicating the VCPU is supposed to run for Θ_i time in every Π_i time. The deferrable server mechanism defines how a VCPU's budget is managed: a VCPU's budget linearly decreases in terms of running time only when the VCPU is running on a core; a VCPU VP_i's budget is replenished to Θ_i in every Π_i period; and a VCPU's remaining budget is discarded at the end of the current period.

1.1.2 Real-time constraints

Task execution in real-time systems must satisfy predefined temporal constraints. For instance, the image processing task of ADAS in automotive systems must finish processing an image before the next image is captured by camera. In addition, many

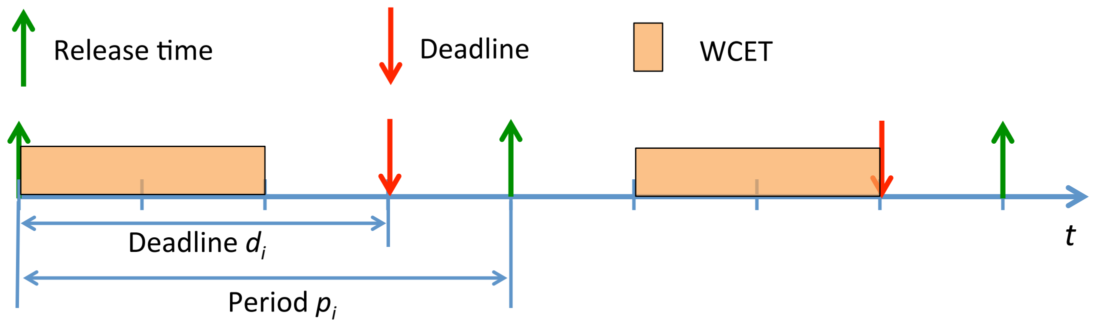


Figure 1.2: Explicit-deadline periodic task model.

real-time tasks are recurrent tasks – they do not terminate during operation. For example, the image processing task keeps running as long as the camera keeps taking images.

Real-time tasks. Liu and Layland [44] introduce the explicit-deadline periodic task model to capture the execution pattern of tasks in real-time systems. The task model has ever since been widely adopted in the real-time community as the foundation to obtain the analytical results. As illustrated in Fig. 1.2, each task τ_i releases a job in each period p_i, which executes for *at most* worst-case execution time (WCET) time and should finish its execution by its deadline d_i. An explicit-deadline periodic task τ_i is defined by $\tau_i = (p_i, e_i, d_i)$, where p_i is the period, e_i is the WCET, and d_i is the relative deadline of τ_i. We require that $0 < e_i \leq d_i \leq p_i$ for all τ_i. A real-time system consists of a set of tasks, which is represented as a task set $\tau = \{\tau_1, ..., \tau_n\}$.

The *utilization* of task τ_i is defined as $u_i = \frac{e_i}{p_i}$, specifying how much processor time the task needs in the worst case in each of its periods. The utilization of a task set τ is the sum of the utilizations of all tasks in the task set: $u = \sum_{\tau_i \in \tau} \frac{e_i}{p_i}$.

The *response time* of a job is the delay from the time when the job is released to the time when the job finishes execution. The response time of a task is the longest response time of all jobs of the task.

A job misses its deadline if its response time is larger than its deadline – that is, the job finishes its execution after its deadline. A task misses its deadline if there exists at least one job of the task missing its deadline.

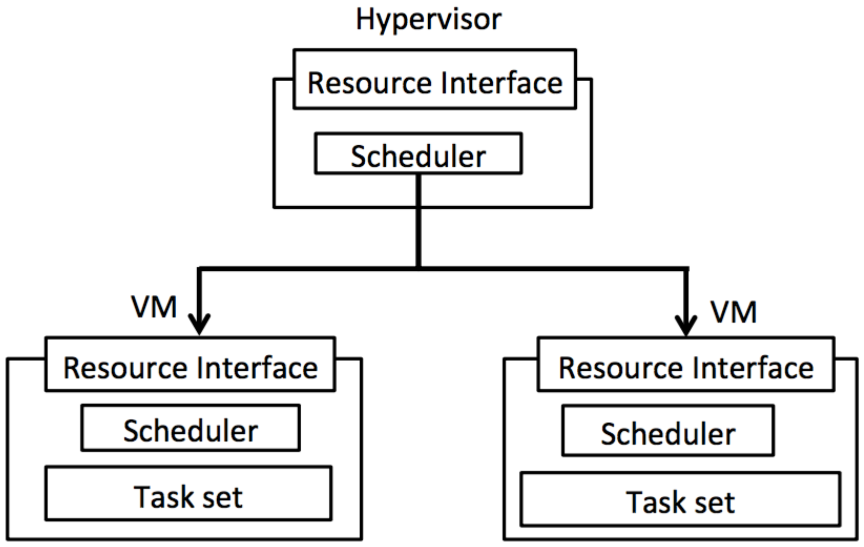


Figure 1.3: Hierarchical systems and compositional analysis.

A task is *schedulable* if none of its jobs misses deadline – all of its jobs finish before their deadlines. A system (or a task set) is *schedulable* if all of its tasks are schedulable. A *hard real-time (HRT)* system requires no deadline miss for its tasks. In contrast, a *soft real-time (SRT)* system allows some deadline misses of tasks. In this dissertation, we focus on HRT requirement, although our proposed system can work for SRT requirement as well.

The *schedulability test or schedulability analysis* provides the sufficient condition for determining whether a system is schedulable. If a system is claimed schedulable by a schedulability test, all of its tasks are schedulable even in the worst case.

1.1.3 Compositional analysis

Virtualization systems distribute hardware resources, such as CPU resource, to VMs in a hierarchical manner. Hypervisor (which is the root component) has all hardware resources. Using the scheduling algorithm it has, the hypervisor allocates the resources to VMs (which are child components); each VM further redistributes its allocated resources to its tasks.

In order to guarantee the schedulability of a virtualization system, system designers must determine the total amount of resources required by the entire system and the amount of resource required by each VM. Conceptually, tasks in a VM are schedulable if the resource demand of the tasks is no larger than the resource supply of the VM. And the entire system is schedulable if the resource requirement of the hypervisor is no larger than the total amount of resource provided by the hardware.

Compositional analysis provides a method to compute the resource requirement of each VM in a compositional manner [69] [41]. Under compositional analysis illustrated in Fig. 1.3, each VM has a resource interference, such as Periodic Resource Model [57] and Multicore Periodic Resource (MPR) model [56], which specifies the amount of resource the VM can provide to its tasks. The compositional analysis first independently abstracts the resource demand of tasks in each VM into the resource interface of the VM. Then the analysis transfers the resource interface of each VM into interface tasks (which are VCPUs) of the hypervisor and further abstracts the resource demand of those interface tasks into the resource interface of the hypervisor.

1.1.4 Requirements of real-time virtualization

A real-time virtualization system is a virtualization system that satisfies the real-time constraints – that is, real-time tasks in each VM are schedulable. Real-time virtualization requires (i) analysis techniques to tell whether a virtualization system is schedulable under given hardware resources in the worst case, and (ii) system techniques to guarantee that a virtualization system claimed schedulable by analysis will never witness a task's missing deadline.

Compositional analysis is an analysis technique that can be used to compute the amount of CPU hardware resource required to guarantee the schedulability of a virtualization system. If the provided hardware resource is no less than the required resource, the virtualization system will be claimed schedulable by the compositional analysis. A number of compositional analysis techniques for multicore systems have been developed (e.g., [16], [31], [43]), but existing theories assume a somewhat idealized platform in which all overhead is negligible. In practice, the platform overhead – especially the cost of cache misses – can substantially interfere with the execution of tasks. As a result, the computed interfaces can underestimate the resource requirements of the tasks within the underlying components. One goal of this dissertation is to remove this assumption by accounting for the cache overhead in the interfaces.

Real-time schedulers are a system technique that has been used to guarantee the configured CPU resource for VMs (when implemented in hypervisor) and tasks (when implemented in VMs). A number of real-time schedulers have been designed and implemented (e.g., [68], [69], [27], [24]), but real-time schedulers manage only the CPU resource, not the other hardware resources – especially the cache resource that may lead to extra cache misses to tasks. As a consequence, tasks can interfere with each other through the other resources which are neither considered in analysis nor eliminated by system techniques, causing tasks to miss deadline even when the system is claimed schedulable by analysis. The other goal of this dissertation is to design and implement cache management techniques to mitigate the impact of cache on systems' real-time constraints.

1.2 Cache challenges for real-time virtualization on multicore

In order to bridge the speed gap between processor and memory without sacrificing the memory capacity, a hierarchy of cache, each of which has a smaller capacity but faster speed than the following, is built on the multicore platform. An example is shown in Fig. 1.4. Each core has a private cache that is only accessible by the core. The processor has a shared cache that is accessible by all cores.

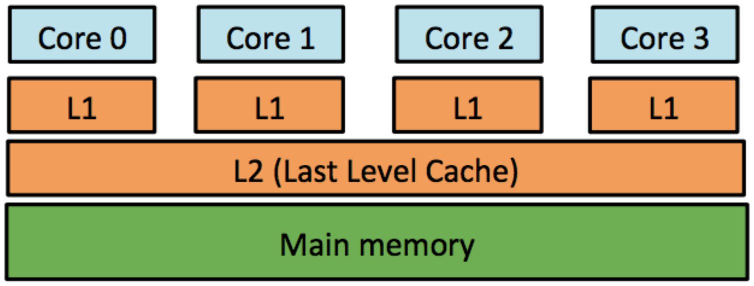


Figure 1.4: Cache hierarchy.

1.2.1 Cache interference

When two code sections are mapped to the same cache set, one section can evict the other section's cache blocks from the cache, which causes a cache miss when the latter accesses the evicted cache block. If the two code sections belong to the same task, this cache miss is an *intrinsic* cache miss; otherwise, it is an *extrinsic* cache miss [18]. The interference due to intrinsic cache misses of a task can typically be statically analyzed or profiled based solely on the task; however, extrinsic cache misses depend on the interference between tasks during execution. In this dissertation, we assume that the tasks' WCETs already include intrinsic cache-related interference, and we will focus on the extrinsic cache-related interference.

Cache interference can be categorized into private cache overhead and shared cache interference, depending on at which level of cache the extra cache misses occur:

- Private cache overhead occurs when a task resumes execution and reloads its contents into private cache that are evicted by another task while the task is not running. A task may experience one private cache overhead whenever the task resumes: the one private cache overhead is the latency of multiple extra cache misses the task experience at its resumption. The total amount of private cache overheads a task experience in each of its periods is determined by the value of one private cache overhead and the number of private cache overheads in the period. The private cache overhead is avoided if each job keeps running until it finishes execution.

- Shared cache interference occurs when a *running task* reloads its contents into *shared cache* that are evicted by another *concurrently running tasks* on another core. In the worst-case scenario, each cache hit access of a task can be turned into cache miss access due to the shared cache interference. Shared cache interference is avoided if each cache area can only be accessed by one core at any time.

1.2.2 Challenges of cache-aware analysis

Cache-aware analyses study the effect of cache interference on the real-time performance of multicore virtualization systems. If a system is claimed schedulable by cache-aware analysis, the system should be schedulable under the presence of cache interference in practice.

Analyzing the *private cache overhead* on multicore virtualization systems is challenging because virtualization introduces additional overhead that is difficult to predict. For instance, when a VCPU resumes after being preempted by a higher-priority VCPU, a task executing on it may experience a cache miss, since its cache blocks may have been evicted from the cache by the tasks that were executing on the preempting VCPU. Similarly, when a VCPU is migrated to a new core, all its cached code and data remain in the old core; therefore, if the tasks later access content that was cached before the migration, the new core must load it from memory rather than from its cache

Another challenge comes from the fact that cache misses that can occur when a VCPU finishes its budget and stops its execution. For instance, suppose a VCPU is currently running a task `A` that has not finished its execution when the VCPU finishes its budget, and that `A` is migrated to another VCPU of the same domain that is either idle or executing a lower priority task `B` (if one exists). Then `A` can incur a cache miss if the new VCPU is on a different core, and it can trigger a cache miss in `B` when `B` resumes. This type of overhead is difficult to analyze, since it is in general not possible to determine statically when a VCPU finishes its budget or which task is affected by the VCPU completion.

Analyzing the *shared cache interference* is theoretically challenging. Since concurrent running tasks can access any cache line at any time, in the worst-case scenario, concurrent running tasks always access the same cache line, turning each cache hit to cache miss. Due to the fact that the task information in one VM is not available to another VM, the analysis has to assume the worst-case scenario to upper bound the impact of shared cache interference, which leads to a pessimistic analysis. Further, even if precise analyses were possible, they would still not mitigate the shared cache interference.

1.2.3 Challenges of cache management

Cache management aims to mitigate the cache interference incurred to tasks for improving the real-time performance of the entire system. At the high level, cache management techniques divide cache into cache partitions and allocate non-overlapped cache partitions to tasks. Since each task has its own dedicated cache partitions, cache interference is mitigated, potentially increasing the system's real-time performance. However, since each task can only use a fraction of cache under cache management, instead of the entire cache without cache management, the WCET of each task may increase, potentially decreasing the system's real-time performance. A cache management technique is useful if it can improve systems' real-time performance over no cache management in general.

Cache management mainly focuses on the shared cache, instead of private cache, because partitioning a small private cache to mitigate the cache interference is often "not required but instead detrimental to the provable system performance" as demonstrated in [12]. This is because the cost of increased WCET with private cache partitioning is often larger than the cost of private cache overhead without cache management.

Managing shared cache is challenging because hardware exposes limited functionalities to system software, such as OS and hypervisor, to directly manage the cache. Although recent COTS processors allow system software to divide cache into equal-size partitions, the supported number of partitions is very limited, making each cache partition relatively large. For instance, the Intel Cache Allocation Technology (CAT) [6] can divide the 20 MB cache on Intel Xeon 2618L v3 processor into only 20 equal-size cache partitions. Each cache partition is 1 MB. When a 1 MB cache partition is reserved for a task, the task may not use all cache areas in the cache partition and the unused cache areas cannot be re-used by other tasks, wasting the scarce cache resource.

In addition, the hardware-based cache partitioning introduces constraints for system software to manage the cache. For instance, the Intel CAT only allows continuous cache partitions to be allocated to tasks. Since unallocated cache partitions may not be contiguous, fragmentation of cache partitions may happen, causing low cache resource utilization.

Managing shared cache for virtualization systems is even more challenging due to the extra abstraction layer introduced by virtualization. OS in each VM is deprivileged in virtualization for VM isolation – preventing a VM from affecting another VM's functionality. However, privilege is required for OS to control the cache. Hypervisor must provide a mechanism for OS to manage the cache for its tasks without breaking the VM isolation provided by virtualization.

1.3 Contributions and organizations

This dissertation proposes *cache-aware real-time virtualization* that provides real-time guarantee to tasks in virtualization systems under the presence of cache interferences. The cache-aware real-time virtualization provides (i) private cache-aware compositional analysis that analyzes the impact of private cache overhead on the timing guarantees; (ii) dynamic shared cache management and analysis that mitigates the shared cache interference and that analyzes the overhead introduced by the shared cache management; (iii) a holistic framework that integrates shared cache allocation with memory bandwidth regulation mechanisms to mitigate potential interference among concurrent tasks.

In Chapter 3, we present the private cache-aware compositional analysis. Specifically, we introduce DMPR, a deterministic extension of the multiprocessor resource periodic model to better represent component interfaces on multicore virtualization platforms; we present a DMPR-based compositional analysis for systems without cache-related overhead; we characterize different types of events that cause cache misses in the presence of virtualization; and we propose two approaches, task-centric and model-centric, to account for the cache-related overhead. Based on the results, we develop the corresponding cache-aware compositional analysis methods.

In Chapter 4, we explore the dynamic cache management and analysis for non-virtualized systems, which can later be used inside a VM in virtualization systems. We investigate the feasibility of global preemptive scheduling with dynamic job-level cache allocation. We present gFPca, a cache-aware variant of the global preemptive fixed-priority (gFP) algorithm, together with its implementation and analysis. gFPca allocates cache to jobs dynamically at run time when they begin or resume, and it allows high-priority tasks to preempt low-priority tasks via both CPU and cache resources. It also allows low-priority tasks to execute when high-priority tasks are unable to execute due to insufficient cache resource, thus further improving the cache and CPU utilizations. Since preemption is allowed, tasks may experience cache overhead – e.g., upon resuming from a preemption, a task may need to reload its cache content in the cache partitions that were used by its higher-priority tasks; therefore, we develop a new method to account for such cache overhead.

In Chapter 5, we present vCAT, a dynamic cache management framework for virtualization systems that can deliver strong shared cache isolation at both VM and task levels, and that can be configured for both static and dynamic allocations. vCAT virtualizes the Intel CAT in software for achieving hypervisor- and VM-level cache allocations. To illustrate the feasibility of our approach, we provide a proof-of-concept prototype of vCAT on top of Xen and LITMUS^{RT}.

In Chapter 6, we propose vC^{2}M, a holistic solution towards timing isolation in multicore virtualization systems. On the system angle, vC^{2}M integrates both the shared cache and memory bandwidth management to provide better isolation among tasks and VMs; this is done by leveraging the vCAT in Chapter 5 and a new memory bandwidth regulation mechanism for virtualization. On the theory side, vC^{2}M provides an efficient resource allocation policy for tasks and VMs that can minimize resources while guaranteeing schedulability. Specifically, given a set of tasks on the VMs and a given hardware configuration, vC^{2}M will compute both (i) the assignment of tasks to virtual CPUs (VCPUs) and VCPUs to cores, and (ii) the amount of CPU, cache, and bandwidth resources for each task and each VCPU, to guarantee schedulability while minimizing resource usage. We have implemented a Xen-based prototype of vC^{2}M. We evaluated vC^{2}M, showing that it can be implemented with minimal overhead, and that it provides substantial benefits in reducing tasks' WCETs.

We discuss the related work in cache-aware analysis and management in Chapter 2, before we present the details of our *cache-aware real-time virtualization* in Chapter 3 to Chapter 6. In the end, we conclude this dissertation with discussion of future work.

Chapter 2

Related work

2.1 Cache-aware analysis

Cache-aware analyses primarily focus on analyzing the impact of *private cache* on systems' schedulability. Private cache-aware analyses, from a high-level perspective, consist of two steps: (i) obtaining the cost of one private cache overhead for a task at the task's resumption event; (ii) accounting each private cache overhead of each task into the overhead-free schedulability analysis.

We first review the approaches of obtaining the cost of one private cache overhead; we then discuss the private cache-aware schedulability tests.

2.1.1 Cost of one private cache overhead

Precise analysis is a common approach to obtain the cost of one private cache overhead. Lee et al. [40] introduced the concept of *Useful Cache Block (UCB)* and *Evicting Cache Block (ECB)*: a memory block m is a UCB at a program point ρ if (a) the memory block m may be cached at ρ and (b) the memory block m may be reused as a cache hit by the program after ρ. When the preemption occurs at the program point ρ, only the UCBs at the program point ρ may need additional reloads. An ECB is a memory block accessed during the execution of a *preempting task*. The cost of reloading a cache block, which is also called as a cache line, is specified as BRT.

The cost of one private cache overhead can be analyzed with four approaches: (i) the *ECB-only* approach [26] [61] that uses the ECBs of the preempting task to bound the cost; (ii) the *UCB-only* approach [40] that uses the number of UCBs to bound the cost; (iii) the *UCB-Union* approach [60] that considers both the preempting and preempted task to calculate the cost; and (iv) the *ECB-Union* approach [11] that considers the union of ECBs of preempting tasks to calculate the cost.

Measurement is another approach to obtain the cost of one-private cache overhead. There exist two types of measurement approaches to measure the cost of one private cache overhead: trace-driven memory simulation approach and real hardware based measurement approach.

The trace-driven memory simulation approach [58] [51] uses a simulation framework to record the execution trace of tasks and examine the private cache overhead based on the collected execution trace. The strength of trace-driven memory simulation approaches is that they can control simulation environment to evaluate the effect of different cache configurations on the cache interference. The weakness of this type of approaches is that it replies on accurate architectural models, which may not be available for Commercial Off-The-Shelf (COTS) hardware, and representative memory traces, which are difficult to collect.

The real hardware based measurement approach [42] [29] [62] [17] directly measures the cost of one private cache overhead on a real COTS hardware. This type of measurement approaches does not require the model of the hardware and can be applied to many COTS processors. However, the measurement approach can not provide a safe upper bound of the cache overhead since it has no guarantee that the worst-case scenario of cache overhead will always occur in the measurement.

2.1.2 Private cache-aware schedulability test

Private cache-aware schedulability tests integrate the cost of private cache overhead into the overhead-free schedulability test to determine if a task set is schedulable under the influence of private cache overhead.

The first approach of private cache-aware schedulability tests [26] [33] [24] extends each task's WCET with the cost of one private cache overhead of the task and applies the overhead-free schedulability test for the task set with extended WCETs. The cost of one private cache overhead for a task can be obtained with one of the approaches discussed above. This approach can be easily applied to existing overhead-free schedulability test. However, this approach may significantly overestimate the overall cost of private cache overhead.

The second approach observes that each additional cache overhead of a task may result in a smaller cost than the previous one. Instead of upper bounding the cost of one private cache overhead, the second approach [59] [11] [47] directly computes the total cost of all private cache overheads a task may experience and extends the overhead-free schedulability test by considering the total private cache overhead as a special workload.

2.2 Cache management

Cache management techniques first use cache partitioning techniques to divide shared cache into partitions and then allocate cache partitions to cores/tasks. Cache partitioning techniques can be grouped into software-based approach and hardware-based approach, depending on if the cache partitioning technique replies on any special hardware.

2.2.1 Software-based cache partitioning techniques

Cache controller uses part of memory address bits, denoted as *index-bits*, to locate cache sets. We can calculate the number of index-bits, as illustrated in Fig. 2.1, as follows: suppose we have an α-way-associative shared cache on an m core platform; the shared cache consists of multiple s-byte equal-size cache slices, connected by the ring bus among cores; and the cache line size is 2^l bytes. We can calculate that each cache slice has $\frac{s}{\alpha \times 2^l}$ cache sets and that the number of index-bits is $c = log\frac{s}{\alpha \times 2^l}$. For example, Intel Xeon E5-2618L v3 processor has a 20-way-associative shared cache on 8 cores; each core has one $\frac{20MB}{8} = 2560KB$ cache slice whose cache line size is 2^6 bytes. The number of index-bits is calculated as $c = log\frac{2560KB}{20 \times 2^6 B} = 11$.

The software-based cache partition techniques divide cache into partitions by grouping cache sets based on parts of their index-bits. The memories that map to the same cache partition are grouped together. In order to allocate a specific cache partition to a task, the software-based techniques always allocate the memories from the corresponding memory group.

The software-based cache partitioning can be achieved by a page-coloring technique and a compiler-based technique.

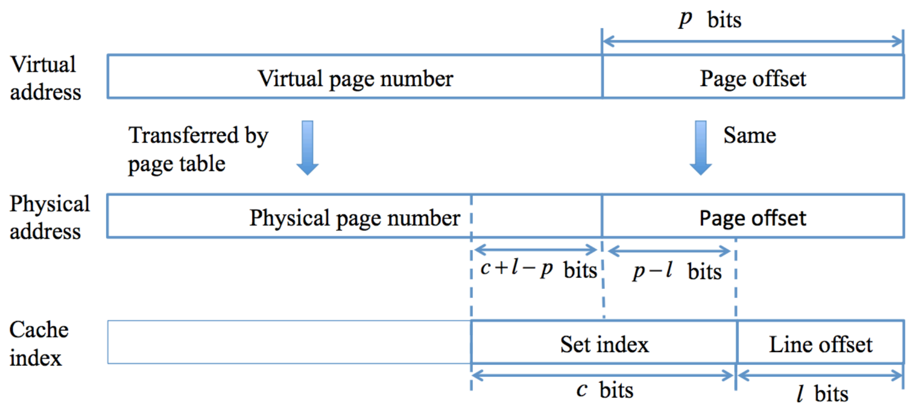


Figure 2.1: Software-based cache partition mechanism. On Intel Xeon E5-2618L v3 processor, $l = 6$, $c = 11$, $p = 12$.

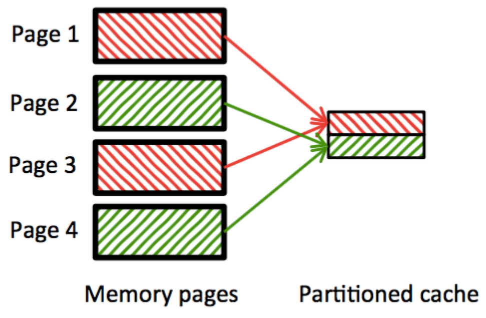
Page coloring technique

The page coloring technique controls a task's memory page allocation in order to control which cache area the task will use. We describe operating systems' paging mechanism before we explain the page coloring technique.

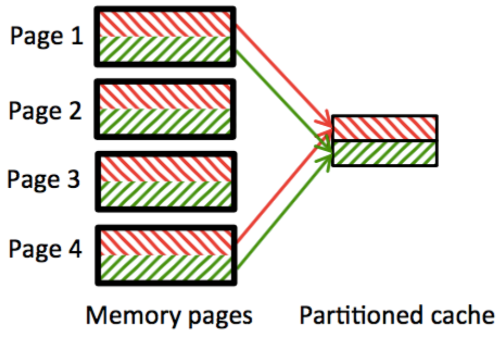
Paging mechanism. Modern operating systems use the paging mechanism to transfer a virtual address `va` to a physical address `pa`. The virtual address `va` consists of two non-overlapped parts as illustrated in Fig. 2.1: the virtual page number, which is denoted as $Bits[v, p]$, and the page offset, which is denoted as $Bits[p-1, 0]$. The paging mechanism transfers the virtual page number to the physical page number by looking up the page table. The page offset of the virtual address is the same with that of the corresponding physical address. The paging mechanism constructs the physical address `pa` for the virtual address `va` by concatenating the physical page number and the page offset.

Page coloring. As illustrated in Fig. 2.1, the physical page number has $c + l - p$ bits overlapped with the cache's index-bits. We call these overlapped bits as *cache-color-bits*. We divide the cache into 2^{c+1-p} non-overlapped areas and call a cache area as a cache color. The cache sets in the same cache color are indexed by the same cache-color-bits. The page coloring technique organizes the memory pages whose addresses have the same value of the cache-color-bits into the same cache-color group. The memory pages in different groups are mapped to different cache colors.

In order to allocate a specific cache color to a task, OS allocates memory pages from the cache-color group to the task. If a cache color is allocated for only one task, the task will not be interfered by other tasks in that cache color. Fig. 2.2(a) illustrates an example that uses the page coloring-based technique to divide the cache into two partitions.



(a) Page coloring partition



(b) Compiler-based partition

Figure 2.2: Example of dividing the cache into two partitions by software-based partition techniques. A color represents a partition.

Compiler-based partition technique

The compiler-based cache partition technique [52] controls the page offset bits, instead of the page number bits, to control which cache area a task can use.

As shown in Fig. 2.1, the page offset of a virtual address has $p - l$ bits overlapped with the cache's index-bits. These overlapped bits are *cache-partition-bits*. We divide the cache into 2^{p-l} non-overlapped cache partitions based on the cache-partition-bits. We categorize the virtual memory addresses with the same cache-partition-bits into the same cache-partition group. In order to allocate a cache partition to a task, the compiler rearranges the task's virtual memory layout when it compiles the task, so that the task will use the virtual memory from the cache-partition group. Fig. 2.2(b) illustrates an example that uses the compiler-based technique to divide the cache into two partitions.

2.2.2 Hardware-based cache partitioning techniques

There exist two hardware-based cache partitioning techniques on COTS processors: (1) the Cache Allocation Technology for Intel processors; (2) the Lockdown-by-Master (LbM) technology for ARM processors. In this section, we first review these two hardware-based cache partitioning techniques; we then discuss the ColoredLockdown technique that combines both the page coloring technique and the LbM technique to provide finer-granularity cache partitioning.

Intel Cache Allocation Technology (CAT)

Intel introduces the Cache Allocation Technology (CAT) that allows system software (such as OS, hypervisor or VMM) to control the allocation of the shared cache to each core. Intel CAT divides the shared cache into N non-overlapped equal-size cache partitions; for instance, $N = 20$ for the Intel Xeon E5-2618L v3 processor. System softwares can allocate a set of such cache partitions to a core by programming two model-specific registers (MSR): (1) the Class of Service (COS) register, which has an N-bit Capacity Bitmask (CBM) field to specify a particular cache partition set, and (2) the IA32_PQR_ASSOC (PQR) register of each core, which has a COS field for linking a particular COS to the core; when the COS field is set to the ID of a COS register, all cache allocation requests from the core will be enforced to the cache partitions specified by the CBM of the COS register. For example, to allocate partitions 0 to 3 to a core, we set 1s for the bits 0 to 3 (and zeroing the remaining) of the CBM field of the associated COS register.

According to the Intel 64 and IA-32 Architectures Software Developer's Manual (SDM) [2] and the experimental studies [6] [72], Intel CAT has the following constraints: (1) the number of cache partitions per core must be at least two [1] and should not exceed the number of available partitions; (2) the partition set of a core can only be made of contiguous cache partitions; and (3) the CAT only controls cache allocation requests (i.e., cache miss requests) and does not control cache lookup requests (i.e., cache hit requests).

According to [1], Intel CAT is available for 6 types of Intel Haswell processors and all Intel Xeon processor D CPUs. Intel CAT is not backward compatible. Although Intel Skylake processors are one-generation newer than Intel Xeon processor

[1]This constraint may not exist for some Intel processors. For example, Intel broadwell processors allow one cache partition per core.

D processors, the Intel Skylake processors do not support the Intel CAT technology.

ARM Lockdown-by-Master (LbM) technology

The *Lockdown-by-Master* technology, supported by the PL310 cache controller on ARM processors, can be used to partition the last level cache into 16 non-overlapped equal-size partitions. The LbM allows certain ways to be marked as unavailable for allocation, such that the cache allocation (which allocates cache lines for cache misses) only happens in the remaining ways that are not marked as unavailable. Each core P_i has a per-core lockdown register R_i, where a bit q in R_i is one if the cache allocation *cannot* happen in the cache way q for the memory access from the core P_i, and zero otherwise.[2]

The PL310 cache controller is widely used on ARM Cortex A9 processors, which implements the ARMv7-A architecture. The PL310 cache controller is not backward compatible either. None of current ARMv8-A architecture-based processors, such as ARM Cortex A53, implements the PL310 cache controller.

Recent studies [49] [38] [70] use the LbM technology to achieve isolation in the shared cache.

Colored Lockdown technique

The Colored Lockdown technique [49] divides the shared cache into equal-size cache partitions by using both the page coloring technique and the LbM technique.

The Colored Lockdown technique first uses the page coloring technique to divide the shared cache into cache colors and groups the memory pages based on their assigned cache colors. In order to allocate cache lines in the cache way i for a memory page PF with the cache color c, where PF is accessed by the core P, the Colored Lockdown technique conducts the following steps: (1) it first ensures none of the lines in the page PF are cached in any level of caches by flushing the memory

[2]To be precise, each core has two separate registers for instruction and data access respectively

page PF from the cache; this step makes sure the following accesses to the page PF are cache misses; (2) it locks all ways but the way i for the core P, so that the following cache misses from the core P can only happen in the way i; (3) it sequentially reads the page PF, with preemption disabled, on the core P, so that the page PF is deterministically loaded in the way i; (4) it restores the cache way lock status for the core P as it was before the step (2). After the step (4), the access to the page PF will always be cache hit in the specific cache way i.

2.2.3 Comparison of partition techniques

We evaluate a cache partition technique based on the following metrics:

- COTS hardware support: COTS hardware is usually cheaper than specialized hardware. A technique with COTS hardware support is more cost-effective than the one without such support.

- Super-page support: because super page technique (which usually uses 2MB or 1GB page) can dramatically reduce TLB misses and improve performance for many applications, modern OSs (e.g., Linux) support the super page technique. If a partitioning technique cannot support the super page technique, the partitioning technique is expected to introduce extra performance penalties (such as extra TLB misses) for applications that may benefit from the super page technique.

- No memory copy for reconfiguring partitions for a task: memory copy is much slower than cache access. If a cache partitioning technique requires copying memory to reconfigure a task's partitions, the technique may not be suitable for dynamic cache partitioning managements. The software-based approach, which requires changing tasks' memory layouts to partition the cache, usually requires memory copy for reconfiguring cache partitions.

- The maximum number of partitions: given a cache with fixed size, the more partitions a partition technique can support, the finer-granularity control we can have over the cache.

We compare the cache partition techniques, discussed in this section, in Table 2.1.

Table 2.1: Comparison of cache partition techniques

Metrics	Software-based		Hardware-based		
	Page coloring	Compiler-based	CAT	LbM	Colored Lockdown
Hardware support	COTS	COTS	Intel [3]	ARM [4]	ARM [5]
Super-page support	No	Yes	Yes	Yes	No
No memory copy for cache reconfiguration	No	No	Yes	Yes	Yes-No[6]
Number of partitions	32[7]	64[8]	20	16	512[9]

[3]It is supported on some Intel processors.

[4]It is supported on ARM processors with the PL310 cache controller.

[5]It is supported on ARM processors with the PL310 cache controller.

[6]It does not requires memory copy for reconfiguring cache ways for tasks, but it requires memory copy for reconfiguring cache colors.

[7]We assume 4KB memory page and 2MB 16-way-associative cache slice.

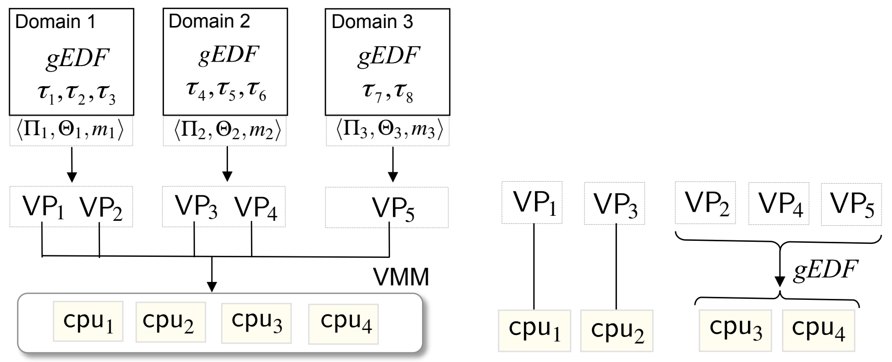
[8]We assume 4KB memory page and 64B cache line. Each page can be mapped to $\frac{4KB}{64B} = 64$ different cache lines

[9]Colored lockdown can partition cache into 32 colors and 16 ways independently. The total number of partitions is $32 \times 16 = 512$.

Chapter 3

Private cache-aware compositional analysis

We have agreed that *real-time virtualization on multicore platforms* is a solution to satisfy the increasing resource demand, to reduce the system complexity, to provide the functional isolation, and to achieve the real-time constraints for the multi-tenancy safety-critical systems. We also realize that *cache interference* is a major challenge in realizing real-time virtualization. In this chapter, we will present a private cache-aware compositional analysis technique that can be used to ensure timing guarantees of tasks scheduled on a multicore virtualization platform under the presence of private cache overhead. Our technique improves on previous multicore compositional analyses by accounting for the cache-related overhead in the VMs' interfaces, and it addresses the new virtualization specific challenges in the overhead analysis. To demonstrate the utility of our technique, we report results from an extensive evaluation based on randomly generated workloads.



(a) Task and VCPU scheduling. (b) Scheduling of VCPUs.

Figure 3.1: Compositional scheduling on a virtualization platform.

3.1 System descriptions

The system we consider consists of multiple real-time components that are scheduled on a multicore virtualization platform, as is illustrated in Fig. 3.1(a). Each component corresponds to a *domain* (virtual machine) of the platform and consists of a set of tasks; these tasks are scheduled on a set of virtual processors (VCPUs) by the domain's scheduler. The VCPUs of the domains are then scheduled on the physical cores by the hypervisor, which is also specified as virtual machine monitor (VMM).

Each task τ_i within a domain is an explicit-deadline periodic task, defined by $\tau_i = (p_i, e_i, d_i)$, where p_i is the period, e_i is the worst-case execution time (WCET), and d_i is the relative deadline of τ_i. We require that $0 < e_i \le d_i \le p_i$ for all τ_i.

Each VCPU is characterized by $\mathsf{VP}_j = (\Pi_j, \Theta_j)$, where Π_j is the VCPU's period and Θ_j is the resource budget that the VCPU services in every period, with $0 \le \Theta_j \le \Pi_j$. We say that VP_j is a *full* VCPU if $\Theta_j = \Pi_j$, and a *partial* VCPU otherwise. We assume that each VCPU is implemented as a periodic server [55] with period Π_j and maximum budget time Θ_j. The budget of a VCPU is replenished at the beginning of each period; if the budget is not used when the VCPU is scheduled to run, it is wasted. We assume that each VCPU can execute only one task at a time. Like in most real-time scheduling research, we follow the conventional real-time task model in which each task is a single thread in this work; an extension to parallel task models is an interestin g but also challenging research direction, which we plan to investigate in our future work.

We assume that all cores are identical and have unit capacity, i.e., each core provides t units of resource (execution time) in any time interval of length t. Each core has a private cache[10], all cores share the same memory, and the size of the memory is sufficiently large to ensure that all tasks (from all domains) can reside in memory at the same time, without conflicts.

Scheduling of tasks and VCPUs. We consider a hybrid version of the Earliest Deadline First (EDF) strategy. As is shown in Fig. 3.1, tasks within each domain are scheduled on the domain's VCPUs under the global EDF (gEDF) [15] scheduling policy. The VCPUs of all the domains are then scheduled on the physical cores under a semi-partitioned EDF policy: *each full VCPU is pinned (mapped) to a dedicated core, and all the partial VCPUs are scheduled on the remaining cores under gEDF.* In the example from Fig. 3.1(b), VP_1 and VP_3 are full VCPUs, which are pinned to the physical cores cpu_1 and cpu_2, respectively. The remaining VCPUs are partial VCPUs, and are therefore scheduled on the remaining cores under gEDF.

Private cache-related overhead. We consider the private cache-related overhead in this chapter. We use $\Delta^{\mathsf{crpmd}}_{\tau_i}$ to denote the maximum time needed to re-load all the useful cache blocks (i.e., cache blocks that will be reused) of a preempted task τ_i when that task resumes (either on the same core or on a different core).[11] Since the overhead for reloading the cache content of a preempted VCPU (i.e., a periodic

[10]In this chapter, we assume that the cores either do not share a cache, or that the shared cache has been partitioned into cache sets that are each accessed exclusively by one core [72] [39]. We believe that an extension to shared caches is possible, and we plan to consider it in our future work.

[11]We are aware that using a constant maximum value to bound the cache-miss overhead of a task may be conservative, and extensions to a finer granularity, e.g., using program analysis, may be possible. However, as the first step, we keep this assumption to simplify the analysis in this work, and we defer such extensions to our future work.

server) upon its resumption is insignificant compared to the task's, we will assume here that it is either zero or is already included in the overhead due to cache misses of the running task inside the VCPU.

Objectives. In the above setting, our goal is to develop a cache-aware compositional analysis framework for the system. This framework consists of two elements: (1) an interface representation that can succinctly capture the resource requirements of a component (i.e., a domain or the entire system); and (2) an interface computation method for computing a minimum-bandwidth cache-aware interface of a component (i.e., an interface with the minimum resource bandwidth that guarantees the schedulability of a component in the presence of cache-related overhead).

Assumptions. We assume that (1) all VCPUs of each domain j share a single period Π_j; (2) all Π_j are known a priori; and (3) each Π_j is available to all domains. These assumptions are important to make the analysis tractable. Assumption 1 is equivalent to using a time-partitioned approach; we make this assumption to simplify the cache-aware analysis in Section 3.7, but it should be easy to extend the analysis to allow different periods for the VCPUs. Assumption 2 is made to reduce the search space, which is common in existing work (e.g., [31]); it can be relaxed by first establishing an upper bound on the optimal period (i.e., the period of the minimum-bandwidth interface) of each domain j, and then searching for the optimal period value based on this bound. Finally, Assumption 3 is necessary to determine how often different events that cause cache-related overhead happen (c.f. Section 3.5), which is crucial for the cache-aware interface computation in Section 3.6 and 3.7. One approach to relaxing this assumption is to treat the period of the VCPUs of a domain as an input parameter in the computation of the overhead that another domain experiences. Such a parameterized interface analysis approach is very general, but making it efficient remains an interesting open problem for future research. We note, however, that although each assumption can be relaxed, the consequence of relaxing all three assumptions requires a much deeper investigation.

3.2 Improvement on multiprocessor periodic resource model

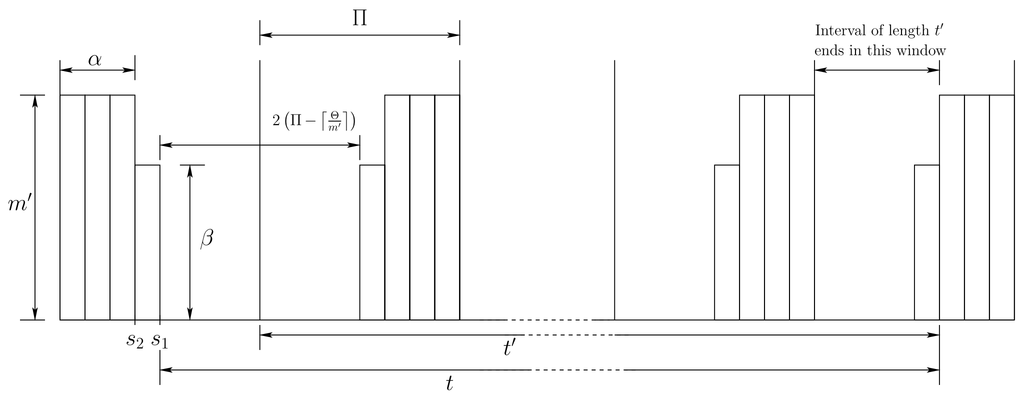
Recall that, when representing a platform, a resource model specifies the characteristics of the resource supply that is provided by that platform; when representing a component's interface, it specifies the total resource requirements of the component that must be guaranteed to ensure the component's schedulability. The resource provided by a resource model R can also be captured by a supply bound function (SBF), denoted by $\mathsf{sbf}_R(t)$, that specifies the minimum number of resource units that R provides over any interval of length t.

In this section, we first describe the existing multiprocessor periodic resource (MPR) model [56], which serves as a basis for our proposed resource model for multicore virtualization platforms. We then present a new SBF for the MPR model that improves upon the original SBF given in [56], thus enabling tighter MPR-based interfaces for components and more efficient use of resource.

3.2.1 Background on MPR

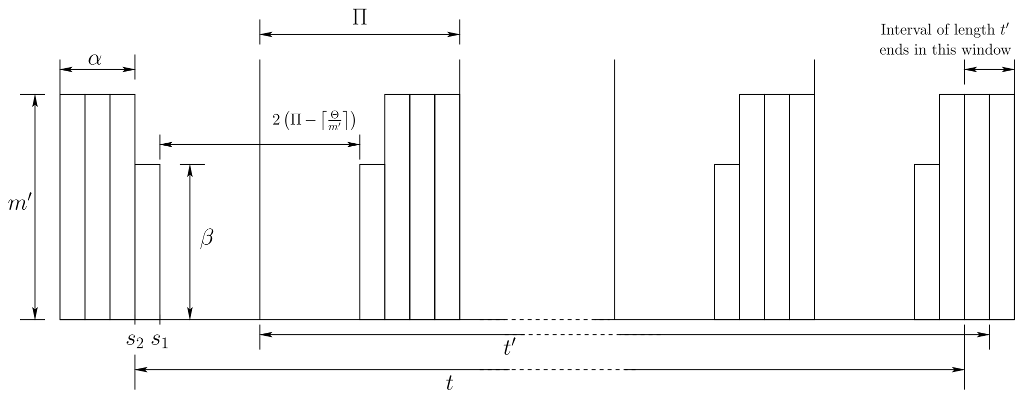
An MPR model $\Gamma = (\tilde{\Pi}, \tilde{\Theta}, m')$ specifies that a multiprocessor platform with a number of identical, unit-capacity CPUs provides $\tilde{\Theta}$ units of resources in every period of $\tilde{\Pi}$ time units, with concurrency at most m' (in other words, at any time instant at most m' physical processors are allocated to this resource model), where $\tilde{\Theta} \leq m'\tilde{\Pi}$. Its resource bandwidth is given by $\tilde{\Theta}/\tilde{\Pi}$.

The worst-case resource supply scenario of the MPR model is shown in Fig. 3.2 [31]. Based on this worst-case scenario, the authors in [31] proposed an SBF that bounds



Π
Interval of length t' ends in this window
α
$2\left(\Pi - \lceil \frac{\Theta}{m'} \rceil\right)$
m'
β
s_2 s_1
t'
t

(a) Case 1



Π
Interval of length t' ends in this window
α
$2\left(\Pi - \lceil \frac{\Theta}{m'} \rceil\right)$
m'
β
s_2 s_1
t'
t

(b) Case 2

Figure 3.2: Worst case resource supply of MPR model.

the resource supplied by the MPR model $\Gamma = (\tilde{\Pi}, \tilde{\Theta}, m')$, which is defined as follows:

$$\tilde{\mathsf{sbf}}_{\Gamma}(t) = \begin{cases} 0, \quad \text{if } t' < 0 \\ \lfloor t'/\tilde{\Pi} \rfloor \tilde{\Theta} + \max\{0, m'x - (m'\tilde{\Pi} - \tilde{\Theta})\}, \text{ if } t' \geq 0 \wedge x \in [1, y] \\ \lfloor t'/\tilde{\Pi} \rfloor \tilde{\Theta} + \max\{0, m'x - (m'\tilde{\Pi} - \tilde{\Theta})\} - (m' - \beta), \text{ if } t' \geq 0 \wedge x \notin [1, y] \end{cases} \tag{3.1}$$

where $\alpha = \left\lfloor \dfrac{\tilde{\Theta}}{m'} \right\rfloor$, $\beta = \tilde{\Theta} - m'\alpha$, $t' = t - \left(\tilde{\Pi} - \left\lceil \dfrac{\tilde{\Theta}}{m'} \right\rceil\right)$, $x = t' - \tilde{\Pi} \left\lfloor \dfrac{t'}{\tilde{\Pi}} \right\rfloor$ and $y = \tilde{\Pi} - \left\lfloor \dfrac{\tilde{\Theta}}{m'} \right\rfloor$.

3.2.2 Improved SBF of the MPR model

We observe that, although the function $\tilde{\mathsf{sbf}}_\Gamma$ given in Eq. (3.1) is a valid SBF for the MPR model Γ, it is conservative. Specifically, the minimum amount of resource provided by Γ over a time window of length t (see Fig. 3.2) can be much larger than $\tilde{\mathsf{sbf}}_\Gamma(t)$ when (i) the resource bandwidth of Γ is equal to its maximum concurrency level (i.e., $\tilde{\Theta}/\tilde{\Pi} = m'$), or (ii) $x \leq 1$, where x is defined in Eq. (3.1). We demonstrate these cases using the two examples below.

Example 3.1. *Let $\Gamma_1 = \langle \tilde{\Pi}, \tilde{\Theta}, m' \rangle$, where $\tilde{\Theta} = \tilde{\Pi} m'$, and Π and m' are any two positive integer values. By the definition of the MPR model, Γ_1 represents a multiprocessor platform with exactly m' identical, unit-capacity CPUs that are fully available. In other words, Γ_1 provides $m't$ time units in every t time units. However, according to Eq. (3.1), we have $\alpha = \left\lfloor \frac{\tilde{\Theta}}{m'} \right\rfloor = \tilde{\Pi}$, $\beta = \tilde{\Theta} - m'\alpha = 0$, $t' = t - \left(\tilde{\Pi} - \left\lceil \frac{\tilde{\Theta}}{m'} \right\rceil\right) = t$, $x = t' - \tilde{\Pi}\left\lfloor \frac{t'}{\tilde{\Pi}} \right\rfloor$, and $y = \tilde{\Pi} - \left\lfloor \frac{\tilde{\Theta}}{m'} \right\rfloor = 0$. Whenever $x \notin [1, y]$, for all $t = t' \geq 0$,*

$$\tilde{\mathsf{sbf}}_{\Gamma_1}(t) = \lfloor t'/\tilde{\Pi} \rfloor \tilde{\Theta} + \max\{0, m'x - (m'\tilde{\Pi} - \tilde{\Theta})\} - (m' - \beta) = m't - m'.$$

As a result, $\tilde{\mathsf{sbf}}_{\Gamma_1}(t) < m't$ for all t such that $x \notin [1, y]$.

Example 3.2. *Let $\Gamma_2 = \langle \Pi = 20, \Theta = 181, m' = 10 \rangle$ and consider $t = 21.1$. From Eq. (3.1), we obtain $\alpha = 18$, $\beta = 1$, $t' = t - 1 = 20.1$, $x = 0.1$, and $y = 2$. Since $x \notin [1, y]$, we have*

$$\begin{aligned}\tilde{\mathsf{sbf}}_{\Gamma_2}(t) &= \lfloor \frac{t'}{\tilde{\Pi}} \rfloor \tilde{\Theta} + \max\{0, m'x - (m'\tilde{\Pi} - \tilde{\Theta})\} - (m' - \beta) \\ &= \lfloor \frac{20.1}{20} \rfloor 181 + \max\{0, 10 \times 0.1 - (10 \times 20 - 181)\} - (10 - 1) = 172.\end{aligned}$$

We reply on the worst-case resource supply scenario of the MPR model shown in Fig. 3.2 to compute the worst-case resource supply of Γ_2 during a time interval of length t. We first compute the worst-case resource supply when $t = 21.1$ based on **Case 1** *in Fig. 3.2:*

- *t starts at the time point s_1;*
- *During the time interval $[s_1, s_1 + (\tilde{\Pi} - \alpha - 1)]$, i.e., $[s_1, s_1 + 1]$, Γ_2 supplies 0 time unit;*
- *During the time interval $[s_1+(\tilde{\Pi}-\alpha-1), s_1+(\tilde{\Pi}-\alpha-1)+\tilde{\Pi}]$, i.e., $[s_1+1, s_1+21]$, Γ_2 supplies $= 181$ time units;*
- *During the time interval $[s_1 + (\tilde{\Pi} - \alpha - 1)+, s_1 + t]$, i.e., $[s_1 + 21, s_1 + 21.1]$, Γ_2 supplies 0 time unit.*

Therefore, Γ_2 supplies 181 time units during a time interval of length $t = 21.1$ based on **Case 1** *in Fig. 3.2.*

Next, we compute the worst-case resource supply when $t = 21.1$ based on **Case 2** *in Fig. 3.2:*

- *t starts at the time point s_2;*
- *During the interval $[s_2, s_2 + (\tilde{\Pi} - \alpha)]$, i.e., $[s_2, s_2 + 2]$ Γ supplies $\beta = 1$ time unit;*
- *During the interval $[s_2 + (\tilde{\Pi} - \alpha), s_2 + 2(\tilde{\Pi} - \alpha)]$, i.e., $[s_2 + 2, s_2 + 4]$, Γ supplies $\beta = 1$ time unit;*
- *During the interval $[s_2 + 2(\tilde{\Pi} - \alpha), s_2 + t]$, i.e., $[s_2 + 4, s_2 + 21.1]$, Γ supplies $(21.1 - 4) \times m' = 171$ time units.*

Therefore, Γ_2 supplies $1 + 1 + 171 = 173$ time units during any time interval of length t based on **Case 2** *in Fig. 3.2. Because the two cases in Fig. 3.2 are the only two possible worst-case scenarios of the MPR resource model [31], the worst-case resource supply of Γ_2 during any time interval of length $t = 21.1$ is 173 time units. Since $\mathsf{sbf}_{\Gamma_2}(t) = 172$, the value computed by Eq. (3.1) under-estimates the actual resource provided by Γ_2.*

Based on the above observations, we introduce a new SBF that can better bound the resource supply of the MPR model. This improved SBF is computed based on the worst-case resource supply scenarios shown in Fig. 3.2.

Lemma 3.1. *The amount of resource provided by the MPR model $\Gamma = \langle \tilde{\Pi}, \tilde{\Theta}, m' \rangle$ over any time interval of length t is at least $\mathsf{sbf}_\Gamma(t)$, where*

$$\mathsf{sbf}_\Gamma(t) = \begin{cases} 0, & t' < 0 \\ \lfloor \frac{t'}{\tilde{\Pi}} \rfloor \tilde{\Theta} + \max\{0, m'x' - (m'\tilde{\Pi} - \tilde{\Theta})\}, & t' \geq 0 \wedge x' \in [1 - \frac{\beta}{m'}, y] \\ \max\{0, \beta(t - 2(\tilde{\Pi} - \lfloor \frac{\tilde{\Theta}}{m'} \rfloor))\} & t' \in [0,1] \wedge x' \notin [1 - \frac{\beta}{m'}, y] \\ \lfloor \frac{t''}{\tilde{\Pi}} \rfloor \tilde{\Theta} + \max\{0, m'x'' - (m'\tilde{\Pi} - \tilde{\Theta}) - (m' - \beta)\}, & t' \geq 1 \wedge x' \notin [1 - \frac{\beta}{m'}, y] \end{cases} \tag{3.2}$$

where

$$\alpha = \lfloor \frac{\tilde{\Theta}}{m'} \rfloor; \qquad \beta = \begin{cases} \tilde{\Theta} - m'\alpha, & \tilde{\Theta} \neq \Pi m' \\ m', & \tilde{\Theta} = \Pi m' \end{cases}; \quad t' = t - (\tilde{\Pi} - \lceil \frac{\tilde{\Theta}}{m'} \rceil); \quad t'' = t' - 1;$$

$$x' = (t' - \tilde{\Pi} \lfloor \frac{t'}{\tilde{\Pi}} \rfloor); \quad x'' = (t'' - \tilde{\Pi} \lfloor \frac{t''}{\tilde{\Pi}} \rfloor) + 1; \qquad y = \tilde{\Pi} - \lfloor \frac{\tilde{\Theta}}{m'} \rfloor.$$

Proof. We will prove that the function $\mathsf{sbf}_\Gamma(t)$ is a valid SBF of Γ based on the worst-case resource supply patterns of Γ shown in Fig. 3.2.

Consider the time interval of length t' (called time interval t') and the black-out interval (during which the resource supply is zero) in Fig. 3.2. By definition, x' is the remaining time of the time interval t' in the last period of Γ, and y is half the length of the black-out interval plus one. There are four cases of x, which determine whether $\mathsf{sbf}_\Gamma(t)$ corresponds to the resource supply of Γ in Case 1 or Case 2 in Fig. 3.2:

- $x' \in [1, y]$: It is easy to show that the value of $\mathsf{sbf}_\Gamma(t)$ in Case 1 is no larger than its value in Case 2. Note that if we shift the time interval of length t in Case 1 by one time unit to the left, we obtain the scenario in Case 2. In doing so, $\mathsf{sbf}_{\Gamma}(t)$ will be increased by β time units from the first period but decreased by at most β time units from the last period. Therefore, the pattern in Case 2 supplies more resource than the pattern in Case 1 when $x' \in [1, y]$.

- $x' \in [1 - \frac{\beta}{m'}, 1]$: As above, if we shift the time interval of length t in Case 1 by one time unit to the left, we obtain the scenario in Case 2. Recall that x' is the remaining time of the time interval of length t' in the last period, $x' \leq 1$ and $y \geq 1$. In shifting the time interval of length t, $\mathsf{sbf}_{\Gamma}(t)$ will lose $(1 - x')m'$ time units while gaining β time units from the first period. Because $x' \geq 1 - \frac{\beta}{m'}$, $\beta - (1 - x')m' \geq 0$. Therefore, $\mathsf{sbf}_{\Gamma}(t)$ gains $\beta - (1 - x')m' \geq 0$ time units in transferring the scenario in Case 1 to the scenario in Case 2. Hence, Case 1 is the worst-case scenario when $x' \in [1 - \frac{\beta}{m'}, 1]$.
- $x' \in [0, 1 - \frac{\beta}{m'})$: It is easy to show that Γ supplies less resource in Case 2 than in Case 1 when we shift the time interval of length t of Case 1 to left by one time unit to get Case 2. Therefore, Case 2 is the worst-case scenario when $x' \in [0, 1 - \frac{\beta}{m'}]$.
- $x' > y$: We can easily show that $\mathsf{sbf}_{\Gamma}(t)$ is no larger in Case 2 than in Case 1. Because $x' > y$, when we shift the time interval t of Case 1 to left by one time unit to get the scenario in Case 2, Γ loses m' time units from the last period but only gains β time units, where $\beta \leq m'$. Therefore, Case 2 is the worst-case scenario when $x' > y$.

From the above, we conclude that Case 1 is the worst-case resource supply scenario when $x' \in [1 - \frac{\beta}{m'}, y]$, and Case 2 is the worst-case resource supply scenario when $x' \notin [1 - \frac{\beta}{m'}, y]$.

Based on the worst-case resource supply scenario under different conditions above, we can derive Eq. 3.2 as follows:

- When $t' < 0$: It is obvious that $\mathsf{sbf}_{\Gamma}(t) = 0$ because Γ supplies no resource in the black-out interval.

- When $t' \geq 0$ and $x' \in [1 - \frac{\beta}{m'}, y]$: Based on the worst-case resource supply scenario in Case 1, Γ has $\lfloor \frac{t'}{\tilde{\Pi}} \rfloor$ periods and provides $\tilde{\Theta}$ time units in each period. Γ has x' remaining time in the last period, which provides $\max\{0, m'x'' - (m'\Pi - \Theta) - (m' - \beta)\}$ time units. Therefore, Γ supplies $\lfloor \frac{t'}{\tilde{\Pi}} \rfloor \tilde{\Theta} + \max\{0, m'x'' - (m'\Pi - \Theta) - (m' - \beta)\}$ time units during time interval t.

- When $t' \in [0, 1]$ and $x' \notin [1 - \frac{\beta}{m'}, y]$: Because $t' \in [0, 1]$, $t \in [-\lceil \frac{\tilde{\Theta}}{m'} \rceil, -\lceil \frac{\tilde{\Theta}}{m'} \rceil + 1]$. Therefore, $t < 2(-\lceil \frac{\tilde{\Theta}}{m'} \rceil) + 2$, where $2(-\lceil \frac{\tilde{\Theta}}{m'} \rceil)$ is the length of the black-out interval. Hence, the worst-case resource supply of Γ during time interval t is $\max\{0, \beta(t - 2(\Pi - \lfloor \frac{\Theta}{m'} \rfloor))\}$.

- When $t' > 1$ and $x' \notin [1 - \frac{\beta}{m'}, y]$, the worst-case resource supply scenario is Case 2. Γ has $\lfloor \frac{t''}{\tilde{\Pi}} \rfloor$ periods and provides $\tilde{\Theta}$ time units in each period. Γ supplies $\max\{0, m'x'' - (m'\tilde{\Pi} - \tilde{\Theta}) - (m' - \beta)\}$ time units during its first and last periods. Therefore, $\mathsf{sbf}_{\Gamma}(t) = \lfloor \frac{t''}{\tilde{\Pi}} \rfloor + \max\{0, m'x'' - (m'\tilde{\Pi} - \tilde{\Theta}) - (m' - \beta)\}$.

The lemma follows from the above results. $\square$

It is easy to verify that, under the two scenarios described in Examples 3.1 and 3.2, $\mathsf{sbf}_{\Gamma_1}(t)$ and $\mathsf{sbf}_{\Gamma_2}(t)$ correspond to the actual minimum resource that Γ_1 and Γ_2 provide, respectively. It is also worth noting that, for the scenario described in Example 3.1, the compositional analysis for the MPR model [31] is compatible[12] with the underlying gEDF schedulability test under the improved SBF but not under the original SBF in Eq. (3.1). In the next example, we further demonstrate the benefits of the improved SBF in terms of resource bandwidth saving.

Example 3.3. *Consider a component C with a taskset $\tau = \{\tau_1 = \cdots = \tau_4 = (200, 100, 200)\}$ that is scheduled under gEDF, and the period of the MPR interface of C is fixed to be 40. Following the interface computation method in [31], the*

[12]We say that a compositional analysis method is compatible with the underlying component's schedulability test it uses if whenever a component C with a taskset τ is deemed schedulable on m cores by the schedulability test, then C is also deemed schedulable under an interface with bandwidth no larger than m by the compositional analysis method.

corresponding minimum-bandwidth MPR interfaces, Γ_1 *and* Γ_2*, of* C *when using the original SBF in Eq.* (3.1) *and when using the improved SBF in Eq.* (3.2) *are obtained as follows:* $\Gamma_1 = \langle 40, 145, 4\rangle$ *and* $\Gamma_2 = \langle 40, 120, 3\rangle$*. Thus, the MPR interface of* C *corresponding to the improved SBF can save* $145/40 - 120/40 = 0.625$ *cores compared to the interface corresponding to the original SBF proposed in [31].*

3.3 Deterministic multiprocessor periodic resource model

In this section, we introduce the deterministic multiprocessor resource model (DMPR) for representing the interfaces. The MPR model described in the previous section is simple and highly flexible because it represents the collective resource requirements of components without fixing the contribution of each processor a priori. However, this flexibility also introduces some extra overhead: it is possible that all processors stop providing resources at the same time, which results in a long worst-case starvation interval (it can be as long as $2(\tilde{\Pi} - \lceil \tilde{\Theta}/m' \rceil)$ time units [31]). Therefore, to ensure schedulability in the worst case, it is necessary to provide more resources than strictly required. However, we can minimize this overhead by restricting the supply pattern of some of the processors. This is a key element of the deterministic MPR that we now propose.

A DMPR model is a deterministic extension of the MPR model, in which all of the processors but one always provide resource with full capacity. It is formally defined as follows.

Definition 3.1. *A DMPR* $\mu = \langle \Pi, \Theta, m\rangle$ *specifies a resource that guarantees* m *full (dedicated) unit-capacity processors, each of which provides* t *resource units in any time interval of length* t*, and one partial processor that provides* Θ *resource units in every period of* Π *time units, where* $0 \leq \Theta < \Pi$ *and* $m \geq 0$*.*

By definition, the resource bandwidth of a DMPR $\mu = \langle \Pi, \Theta, m \rangle$ is $\mathsf{bw}_\mu = m + \frac{\Theta}{\Pi}$. The total number of processors of μ is $m_\mu = m+1$, if $\Theta > 0$, and $m_\mu = m$, otherwise.

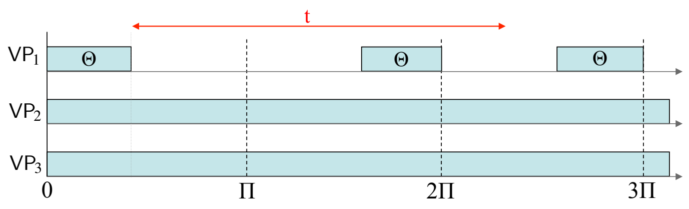


Figure 3.3: Worst-case resource supply pattern of $\mu = \langle \Pi, \Theta, m \rangle$.

Observe that the partial processor of μ is represented by a single-processor periodic resource model $\Omega = (\Pi, \Theta)$ [57]. (However, it can also be represented by any other single processor resource model, such as EDP model [30].) Based on this characteristic, we can easily derive the worst-case supply pattern of μ (shown in Figure 3.3) and its supply bound function, which is given by the following lemma:

Lemma 3.2. *The supply bound function of a DMPR model $\mu = \langle \Pi, \Theta, m \rangle$ is given by:*

$$\mathsf{sbf}_\mu(t) = \begin{cases} mt, & \textit{if } \Theta = 0 \ \vee \ (0 \le t \le \Pi - \Theta) \\ mt + y\Theta + \max\{0, t - 2(\Pi - \Theta) - y\Pi\}, & \textit{otherwise} \end{cases}$$

where $y = \left\lfloor \frac{t-(\Pi-\Theta)}{\Pi} \right\rfloor$, *for all* $t > \Pi - \Theta$.

Proof. Consider any interval of length t. Since the full processors of μ are always available, μ provides the minimum resource supply iff the partial processor provides the worst-case supply. Since the partial processor is a single-processor periodic resource model $\Omega = (\Pi, \Theta)$, its minimum resource supply in an interval of length t is given by [57]: $\mathsf{sbf}_\Omega(t) = 0$, if $\Theta = 0$ or $0 \le t \le \Pi - \Theta$; otherwise, $\mathsf{sbf}_\Omega(t) = y\Theta + \max\{0, t - 2(\Pi - \Theta) - y\Pi\}$ where $y = \left\lfloor \frac{t-(\Pi-\Theta)}{\Pi} \right\rfloor$. In addition, the m full processors of μ provides a total of mt resource units in any interval of length t. Hence, the minimum resource supply of μ in an interval of length t is $mt + \mathsf{sbf}_\Omega(t)$. This proves the lemma. □

It is easy to show that, when a DMPR μ and an MPR Γ have the same period, bandwidth, and total number of processors, then $\mathsf{sbf}_{\mu}(t) \geq \mathsf{sbf}_{\Gamma}(t)$ for all $t \geq 0$, and the worst-case starvation interval of μ is always shorter than that of Γ.

3.4 Overhead-free compositional analysis

In this section, we present our method for computing the minimum-bandwidth DMPR interface for a component, assuming that the cache-related overhead is negligible. The overhead-aware interface computation is considered in the next sections. We first recall some key results for components that are scheduled under gEDF [31].

3.4.1 Component schedulability under gEDF

The demand of a task τ_i in a time interval $[a, b]$ is the amount of computation that must be completed within $[a, b]$ to ensure that all jobs of τ_i with deadlines within $[a, b]$ are schedulable. When $\tau_i = (p_i, e_i, d_i)$ is scheduled under gEDF, its demand in any interval of length t is upper bounded by [31]:

$$\begin{aligned} \mathsf{dbf}_i(t) &= \Big\lfloor \frac{t + (p_i - d_i)}{p_i} \Big\rfloor e_i + CI_i(t), \text{ where} \\ CI_i(t) &= \min\Big\{e_i, \max\Big\{0, t - \Big\lfloor \frac{t + (p_i - d_i)}{p_i} \Big\rfloor p_i\Big\}\Big\}. \end{aligned} \tag{3.3}$$

In Eq. (3.3), $CI_i(t)$ denotes the maximum carry-in demand of τ_i in any time interval $[a, b]$ with $b - a = t$, i.e., the maximum demand generated by a job of τ_i that is released prior to a but has not finished its execution requirement at time a.

Consider a component C with a taskset $\tau = \{\tau_1, ...\tau_n\}$, where $\tau_i = (p_i, e_i, d_i)$, and suppose the tasks in C are schedulable under gEDF by a multiprocessor resource with m' processors. From [31], the worst-case demand of C that must be guaranteed to ensure the schedulability of τ_k in a time interval $(a, b]$, with $b - a = t \geq d_k$ is bounded by:

$$\mathsf{DEM}(t, m') = m'e_k + \sum_{\tau_i \in \tau} \hat{I}_{i,2} + \sum_{i: i \in L_{(m'-1)}} (\bar{I}_{i,2} - \hat{I}_{i,2}) \tag{3.4}$$

$$\begin{aligned}
\text{where} \quad \hat{I}_{i,2} &= \min\big\{ \mathsf{dbf}_i(t) - CI_i(t),\ t - e_k \big\},\ \forall\, i \neq k, \\
\hat{I}_{k,2} &= \min\big\{ \mathsf{dbf}_k(t) - CI_k(t) - e_k,\ t - d_k \big\}; \\
\bar{I}_{i,2} &= \min\big\{ \mathsf{dbf}_i(t),\ t - e_k \big\},\ \forall\, i \neq k, \\
\bar{I}_{k,2} &= \min\big\{ \mathsf{dbf}_k(t) - e_k,\ t - d_k \big\};
\end{aligned}$$

and $L_{(m'-1)}$ is the set of indices of all tasks τ_i that have $\bar{I}_{i,2} - \hat{I}_{i,2}$ being one of the $(m'-1)$ largest such values for all tasks.[13] This leads to the following schedulability test for C:

Theorem 3.3 ([31])**.** *A component C with a task set $\tau = \{\tau_1, ...\tau_n\}$, where $\tau_i = (p_i, e_i, d_i)$, is schedulable under gEDF by a multiprocessor resource model R with m' processors in the absence of overhead if, for each task $\tau_k \in \tau$ and for all $t \geq d_k$, $\mathsf{DEM}(t, m') \leq \mathsf{sbf}_R(t)$, where $\mathsf{DEM}(t, m')$ is given by Eq.* (3.4) *and $\mathsf{sbf}_R(t)$ gives the minimum total resource supply by R in an interval of length t.*

3.4.2 DMPR interface computation

In the absence of cache-related overhead, the minimum resource supply provided by a DMPR model $\mu = \langle \Pi, \Theta, m \rangle$ in any interval of length t is $\mathsf{sbf}_\mu(t)$, which is given by Lemma 3.2. Since each domain schedules its tasks under gEDF, the following theorem follows directly from Theorem 3.3.

Theorem 3.4. *A domain $\mathcal{D}$ with a task set $\tau = \{\tau_1, ...\tau_n\}$, where $\tau_i = (p_i, e_i, d_i)$, is schedulable under gEDF by a DMPR model $\mu = (\Pi, \Theta, m)$ if, for each $\tau_k \in \tau$ and*

[13]Here, d_k and t refer to D_k and $A_k + D_k$ in [31], respectively.

for all $t \geq d_k$,

$$\mathsf{DEM}(t, m_\mu) \leq \mathsf{sbf}_\mu(t), \tag{3.5}$$

where $m_\mu = m + 1$ if $\Theta > 0$, and $m_\mu = m$ otherwise.

We say that μ is a feasible DMPR for $\mathcal{D}$ if it guarantees the schedulability of $\mathcal{D}$ according to Theorem 3.4.

The next theorem derives a bound of the value t that needs to be checked in Theorem 3.4.

Theorem 3.5. *If Eq.* (3.5) *is violated for some value t, then it must also be violated for a value that satisfies the condition*

$$t < \frac{C_\Sigma + m_\mu e_k + U + B}{\frac{\Theta}{\Pi} + m - U_T} \tag{3.6}$$

where C_Σ is the sum of the $m_\mu - 1$ largest e_i; $U = \sum_{i=1}^{n}(p_i - d_i)\frac{e_i}{p_i}$; $U_T = \sum_{i=1}^{n}\frac{e_i}{p_i}$; and $B = 2\frac{\Theta}{\Pi}(\Pi - \Theta)$.

Proof. The proof follows a similar line with the proof of Theorem 2 in [31]. Recall that $\mathsf{DEM}(t, m_\mu)$ is given by Eq. (3.4). According to Eq. (3.4), we have

$$\hat{I}_{i,2} \leq \lfloor \frac{t + (p_i - d_i)}{p_i} \rfloor e_i \leq \frac{t + (p_i - d_i)}{p_i} e_i \leq t\frac{e_i}{p_i} + \frac{p_i - d_i}{p_i} e_i.$$

Therefore,

$$\sum_{i=1}^{n} \hat{I}_{i,2} \leq \sum_{i=1}^{n} t\frac{e_i}{p_i} + \sum_{i=1}^{n} \frac{p_i - d_i}{p_i} e_i = tU_T + U.$$

Because the carry-in workload of τ_i is no more than e_i, we derive $\sum\limits_{i:i \in L_{(m_\mu - 1)}} (\bar{I}_{i,2} - \hat{I}_{i,2}) \leq C_\Sigma$. Thus,

$$\mathsf{DEM}(t, m_\mu) \leq m_\mu e_k + tU_T + U + C_\Sigma.$$

Further, $\mathsf{sbf}_\mu(t)$ gives the worst-case resource supply of the DMPR model $\mu = \langle \Pi, \Theta, m \rangle$ over any interval of length t. Based on Lemma 3.2, the resource supply of

μ is total resource supply of one partial VCPU (Π, Θ) and m full VCPUs. From [57], the resource supply of the partial VCPU (Π, Θ) over any interval of length t is at least $\frac{\Theta}{\Pi}(t - 2(\Pi - \Theta))$. In addition, the resource supply of m full VCPUs over any interval of length t is mt. Hence, the resource supply of μ over any interval of length t is at least $mt + \frac{\Theta}{\Pi}(t - 2(\Pi - \Theta))$. In other words,

$$\mathsf{sbf}_\mu(t) \geq mt + \frac{\Theta}{\Pi}(t - 2(\Pi - \Theta)).$$

Suppose Eq. (3.5) is violated, i.e., $\mathsf{DEM}(t, m_\mu) > \mathsf{sbf}_\mu(t)$ for some value t. Then, combine with the above results, we imply

$$m_\mu e_k + tU_T + U + C_\Sigma > mt + \frac{\Theta}{\Pi}(t - 2(\Pi - \Theta)),$$

which is equivalent to

$$t < \frac{C_\Sigma + m_\mu e_k + U + B}{\frac{\Theta}{\Pi} + m - U_T}.$$

Hence, if Eq. (3.5) is violated for some value t, then t must satisfy Eq. (3.6). This proves the theorem. $\square$

The next lemma gives a condition for the minimum-bandwidth DMPR interface with a given period Π.

Lemma 3.6. *A DMPR model $\mu^* = \langle \Pi, \Theta^*, m^* \rangle$ is the minimum-bandwidth DMPR with period Π that can guarantee the schedulability of a domain $\mathcal{D}$ only if $m^* \leq m$ for all DMPR models $\mu = \langle \Pi, \Theta, m \rangle$ that can guarantee the schedulability of a domain $\mathcal{D}$.*

Proof. Suppose $m^* > m$ for some DMPR $\mu = \langle \Pi, \Theta, m \rangle$. Then, $m^* \geq m + 1$ and, hence, $\mathsf{bw}_{\mu^*} = m^* + \Theta^*/\Pi \geq m + 1 + \Theta^*/\Pi \geq m + 1$. Since $\Theta < \Pi$, $\mathsf{bw}_\mu = m + \Theta/\Pi < m + 1$. Thus, $\mathsf{bw}_{\mu^*} > \mathsf{bw}_\mu$, which implies that m^* cannot be the minimum-bandwidth DMPR with period Π. Hence the lemma. $\square$

Computing the domains' interfaces. Let $\mathcal{D}_i$ be a domain in the system and Π_i be its given VCPU period (c.f. Section 3.1). The minimum-bandwidth interface of $\mathcal{D}_i$ with period Π_i is the minimum-bandwidth DPRM model $\mu_i = \langle \Pi_i, \Theta_i, m_i \rangle$ that is feasible for $\mathcal{D}_i$. To obtain μ_i, we perform binary search on the number of full processors m'_i, and, for each value m'_i, we compute the smallest value of Θ'_i such that $\langle \Theta'_i, \Pi_i, m'_i \rangle$ is feasible for $\mathcal{D}_i$ (using Theorem 3.4).[14] Then m_i is the smallest value of m'_i for which a feasible interface is found, and, Θ_i is the smallest budget Θ'_i computed for m_i.

Computing the system's interface. The interface of the system can be obtained by composing the interfaces μ_i of all domains $\mathcal{D}_i$ in the system under the VMM's semi-partitioned EDF policy (c.f. Section 3.1). Let D denote the number of domains of the platform.

Observe that each interface $\mu_i = \langle \Pi_i, \Theta_i, m_i \rangle$ can be transformed directly into an equivalent set of m_i full VCPUs (with budget Π_i and period Π_i) and, if $\Theta_i > 0$, a partial VCPU with budget Θ_i and period Π_i. Let $\mathcal{C}$ be a component that contains all the partial VCPUs that are transformed from the domains' interfaces. Then the VCPUs in $\mathcal{C}$ are scheduled together under gEDF, whereas all the full VCPUs are each mapped to a dedicated core.

Since each partial VCPU in $\mathcal{C}$ is implemented as a periodic server, which is essentially a periodic task, we can compute the minimum-bandwidth DMPR interface $\mu_{\mathcal{C}} = \langle \Pi_{\mathcal{C}}, \Theta_{\mathcal{C}}, m_{\mathcal{C}} \rangle$ that is feasible for $\mathcal{C}$ by the same technique used for domains. Combining $\mu_{\mathcal{C}}$ with the full VCPUs of the domains, we can see that the system must be guaranteed $m_{\mathcal{C}} + \sum_{1 \le i \le D} m_i$ full processors and a partial processor, with budget $\Theta_{\mathcal{C}}$ and period $\Pi_{\mathcal{C}}$, to ensure the schedulability of the system. The next theorem directly follows from this observation.

[14]Note that the number of full processors is always bounded from below by $\lfloor U_i \rfloor$, where U_i is the total utilization of the tasks in $\mathcal{D}_i$, and bounded from above by the number of tasks in $\mathcal{D}_i$ or the number of physical platform (if given), whichever is smaller.

Theorem 3.7. *Let $\mu_i = \langle \Pi_i, \Theta_i, m_i \rangle$ be the minimum-bandwidth DMPR interface of domain $\mathcal{D}_i$, for all $1 \le i \le D$. Let $\mathcal{C}$ be a component with the taskset*

$$\tau_{\mathcal{C}} = \{(\Pi_i, \Theta_i, \Pi_i) \mid 1 \le i \le D \wedge \Theta_i > 0\},$$

which are scheduled under gEDF. Then the minimum-bandwidth DMPR interface with period $\Pi_{\mathcal{C}}$ of the system is given by: $\mu_{\mathsf{sys}} = \langle \Pi_C, \Theta_C, m_{\mathsf{sys}} \rangle$, where $\mu_{\mathcal{C}} = \langle \Pi_{\mathcal{C}}, \Theta_{\mathcal{C}}, m_{\mathcal{C}} \rangle$ is a minimum-bandwidth DMPR interface with period $\Pi_{\mathcal{C}}$ of $\mathcal{C}$ and $m_{\mathsf{sys}} = m_{\mathcal{C}} + \sum_{1 \le i \le D} m_i$.

Based on the system's interface, one can easily derive the schedulability of the system as follows (the lemma comes directly from the interface's definition):

Lemma 3.8. *Let M be the number of physical cores of the platform. The system is schedulable if $M \ge m_{\mathsf{sys}} + 1$, or, $M = m_{\mathsf{sys}}$ and $\Theta_C = 0$, where $\langle \Pi_C, \Theta_C, m_{\mathsf{sys}} \rangle$ is the minimum-bandwidth DMPR system's interface.*

The results obtained above assume that the cache-related overhead is negligible. We will next develop the analysis in the presence of cache-related overhead.

3.5 Cache-related overhead scenarios

In this section, we characterize the different events that cause cache-related overhead; this is needed for the cache-aware analysis in Sections 3.6 and 3.7.

Cache-related overhead in a multicore virtualization platform is caused by (1) task preemption within the same domain, (2) VCPU preemption, and (3) VCPU exhaustion of budget. We discuss each of them in detail below.

3.5.1 Task-preemption event

Since tasks within a domain are scheduled under gEDF, a newly released higher-priority task preempts a currently executing lower-priority task of the *same* domain, if none of the domain's VCPUs are idle. When a preempted task resumes its execution, it may experience cache misses: its cache content may have been evicted from the cache by the preempting task (or tasks with a higher priority than the preempting task, if a nested preemption occurs), or the task may be resumed on a different VCPU that is running on a different core, in which case the task's cache content may not be present in the new core's cache. Hence the following definition:

Definition 3.2 (Task-preemption event). *A task-preemption event of τ_i is said to occur when a job of another task τ_j in the same domain is released and this job can preempt the current job of τ_i.*

Fig. 3.4 illustrates the worst-case scenario of the overhead caused by a task-preemption event. In the figure, a preemption event of τ_1 happens at time $t = 3$ when τ_3 is released (and preempts τ_1). Due to this event, τ_1 experiences a cache miss at time $t = 5$ when it resumes. Since τ_1 resumes on a different core, all the cache blocks it will reuse have to be reloaded into new core's cache, which results in cache-related preemption/migration overhead on τ_1. (Note that the cache content of τ_1 is not necessarily reloaded all at once, but rather during its remaining execution after it has been resumed; however, for ease of exposition, we show the combined overhead at the beginning of its remaining execution).

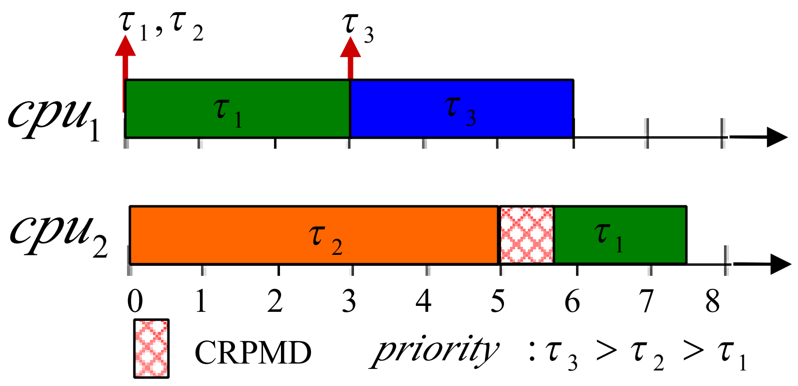


Figure 3.4: Cache-related overhead of a task-preemption event.

Since gEDF is work-conserving, tasks do not suspend themselves, and each task resumes at most once after each time it is preempted. Therefore, each task τ_k experiences the overhead caused by each of its task-preemption events at most once, and this overhead is bounded from above by $\Delta^{\mathsf{crpmd}}_{\tau_k}$.

Lemma 3.9. *A newly released job of τ_j preempts a job of τ_i under gEDF only if $d_j < d_i$.*

Proof. Suppose $d_j \geq d_i$ and a newly released job J_j of τ_j preempts a job J_i of τ_i. Then, J_j must be released later than J_i. As a result, the absolute deadline of J_j is later than J_i's (since $d_j \geq d_i$), which contradicts the assumption that J_j preempts J_i under gEDF. This proves the lemma. $\square$

The maximum number of task-preemption events in each period of τ_i is given by the next lemma.

Lemma 3.10 (Number of task-preemption events). *The maximum number of task-preemption events of τ_i under gEDF during each period of τ_i, denoted by $N^1_{\tau_i}$, is bounded by*

$$N^1_{\tau_i} \leq \sum_{\tau_j \in \mathsf{HP}(\tau_i)} \left\lceil \frac{d_i - d_j}{p_j} \right\rceil \tag{3.7}$$

where $\mathsf{HP}(\tau_i)$ is the set of tasks τ_j within the same domain with τ_i with $d_j < d_i$.

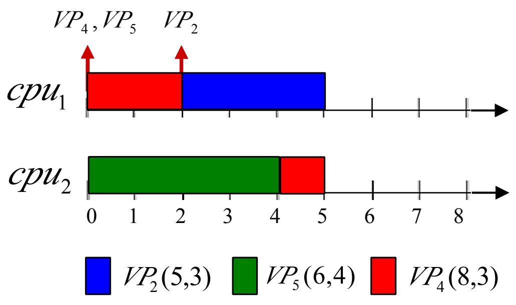
Proof. Let τ_i^c be the current job of τ_i in a period of τ_i, and let r_i^c be its release time. From Lemma 3.9, only jobs of a task τ_j with $d_j < d_i$ and in the same domain can preempt τ_i^c. Further, for each such τ_j, only the jobs that are released after τ_i^c and that have absolute deadlines no later than τ_i^c's can preempt τ_i^c. In other words, only jobs that are released within the interval $(r_i^c, r_i^c + d_i - d_j]$ can preempt τ_i^c. As a result, the maximum number of task-preemption events of τ_i under gEDF is no more than $\sum_{\tau_j \in \mathsf{HP}(\tau_i)} \left\lceil \frac{d_i - d_j}{p_j} \right\rceil$. $\square$

3.5.2 VCPU-preemption event

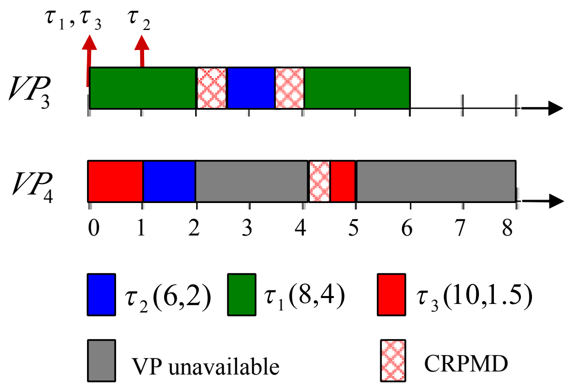
Definition 3.3 (**VCPU-preemption event**)**.** *A VCPU-preemption event of* VP_i *occurs when* VP_i *is preempted by a higher-priority VCPU* VP_j *of another domain.*

When a VCPU VP_i is preempted, the currently running task τ_l on VP_i may migrate to another VCPU VP_k of the same domain and may preempt the currently running task τ_m on VP_k. This can cause the tasks running on VP_k experiences cache-related preemption or migration overhead twice in the worst case, as is illustrated in the following example.

Example 3.4. *The system consists of three domains* $\mathcal{D}_1$*-*$\mathcal{D}_3$*.* $\mathcal{D}_1$ *has VCPUs* VP_1 *(full) and* VP_2 *(partial);* $\mathcal{D}_2$ *has VCPUs* VP_3 *(full) and* VP_4 *(partial); and* $\mathcal{D}_3$ *has one partial VCPU* VP_5*. The partial VCPUs of the domains –* $\mathsf{VP}_2(5,3)$*,* $\mathsf{VP}_4(8,3)$ *and* $\mathsf{VP}_5(6,4)$ *– are scheduled under gEDF on* cpu_1 *and* cpu_2*, as is shown in Fig. 3.5(a). In addition, domain* $\mathcal{D}_2$ *consists of three tasks,* $\tau_1(8,4,8)$*,* $\tau_2(6,2,6)$ *and* $\tau_3(10,1.5,10)$*, which are scheduled under gEDF on its VCPUs (Fig. 3.5(b)).*



(a) Scheduling scenario of VCPUs.



(b) Cache overhead of tasks in $\mathcal{D}_2$.

Figure 3.5: Cache overhead due to a VCPU-preemption event.

As is shown in Fig. 3.5(a), a VCPU-preemption event occurs at time $t = 2$*, when* VP_4 *(of* $\mathcal{D}_2$*) is preempted by* VP_2*. Observe that, within* $\mathcal{D}_2$ *at this instant,* τ_2 *is running on* VP_4 *and* τ_1 *is running on* VP_3*. Since* τ_2 *has an earlier deadline than* τ_1*,* *it is migrated to VP_3 and preempts τ_1 there. Since VP_3 is mapped to a different core from cpu_1, τ_2 has to reload its useful cache content to the cache of the new core at $t = 2$. Further, when τ_1 resumes at time $t = 3.5$, it has to reload the useful cache blocks that may have been evicted from the cache by τ_2. Hence, the VCPU-preemption event of VP_4 causes overhead for both of the tasks in its domain.*

Lemma 3.11. *Each VCPU-preemption event causes at most two tasks to experience a cache miss. Further, the cache-related overhead it causes is at most $\Delta^{\mathsf{crpmd}}_{\mathsf{C}} = \max_{\tau_i \in C} \Delta^{\mathsf{crpmd}}_{\tau_i}$, where C is the component that has the preempted VCPU.*

Proof. At most one task is running on a VCPU at any time. Hence, when a VCPU VP_i of C is preempted, at most one task (τ_m) on VP_i is migrated to another VCPU VP_j, and this task preempts at most one task (τ_l) on VP_j. As a result, at most two tasks (i.e., τ_m and τ_l) incur a cache miss because of the VCPU-preemption event. (Note that τ_l cannot immediately preempt another task τ_n because otherwise, τ_m would have migrated to the VCPU on which τ_n is running and preempted τ_n instead.) Further, since the overhead caused by each cache miss in C is at most $\Delta^{\mathsf{crpmd}}_{\mathsf{C}} = \max_{\tau_i \in C} \Delta^{\mathsf{crpmd}}_{\tau_i}$, the maximum overhead caused by the resulting cache misses is at most $2\Delta^{\mathsf{crpmd}}_{\mathsf{C}}$. □

Since the partial VCPUs are scheduled under gEDF as implicit-deadline tasks (i.e., the task periods are equal to their relative deadlines), the number of VCPU-preemption events of a partial VCPU VP_i during each VP_i's period also follows Lemma 3.10. The next lemma is implied directly from this observation.

Lemma 3.12 (Number of VCPU-preemption events). *Let $\mathsf{VP}_i = (\Pi_i, \Theta_i)$ for all partial VCPUs VP_i of the domains. Let $\mathsf{HP}(\mathsf{VP}_i)$ be the set of VP_j with $0 < \Theta_j < \Pi_j < \Pi_i$. Denote by $N^2_{\mathsf{VP}_i}$ and $N^2_{\mathsf{VP}_i,\tau_k}$ the maximum number of VCPU-preemption events of VP_i during each period of VP_i and during each period of τ_k*

inside VP_i*'s domain, respectively. Then,*

$$N^2_{\mathsf{VP}_i} \leq \sum_{\mathsf{VP}_j \in \mathsf{HP}(\mathsf{VP}_i)} \left\lceil \frac{\Pi_i - \Pi_j}{\Pi_j} \right\rceil \tag{3.8}$$

$$N^2_{\mathsf{VP}_i, \tau_k} \leq \sum_{\mathsf{VP}_j \in \mathsf{HP}(\mathsf{VP}_i)} \left\lceil \frac{p_k}{\Pi_j} \right\rceil. \tag{3.9}$$

3.5.3 VCPU-completion event

Definition 3.4 (VCPU-completion event). *A VCPU-completion event of* VP_i *happens when* VP_i *exhausts its budget in a period and stops its execution.*

Like in VCPU-preemption events, each VCPU-completion event causes at most two tasks to experience a cache miss, as given by Lemma 3.13.

Lemma 3.13. *Each VCPU-completion event causes at most two tasks to experience a cache miss.*

Proof. The effect of a VCPU-completion event is very similar to that of a VCPU-preemption event. When VP_i finishes its budget and stops, the running task τ_m on VP_i may migrate to another running VCPU VP_j, and, τ_m may preempt at most one task τ_l on VP_j. Hence, at most two tasks incur a cache miss due to a VCPU-preemption event. □

Lemma 3.14 (Number of VCPU-completion events). *Let* $N^3_{\mathsf{VP}_i}$ *and* $N^3_{\mathsf{VP}_i, \tau_k}$ *be the number of VCPU-completion events of* VP_i *in each period of* VP_i *and in each period of* τ_k *inside* VP_i*'s domain. Then,*

$$N^3_{\mathsf{VP}_i} \leq 1 \tag{3.10}$$

$$N^3_{\mathsf{VP}_i, \tau_k} \leq \left\lceil \frac{p_k - \Theta_i}{\Pi_i} \right\rceil + 1 \tag{3.11}$$

Proof. Eq. (3.10) holds because VP_i completes its budget at most once every period. Further, observe that τ_i experiences the worst-case number of VCPU-preemption events when (1) its period ends at the same time as the budget finish time of VP_i's current period, and (2) VP_i finishes its budget as soon as possible (i.e., B_i time units from the beginning of the VCPU's period) in the current period and as late as possible (i.e., at the end of the VCPU's period) in all its preceding periods. Eq. (3.11) follows directly from this worst-case scenario. $\square$

VCPU-stop event. Since a VCPU stops its execution when its VCPU-completion or VCPU-preemption event occurs, we define a *VCPU-stop event* that includes both types of events. That is, a VCPU-stop event of VP_i occurs when VP_i stops its execution because its budget is finished *or* because it is preempted by a higher-priority VCPU. Since VCPU-stop events include both VCPU-completion events and VCPU-preemption events, the maximum number of VCPU-stop events of VP_i during each VP_i's period, denoted as $N^{\mathsf{stop}}_{\mathsf{VP}_i}$, satisfies

$$N^{\mathsf{stop}}_{\mathsf{VP}_i} = N^2_{\mathsf{VP}_i} + N^3_{\mathsf{VP}_i} \leq \sum_{\mathsf{VP}_j \in \mathsf{HP}(\mathsf{VP}_i)} \left\lceil \frac{\Pi_i - \Pi_j}{\Pi_j} \right\rceil + 1 \tag{3.12}$$

Overview of the overhead-aware compositional analysis. Based on the above quantification, in the next two sections we develop two different approaches, task-centric and model-centric, for the overhead-aware interface computation. Although the obtained interfaces by both approaches are safe and can each be used independently, we combine them to obtain the interface with the smallest bandwidth as the final result.

3.6 Task-centric compositional analysis

This section introduces two task-centric analysis methods to account for the cache-related overhead in the interface computation. The first, denoted as BASELINE, accounts for the overhead by inflating the WCET of every task in the system with the maximum overhead it experiences within each of its periods. The second, denoted as TASK-CENTRIC-UB, combines the result of the first method using an upper bound on the number of VCPUs that each domain needs in the presence of cache-related overhead. We describe each method in detail below.

3.6.1 BASELINE: Analysis based on WCET-inflation

As was discussed in Section 3.5, the overhead that a task experiences during its lifetime is composed of the overhead caused by task-preemption events, VCPU-preemption events and VCPU-completion events. In addition, when one of the above events occurs, each task τ_k experiences at most one cache miss overhead and, hence, a delay of at most $\Delta^{\mathsf{crpmd}}_{\tau_k}$. From [24], the cache overhead caused by a task-preemption event can be accounted for by inflating the higher-priority task τ_i of the event with the maximum cache overhead caused by τ_i. From Lemmas 3.12 and 3.14, we conclude that the maximum overhead τ_k experiences within each period is

$$\delta^{\mathsf{crpmd}}_{\tau_k} = \max_{\tau_i \in \mathsf{LP}(\tau_k)} \{\Delta^{\mathsf{crpmd}}_{\tau_i}\} + \Delta^{\mathsf{crpmd}}_{\tau_k}(N^2_{\mathsf{VP}_i,\tau_k} + N^3_{\mathsf{VP}_i,\tau_k})$$

where $\mathsf{LP}(\tau_k)$ is the set of tasks τ_i within the same domain with τ_k with $d_i > d_k$ and VP_i is the partial VCPU of the domain of τ_k. As a result, the worst-case execution time of τ_k in the presence of cache overhead is at most

$$e'_k = e_k + \delta^{\mathsf{crpmd}}_{\tau_k}. \tag{3.13}$$

Thus, we can state the following theorem:

Theorem 3.15. *A component with a taskset $\tau = \{\tau_1, ...\tau_n\}$, where $\tau_k = (p_k, e_k, d_k)$, is schedulable under gEDF by a DMPR model μ in the presence of cache-related overhead if its inflated taskset $\tau' = \{\tau'_1, ...\tau'_n\}$ is schedulable under gEDF by μ in the absence of cache-related overhead, where $\tau'_k = (p_k, e'_k, d_k)$, and e'_k is given by Eq. 3.13.*

Based on Theorem 3.15, we can compute the DMPR interfaces of the domains and the system by first inflating the WCET of each task τ_k in each domain with the overhead $\delta^{\mathsf{crpmd}}_{\tau_\mathsf{k}}$ and then applying the same method as the overhead-free interface computation in Section 3.4.2.[15]

3.6.2 TASK-CENTRIC-UB: Combination of BASELINE with an upper bound on the number of VCPUs

Recall from Section 3.5 that, VCPU-preemption events and VCPU-completion events happen only when the component has a partial VCPU. Therefore, the taskset in a component with no partial VCPU experiences only the cache overhead caused by task-preemption events. Recall that when a task-preemption event happens, the corresponding lower-priority task τ_i experiences a cache miss delay of at most $\Delta^{\mathsf{crpmd}}_{\tau_\mathsf{i}}$. Thus, the maximum cache overhead that a high-priority task τ_k causes to any preempted task is $\max_{\tau_i \in \mathsf{LP}(\tau_k)} \Delta^{\mathsf{crpmd}}_{\tau_\mathsf{i}}$, where $\mathsf{LP}(\tau_k)$ is the set of tasks τ_i within the same domain with τ_k that have $d_i > d_k$. As a result, the worst-case execution time of τ_k in the presence of cache overhead caused by task-preemption events is at most

$$e''_k = e_k + \max_{\tau_i \in \mathsf{LP}(\tau_k)} \Delta^{\mathsf{crpmd}}_{\tau_\mathsf{i}}, \tag{3.14}$$

where $\tau_i \in \mathsf{LP}(\tau_k)$ if $d_i > d_k$. This implies the following lemma:

Lemma 3.16. *A component with a taskset $\tau = \{\tau_1, ..., \tau_n\}$, where $\tau_k = (p_k, e_k, d_k)$, is schedulable under gEDF by a DMPR model $\bar{\mu} = \langle \Pi, 0, \bar{m} \rangle$ in the presence of cache-related overhead if its inflated taskset $\tau'' = \{\tau''_1, ..., \tau''_n\}$ is schedulable under gEDF by $\mu'' = \langle \Pi, \Theta'', m'' \rangle$ in the absence of cache-related overhead, where $\tau''_k = (p_k, e''_k, d_k)$, e''_k is given by Eq. 3.14, and $\bar{m} = m'' + \lceil \frac{\Theta''}{\Pi} \rceil$. Further, the maximum number of full VCPUs of the interface of the taskset τ in the presence of cache overhead is $\bar{m}$.*

Proof. First, observe that the inflated taskset τ'' safely accounts for all the cache

[15]Note that we inflate only the tasks' WCETs and not the VCPUs' budgets, since $\delta^{\mathsf{crpmd}}_{\tau_\mathsf{k}}$ includes the overhead for reloading the useful cache content of a preempted VCPU when it resumes.

overhead experienced by τ. This is because (1) inflating the worst-cache execution time of each task τ_k with $\max_{\tau_i \in \mathsf{LP}(\tau_k)} \Delta^{\mathsf{crpmd}}_{\tau_i}$ is safe to account for the cache overhead delay caused by task-preemption events (as was proven in [24]), and (2) the DMPR model $\bar{\mu}$ has no partial VCPU and thus, τ does not experience any cache overhead caused by VCPU-preemption events or VCPU-completion events. Further, based on Lemma 3.2, one can easily show that the resource supply bound function $\mathsf{sbf}_{\mu}(t)$ of a DMPR model $\mu = \langle \Pi, \Theta, m \rangle$ is monotonically non-decreasing with the budget of μ when the period of μ is fixed. In other words, $\mathsf{sbf}_{\bar{\mu}}(t) \geq \mathsf{sbf}_{\mu''}(t)$ for all t. Combine the above observations, we imply that τ is schedulable under the resource model $\bar{\mu}$ in the presence of cache overhead if τ'' is schedulable under the resource model μ'' in the absence of cache overhead. This proves the first part of the lemma.

Since τ is schedulable under the resource model $\bar{\mu}$ in the presence of cache overhead, the number of full VCPUs of the overhead-aware interface of τ is always less than or equal to the ceiling of the bandwidth of $\bar{\mu}$, which is exactly $\bar{m}$. $\square$

Note that the maximum number of full VCPUs given by Lemma 3.16 can be larger or smaller than the interface bandwidth computed by the BASELINE method, as is illustrated in the following two examples.

Example 3.5. *Consider a system* Sys_1 *consisting of two domains,* C_1 *and* C_2*, with workloads* $\tau_{C_1} = \{\tau_1^1 = \cdots = \tau_1^3 = (100, 40, 100)\}$ *and* $\tau_{C_2} = \{\tau_2^1 = \cdots = \tau_2^3 = (100, 40, 100)\}$*, respectively. Suppose that* Sys_1 *employs the hybrid EDF scheduling strategy described in Section 3.1; the periods of DMPR interfaces of* C_1*,* C_2 *and* Sys_1 *are set to* $80, 40$ *and* 20*, respectively; and the cache overhead per task is* 1*. Then, the DMPR cache-aware interface of* C_1 *computed using the* BASELINE *method is* $\mu_{C_1} = \langle 80, 76, 1 \rangle$*, which has a bandwidth of* $1 + 76/80 = 1.95$*.*

In contrast, if we only consider the cache overhead caused by task-preemption events, then the interface of the system is given by $\mu''_{C_1} = \langle 80, 64, 1 \rangle$*. Based on Lemma 3.16, the maximum number of full VCPUs of* C_1 *is* $1 + 64/80 = 2$*, and the corresponding DMPR interface is* $\bar{\mu}_{C1} = \langle 80, 0, 2 \rangle$*. Thus, the interface computed by* *the* BASELINE *method has a smaller bandwidth than the maximum number of full VCPUs given by Lemma 3.16.*

Example 3.6. *Consider a system* Sys_2 *that is identical to the system* Sys_1 *in Example 3.5, except that the cache overhead for each task is* 5 *instead of* 1*. In this case, the cache-aware interface of* C_1 *computed using the* BASELINE *method is* $\bar{\mu}_{C_1} = \langle 80, 72, 2\rangle$*, which has a bandwidth of* $2 + 72/80 = 2.9$*. In contrast, if we consider only the cache overhead caused by task-preemption events, then the interface of the system is given by* $\mu''_{C_1} = \langle 80, 74, 1\rangle$*. Based on Theorem 3.16, the maximum number of full VCPUs is* $1 + 74/80 = 2$*. Therefore, the interface computed by the* BASELINE *method has a larger bandwidth than the maximum number of full VCPUs given by Lemma 3.16.*

Since the interface $\bar{\mu}$ given by Lemma 3.16 does not always have a smaller bandwidth than the interface computed using the BASELINE method, we combine the two interfaces to derive the minimum-bandwidth DMPR interface in the presence of overhead, as is given by Theorem 3.17. The correctness of this theorem is derived directly from the correctness of Lemma 3.16 and Theorem 3.15.

Theorem 3.17. *Let* C *be a component with a taskset* $\tau = \{\tau_1, ..., \tau_n\}$ *that is schedulable by the gEDF scheduler, where* $\tau_k = (p_k, e_k, d_k)$ *for all* $1 \leq k \leq n$*. Suppose* $\mu'_C = \langle \Pi, \Theta', m'\rangle$ *is the feasible DMPR interface given by Theorem 3.15, and* m'' *is the maximum number of full VCPUs of* C *given by Lemma 3.16. Then, the component* C *is schedulable under the DMPR interface* μ_C*, where* $\mu_C = \mu'_C$ *if* $m'' > m' + \frac{\Theta'}{\Pi}$*, and* $\mu_C = \langle \Pi, 0, m''\rangle$ *otherwise.*

Interface computation under the TASK-CENTRIC-UB method: Based on the above results, the overhead-aware interface for a system can be obtained by first computing the interface for each domain using Theorem 3.17, and then computing the system's interface by applying the overhead-free interface computation in Section 3.4.

3.6.3 TASK-CENTRIC-UB vs. BASELINE

As was discussed in Section 3.6.2, the interface of a domain computed by the TASK-CENTRIC-UB method always has a bandwidth no larger than the bandwidth of the interface computed by the BASELINE method. We will show that this relationship also holds for the interfaces at the system level. We first define the dominance relation between any two analysis methods as follows:

Definition 3.5. *A compositional analysis method CSA is said to dominate another compositional analysis method CSA' iff for any system S, the interface bandwidth of S when computed using CSA is always less than or equal to the interface bandwidth of S when computed using CSA'.*

Lemma 3.18. *The* TASK-CENTRIC-UB *method always dominates the* BASELINE *method.*

Proof. Consider a system S with D domains, $\{C_1, ..., C_D\}$. Let $\mu_{C_i} = \langle \Pi_i, \Theta_i, m_i \rangle$ and $\mu'_{C_i} = \langle \Pi_i, \Theta'_i, m'_i \rangle$ be the minimum-bandwidth DMPR interfaces of C_i under the TASK-CENTRIC-UB method and the BASELINE method, respectively. We have the following:

- Under the TASK-CENTRIC-UB method, the system has a set of partial VCPUs, $\mathsf{VP}_{\mathsf{part}} = \{\mathsf{VP}_1 = (\Pi_1, \Theta_1), ..., \mathsf{VP}_D = (\Pi_D, \Theta_D)\}$, and $(m_1 + ... + m_D)$ full VCPUs. Based on the analysis in Section 3.4, the minimum-bandwidth DMPR interface of S is given by $\mu_S = \langle \Pi_C, \Theta_C, m_S \rangle$, where $\mu_C = \langle \Pi_C, \Theta_C, m_C \rangle$ is the minimum-bandwidth DMPR interface for $\mathsf{VP}_{\mathsf{part}}$ and $m_S = m_C + \sum_{1 \le i \le D} m_i$.
- Under the BASELINE method, the system has a set of partial VCPUs, $\mathsf{VP}'_{\mathsf{part}} = \{\mathsf{VP}'_1 = (\Pi_1, \Theta'_1),\ ..., \mathsf{VP}_D = (\Pi_D, \Theta'_D)\}$ and $(m'_1 + ... + m'_D)$ full VCPUs. Therefore, the minimum-bandwidth DMPR interface system is given by $\mu'_S = \langle \Pi_C, \Theta'_C, m'_S \rangle$, where $\mu'_C = \langle \Pi, \Theta'_C, m'_C \rangle$ is the minimum-bandwidth DMPR interface of the partial VCPU set $\mathsf{VP}'_{\mathsf{part}}$, and $m'_S = m'_C + \sum_{1 \le i \le D} m'_i$.

From Theorem 3.17, there are two cases for the relationship between μ_{C_i} and μ'_{C_i}:

1. $\Theta_i = \Theta'_i$ and $m_i = m'_i$, if the interface bandwidth computed by the BASELINE method is less than or equal to the maximum number of full VCPUs of C_i given by Lemma 3.16 (i.e., $m'_i + \frac{\Theta'_i}{\Pi} \leq m_i + \frac{\Theta_i}{\Pi}$);

2. $\Theta_i = 0$ and $m_i \leq m'_i$, otherwise.

We can conclude from the above cases that for all partial VCPUs VP_i and VP'_i computed respectively by the TASK-CENTRIC-UB method and the BASELINE method, $\mathsf{VP}_i = \mathsf{VP}'_i$, or VP_i has budget equal to 0 whereas VP'_i has budget larger than 0. In other words, $\mathsf{VP}_{\mathsf{part}} \subseteq \mathsf{VP}'_{\mathsf{part}}$.

Because $\mathsf{VP}_{\mathsf{part}}$ is only a subset of $\mathsf{VP}'_{\mathsf{part}}$, we can derive from Eq. (3.4) that the resource demand of $\mathsf{VP}_{\mathsf{part}}$ is always less than or equal to the resource demand of $\mathsf{VP}'_{\mathsf{part}}$. Therefore, if $\mathsf{VP}'_{\mathsf{part}}$ is schedulable under the DMPR interface μ'_C, then $\mathsf{VP}_{\mathsf{part}}$ is also schedulable under μ'_C. Because μ_C is the bandwidth-optimal DMPR interface of $\mathsf{VP}_{\mathsf{part}}$, the bandwidth of μ_C is no larger than the bandwidth of μ'_C, i.e., $\frac{\Theta_C}{\Pi_C} + m_C \leq \frac{\Theta'_C}{\Pi_C} + m'_C$. In addition, $\sum_{1 \leq i \leq D} m_i \leq \sum_{1 \leq i \leq D} m'_i$, because $m_i \leq m'_i$. Hence, the bandwidth of μ_S, which is equal to $\frac{\Theta_C}{\Pi_C} + m_C + \sum_{1 \leq i \leq D} m_i$, is no larger than the bandwidth of μ'_S, which is $\frac{\Theta'_C}{\Pi_C} + m'_C + \sum_{1 \leq i \leq D} m'_i$. This proves the lemma. □

3.7 Model-centric compositional analysis

Recall from Section 3.5 that each VCPU-stop event (i.e., VCPU-preemption or VCPU-completion event) of VP_i causes at most one cache miss overhead for at most two tasks of the same domain. However, since it is unknown which two tasks may be affected, the BASELINE method in Section 3.6 assumes that *every* task τ_k of the same domain is affected by *all* the VCPU-stop events of VP_i (and thus includes all of the corresponding overheads in the inflated WCET of the task). While this approach is safe, it is very conservative, especially when the number of tasks or the number of events is high.

In this section, we propose an alternative method, called MODEL-CENTRIC, that avoids the above assumption to minimize the pessimism of the analysis. The idea is to account for the total overhead due to VCPU-stop events that is incurred by all tasks in a domain, rather than by each task individually. This combined overhead is the overhead that *the domain as a whole* experiences due to VCPU-stop events under a given DMPR interface μ of the domain (since the budget of the partial VCPU of a domain is determined by the domain's interface). Therefore, the effective resource supply that a domain receives from a DMPR interface μ in the presence of VCPU-stop events is the total resource supply that μ provides, less the combined overhead.

3.7.1 Challenge: Resource parallel supply problem

Based on the overhead scenarios in Section 3.5, at first it seems possible to account for the overhead of the VCPU-preemption and VCPU-completion events by inflating the budget of an overhead-free interface with the cache-related overhead caused by the VCPU-preemption and VCPU-completion events that occur within a period of the overhead-free interface. However, this interface budget inflation approach is unsafe, due to the resource parallel supply under multicore interfaces. We illustrate this via the following scenario.

Example 3.7. *Consider a system with a single component C that has a workload $\tau = \{\tau_1 = \tau_2 = (2, 0.1, 2), \tau_3 = (2, 1.81, 2)\}$, which is scheduled under gEDF. We assume that ties are broken based on increasing order of tasks' indices, i.e., a task with a smaller index has a higher priority. Suppose the cache overhead for each task is given by $\Delta^{\mathsf{crpmd}}_{\tau_1} = \Delta^{\mathsf{crpmd}}_{\tau_2} = 0.05$ and $\Delta^{\mathsf{crpmd}}_{\tau_3} = 0.2$. (The time unit is ms.) In this example, we consider only the cache overhead caused by VCPU-preemption and VCPU-completion events and assume that there are no other types of overhead.*

Based on the overhead-free anlaysis in Section 3.4, the taskset τ is schedulable under the DMPR interface $\mu = \langle 2, 1.01, 2\rangle$. Since the interface has only one partial

VCPU and this partial VCPU is not preempted by any other (full) VCPUs, the taskset τ in C experiences no VCPU-preemption event. In addition, at most one VCPU-completion event happens in a period of the DMPR interface μ. Further, based on Section 3.5, each VCPU-completion event causes at most two tasks to experience a cache miss. Therefore, the total cache overhead delay in a DMPR interface's period is at most $2\max_{1\le i\le 3}\{\Delta^{\mathsf{crpmd}}_{\tau_i}\} = 0.4$.

Suppose we inflate the budget of the overhead-free DMPR interface μ with the total cache overhead delay of 0.4*. Then, we obtain the DMPR interface* $\mu' = \langle 2, 1.41, 2\rangle$*. However, the taskset τ is not schedulable under μ', as is illustrated by Fig. 3.6.*

Fig. 3.6(a) shows the resource supply pattern of μ', and Fig. 3.6(b) shows the release and schedule patterns of the tasks in τ. Here, the tasks τ_1, τ_2, and τ_3 are released at $t = 1.01$*. τ_3 migrates from* VCPU_3 *to* VCPU_2 *at* $t = 1.41$ *and occurs a delay of* $\Delta^{\mathsf{crpmd}}_{\tau_3} = 0.2$ *time units to reload its cache content (because* VCPU_3 *completes its budget at* $t = 1.41$*). τ_3 keeps running on* VCPU_2 *for* 1.41 *time units and finishes its execution at* $t = 3.02$*. Since τ_3's absolute deadline is* $t = 3.01$*, τ_3 misses its deadline.*

The flaw in the cache-aware analysis approach that naïvely inflates the interface's budget comes from the resource parallel supply problem of the global multicore scheduling. In the above scenario, when τ_3 experiences cache overhead, its worst-case execution time is enlarged and thus, it needs more CPU time to execute. However, inflating the budget of the interface cannot guarantee that τ_3 receives the inflated budget, e.g., when part of the inflated budget is assigned to a VCPU that supplies resource in parallel with the VCPU on which τ_3 is running. Because τ_3 is not a parallel task and cannot execute on two cores at the same time, τ_3 does not fully utilize the inflated budget. As a result, although the extra budget is enough to account for the cache overhead τ_3 experiences, the inflated budget is not enough to guarantee the schedulability of the taskset under the resource model with inflated budget.

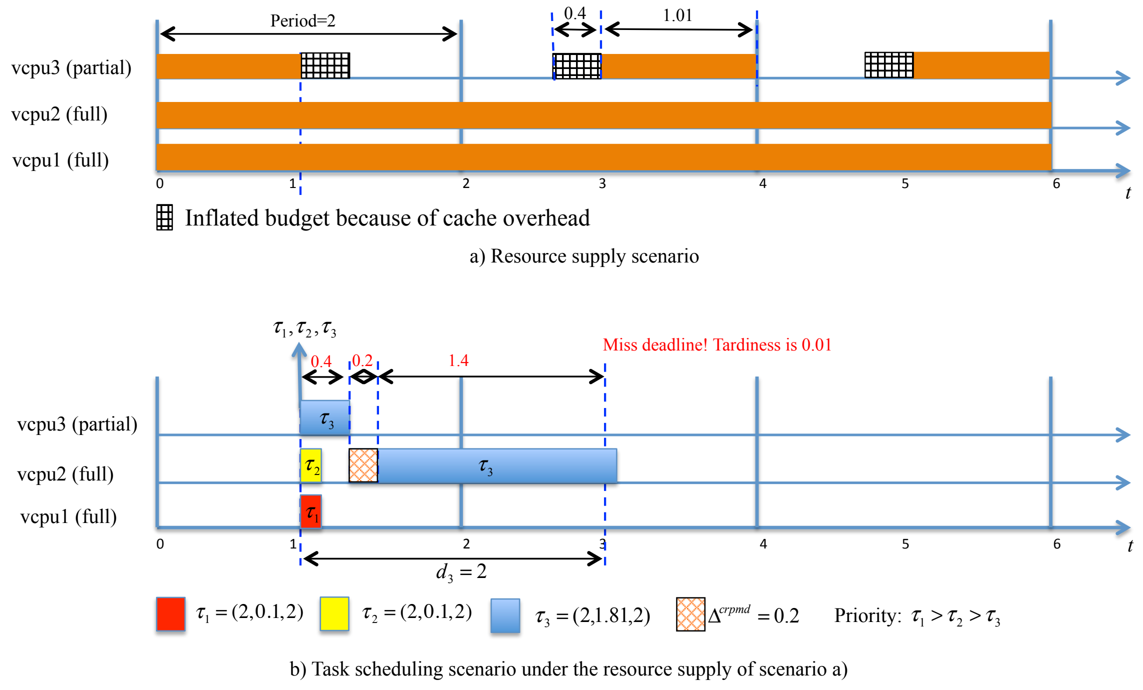


Figure 3.6: Scenario of unsafe analysis of inflating interface's budget.

It is worth noting that the above overhead-aware analysis based on interface budget inflation is only safe under the assumption that the resource demand of a taskset is independent of the resource supply of the interface. However, this assumption is incorrect in the multicore setting: both the resource demand of a taskset in Eq. 3.4 and the resource supply of a resource mdoel in Lemma 3.2 depend on the number of VCPUs of a component, and they are coupled in terms of the number of VCPUs.

In the next section, we present an alternative approach that explicitly considers the effect of cache overhead on the SBF of the interface of *each* VCPU.

3.7.2 Cache-aware effective resource supply of a DMPR model

We first analyze the effective resource supply of a DMPR model μ, i.e., the supply it provides to a domain in the presence of the overhead caused by VCPU-stop events. We then combine the results with the overhead caused by task-preemption events to derive the schedulability and the interface of a domain.

Consider a DMPR interface $\mu = (\Pi, \Theta, m)$ of a domain $\mathcal{D}_i$, and recall that μ provides one partial VCPU $\mathsf{VP}_i = (\Pi, \Theta)$ and m full VCPUs to $\mathcal{D}_i$. Then, in the presence of overhead due to VCPU-stop events, the effective resource supply of μ consists of the effective resource supply of VP_i and the effective resource supply of m full processors. Here, the effective budget (resource) of a VCPU is the budget (resource) that is used solely to execute the tasks running on the VCPU, rather than to handle the cache misses that are caused by VCPU-stop events. We quantify each of them below.

For ease of exposition, we say that a VCPU incurs a CRPMD if the task running on the VCPU incurs the overhead caused by a VCPU-stop event, and we call a time interval $[a, b]$ an *overhead interval* of a VCPU if the effective resource the VCPU provides during $[a, b]$ is zero. (Note that the first overhead interval of VP_i in a period cannot start before VP_i begins its execution.) Finally, we call $[a, b]$ a *black-out interval* of a VCPU if it consists of overhead intervals or intervals during which the VCPU provides no resources.

Effective resource supply of the partial VCPU VP_i of μ. Recall that $N^{\mathsf{stop}}_{\mathsf{VP}_i}$ denotes the maximum number of VCPU-stop events of VP_i during each period Π. The next lemma states a worst-case condition for the effective resource supply of VP_i:

Lemma 3.19. *The worst-case effective resource supply of VP_i in each period occurs when VP_i has $N^{\mathsf{stop}}_{\mathsf{VP}_i}$ VCPU-stop events.*

Proof. Because VP_i has a constant budget of Θ in each period Π, the more cache-related overhead it incurs in a period, the fewer effective resources it can supply to (the actual execution of) the tasks in the domain. Since the overhead that a domain's tasks incur in a period of VP_i is highest when VP_i stops its execution as many times as possible, the worst-case effective resource supply of VP_i in a period occurs when VP_i has the maximum number of VCPU-stop events, which is $N^{\mathsf{stop}}_{\mathsf{VP}_i}$ events. Hence, the lemma. $\square$

Based on this lemma, we can construct the worst-case scenario during which the effective resource supply of VP_i is minimal, and we can derive the effective supply bound function according to this worst-case scenario.

Lemma 3.20. *The effective resource supply that* VP_i *provides during* $\mathcal{I}$ *is minimal when (1)* VP_i *provides its budget as early as possible in the current period and as late as possible in the subsequent periods, (2)* VP_i *has as many VCPU-stop events as possible in each period, and (3) the interval* $\mathcal{I}$ *begins in the current period of* VP_i *and the total length of the black-out intervals that overlap with* $\mathcal{I}$ *is maximal.*

Proof. Suppose VP_i provides Θ resource units in each of its period. Denote by $\mathsf{ScenarioA}$ and $\mathsf{ScenarioB}$ the effective resource supply scenarios described in Claim 1 and the worst-case supply scenario. Further, denote by $\mathsf{sbf}^{\mathsf{stop}}_{\mathsf{VP}_i}(t)$ and $\overline{\mathsf{sbf}^{\mathsf{stop}}_{\mathsf{VP}_i}}(t)$ the effective resource supply of VP_i over any interval of length t in $\mathsf{ScenarioA}$ and $\mathsf{ScenarioB}$, respectively. Then, $\mathsf{sbf}^{\mathsf{stop}}_{\mathsf{VP}_i}(t) \geq \overline{\mathsf{sbf}^{\mathsf{stop}}_{\mathsf{VP}_i}}(t)$. Let the effective resource supply in each period of VP_i in $\mathsf{ScenarioB}$ be $\overline{\Theta^*}$. Because there is at most $N^{\mathsf{stop}}_{\mathsf{VP}_i}$ cache misses during each period of VP_i, $\overline{\Theta^*} \geq \Theta - N^{\mathsf{stop}}_{\mathsf{VP}_i}\Delta^{\mathsf{crpmd}}_{\mathsf{VP_i}} = \Theta^*$, where Θ^* is the effective budget that VP_i provides in each period in $\mathsf{ScenarioA}$. There are two cases:

Case 1) $\Theta \leq N^{\mathsf{stop}}_{\mathsf{VP}_i}\Delta^{\mathsf{crpmd}}_{\mathsf{VP_i}}$**:** We have $\mathsf{sbf}^{\mathsf{stop}}_{\mathsf{VP}_i}(t) = 0$. Because $\overline{\mathsf{sbf}^{\mathsf{stop}}_{\mathsf{VP}_i}}(t) \leq \mathsf{sbf}^{\mathsf{stop}}_{\mathsf{VP}_i}(t)$, VP_i can provide at most Θ^* effective budget in each period under $\mathsf{ScenarioB}$, where $\Theta^* = \Theta - N^{\mathsf{stop}}_{\mathsf{VP}_i}\Delta^{\mathsf{crpmd}}_{\mathsf{VP_i}}$. In other words, $\overline{\Theta^*} \leq \Theta^*$. Since $\Theta^* \leq \overline{\Theta^*}$, we obtain $\overline{\Theta^*} = \Theta^*$.

Case 2) $\Theta > N^{\mathsf{stop}}_{\mathsf{VP}_i}\Delta^{\mathsf{crpmd}}_{\mathsf{VP_i}}$**:** There are five sub-cases, as follows:

(a) $t \leq x + z$: We have $\mathsf{sbf}^{\mathsf{stop}}_{\mathsf{VP}_i}(t) = 0$. Because $\overline{\mathsf{sbf}^{\mathsf{stop}}_{\mathsf{VP}_i}}(t) \leq \mathsf{sbf}^{\mathsf{stop}}_{\mathsf{VP}_i}(t)$, VP_i in $\mathsf{ScenarioB}$ must provide its budget as early as possible in the current period and as late as possible in the next period (as is shown in the interval $[t_3, t_5]$ in $\mathsf{ScenarioA}$), so that it can guarantee that $\overline{\mathsf{sbf}^{\mathsf{stop}}_{\mathsf{VP}_i}}(t) = 0$. Further, because VP_i must provide at most Θ^* time units during each period Π, VP_i always provides effective resource when t is enlarged. Therefore, the maximum length of the black-out interval is $x + z$.

(b) $x+z < t \leq x+z+\Theta^*$: Since VP_i provides $\overline{\Theta^*}$ resource units in each period and the whole second period of $\mathsf{ScenarioB}$ overlaps with the interval $\mathcal{I}$, VP_i must provide Θ^* resource units at the end of the Θ^* time unit interval of the second period. Thus, $\mathsf{ScenarioB}$ is the same as $\mathsf{ScenarioA}$ during the interval $[t_5, t_6]$.

(c) $x + z + \Theta^* < t \leq x + 2z + \Theta^*$: $\mathsf{sbf}^{\mathsf{stop}}_{\mathsf{VP}_i}(t) = \Theta^*$ and VP_i in $\mathsf{ScenarioA}$ provides no effective resource during $[t_6, t_7]$. Therefore, VP_i in $\mathsf{ScenarioB}$ also provides no effective resource during $[t_6, t_7]$ (since $\overline{\mathsf{sbf}^{\mathsf{stop}}_{\mathsf{VP}_i}}(t) \leq \Theta^*$).

(d) $x + 2z + \Theta^* < t \leq x + 2z + 2\Theta^*$: Similar to the sub-case (b) above, VP_i in $\mathsf{ScenarioB}$ must provide Θ^* time units during $[t_7, t_8]$ (because otherwise, it cannot provide Θ^* time units in each period).

(e) By repeating the sub-cases (c) and (d), we can prove that VP_i in $\mathsf{ScenarioB}$ provides no less effective resource than that in $\mathsf{ScenarioA}$.

From the above, we imply that $\mathsf{ScenarioA}$ is the worst-case effective resource supply scenario of VP_i. Hence, the lemma. □

Lemma 3.21. *The effective supply bound function of the partial VCPU* $\mathsf{VP}_i = (\Pi, \Theta)$ *of a resource model* $\mu = (\Pi, \Theta, m)$ *of a component* C *is*

$$\mathsf{sbf}^{\mathsf{stop}}_{\mathsf{VP}_i}(t) = \begin{cases} y\Theta^* + \max\{0, t - x - y\Pi - z\}, & \textit{if } \Theta > N^{\mathsf{stop}}_{\mathsf{VP}_i}\Delta^{\mathsf{crpmd}}_{\mathsf{VP_i}} \\ 0, & \textit{otherwise} \end{cases} \tag{3.15}$$

where $\Delta^{\mathsf{crpmd}}_{\mathsf{VP_i}} = \max_{\tau_i \in C}\{\Delta^{\mathsf{crpmd}}_{\tau_i}\}$, $\Theta^* = \Theta - N^{\mathsf{stop}}_{\mathsf{VP}_i}\Delta^{\mathsf{crpmd}}_{\mathsf{VP_i}}$, $x = \Pi - \Delta^{\mathsf{crpmd}}_{\mathsf{VP_i}} - \Theta^*$, $y = \lfloor \frac{t-x}{\Pi} \rfloor$ *and* $z = \Pi - \Theta^*$.

Proof. Let $\mathcal{I}$ be any interval of length t. We will prove the lemma based on the worst-case resource supply scenario given by Lemma 3.20.

Fig. 3.7 illustrates the worst-case scenario described in Lemma 3.20, where $\mathcal{I}$ begins at time t_3 and the intervals during which VP_i provides effective resources are $[t_2, t_3]$, $[t_5, t_6]$ and $[t_7, t_8]$:

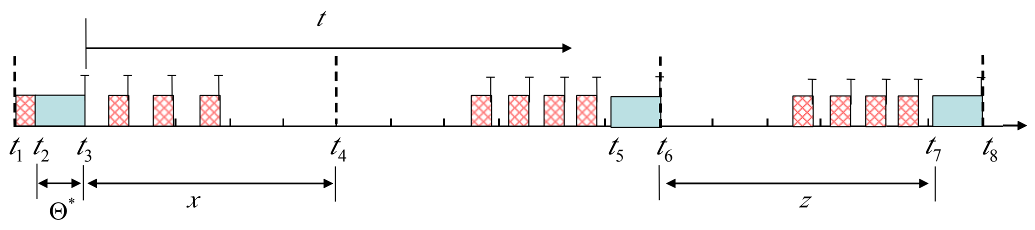


Figure 3.7: Worst-case effective resource supply of $\mathsf{VP}_i = (\Pi, \Theta)$.

In the figure, the first overhead interval of VP_i in a period starts when VP_i first begins its execution in that period. This first overhead interval is caused by the VCPU-completion event of VP_i that occurs in the previous period. Recall from Lemma 3.19 that the maximum number of VCPU-stop events of VP_i in a period Π is $N^{\mathsf{stop}}_{\mathsf{VP}_i}$. Further, according to the gEDF scheduling of component C, any task in C may run the partial VCPU and experience the cache overhead caused by the VCPU-stop event. Therefore, the maximum overhead a task in component C experiences due to a VCPU-stop event of VP_i is $\Delta^{\mathsf{crpmd}}_{\mathsf{VP_i}} = \max_{\tau_i \in C}\{\Delta^{\mathsf{crpmd}}_{\tau_i}\}$. As a result, the effective budget is $\Theta^* \geq \Theta - N^{\mathsf{stop}}_{\mathsf{VP}_i}\Delta^{\mathsf{crpmd}}_{\mathsf{VP_i}}$. Further, we have:

$$\begin{aligned}
t_3 - t_2 &\geq \Theta - (N^{\mathsf{stop}}_{\mathsf{VP}_i} - 1)\Delta^{\mathsf{crpmd}}_{\mathsf{VP_i}} - (t_2 - t_1) = \Theta^* + \Delta^{\mathsf{crpmd}}_{\mathsf{VP_i}} - (t_2 - t_1);\\
x &= t_4 - t_3 = (t_4 - t_1) - (t_3 - t_2) - (t_2 - t_1) \leq \Pi - \Delta^{\mathsf{crpmd}}_{\mathsf{VP_i}} - \Theta^*;\\
z &= t_7 - t_6 = (t_8 - t_6) - (t_8 - t_7) \leq \Pi - \Theta^*.
\end{aligned}$$

Based on this information, we can derive the minimum effective resource supply during the interval $\mathcal{I}$ as follows: if $\Theta \leq N^{\mathsf{stop}}_{\mathsf{VP}_i}\Delta^{\mathsf{crpmd}}_{\mathsf{VP_i}}$, then $\Theta^* = 0$ and $\mathsf{sbf}^{\mathsf{stop}}_{\mathsf{VP}_i} = 0$; otherwise, $\mathsf{sbf}^{\mathsf{stop}}_{\mathsf{VP}_i}(t) = y\Theta^* + \max\{0, t - x - y\Pi - z\}$. In addition, $\mathsf{sbf}^{\mathsf{stop}}_{\mathsf{VP}_i}(t)$ is minimal when $\Theta^* = \Theta - N^{\mathsf{stop}}_{\mathsf{VP}_i}\Delta^{\mathsf{crpmd}}_{\mathsf{VP_i}}$ and $x = \Pi - \Delta^{\mathsf{crpmd}}_{\mathsf{VP_i}} - \Theta^*$. Therefore, Equation 3.15 gives the minimum effective resource supply of the worst-case effective resource supply scenario described in Lemma 3.20. This proves the lemma. □

Effective resource supply of all m full VCPUs of μ. Similar to the partial-VCPU case, we can also establish a worst-case condition for the total effective resource supply of the full VCPUs:

Lemma 3.22. *The m full VCPUs provide the worst-case total effective resource supply when they incur $N^{\mathsf{stop}}_{\mathsf{VP}_i}$ CRPMDs in total during each period Π of the partial VP_i of μ.*

Proof. Because the total resource supply of m full VCPUs in any interval of length t is always mt, these VCPUs together provide the least effective resource supply when they incur the maximum number of CRPMDs. Recall from Section 3.5 that, when a VCPU-stop event of the partial VCPU VP_i of a domain $\mathcal{D}_i$ occurs, it causes one CRPMD in a full VCPU of the same domain. Hence, the total number of CRMPDs that these full VCPUs incur together is the number of VCPU-stop events of the partial VCPU VP_i of the same domain. The lemma then follows from a combination with Lemma 3.19. □

The next lemma gives the worst-case supply scenarios of m full VCPUs. Fig. 3.8 illustrates one of the conditions under this worst-case scenario.

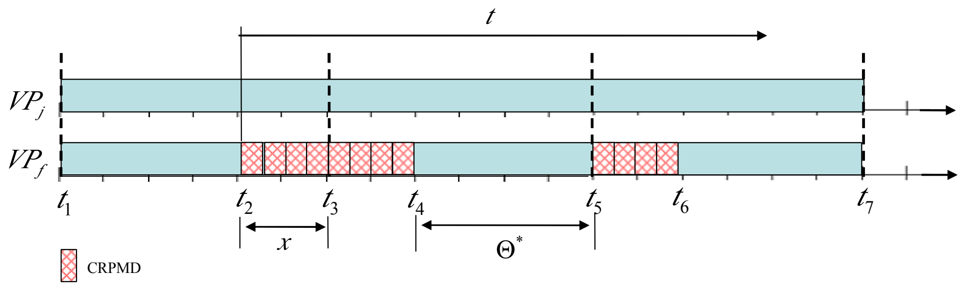


Figure 3.8: Worst-case resource supply of m full VCPUs of μ.

Lemma 3.23. *The worst-case effective resource supply of m full VCPUs of μ in any interval $\mathcal{I}$ of length t occurs when (1) all the $N^{\mathsf{stop}}_{\mathsf{VP}_i}$ CRPMDs are experienced by one full VCPU VP_f in each period Π of VP_i, (2) VP_f incurs the overhead as late as possible in the first period and as early as possible in the rest of periods of VP_i, (3) the maximum overhead cost of each CRPMD overhead is $\Delta^{\mathsf{crpmd}}_{\mathsf{VP_i}}$, and (4) the interval $\mathcal{I}$ begins when the first CRPMD occurs in the first period.*

Proof. We denote the effective resource supply scenario given by Lemma 3.23 (see Fig. 3.8) by $\mathsf{ScenarioA}$, and let $\mathsf{ScenarioB}$ be a worst-case effective resource supply scenario of the m full VCPUs. Let $x = N^{\mathsf{stop}}_{\mathsf{VP}_i}\Delta^{\mathsf{crpmd}}_{\mathsf{VP_i}}$. We will prove that the m full VCPUs provides no less effective resource in $\mathsf{ScenarioB}$ than in $\mathsf{ScenarioA}$ with the following arguments:

1. While a full VCPU VP_f is experiencing a CRPMD, the resource provided by any other full VCPU VP_j is unavailable to the task currently running on VP_f (since this task cannot execute on more than one VCPUs at any given time). Since it is unknown which exact task in the domain is running on VP_f, it is unknown whether VP_j is available to a given task. Hence, we consider VP_j as unavailable to every task while VP_f is experiencing the overhead, so as to guarantee the safety of the schedulability analysis. Recall from Lemma 3.22 that, all m full VCPUs incur $N^{\mathsf{stop}}_{\mathsf{VP}_i}$ CRPMDs in each period. The unavailable intervals of each period Π is maximized when all these $N^{\mathsf{stop}}_{\mathsf{VP}_i}$ CRPMDs are incurred by one full VCPU VP_f in each period Π of VP_i. Hence, $\mathsf{ScenarioB}$ must obey Condition (1).

2. The maximum total length of the unavailable intervals of m full VCPUs in each period is $x = N^{\mathsf{stop}}_{\mathsf{VP}_i}\Delta^{\mathsf{crpmd}}_{\mathsf{VP_i}}$. The maximum black-out interval happens when the unavailable intervals in two periods are consecutive and the maximum cost of each CRPMD is $\Delta^{\mathsf{crpmd}}_{\mathsf{VP_i}}$. Therefore, the full VCPU VP_f should incur the overhead as late as possible in the first period and as early as possible in the

second period of VP_i in order for the black-out interval to be maximized. In addition, the interval $\mathcal{I}$ should begin when the first CRPMD occurs in the first period. Hence, ScenarioB should obey the conditions (3) and (4), and the m full VCPUs provide no less effective resource in ScenarioB than in ScenarioA when $t \leq 2x$.

3. When $x + k\Pi < t < 2x + k\Pi$ ($k \in N$), because m full VCPUs must provide $m(\Pi - x)$ effective resource units in each period and the interval t has k periods, the m full VCPUs in ScenarioB should provide at least $km(\Pi - x)$ effective resource units during a time interval of length t. Because $t > x + k\Pi$, the m full VCPUs in ScenarioB have already provided $km(\Pi - x)$ effective resource units during the interval of length $x + k\Pi$. Therefore, they must provide no effective resource in the remaining time interval of length $t - (x + k\Pi)$ (otherwise, the m full VCPUs would provide more effective resource in ScenarioB than in ScenarioA.) Hence, VP_f should incur the overhead as early as possible in all periods (except for the first period) of VP_i. Hence, by combining the the arguments (2) and (3), we imply that ScenarioB must obey Condition (2) and the m full VCPUs provide no less effective resource in ScenarioB than in ScenarioA when $x + k\Pi < t < 2x + k\Pi$.

4. When $2x + k\Pi < t < x + (k + 1)\Pi$ ($k \in N$), the m full VCPUs in ScenarioB provides no effective resource during $[x+k\Pi, 2x+k\Pi]$ according to the argument (3). In addition, the m full VCPUs in ScenarioB must provide $m(\Pi-x)$ effective resource units during $[x + k\Pi, x + (k + 1)\Pi]$, i.e., the $(k + 1)^{th}$ period of VP_i, in order to guarantee $m(\Pi - x)$ effective resource units during the $(k + 1)^{th}$ period of VP_i. Therefore, the m VCPUs in ScenarioB always provides the same effective resource during $[2x + k\Pi, x + (k + 1)\Pi]$ as in ScenarioA. Hence, they provide no less effective resource in ScenarioB than in ScenarioA when $2x + k\Pi < t < x + (k + 1)\Pi$.

Because the m full VCPUs provide no less effective resource in $\mathsf{ScenarioB}$ than in $\mathsf{ScenarioA}$, and $\mathsf{ScenarioB}$ is a worst-case effective resource supply scenario, we imply that $\mathsf{ScenarioA}$ is also a worst-case effective resource supply scenario of the m full VCPUs. Hence, the lemma. □

The next lemma gives the effective SBF of the m full VCPUs of μ based on the worst-case scenario described in Lemma 3.23.

Lemma 3.24. *The effective resource supply bound function of the m full VCPUs of μ is given by:*

$$\mathsf{sbf}^{\mathsf{stop}}_{\mathsf{VPs}}(t) = \begin{cases} m\big(y\Theta' + \max\{0, t - y\Pi - 2x\}\big) & \textit{if } \Theta \neq 0 \\ mt & \textit{if } \Theta = 0 \end{cases} \tag{3.16}$$

where $x = N^{\mathsf{stop}}_{\mathsf{VP}_i}\Delta^{\mathsf{crpmd}}_{\mathsf{VP_i}}$, $y = \lfloor \frac{t-x}{\Pi} \rfloor$ and $\Theta' = \Pi - x$.

Proof. The effective resource supply bound function $\mathsf{sbf}^{\mathsf{stop}}_{\mathsf{VPs}}(t)$ of the resource supply scenario given by Lemma 3.23 is given by: When $t < 2x$, $\mathsf{sbf}^{\mathsf{stop}}_{\mathsf{VPs}}(t) = 0$; When $x + k\Pi < t < 2x + k\Pi$, $\mathsf{sbf}^{\mathsf{stop}}_{\mathsf{VPs}}(t) = km(\Pi - x)$; When $2x + k\Pi < t < x + (k+1)\Pi$, $\mathsf{sbf}^{\mathsf{stop}}_{\mathsf{VPs}}(t) = km(\Pi - x) + m(t - 2x - k\Pi)$. Equation 3.16 is derived by rearranging the equations of $\mathsf{sbf}^{\mathsf{stop}}_{\mathsf{VPs}}(t)$. Since the resource supply scenario given by Lemma 3.23 is a worst-case scenario, $\mathsf{sbf}^{\mathsf{stop}}_{\mathsf{VPs}}(t)$ is the effective resource supply bound function of the m full VCPUs of μ. □

Effective resource supply of a DMPR model The next lemma gives the effective resource supply that a DMPR interface $\mu = (\Pi, \Theta, m)$ provides to a domain $\mathcal{D}_i$ after having accounted for the overhead due to VCPU-stop events. The lemma is a direct consequence of Lemmas 3.21 and 3.24.

Lemma 3.25. *The effective resource supply of a DMPR interface $\mu = \langle \Pi, \Theta, m \rangle$ of a domain $\mathcal{D}_i$ after having accounted for the overhead due to VCPU-stop events is* *given by:*

$$\mathsf{sbf}_{\mu}^{\mathsf{stop}}(t) = \mathsf{sbf}_{\mathsf{VP}_i}^{\mathsf{stop}}(t) + \mathsf{sbf}_{\mathsf{VPs}}^{\mathsf{stop}}(t),\ \forall\, t \geq 0. \tag{3.17}$$

Here, $\mathsf{sbf}_{\mathsf{VP}_i}^{\mathsf{stop}}(t)$ *is the effective resource supply of the partial VCPU* $\mathsf{VP}_i = (\Pi, \Theta)$*, which is given by Eq.* (3.15)*, and* $\mathsf{sbf}_{\mathsf{VPs}}^{\mathsf{stop}}(t)$ *is the effective resource supply of the* m *full VCPUs of* μ*, which is given by Eq.* (3.16)*.*

Proof. Since the resource supply of a DMPR interface is the total effective resource supply of its partial VCPU and full VCPUs, the lemma directly follows from the definition of $\mathsf{sbf}_{\mathsf{VP}_i}^{\mathsf{stop}}(t)$ and $\mathsf{sbf}_{\mathsf{VPs}}^{\mathsf{stop}}(t)$. % □

Note that, when no partial VCPU exists for interface $\mu = \langle \Pi, 0, m \rangle$, the effective resource supply of μ is equal to the resource supply of μ, i.e., $\mathsf{sbf}_{\mu}^{stop}(t) = mt$.

3.7.3 DMPR interface computation under MODEL-CENTRIC method

Based on the effective supply function, we can develop the component schedulability test as follows.

Theorem 3.26. *Consider a domain* $\mathcal{D}_i$ *with a taskset* $\tau = \{\tau_1, ...\tau_n\}$*, where* $\tau_k = (p_k, e_k, d_k)$*. Let* $\tau'' = \{\tau''_1, ...\tau''_n\}$*, where, for all* $1 \leq k \leq n$*,* $\tau''_k = (p_k, e''_k, d_k)$ *and* $e''_k = e_k + max_{\tau_i \in LP(\tau_k)} \Delta_{\tau_i}^{\mathsf{crpmd}}$ *(and recall that* $\mathsf{LP}(\tau_k) = \{\tau_i | d_i > d_k\}$*) . Then,* $\mathcal{D}_i$ *is schedulable under gEDF by a DMPR model* μ *in the presence of cache-related overhead, if the inflated taskset* τ'' *is schedulable under gEDF by the effective resource supply* $\mathsf{sbf}_{\mu}^{\mathsf{stop}}(t)$ *in the absence of overhead.*

Proof. Since τ'' includes the overhead that τ incurs due to task-preemption events, if $\mathsf{sbf}_{\mu}^{\mathsf{stop}}(t)$ is sufficient to schedule τ'' assuming negligible overhead, then it is also sufficient to schedule τ in the presence of task-preemption events. As $\mathsf{sbf}_{\mu}^{\mathsf{stop}}(t)$ gives the effective supply that μ provides to τ after having accounted for the overhead due to VCPU-stop events, μ provides sufficient resources to schedule τ in the presence of the overhead from all types of events. This proves the theorem. $\Box$

Based on the above results, we can generate a cache-aware minimum-bandwidth DMPR interface for a domain in the same manner as in the overhead-free case, except that we use the effective resource supply and the inflated taskset in the schedulability test. Similarly, the system's interface can be computed from the interfaces of the domains in the exact same way as the overhead-free interface computation.

3.8 Hybrid cache-aware DMPR interface

Recall from Section 3.6 that the TASK-CENTRIC-UB method always dominates the BASELINE method. However, neither of these analysis methods dominates the MODEL-CENTRIC method, and vice versa. We demonstrate this using two example systems, where the TASK-CENTRIC-UB method gives a smaller interface bandwidth in the first system but a larger interface bandwidth in the second system compared to the interface bandwidth given by the MODEL-CENTRIC method.

Example 3.8. *Let Sys_1 be a system consisting of two domains C_1 and C_2 that are scheduled under the hybrid EDF scheduling strategy (c.f. Section 3.1) and that have workloads $\tau_{C_1} = \{\tau_1^1 = ... = \tau_1^4 = (200, 100, 200)\}$ and $\tau_{C_2} = \{\tau_2^1 = \tau_2^2 = (200, 100, 200)\}$, respectively. By applying the analysis in Sections 3.6.2 and 3.7, the interfaces of the system under* TASK-CENTRIC-UB *and under* MODEL-CENTRIC *are computed to be $\mu_{\mathsf{Sys}_1} = \langle 20, 17, 5\rangle$ and $\mu'_{\mathsf{Sys}_1} = \langle 20, 19, 5\rangle$, respectively. Thus, the system's interface under* TASK-CENTRIC-UB *has a smaller bandwidth than that of the interface computed under* MODEL-CENTRIC.

Example 3.9. *Let Sys_2 be a system consisting of two domains C_1 and C_2 that are scheduled under the hybrid EDF scheduling strategy and that have workloads $\tau_{C_1} = \{\tau_1^1, ..., \tau_1^5 = (100, 5, 100)\}$ and $\tau_{C_2} = \{\tau_2^1, ..., \tau_2^5 = (100, 5, 100)\}$, respectively.*

The interfaces of this system under TASK-CENTRIC-UB *and under* MODEL-CENTRIC *are given by* $\mu_{\mathsf{Sys}_2} = \langle 20, 0, 4\rangle$ *and* $\mu'_{\mathsf{Sys}_2} = \langle 20, 14, 3\rangle$*, respectively. Thus, the system's interface under* TASK-CENTRIC-UB *has a larger bandwidth than that of the interface computed under* MODEL-CENTRIC*.*

One can also show that neither MODEL-CENTRIC nor BASELINE dominates one another. For instance, consider the system Sys_1 in Example 3.8. The interface of the whole system under the BASELINE method is $\mu''_{\mathsf{Sys}_1} = \langle 20, 17, 5\rangle$, which has a smaller bandwidth than the interface μ'_{Sys_1} computed using the MODEL-CENTRIC method. Further, since the TASK-CENTRIC-UB method dominates the BASELINE method but not the MODEL-CENTRIC method, the BASELINE method also does not dominate the MODEL-CENTRIC method.

From the above observations, we can derive the minimum interface of a component from the ones computed using the TASK-CENTRIC-UB and MODEL-CENTRIC methods (since TASK-CENTRIC-UB method always dominates BASELINE), as stated by Theorem 3.27. The theorem is trivially true, since both interfaces computed using the TASK-CENTRIC-UB and MODEL-CENTRIC methods are safe. We refer to this analysis as the HYBRID method.

Theorem 3.27 (Hybrid cache-aware interface). *The minimum cache-aware DMPR interface of a domain* $\mathcal{D}_i$ *(a system* $\mathcal{S}$*) is the interface that has a smaller resource bandwidth between* μ_{task} *and* μ_{model}*, where* μ_{task} *and* μ_{model} *are the minimum-bandwidth DMPR interfaces of* $\mathcal{D}_i$ *(*$\mathcal{S}$*) computed using the* TASK-CENTRIC-UB *and the* MODEL-CENTRIC *methods, respectively.*

Discussion. We observe that the schedulability analysis under gEDF in the absence of overhead (Theorem 3.3) is only a sufficient test, and that its pessimism degree varies significantly with the characteristics of the taskset. For instance, under the same multiprocessor resource, one taskset with a larger total utilization may be schedulable while another with a smaller total utilization may not be schedulable.

As a result, it is possible that the overhead-aware interface of a domain (system) may require less resource bandwidth than the overhead-free interface of the same domain (system).

3.9 Evaluation

To evaluate the benefits of our proposed interface model and cache-aware compositional analysis, we performed simulations using randomly generated workloads. We had five main objectives for our evaluation: (1) determine how much resource bandwidth the interfaces computed using the improved SBF (Section 3.2.2) can save compared to the interfaces computed using the original SBF proposed in [31]; (2) determine how much resource bandwidth the DMPR model can save compared to the MPR model; (3) evaluate the relative performance of the HYBRID method and the BASELINE method; (4) study the impact of task parameters (e.g., the range of taskset utilization, the distribution of task's utilization, the period range of tasks) on the interfaces under the HYBRID and BASELINE methods; and (5) evaluate the performance of the HYBRID analysis when using a cache overhead value per task and when using the maximum cache overhead value for the entire system.

3.9.1 Experimental setup

Key factors. We focus on the following five key factors that can affect the performance of a cache-aware compositional analysis:[16]:

- *Utilization of a task set.* Tasks with larger utilizations tend to have a larger number of tasks; thus, each task tends to experience more cache overhead during its lifetime because there are more other tasks that can preempt it.

[16]We assume other factors are same when we discuss one factor's impact on the cache-aware analysis

- *Distribution of task utilizations.* High-utilization tasks are more sensitive to cache overhead and can more easily become unschedulable because of this overhead than tasks with small utilization.

- *Periods of the tasks.* If two tasks have the same utilization and experience the same cache overhead, the task with the smaller period has a higher probability of missing its deadline because of the overhead than the task with the larger period because the former has a smaller relative deadline. Therefore tasks with smaller period are more sensitive to cache overhead.

- *Number of tasks in a task set.* In the BASELINE approach and the task-centric approach from Section 3.6, when a VCPU-stop event happens, each task's worst-case execution time is inflated by the cache overhead caused by this event, even though at most two tasks actually experience the cache overhead that the event has caused. Hence, these two approaches will become more and more pessimistic as the number of tasks increases.

- *Cost of cache overhead per event.* If the cost of cache overhead increases, tasks will experience longer delays when task-preemption or VCPU-stop events occur.

Workload. In order to evaluate the impact of the above five factors on the performance of overhead-free and overhead-aware compositional analysis, we generated a number of synthetic real-time workloads with randomly generated periodic task sets that span a range of different parameters for each of these factors. Below, we explain how the parameters were chosen.

We picked the *task set utilizations* from the interval $[0, 24]$, with increments of 0.2, to be consistent with the ranges used in [22] and [24]. However, we observed that a smaller interval is sufficient to demonstrate the relative performance of overhead-free and overhead-aware compositional analysis; hence, we used the range $[0, 5]$, again with increments of 0.2, when evaluating the impact of the other factors on overhead-aware compositional analysis.

The *tasks' utilizations* were drawn from one of four distributions: one uniform distribution over the range $[0.001, 0.1]$ and three bimodal distributions; in the latter, the utilization was distributed uniformly over either $[0.1, 0.5)$ or $[0.5, 0.9]$, with respective probabilities of 8/9 and 1/9 (light), 6/9 and 3/9 (medium), and 4/9 and 5/9 (heavy). These probabilities are consistent with the ones used in [17] and [24]. The *periods* of the tasks were drawn from a uniform distribution over one of the following three ranges: $(350ms, 850ms)$, $(550ms, 650ms)$, and $(100ms, 1100ms)$; all periods were integer. These distributions are identical to those used in [41]. The *number of tasks* in a task set ranged from $[0, 300]$ with increments of 20.

The *cost of cache overhead per event* was chosen based on the cache overhead ratio, which we define as the cache overhead of a task τ_i divided by the worst-case execution time of τ_i. We picked the cache overhead ratio from the range $[0, 0.1]$ with increments of 0.01. This range was chosen based on measurements of the L2 cache miss overhead of tasks on our experimental platform; we found that the cost of missing the L2 private cache but hitting the L3 shared cache was $0.02ms$ when the working set size was $256KB$ (the L2 private cache size). Because the L3 cache hit latency is very small (less than 100 cycles), the cache overhead per task-preemption or VCPU-stop event is only $0.02ms$. Therefore, the cache overhead ratio was less than 0.02 for any task we measured that had a worst-case execution time of more than $2ms$.

Overhead measurements. For our measurements, we used a Dell Precision T3610 six-core workstation with the RT-Xen 2.0 platform [69]; each domain was running LITMUS^{RT} 2012.3 [27] [24]. The scheduler was gEDF in the domains and semi-partitioned EDF in the VMM, as described in Section 3.1. We allocated a full-capacity VCPU to one domain and pinned this VCPU to a physical core of its own; this was done to avoid interference from domain 0 (the administrative domain in RT-

Xen), which was pinned to a different core. We measured the cache overhead of the cache-intensive program ρ as follows. First we warmed up the cache by accessing all the cache content of the program; then we used the time stamp counter to measure the time l_{hit} it takes to access the same content again. Because the cache was warm, l_{hit} is the cache hit latency of this program. Next, we allocated an array of the same size as the private L2 cache and loaded this into the same core's L2 cache in order to pollute the cache content of ρ. Finally, we again accessed all the cache content of ρ and recorded the cache miss latency l_{miss}. The cache overhead of the program ρ per task-preemption or VCPU-stop event is then $l_{miss} - l_{hit}$.

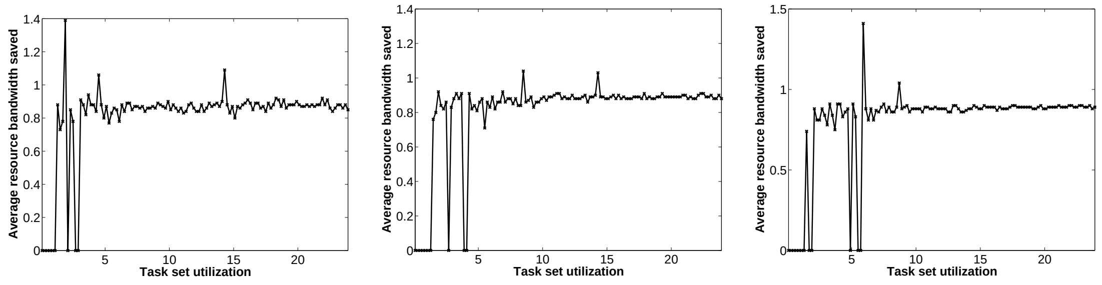
3.9.2 Overhead-free analysis

We begin with an empirical comparison of the overhead-free analyses. For this purpose, we set up four domains with harmonic periods, and we randomly generated tasks and uniformly distributed them across the four domains. To be consistent with [54], we generated 25 task sets per task set utilization or task set size.

MPR with improved SBF vs. MPR with original SBF. To estimate the impact of the improved SBF, we generated 625 tasksets with taskset utilizations ranging from 0.1 to 24, with increments of 0.2. The task utilizations were drawn from the bimodal-light distribution as described earlier; the tasks' periods were uniformly distributed across $[350ms, 850ms]$. For each taskset we generated, we distributed the tasks into *one domain*, and we then computed the overhead-free interface of the domain using MPR with the improved SBF, as well as using the original MPR. Fig. 3.9(a) shows the average bandwidth savings due to the improved SBF. We observe that, across all taskset utilizations, MPR with the improved SBF always requires either the same or less resource bandwidth than MPR with the original SBF. We also observe that MPR with the improved SBF saves over 0.8 cores when the taskset utilization is larger than 5. Fig. 3.9(b) and 3.9(c) show the average resource bandwidth savings with the other two bi-modal distributions; we observe that, in all three cases, MPR with the improved SBF consistently outperformed MPR with the original SBF.

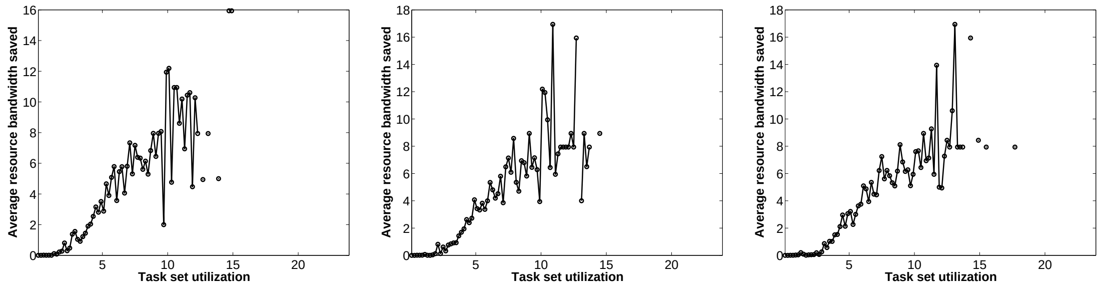
DMPR vs. MPR with the original SBF. To compare DMPR to MPR with the original SBF on the whole system, we distributed the tasks in each taskset over **four domains** and we then computed the overhead-free interface of the whole system using both DMPR and MPR with the original SBF. Fig. 3.10(a) shows the average bandwidth savings of DMPR for different taskset utilizations. Our results show that DMPR consistently saves bandwidth relative to MPR with the original SBF for up to 16 cores. There are very few data points beyond this point because we can only compute the average bandwidth savings when both analyses return valid interfaces for the same taskset; however, for taskset utilizations above 16, MPR generally fails to compute a valid interface for the system.

As shown in Fig. 3.11(a), the fraction of tasksets with valid interfaces under MPR with the original SBF decreases with increasing taskset utilization. This is because the original SBF of MPR is pessimistic and cannot provide $m't$ time units with interface $\Gamma = \langle, m', m' \rangle$. Once the interfaces of the leaf components (i.e., domains) have been computed, these interfaces are transferred to VCPUs as the workload of the top component. When some of those VCPUs have utilization 1, the resource demand increases faster than the resource supply of MPR with the original SBF; hence, MPR cannot find a valid interface. DMPR does not have this problem because it can always supply $m't$ time units with bandwidth m'; hence, the fraction of tasksets with valid interfaces is always 1. As Fig. 3.11(b) and Fig. 3.11(c) show, the results for the other two bimodal distributions are similar: DMPR is consistently able to compute interfaces for all tasksets, whereas MPR with the original SBF finds fewer and fewer interfaces as the taskset utilization increases.



(a) Bimodal-light. (b) Bimodal-medium. (c) Bimodal-heavy.

Figure 3.9: Average resource bandwidth saved: MPR with improved SBF vs. MPR with original SBF.



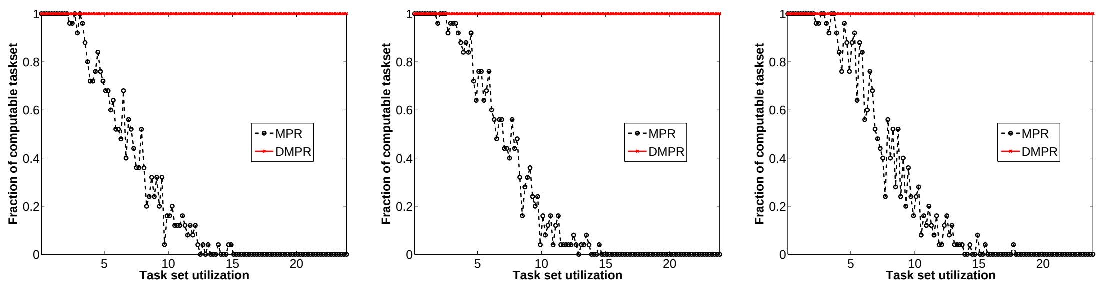
(a) Bimodal-light. (b) Bimodal-medium. (c) Bimodal-heavy.

Figure 3.10: Average resource bandwidth saved: DMPR vs. MPR with original SBF.

3.9.3 Comparison of HYBRID cache-aware analysis vs. BASELINE cache-aware analysis

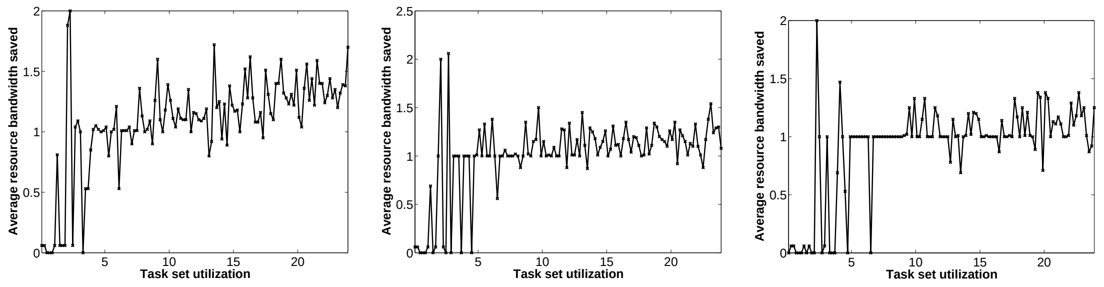
Next, we compared the performance of the two overhead-aware analysis approaches. For this we used the same tasksets and system configuration as for the previous experiment, but we additionally computed DMPR interfaces for each taskset using the respective approach.

Impact of taskset utilization. Fig. 3.13(a) shows the average resource bandwidth savings of the HYBRID approach compared to the BASELINE approach for each taskset utilization. We observe that a) HYBRID reduced the resource bandwidth in all cases, and that b) more and more cores are being saved as the taskset utilization increases. Note that, as the taskset utilization increases, the interface bandwidth can sometimes



(a) Bimodal-light. (b) Bimodal-medium. (c) Bimodal-heavy.

Figure 3.11: Fraction of taskset with valid interfaces: DMPR vs. MPR with original SBF.

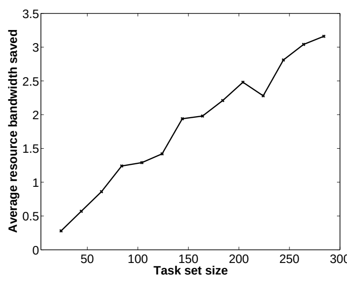


(a) Bimodal-light. (b) Bimodal-medium. (c) Bimodal-heavy.

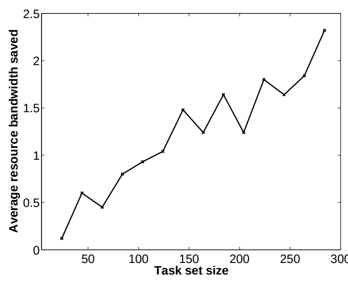
Figure 3.12: Average resource bandwidth saved: HYBRID vs. BASELINE.

decrease. One reason for this is that the underlying gEDF schedulability test is only sufficient, and is not strictly dependent on the taskset utilization; in other words, it is possible that a taskset with a high utilization is schedulable but another with a lower utilization is not. We also observe that, as discussed earlier, the relative performance of the HYBRID and BASELINE analyses is easy to see even for small taskset utilizations; this is why we only compare the two overhead-aware analysis for taskset utilizations $[0, 5]$ instead of the larger $[0, 24]$ range.

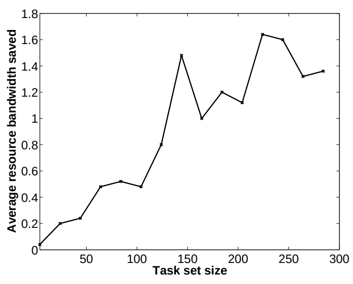
Impact of task utilization. Fig. 3.13(a)-Fig. 3.13(c) show the average resource bandwidth savings for different taskset utilizations and each of the three bimodal distributions. We observe that, in all three cases, the HYBRID approach consistently outperformed the BASELINE approach. Further, as the taskset utilization increases, the savings also increase and remain steady at approximately one core once the



(a) Bimodal-light.

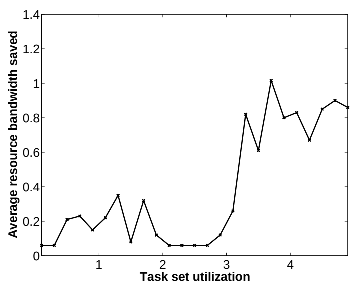


(b) Bimodal-medium.

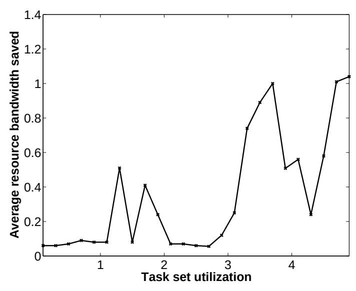


(c) Bimodal-heavy.

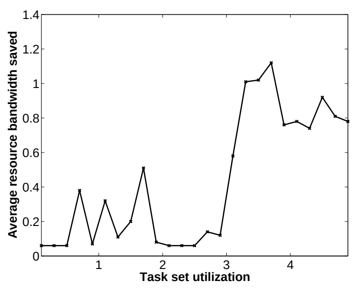
Figure 3.13: Average resource bandwidth saved: HYBRID vs. BASELINE.



(a) Task period: [100, 1100]ms.



(b) Task period: [350, 850]ms.



(c) Task period: [550, 650]ms.

Figure 3.14: Average resource bandwidth saved under different ranges of tasks' periods

taskset utilization has reached 10.

Impact of taskset size. We investigated the impact of the number of tasks (i.e., the taskset size) on the average bandwidths saving of the HYBRID approach compared to the BASELINE approach. For this experiment, we generated a set of tasksets with sizes between 4 to 300, with increments of 20, and with 25 tasksets per size. As before, we tried each of the three bimodal distributions we discussed in Section 3.9.1.

Fig. 3.13(a)-Fig. 3.13(c) show the average resource bandwidth savings for different taskset sizes with each of the three bi-modal distributions. We observe that a) the HYBRID approach consistently outperforms the BASELINE approach, and b) the savings increase with the number of tasks. This is expected because the BASELINE technique inflates the WCET of every task with all the cache-related overhead each task experiences; hence, its total cache overhead increases with the size of the taskset.

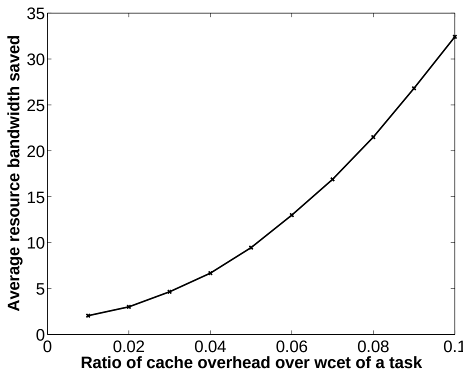


Figure 3.15: Average bandwidth saving under different ratios of cache overhead to task WCET.

Impact of task period distribution. We further investigated the impact of the distribution of tasks' periods on the average bandwidth savings of the HYBRID approach compared to the BASELINE approach. For this experiment, we generated a number of tasksets with taskset utilizations in the range $[0, 5]$ with increments of 0.2, and, as usual, 25 tasksets per taskset utilization. The individual tasks' utilizations were drawn from the bi-modal light distribution. For the tasks' periods, we tried each of the three distributions that were discussed in Section 3.9.1. Fig. 3.14(a)-Fig. 3.14(c) show the average resource bandwidth saving for three different distribution of tasks' periods; in all three cases, the HYBRID approach consistently outperforms the BASELINE approach.

Impact of cost of cache overhead. We first generated 25 tasksets with taskset utilization 4.9 and uniformly distributed the tasks of each taskset over four domains with harmonic periods. The tasks' utilizations were uniformly distributed in $[0.001, 0.1]$, and their periods were uniformly distributed in $[350ms, 850ms]$. We then modified the cache overhead of tasks of the 25 tasksets and generated a set of tasksets with cache-related overhead ratio $[0, 0.1]$ with increments of 0.01 based on the 25 tasksets. Recall from Section 3.9.1 that we define the cache-related overhead ratio of a task τ_i to be the cost of one cache-related overhead of τ_i divided by the worst-case execution time of τ_i.

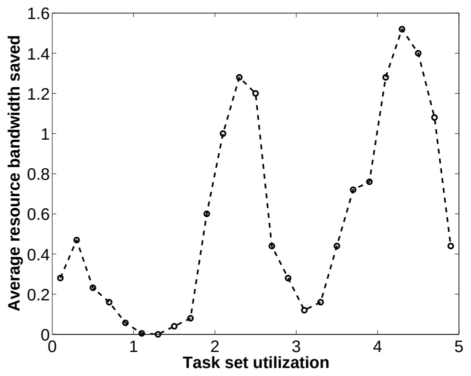


Figure 3.16: Average bandwidth saving of HYBRID with cache overhead per task over HYBRID with maximum cache overhead of system (Ratio of overhead over wcet is uniformly in [0,0.1])

Fig. 3.15 shows the average resource bandwidth savings of the HYBRID approach over the BASELINE approach for each cache overhead ratio. We observe that the HYBRID approaches saves more resources as the cache-related overhead ratio increases. This is expected because tasks' utilizations are uniformly distributed over $[0.001, 0.1]$ and a taskset has more tasks than the number of VCPUs. Since the BASELINE approach inflates the WCET of every task with all the cache-related overheads any task can experience, its total cache overhead increases as the cost of one cache-related overhead increases.

Impact of per-task cache overheads. When different tasks can have different costs for cache-related overheads, it is pessimistic to simply use the largest cache overhead in the system, as we did in [74]. To evaluate the impact of considering cache overheads *per task*, we generated tasks with different cache-related overhead ratios, drawn from an uniform distribution over $[0, 0.1]$. We then calculated the system's interface with the HYBRID analysis using the following two approaches: (1) Using a per-task cost of cache overheads to compute the HYBRID analysis, as we did in this work; and (2) Using the upper bound for the cache overhead in the system as the cost for each task, as we did in [74].

Fig. 3.16 shows the average resource bandwidth savings of the HYBRID approach with per-task cache overheads relative to the more pessimistic approach. We observe that the HYBRID approach with per-task cache overheads consistently outperformed the pessimistic approach; however, the saving does *not* increase as the taskset utilization increases. This is because the TASK-CENTRIC-UB approach only considers the cache overhead caused by task-preemption events, and each task's WCET is only inflated with one cache overhead. Therefore, the pessimistic HYBRID analysis with system's maximum cache overhead may have the same upper-bounded number of full VCPUs as the HYBRID analysis with cache overhead per task. When both analyses use the upper-bounded number of full VCPUs as the components' interface, the HYBRID analysis with per-task cache overheads will have the same interface bandwidth as the pessimistic analysis and thus saves no resources; however, (2) if both HYBRID analyses choose the interfaces computed by the MODEL-CENTRIC analysis, the HYBRID analysis with per-task cache overheads will save resources relative to the pessimistic approach because every time one cache-related overhead happens, the pessimistic approach will have more cache overhead.

3.9.4 Performance in theory vs. in practice

We also validated the correctness of the cache-aware interfaces (and the invalidity of the overhead-free interfaces) in practice. For this experiment, we first computed the domains' interfaces, and we then ran the generated tasks on our RT-Xen experimental platform. The periods and budgets of the domains in RT-Xen were chosen to be those of the respective computed interfaces. We then computed the schedulability and deadline miss ratios of the tasks, based on the theoretical schedulability test and the measurements on the RT-Xen platform. Table 3.1 shows the schedulability and deadline miss ratios of these methods.[17]

We observe that the overhead-free MPR and DMPR interfaces significantly underestimate the tasks' resource requirements: even though the tasks were claimed

[17]We note that the interfaces given by the HYBRID method and the BASELINE method are the same as the interfaces given by the cache-aware hybrid analysis method and task-centric analysis method proposed in the conference version [74], respectively.

	Schedulable		Deadline miss ratio	
	Theory	RT-Xen	Theory	RT-Xen
Overhead-free MPR	Yes	No	N/A	78%
Overhead-free DMPR	Yes	No	N/A	78%
HYBRID	No	No	N/A	0.07%
BASELINE	No	No	N/A	7%

Table 3.1: Performance in theory vs. in practice.

to be schedulable by the computed interfaces, 78% of the jobs missed their deadlines. The experimental results also confirm that our cache-aware analysis correctly estimated the resource requirements of the system in practice: the theory predicted that the tasks would not be schedulable, and this was confirmed in practice by the nonzero deadline miss ratio, which was 0.07% for the HYBRID approach and 7% for the task-centric approach. We also observe that the HYBRID approach had fewer deadline misses than, and thus outperformed, the task-centric approach.

3.10 Conclusion

In this chapter, we have presented a private cache-aware compositional analysis technique for real-time virtualization multicore systems. Our technique accounts for the cache overhead in the component interfaces, and thus enables a safe application of the analysis theories in practice. We have developed three different approaches, BASELINE, TASK-CENTRIC-UB and MODEL-CENTRIC, for analyzing the cache-related overhead and for testing the schedulability of components in the presence of cache overhead. We have also introduced an improved supply bound function for the MPR model and a deterministic extension of the MPR model, which improve the interface resource efficiency, as well as accompanying overhead-aware interface computation methods. Our evaluation on synthetic workloads shows that our improved SBF and the DMPR interface model can help reduce resource bandwidth by a significant factor compared to the MPR model with the existing SBF, and that a hybrid of TASK-

CENTRIC-UB and MODEL-CENTRIC achieves significant resource savings compared to the BASELINE method (which is based solely on WCET inflation).

Chapter 4

Shared cache-aware scheduling and analysis for operating systems

We have solved the analysis challenge of the private cache overhead; we now arrive at the challenge of the shared cache interference. Before we explore the shared cache management techniques for virtualization systems in the next chapter, we start with the non-virtualized systems, which is simpler than virtualization systems and can be applied as the cache management technique in VMs for virtualization systems.

As discussed in Section 1.2, although shared cache can help increase the average performance, it also makes the worst-case timing analysis much more challenging due to the complex inter-core shared-cache interference: when tasks running simultaneously on different cores access memories that are mapped to the same cache set, they may evict each other's cache content from the cache, resulting in cache misses that are hard to predict.

One effective approach to bounding the inter-core cache interference is *cache partitioning*, which can be done using mechanisms such as page coloring [35] or way partitioning [49]. The idea is to divide the shared cache into multiple cache partitions and assign them to different tasks, such that tasks running simultaneously on different cores always use different cache partitions. Since tasks running concurrently never access one another's cache partitions in this approach, the cache interference due to concurrent cache accesses can be eliminated, thus reducing the overall cache overhead and improving the worst-case response times of the tasks.

In this chapter, we investigate the feasibility of *global preemptive* scheduling with dynamic *job-level* cache allocation. We present gFPca, a cache-aware variant of the global preemptive fixed-priority (gFP) algorithm, together with its analysis and implementation. gFPca allocates cache to jobs dynamically at run time when they begin or resume, and it allows high-priority tasks to preempt low-priority tasks via *both CPU and cache* resources. It also allows low-priority tasks to execute when high-priority tasks are unable to execute due to insufficient cache resource, thus further improving the cache and CPU utilizations. Since preemption is allowed, tasks may experience cache overhead – e.g., upon resuming from a preemption, a task may need to reload its cache content in the cache partitions that were used by its higher-priority tasks; therefore, we develop a new method to account for such cache overhead.

4.1 System model

We consider a multi-core platform with M identical cores and a shared cache that is accessible by all cores. The cache is partitioned into A equal cache partitions; we achieved this using the way partition mechanism [49]. The latency of reloading one partition is upper bounded by the maximum cache partition reload time, denoted by PRT. The value of PRT can be derived from the number of cache lines per partition and the maximum reloading time of one cache line. As a first step, this paper focuses on the shared-cache interference and considers only data caches; we assume that the effects of other resource interferences, such as that of private caches and memory bus, are negligible or have been included in the tasks' WCETs.

The system consists of a set of independent explicit-deadline sporadic tasks, $\tau =$

$\{\tau_1, ..., \tau_n\}$. Each task τ_i is defined by $\tau_i = (p_i, e_i, d_i, A_i)$, where p_i, e_i and d_i are the minimum inter-arrival time (which we refer to as the period), worst-case execution time (WCET) and relative deadline of τ_i, and A_i is the number of cache partitions that τ_i can use. Note that different values of A_i may lead to different values of e_i; our analysis holds for any given value of A_i and corresponding e_i. (In our numerical evaluation, A_i was chosen to be the smallest number of cache partitions that leads to the minimum WCET for τ_i.) In addition, although the number of partitions allocated to τ_i is fixed, under our scheduling approach, the exact partitions allocated to each job of τ_i may change whenever it begins its execution or resumes from a preemption.

We require that $0 < e_i \leq d_i \leq p_i$ and $A_i \leq A$ for all $\tau_i \in \tau$, where A is the total number of partitions of the shared cache. Each task has a fixed and unique priority; without loss of generality, we assume that the tasks in τ are sorted by their priorities, i.e., τ_i has higher priority than τ_j iff $i < j$.

Cache-related overhead. We assume that the WCET of each task already includes intrinsic cache-related overhead, and we focus on the extrinsic cache overhead. By abuse of terminology, throughout the paper, we refer to *one cache overhead* of a task as the time the task takes to reload its evicted cache content when it resumes from a preemption, and *total cache overhead* of a task as the total amount of time the task takes to reload its evicted cache content throughout the execution of a job of the task. We assume that the operating system does not affect the shared cache state of tasks; for example, one way to avoid the shared cache interference between the OS and tasks is to dedicate a specific area of the cache to the OS. In this paper, we consider only the shared cache overhead and defer the incorporation of the private cache overhead to future work.

ECP and UCP. We say that a task accesses a partition if it accesses any line(s) within that partition. We define an *Evicting Cache Partition* (ECP) of a task to be a cache partition that the task can access, and we denote by ECP_k the set of

ECPs of τ_k during an uninterrupted execution interval of τ_k. Note that ECP_k varies across different continuous execution intervals of τ_k, but $|\mathsf{ECP}_k| \leq A_k$ by definition. In addition, we define a *Useful Cache Partition* (UCP) of τ_k to be a cache partition that τ_k accesses at some time point and later accesses again as cache hit, when τ_k executes alone in the system. The set of UCPs of τ_k is denoted by UCP_k; by definition, $\mathsf{UCP}_k \subseteq \mathsf{ECP}_k$.

4.2 gFPca scheduling algorithm

We now present the gFPca algorithm. Like global fixed priority (gFP) scheduling, gFPca also schedules tasks based on their priorities; however, a task is only executed if there are sufficient cache partitions for it (including also the partitions obtained by preempting one or more lower-priority tasks), and low-priority tasks can execute if all pending high-priority tasks are unable to execute.

Specifically, gFPca makes scheduling decisions whenever a task releases a new job or finishes its current job's execution (or is blocked or unblocked via resources other than cache and CPU). At each scheduling point, it tries to schedule pending tasks in decreasing order of priority. For each pending task τ_i:

Step 1) First, gFPca looks for an idle core; if none exists, it considers the core that is executing the lowest-priority task among all currently executing tasks with lower priority than τ_i, if such tasks exist. If no such core is found, it returns.

Step 2) Next, gFPca tries to find A_i cache partitions for τ_i, considering the idle partitions first and then the partitions obtained by preempting τ_i's lower-priority tasks (chosen in increasing order of priority). If successful, it will reserve those A_i partitions for τ_i, preempt the lower-priority tasks that are using those partitions or using the core chosen in Step 1, and schedule τ_i to run on the chosen core. (When more than A_i partitions are found, gFPca gives preference to the ones that still hold the cache content of the task τ_i.) Otherwise, gFPca will move to the next task and repeat the process from Step 1. gFPca imposes no constraints among the partitions allocated to a task; however, both its cache allocation and analysis can easily be modified to incorporate potential constraints, e.g., one that imposes contiguous partitions. Due to space limitation, we omit the details here.

Under gFPca, cache partitions are allocated to each job *dynamically* at run time when it begins its execution and when it resumes. Whenever this occurs, the system maps some or all of the memory accesses of the task to the allocated partitions (which may include those previously belonged to a preempted task). When a preempted task resumes, it needs to reload its information from the memory to the cache, if this information has been polluted by higher-priority tasks or if it is assigned new cache partitions. Our analysis considers the costs of mapping the memory accesses and reloading the memory content into the cache. In our implementation, reassigning partitions can be done by simply resetting the registers that control the cache partitions (without the need to copy memory pages), which takes only about a few cycles; therefore, we consider the overhead of reassigning partitions as part of the context switch overhead in our analysis.

4.3 Implementation

We implemented gFPca within LITMUS^{RT} on the Freescale I.MX6 quad-core evaluation board, which supports way partitioning through the PL310 cache controller. For comparison, we also implemented the existing non-preemptive nFPca in [32] and the cache-agnostic gFP schedulers.

4.3.1 Dynamic cache control

We utilized the *Lockdown by Master (LbM)* mechanism, supported by the PL310 controller, for our cache allocation (using a similar approach as [49, 65]). The LbM allows certain ways to be marked as unavailable for allocation, such that the cache allocation (which allocates cache lines for cache misses) only happens in the remaining ways that are not marked as unavailable. Each core P_i has a per-CPU lockdown register R_i, where a bit q in R_i is one if the cache allocation cannot happen in the cache way q for the memory access from the core P_i, and zero otherwise. (To be precise, each core has two separate registers for instruction and data access, but we focus on data access in this paper.)

Challenge. To reserve the set of cache partitions S_k (represented as a bitmask) for a task τ_k on a core, we set the lockdown register of the core to be the bitwise complement of S_k. However, this alone cannot guarantee that τ_k will not access cache partitions outside S_k, because the *LbM* cannot control where the cache lookup (i.e., cache hit) occurs. As a result, tasks running concurrently on different cores may still access each other's cache partitions, even if the register is set.

Approach: Recall that the actual cache partitions allocated to a task varies from one preemption point to the next (even within the same job of the task). One way to address the above challenge is to flush the partitions allocated to each task τ_k when it completes a job or is preempted [65]. However, this approach prevents a task from reusing its content in the cache when possible: if a partition reserved for τ_k has not been used by any other task when τ_k resumes or releases a new job, then τ_k should be able to reuse the content inside that partition; this would not be possible if we had flushed the task's partitions when it was preempted or finished its previous job.

Since the cost of flushing a cache way is relatively expensive compared to other scheduler-related overhead[18], we minimized cache flushes through *selective flushing*. The idea is to select from the reserved partitions of τ_k all the partitions that may hold the content of other tasks, and *only* flush the selected partitions *when τ_k resumes or releases a new job*.

To flush a cache partition, we leveraged the hardware cache maintenance operations to clean and invalidate the specific cache ways that need to be flushed. (This

[18]The cost of flushing one cache way depends on the contention on components of the cache controller. Our measurement shows that the worst-case cost of flushing one cache way is 0.12ms.

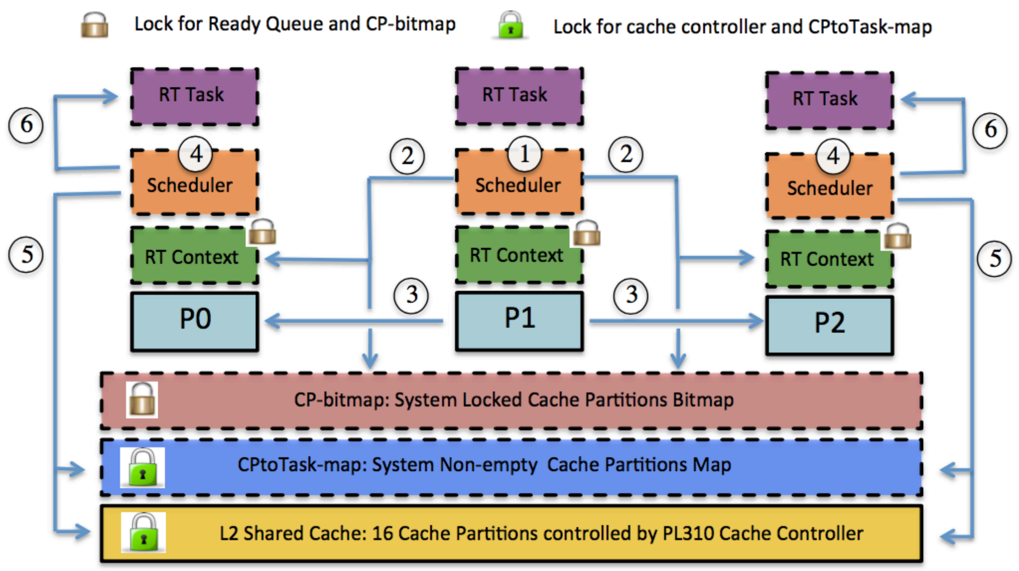

Lock for Ready Queue and CP-bitmap
Lock for cache controller and CPtoTask-map
RT Task
RT Task
RT Task
Scheduler
Scheduler
Scheduler
RT Context
RT Context
RT Context
P0
P1
P2
1
2
2
3
3
4
4
5
5
6
6
CP-bitmap: System Locked Cache Partitions Bitmap
CPtoTask-map: System Non-empty Cache Partitions Map
L2 Shared Cache: 16 Cache Partitions controlled by PL310 Cache Controller

Figure 4.1: Scheduling architecture. Dotted-line boxes enclose software components. Solid-line boxes enclose hardware comp.

is different from the approach in [65], which loads pages to the cache partitions to evict the previous content from the cache.) Our approach guarantees cache isolation among concurrently running tasks (since no task can use the reserved cache partitions of another task), and it helps to minimize the cache management overhead (since a task may use the previously – rather than currently – reserved partitions until they are reserved and flushed by another task). Note that when the cache content of a task τ_k is flushed from its previously reserved partitions (by another task), then τ_k may need to reload its content to its current reserved partitions; we account for such overhead in our analysis.

4.3.2 Scheduling architecture

Fig. 4.1 shows a high-level overview of the scheduling architecture for gFPca. Our implementation extended various components in LITMUS^{RT} to incorporate gFPca's cache management and scheduling behavior. Most notable extensions include: (1) *RT Task*: We extended the *rt_params* field, which holds the timing information of a real-time task, with the cache information (i.e., the number of cache partitions, the set of currently used partitions, and the set of previously used partitions). (2) *RT-Context*: We extended the *cpu_ entry* data structure, which holds the real-time context of a core, with a new field called *preempting* to indicate whether the core is *preempted via cache*. (3) *Scheduling real-time domain*, which holds all (global) information of the cores and real-time tasks, such as the release and ready queues (not shown in Fig. 4.1). We extended the scheduling domain to include two new components: CP-bitmap and CPtoTask-map. CP-bitmap is a bitmap that indicates whether a cache partition is locked (i.e., reserved for some task). CPtoTask-map maps each partition to a task that it belongs (if any). The architecture also includes the PL310 cache controller that controls the 16 cache partitions of the L2 shared cache. For synchronization, we used three global spin locks: one for the release queue; one for the ready queue, RT-Context, and CP-bitmap; and one for CPtoTask-map and the cache controller's registers.

The gFPca scheduler: The steps in Fig. 4.1 illustrates how the scheduler on a core works in a nutshell. Specifically, when a scheduling event (task-release, task-finish, task-blocked on other resources such as I/O, or task-unblocked event) arrives at a core (e.g., P1), the scheduler on that core will be invoked. Once being invoked, the scheduler performs Steps 1–3:

Step 1) Executes the *check_for_preemption* function, which implements the gFPca algorithm (described in Section 4.2), to determine: the highest-priority ready task that can execute next, the core to execute the task, the cache partitions to reserve for the task, and the currently running tasks to be preempted. The scheduler then continues to the next highest-priority ready task, until no more ready task can be scheduled. For the example in Fig. 4.1, the scheduler on P1 decides to preempt the tasks currently running on P0 and P2 (say τ_i and τ_j, respectively) and schedule the ready task (say τ_k) on P0.

Step 2) Updates CP-bitmap to reflect the new locked cache partitions, and updates the RT-Context of the preempted cores and the core(s) that will run the scheduled tasks. In Fig. 4.1, P1's scheduler modifies CP-bitmap by unmarking the cache partitions that were assigned to τ_i and τ_j and then marking the partitions that will be reserved for τ_k. In addition, it updates P0's *linked task* (i.e., the real-time task to execute next) to be τ_k, P2's linked task to be NULL and P2's *preempting* field to be true (to indicate that P2 is preempted via cache only).

Step 3) Sends an Inter-Processor Interrupt (IPI) to each preempted core and each core that will run a scheduled task, to notify the preempted core to preempt its currently running task and the scheduled core to execute its linked task (e.g., P0 to preempt τ_i and run τ_k, and P2 to preempt τ_j).

When a core receives the above IPI, the scheduler on that core will be invoked, and it will perform the next three steps:

Step 4) Moves the linked task (configured in Step 2) to the core, and updates the scheduled task of the core to be the linked task. (If the linked task is NULL, the scheduler will pick a non-real-time task to execute on the core. We assume that non-real-time tasks do not interfere with the real-time tasks.)

Step 5) Determines which of the cache partitions reserved for the linked task should be flushed (i.e., if used by other tasks), flushes those partitions, and updates CPtoTask-map to reflect the new mapping of partitions to tasks.

Step 6) Starts executing the linked task.

4.3.3 Run-time overhead

We used the feather-trace tool to measure the overheads, as in earlier LITMUS^{RT}-based studies (e.g., [23, 24]). Since the tool uses the timestamp counter to track the start and finish time of an event in cycles, we first validated that the timestamp counter on our board has a constant speed (necessary for precise conversion from cycles to nanoseconds). Since the timestamp counter on each core of the board is not synchronized, we also modified the tool to use the system-wide monotonically-increasing timer (in nanosecond) to trace the Inter-Processor Interrupt (IPI) delay.

We randomly generated periodic tasksets of size ranging between 50 to 450 tasks, with a step of 50. We generated 10 tasksets per taskset size (i.e., 90 tasksets in total) under each scheduler. Under each scheduler, we traced each taskset for 30 seconds, and measured all size types of overhead: release overhead, release latency, scheduling overhead, context switch overhead, IPI delay, and tick overhead (as defined in [5]). We removed the outliers using the method in [23] and computed the worst-case and average-case overheads.

	Taskset size: 50			Taskset size: 450		
	$gEDF$	gFPca	nFPca	$gEDF$	gFPca	nFPca
Release	5.72	5.86	4.74	7.73	23.92	5.45
Sched	8.64	7.75	7.57	11.88	20.07	15.25
CXS	4.23	138.72	142.46	7.31	159.84	162.93
IPI	4.06	3.64	4.12	3.92	3.84	4.03

Table 4.1: Average overhead (μs) under different schedulers with cache-read workload.

Table 4.1 shows the average overheads for taskset size of 50 and 450 under the gFPca and nFPca schedulers, as well as the existing $gEDF$ scheduler in LITMUS^{RT} for comparison. The results show that the release, scheduling, and IPI delay overheads of the gFPca and nFPca schedulers are similar to that of $gEDF$. However, gFPca and nFPca have a larger context switch overhead than $gEDF$ does, which is expected because they may need to flush cache partitions during a context switch, as described in the implementation description. The gFPca scheduler incurs higher worst-case overheads than the $gEDF$ scheduler, which is not surprising because the scheduling algorithm gFPca has a higher complexity than $gEDF$. All measured overhead values can be found in [71].

In the coming sections, we present the schedulability analysis of gFPca, first assuming the absence of overhead and then considering all types of the overhead. As the analysis of the cache-related preemption and migration delay (CRPMD) overhead is most challenging, we focus on the analysis of the CRPMD overhead in the main context and present the extension to the remaining types of overhead in Section 4.5.7. Note that our evaluation considered all these overheads.

4.4 Overhead-free analysis

The overhead-free schedulability analysis of gFPca can be established using a similar idea as that of nFPca [32]. As usual, the processor demand of a task τ_i in any interval $[a, b]$ is the amount of processing time required by τ_i in $[a, b]$ that has to complete at or before b. When task τ_i is scheduled under gFPca, τ_i has the maximum amount of computation in a period of another task τ_k when the first job of τ_i starts executing at the release time of τ_k and the following jobs of τ_i execute as early as possible, as illustrated in Fig. 4.2. Hence, the worst-case demand of τ_i in a period of τ_k is given by [20]:

$$W_i^k = NJ_i^k \cdot e_i + \min\{d_k + d_i - e_i - NJ_i^k \cdot p_i,\ e_i\}, \tag{4.1}$$

where $NJ_i^k = \lfloor \frac{d_k + d_i - e_i}{p_i} \rfloor$ is the maximum number of jobs of τ_i that have the entire executions falling within a period of τ_k.

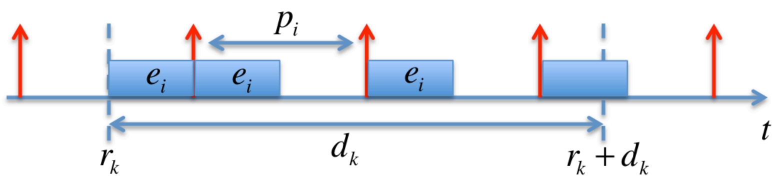


Figure 4.2: Worst-case demand of τ_i in a period of τ_k scenario.

The length of τ_k's busy interval, denoted by B_k, is the total length of all subintervals in a period of τ_k during which it cannot execute. The busy interval of τ_k can be grouped into two categories: (1) CPU-busy interval, during which all cores are busy executing other higher-priority tasks; and (2) cache-busy interval, during which at least one core is available (i.e., idle or executing a lower-priority task) and at least $A - A_k + 1$ cache partitions are assigned to τ_k's higher-priority tasks.

Consequently, the workload of τ_i in a period of τ_k consists of two types: (1) *CPU-interference workload*, α_i^k, which is the workload of τ_i when it executes in the CPU-busy interval of τ_k; and (2) *cache-interference workload*, β_i^k, which is the workload of τ_i when it executes in the cache-busy interval of τ_k. Since τ_k cannot execute when its higher-priority tasks collaboratively keep the CPU busy, and because the system has M cores, the length of the CPU-busy interval of τ_k is bounded by $\frac{1}{M}\sum_{i<k}\alpha_i^k$. Because each higher-priority task executes β_i^k time units with A_i cache partitions occupied, and because higher-priority tasks only need to occupy $A - A_k + 1$ cache partitions to prevent τ_k from execution, the combined cache resources (i.e., the number of partitions occupied in an interval multiplied by the interval length) that need to be used by all other tasks to block τ_k from execution during τ_k's cache-busy interval is bounded above by $\sum_{i<k}\min\{A_i, A - A_k + 1\}\beta_i^k$. Therefore, the length of the cache-busy interval of τ_k is bounded above by $\sum_{i<k}\frac{\min\{A_i, A-A_k+1\}}{A-A_k+1}\beta_i^k$. Since the length of the busy interval of τ_k is no more than the sum of the length of the CPU-busy interval and the length of the cache-busy interval, it is bounded above by:

$$\sum_{i<k}\left(\frac{1}{M}\alpha_i^k + \frac{\min\{A_i, A - A_k + 1\}}{A - A_k + 1}\beta_i^k\right).$$

Further, in each period of τ_k, the CPU/cache-interference workload of a higher-priority task τ_i must satisfy the following constraints: (1) the combination of the CPU-interference workload and cache-interference workload of τ_i cannot exceed the workload of τ_i, i.e., $\alpha_i^k + \beta_i^k \le W_i^k$; and (2) the CPU/cache-interference workload of all τ_i should be no more than the length of the CPU/cache-busy interval of τ_k, i.e., $\alpha_i^k \le \sum_{i<k}\frac{1}{M}\alpha_i^k$ and $\beta_i^k \le \sum_{i<k}\frac{\min\{A_i, A-A_k+1\}}{A-A_k+1}\beta_i^k$.

Based on the above discussion, we obtain the following:

Lemma 4.1. *The maximum length B_k of the busy interval of τ_k is bounded by $\widehat{B}_k$,* *where $\widehat{B}_k$ is the optimal solution of the following Linear Programming (LP) problem:*

$$\textbf{\textit{maximize}} \quad \sum_{i<k} \left(\frac{1}{M}\alpha_i^k + \frac{\min\{A_i, A - A_k + 1\}}{A - A_k + 1}\beta_i^k \right)$$

$$\begin{aligned} \textbf{\textit{subject to}} \quad & \alpha_i^k + \beta_i^k && \leq W_i^k, \ \ \forall i < k \\ & \alpha_i^k && \leq \textstyle\sum_{i<k} \frac{1}{M}\alpha_i^k \\ & \beta_i^k && \leq \textstyle\sum_{i<k} \frac{\min\{A_i, A-A_k+1\}}{A-A_k+1}\beta_i^k \end{aligned}$$

Proof. The lemma holds by construction as discussed above. □

The next theorem follows as a result of Lemma 4.1.

Theorem 4.2. *A taskset τ is schedulable under the* gFPca *algorithm if each task τ_k in τ satisfies $\widehat{B}_k \leq d_k - e_k$.*

Proof. Suppose τ is unschedulable. Then, there exists a task τ_k that is unschedulable, which implies that the length of its busy interval (B_k) is larger than the length of its slack interval, i.e., the maximum waiting or blocking time that τ_i can accommodate before missing its deadline, which is given by $d_i - e_i$. In addition, we can easily show that the CPU-interference workload α_i^k and the cache-interference workload β_i^k of each high-priority task τ_i within a period of τ_k satisfy the constraints in the above LP formulation; therefore, the maximum length of the busy interval (i.e., $\widehat{B}_k$), calculated by the LP, is no less than B_k. In other words, $\widehat{B}_k \geq B_k > d_k - e_k$. By contraposition, we imply the theorem. □

Theorem 4.3. *Given a taskset $\widetilde{\tau} = \{ \widetilde{\tau}_1, ..., \widetilde{\tau}_n \}$, where $\widetilde{\tau}_i = (p_i, \widetilde{e}_i, d_i, A_i)$ for all $1 \leq i \leq n$. Let $\tau = \{\tau_1, ..., \tau_n\}$ be any task set with $\tau_i = (p_i, e_i, d_i, A_i)$ and $e_i \leq \widetilde{e}_i$ for all $1 \leq i \leq n$. Then, τ is schedulable under the* gFPca *algorithm if $\widetilde{\tau}$ satisfies the* gFPca *schedulability conditions given by Theorem 4.2.*

Proof. We will show that if τ is unschedulable under gFPca, then $\widetilde{\tau}$ will be deemed unschedulable under Theorem 4.2.

Indeed, if τ is unschedulable under gFPca, then there exists a task $\tau_k \in \tau$ that misses its deadline. Let B_k be the maximum length of the busy interval of τ_k. Then, $\widehat{B}_k \geq B_k$ due to Lemma 4.1. Since τ_k misses its deadline, $B_k > d_k - e_k$. Combining this with $\widetilde{e_i} \geq e_i$ and $\widehat{B}_k \geq B_k$, we obtain $\widehat{B}_k > d_k - \widetilde{e_k}$. Thus, the taskset $\widetilde{\tau}$ is deemed unschedulable by Theorem 4.2. $\square$

4.5 Overhead-aware analysis

Insight. We observe that under gFPca, the cache effects τ_i has on a lower-priority task τ_k comes from not only direct preemption (i.e., τ_i is released and preempts τ_k) but also indirect preemption: when τ_i is released, it is possible that τ_i and τ_k are scheduled to run whereas an intermediate-priority τ_j $(i < j < k)$ is blocked due to insufficient cache for it; when τ_i finishes, τ_k is preempted by τ_j because there is now sufficient cache for τ_j to execute. Due to this behavior, existing approaches, such as [61], cannot be applied.

Our idea is to account for the overhead by analyzing the source events that cause cache overhead, and analyze the combined total overhead they cause to a task. As not every task experiences (extrinsic) overhead, e.g., the highest-priority task, we also derive the necessary conditions under which a task may experience overhead. Specifically, we first identify the cache-related task events and establish the necessary conditions under which these events cause a task to experience overhead. These conditions are then used to derive the set of tasks that may preempt a task τ_k via CPU or cache resource. Finally, we analyze the total overhead of τ_k that is caused by the cache-related events of other tasks and include it into τ_k's WCET, then we apply the overhead-free schedulability analysis on the inflated taskset. For simplicity, we will simply write 'overhead' in place of 'cache overhead'.

4.5.1 Overhead analysis challenges

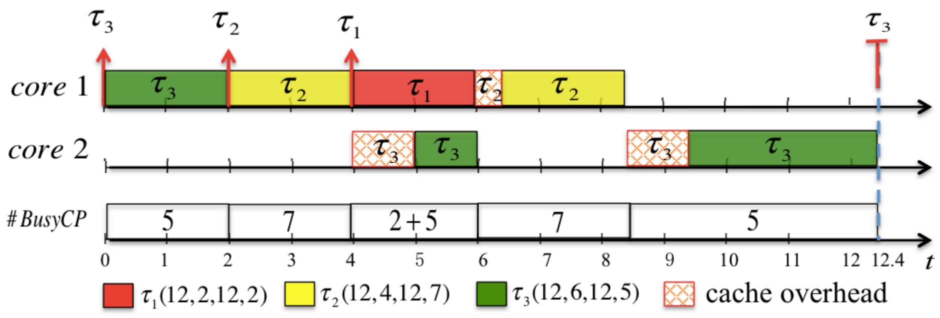
Existing overhead accounting approaches [13, 48, 24, 54, 74] typically work as follows:

- first, analyze for each task τ_i either (a) the maximum cache overhead, θ_i, that τ_i causes to its lower-priority tasks, or (b) the maximum cost of one cache overhead, Δ_i, that τ_i incurs upon resuming from a preemption;
- then, incorporate the overhead into the analysis by inflating the tasks' WCETs based on the obtained θ_i or Δ_i.

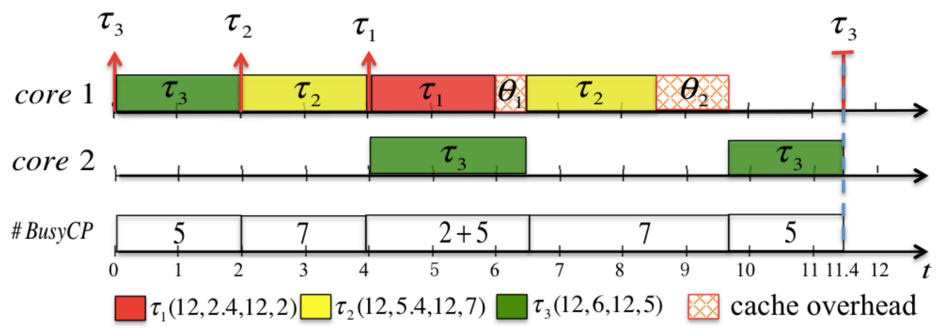
It seems intuitive at first to apply the same approach for gFPca; unfortunately, a naïve computation of θ_i or WCET inflation based on Δ_i can lead to unsafe analysis results for gFPca. (Note: these apply only to gFPca, not gFP.) We will show this using an example.

Naïve WCET inflation based on θ_i. We first compute θ_i for each task τ_i ($i < n$) and then inflate τ_i's WCET by the overhead θ_i. For this, we extend the method used in the uniprocessor setting [61]. Specifically, when a higher-priority task τ_i preempts τ_k on a uniprocessor, the cache lines that τ_i may evict from the cache must be the cache lines it can access, i.e., its ECBs (c.f. Section 6.4). Let BRT be the maximum latency of reloading one cache line and ECB_i be the ECBs of τ_i. Thus, the private-cache overhead caused by τ_i is bounded by [61]: $\theta_i^{\mathsf{uni}} = \mathsf{BRT} \times |\mathsf{ECB}_i|$. It seems intuitive to apply the same idea to gFPca by using the ECPs of τ_i, since the partitions that τ_i evicts should be the partitions it can access. Recall that PRT is the latency of reloading one cache partition and ECP_i is τ_i's ECPs. Then, the cache overhead caused by τ_i is bounded by $\theta_i = \mathsf{PRT} \times |\mathsf{ECP}_i|$. However, this bound is unsafe when applied to gFPca, as shown in the example below.

Counter Example 1. *Consider a taskset $\tau = \{ \tau_1, \tau_2, \tau_3 \}$, with $\tau_1 = (12, 2, 10, 2)$, $\tau_2 = (12, 4, 11, 7), \tau_3 = (12, 6, 12, 5)\}$, and priority order $\tau_1 \succ \tau_2 \succ \tau_3$. Suppose τ is scheduled using* gFPca *on a dual-core platform with 8 cache partitions, and τ_1, τ_2,*



(a) Actual execution in the presence of overhead.



(b) Inflating WCETs of high-priority tasks τ_i with θ_i.

Figure 4.3: Actual execution and unsafe overhead accounting scenarios for Counter Example 1.

and τ_3 are released at time 4, 2, and 0, respectively. Suppose $\mathsf{PRT} = 0.2$. *Then, as illustrated in Fig. 4.3(a), τ_3 finishes at $t = 12.4$ and thus misses its deadline.*

However, from $\theta_i = \mathsf{PRT} \times |\mathsf{ECP}_i|$, we obtain $\theta_1 = 0.4$ and $\theta_2 = 1.4$. If we inflate τ_1 and τ_2 with θ_1 and θ_2, respectively, then their inflated WCETs are $e_1' = 2.4$ and $e_2' = 5.4$. As illustrated in Fig. 4.3(b), this leads to τ_3 finishing at $t = 11.4$ and meeting its deadline. Clearly, the inflated WCETs are insufficient to account for the actual overhead τ_3 experiences.

Alternatively, if we inflate the WCET of each low-priority task (τ_2 and τ_3) with the total cache overhead caused by all of its higher-priority tasks, the inflated WCET of each task will be: $e_1 = 2$, $e_2' = e_2 + \lceil p_2/p_1 \rceil \times \theta_1 = 4.4$; and $e_3' = e_3 + \lceil p_3/p_1 \rceil \times \theta_1 + \lceil p_3/p_2 \rceil \times \theta_2 = 7.8$. Then τ_3 would finish at $t = 12.2$ which is earlier than its actual finish time.

As Fig. 4.3(a) illustrates, under gFPca the cache effects τ_i has on a lower-priority task τ_k comes from not only direct preemption (i.e., when τ_i is released and preempts τ_k) but also indirect preemption: when τ_i is released, it is possible that τ_i and τ_k are scheduled to run whereas an intermediate-priority τ_j $(i < j < k)$ is blocked due to insufficient cache; when τ_i finishes, τ_k is preempted by τ_j because there is now sufficient cache for τ_j to execute. Therefore, the number of cache partitions of τ_k that are evicted can be as large as $|\mathsf{ECP}_j \cup \mathsf{ECP}_i|$ (which is more than $|\mathsf{ECP}_i|$).

4.5.2 Cache-related task events

Under gFPca, the system has five types of task events: task-release, task-finish, task-preemption, task-resumption, and task-migration events. Because the cache is shared by all cores, no overhead is incurred when a task migrates from one core to another; therefore, a task-migration event of a task does not lead to any overhead and we only need to consider the other four types of task events.

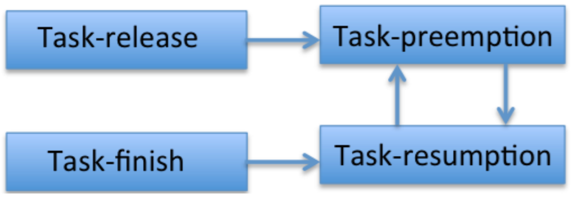


Figure 4.4: Causal relations of task events.

A task-preemption event of τ_k occurs when the CPU or cache resource allocated to τ_k is reduced. Because new jobs are released when task-release events occur and existing jobs resume when task-resumption events occur, a higher-priority task τ_i with the task-release or task-resumption event may take the CPU and/or cache resource from τ_k, thus leading to a task-preemption event of τ_k. Similarly, because running jobs may stop at task-preemption and task-finish events, and the released CPU or cache resource may be allocated to τ_k, both task-preemption and task-finish events of τ_i may lead to a task-resumption event of τ_k. Further, a task-preemption event may lead to a task-resumption event and vice versa.

If the arrival of a task event A may lead to the arrival of another task event B, then we say A causes B, denoted as $A \rightarrow B$. The causal relations of task events are illustrated in Fig. 4.4. It is clear from the figure that the task-release and task-finish events are the root causes of the other events. Since a task experiences overhead only at its task-resumption events, which are caused by task-release and task-finish events of other tasks, if the task-release and task-finish events are eliminated, the overhead will also be eliminated.

Lemma 4.4. *Task-release events and task-finish events are the source events that cause overhead in a system.*

Proof. As illustrated in Fig. 4.4, other task events, i.e., task-resumption event and task-preemption event, are caused by the task-release event and the task-finish event. If there is no task-release event or task-finish event in a system, the other task events will not exist and hence the overhead will not occur. □

Based on Lemma 4.4, if we can compute a bound on the overhead that each task-release event and each task-finish event of a higher-priority task τ_i cause to a lower-priority task τ_k, then we can safely account for the total overhead of τ_k. To derive this bound, we will analyze the set of tasks that can preempt τ_k based on the necessary conditions of task-preemption events, which we now establish.

4.5.3 Conditions of task-preemption events

The overhead that a task τ_k experiences come from its preemption events, which are caused by the task-release and task-finish events of its higher-priority tasks. A higher-priority task τ_i may preempt τ_k via either CPU and/or cache resources; however, no task-preemption event of τ_k occurs if the number of cores is larger than the number of tasks in the system and the number of cache partitions of the platform is sufficient for all tasks. The next lemmas state the conditions of a preemption via CPU and cache resources, respectively.

Lemma 4.5. *If a task τ_i preempts a task τ_k's CPU resource at time t, then τ_i must have higher priority than τ_k and the number of tasks with higher priority than τ_k must be at least the total number of cores in the system, i.e., $\sum_{j<k} 1 \geq M$.*

Proof. Suppose the number of tasks with higher priority than τ_k is smaller than M, there must exist a core that is either idle or executing a lower-priority task τ_l $(k < l)$ at time t. Then τ_i should either execute on the idle core or preempt the CPU resource of τ_l instead of preempting the CPU resource of τ_k. This contradicts the fact that τ_i preempts τ_k. Therefore, this lemma holds. $\square$

Lemma 4.6. *If τ_i preempts τ_k's cache resource at t, then τ_i must have higher priority than τ_k and the total number of cache partitions of τ_j with $j < k$ must be larger than $A - A_k$, where A is the number of cache partitions of the cache.*

Proof. Suppose $\sum_{j \leq k} |ECP_j| \leq A$, the set of tasks whose priority are no smaller than τ_k can reside in the cache at the same time without interfering each other. Therefore, τ_i should not preempt τ_k's cache resource. This contradicts to the fact that τ_i preempts τ_k's cache resource. $\square$

Let ρ_k and κ_k be the maximum sets of tasks that may preempt τ_k via CPU and cache resources, respectively. Due to the above lemmas, we have:

$$\rho_k = \{\tau_i \mid i < k \text{ and } \sum_{j<k} 1 \geq M\} \tag{4.2}$$

$$\kappa_k = \{\tau_i \mid i < k \text{ and } \sum_{j \leq k} A_j > A\} \tag{4.3}$$

As a result, the set of tasks that may preempt τ_k via either CPU or cache or both resources is $\rho_k \cup \kappa_k$.

4.5.4 Overhead caused by a task-release event

Based on the established conditions of a task-preemption event of τ_k, we can analyze the overhead of τ_k that is caused by one task-release event of a higher-priority task

τ_i.

Observe that when τ_i releases a job at time t_1, the cache partitions τ_i may access and pollute are in ECP_i. If τ_k is preempted at the task-release event of τ_i, τ_i can directly evict all cache partitions in ECP_i that τ_k may use in the worst case.

Further, another higher-priority task τ_j of τ_k may release a job at time t_1 as well. Although such a task-release event may also cause overhead to τ_k, this overhead will be considered as the overhead caused by τ_j's task-release events (rather than by τ_i's). Further, under gFPca, a lower-priority task τ_l may also pollute the cache partitions of τ_k while τ_k is being preempted due to a task-release event of τ_i. However, not every lower-priority task τ_l can pollute the cache partitions of τ_k.

Lemma 4.7. *When a release-event of τ_i occurs, if τ_k is preempted but a lower-priority task τ_l ($k < l$) either resumes from a preemption or releases a new job and this job is executed, then the number of cache partitions of τ_l must be less than that of τ_k, i.e., $A_l < A_k$.*

Proof. Suppose $A_l \geq A_k$ and τ_k is preempted at t_0 and resumes at t_3, where $t_0 < t_3$. Because τ_l starts running either by release a new job or resuming from preemption during $[t_0, t_3]$, τ_l must be able to acquire A_l cache partitions and one core to run during $[t_0, t_3]$. However, because $A_l \geq A_k$ and $l > k$, τ_k should preempt τ_l and resumes from preemption during $[t_0, t_3]$ based on the gFPca scheduling. This contradicts to the fact that τ_k is not running during $[t_0, t_3]$. □

Let $\phi^r_{i,k}$ denote the set of useful cache partitions of τ_k that may be polluted due to a task-release event of τ_i. When a task-release event of τ_i occurs, there are three scenarios: (1) τ_i does not preempt τ_k (as there are sufficient CPU and cache resources for τ_i), in which case τ_k experiences no overhead due to this task-release event of τ_i; (2) τ_i preempts τ_k by taking only τ_k's CPU resource, in which case only the lower-priority tasks of τ_k may pollute the UCPs of τ_k; and (3) τ_i preempts τ_k by taking τ_k's cache resource, in which case both τ_i and lower-priority tasks of τ_k may pollute the UCPs of τ_k. Therefore, $\phi^r_{i,k}$ can be calculated as follows:

$$\phi^r_{i,k} = \begin{cases} \mathsf{UCP}_k \cap \big(\mathsf{ECP}_i \cup (\underset{k<l,A_l<A_k}{\cup} \mathsf{ECP}_l)\big), & \text{if } \tau_i \in \kappa_k \\ \mathsf{UCP}_k \cap \big(\underset{k<l,A_l<A_k}{\cup} \mathsf{ECP}_l\big), & \text{if } \tau_i \notin \kappa_k \wedge \tau_i \in \rho_k \\ \emptyset, & \text{if } \tau_i \notin \{\kappa_k \cup \rho_k\} \end{cases} \tag{4.4}$$

Given any two sets S_1 and S_2, we have $|S_1 \cup S_2| \leq |S_1| + |S_2|$ and $|S_1 \cap S_2| \leq \min\{|S_1|, |S_2|\}$. Hence,

$$|\phi^r_{i,k}| \leq \begin{cases} \min\{|\mathsf{UCP}_k|, |\mathsf{ECP}_i| + \sum_{k<l,A_l<A_k} |\mathsf{ECP}_l|\}, & \text{if } \tau_i \in \kappa_k \\ \min\{|\mathsf{UCP}_k|, \sum_{k<l,A_l<A_k} |\mathsf{ECP}_l|\}, & \text{if } \tau_i \notin \kappa_k \wedge \tau_i \in \rho_k \\ 0, & \text{if } \tau_i \notin \{\kappa_k \cup \rho_k\} \end{cases} \tag{4.5}$$

Denote by $\Delta^r_{i,k}$ the overhead of τ_k that is caused by a task-release event of τ_i, where $i < k$. Then,

$$\Delta^r_{i,k} \leq \mathsf{PRT} \cdot |\phi^r_{i,k}|. \tag{4.6}$$

4.5.5 Overhead caused by a task-finish event

When a task τ_i finishes its execution at time t_2, the overhead that task τ_k may experience due to this task-finish event falls into the following cases:

Case 1) τ_k is not running at t_2: If τ_k finishes before or at t_2, then clearly the task-finish event causes no overhead to τ_k. If it has not finished its execution at t_2, this task-finish event also does not bring any overhead to τ_k, because even though τ_i might have polluted τ_k's cache before t_2, the pollution is caused by other task-release or task-finish events of τ_i and should be accounted in the cost of those events.

Case 2) τ_k is running at t_2: If τ_k continues to run after t_2, then it incurs no overhead as it is not preempted. However, if τ_k is preempted at t_2, then it must be preempted by another higher-priority task τ_j that is resumed at t_2 when τ_i finishes, in which case τ_j can access and pollute any cache partitions in ECP_j. However, as stated in the next two lemmas, at most one task τ_j with $i < j < k$ can resume *and* preempt τ_k at t_2, and the number of cache partitions this task can access should be more than that of τ_k.

Lemma 4.8. *If a task τ_j, where $i < j < k$, resumes and preempts τ_k at a task-finish event of τ_i, then $A_j > A_k$.*

Proof. We will prove this lemma by contradiction. Suppose $A_j \leq A_k$. Let the task-finish event of τ_i occur at t_2. Because τ_k is preempted by τ_j at t_2 when τ_j resumes, τ_k must be running at $t_2 - \epsilon$, where ϵ is an infinite small time interval. Because $A_j \leq A_k$ and τ_k is running at $t_2 - \epsilon$, τ_j should preempt τ_k and resumes at $t_2 - \epsilon$ based on the gFPca scheduling. This contradicts to the fact that τ_j resumes at t_2. Hence, it is proved. $\square$

Lemma 4.9. *There exists at most one task τ_j with $i < j < k$ that can resume* and *preempt τ_k at a task-finish event of τ_i.*

Proof. Suppose there exist two tasks τ_j and τ_j' at the task-release event of τ_i at t_2, such that both tasks resume *and* preempt the low priority task τ_k. When a task preempts another task, the preempting task has to acquire the cpu or cache partition resource from the preempted task. Because both τ_j and τ_j' preempt τ_k at t_2, both τ_j and τ_j' should preempt parts of τ_k's resource.

Suppose either τ_j or τ_j' does not preempt any cache partition resource from τ_k. Either τ_j or τ_j' only preempts the cpu resource from τ_k. Then we can switch the cpu used by τ_j and τ_j' so that the task that only preempts τ_k's cpu resource will no longer preempts any resource from τ_k. Because this reduces the number of preemption, the gFPca scheduling will always choose to let only one such task preempt the τ_k in this situation. Therefore, this situation contradicts to the hypothesis that both tasks preempt τ_k.

Suppose both τ_j and τ_j' preempt the cache partition resource from τ_k. τ_j acquires A_j^k cache partitions from τ_j and $A_j - A_j^k$ cache partitions from the system, and τ_j' acquires $A_j^{k'}$ cache partitions from τ_j and $A_j' - A_j^{k'}$ cache partitions from the system. If $A_j^k \leq (A_j' - A_j^{k'})$, we can just let τ_j and τ_j' exchange A_j^k cache partitions so that τ_j will have no cache partitions from τ_k. Therefore, τ_j will no longer preempt τ_k, which contradicts to the fact that both τ_j and τ_j' preempts τ_k. If $A_j^k > (A_j' - A_j^{k'})$, then $A_j^k + A_j^{k'} > A_j'$. Because $A_k \geq A_j^k + A_j^{k'}$, we have $A_k > A_j'$. Because τ_j' resumes *and* preempts τ_k at the task-finish event, $A_j' > A_k$ according to Lemma 4.8, which contradicts to the fact $A_k > A_j'$ we derived from the hypothesis. Hence, it is proved. $\square$

In addition, when τ_k is preempted, lower-priority tasks of τ_k may also resume or release new jobs and these jobs are executed, and thus they may pollute the cache partitions of τ_k. According to Lemma 4.7, only lower-priority tasks τ_l with $k < l$ and $A_l < A_k$ may pollute τ_k's cache partitions while τ_k is being preempted. When a task τ_j $(i < j < k)$ resumes and preempts τ_k at the occurrence of the task-finish event of τ_i, the set of useful cache partitions of τ_k that may be polluted, denoted by $\phi_{i,j,k}^f$, is the same with the set of useful cache partition of τ_k that may be polluted at the task-release event of τ_j. Therefore, $\phi_{i,j,k}^f = \phi_{j,k}^r$ and the size of $\phi_{i,j,k}^f$ is $|\phi_{i,j,k}^f| = |\phi_{j,k}^r|$.

Let $\Delta_{i,k}^f$ denote the overhead of τ_k that is caused by a task-finish event of τ_i, where $i < k$. Because any task τ_j ($i < j < k$ and $A_k < A_j$) may resume and preempt τ_k at the task-finish event of τ_i, we obtain

$$\Delta_{i,k}^f \leq \max_{i<j<k, A_k<A_j} \mathsf{PRT} \cdot |\phi_{i,j,k}^f|. \tag{4.7}$$

4.5.6 Overhead-aware schedulability analysis

In the previous sections, we have computed the maximum overhead that each task-release event and each task-finish event of a higher-priority task τ_i causes to a lower-priority task τ_k. To account for the overall overhead τ_k experiences, we need to compute the number of task-release and task-finish events of higher-priority tasks in each period of τ_k.

Since each job of a task has one task-release event and one task-finish event, it may seem at first that an upper bound on the total number of task-release and task-finish events of all higher-priority tasks in the period of τ_k is $\sum_{i<k} 2\lceil \frac{d_k}{p_i} \rceil + 2$. While this bound is safe, it is not tight because not every task-release event or task-finish event of each job of higher-priority tasks can cause overhead to τ_k, as stated by Lemma 4.10.

Lemma 4.10. *If a task τ_k is preempted at the release time t_1 and again at the finish time t_2 of the same job of a higher-priority task τ_i, then τ_k must have been resumed at some time t_3 during the interval (t_1, t_2) when some other higher-priority task τ_j $(j < k)$ releases or finishes.*

Proof. Because τ_k is preempted at t_1, τ_k is running at $t_1 - \epsilon$ and not running at $t_1 + \epsilon$, where ϵ is an infinite small time interval. Similarly, τ_k is running at $t_2 - \epsilon$ and not running at $t_2 + \epsilon$, since τ_k is preempted at t_2. Because τ_k is not running at $t_1 + \epsilon$ but runs at $t_2 - \epsilon$, where $t_1 < t_2$, τ_k must resume from the not-running status to the running status during $[t_1 + \epsilon, t_2 - \epsilon]$. As discussed in Section 4.5.2, a task resumes only when a task-release event or a task-finish event of a high priority task occurs. It is proved. □

Thus, instead of accounting for the overhead caused by each task-release and each task-finish event of higher-priority tasks, we account for the overhead of τ_k that is caused by *each job* of its higher-priority tasks in a period of τ_k, as follows:

If only one of the task-release and task-finish events of the same job of τ_i may cause overhead to τ_k, the overhead caused by each job of τ_i is $\max\{\Delta^r_{i,k}, \Delta^f_{i,k}\}$. In contrast, if both the task-release and task-finish events of the same job of τ_i may cause overhead to τ_k, the maximum overhead of τ_k that is caused by each job of τ_i is the total overhead caused by the task-release and task-finish events of the job *minus* the minimal overhead caused by the task-release event or the task-finish event of a high-priority task τ_j ($j < k$ and $j \neq i$), i.e., $\Delta^r_{i,k} + \Delta^f_{i,k} - \min_{j<k, j\neq i}\{\Delta^r_{j,k}, \Delta^f_{j,k}\}$. Hence, the overhead of τ_k that is caused by one job of a higher-priority task τ_i is bounded by

$$\delta^k_i \overset{\mathsf{def}}{=} \max\{\Delta^r_{i,k}, \Delta^f_{i,k}, \Delta^r_{i,k} + \Delta^f_{i,k} - \min_{j<k, j\neq i}\{\Delta^r_{j,k}, \Delta^f_{j,k}\}\}.$$

Further, the number of jobs of τ_i in a period of τ_k that have both release and finish events causing τ_k to resume is at most $NI^k_i \overset{\mathsf{def}}{=} \lceil \frac{d_k}{p_i} \rceil$. Since the finish event of the carry-in job of τ_i and the release event of the carry-out job of τ_i in a period of τ_k may also lead to one task-resumption event of τ_k, we imply that the overhead of τ_k that is caused by all of its higher-priority tasks is upper bounded by

$$\delta^k = \sum_{i=1}^{k-1} \delta^k_i \cdot NI^k_i + \Delta^f_{i,k} + \Delta^r_{i,k} \tag{4.8}$$

The overhead-aware analysis can now be done by first inflating the WCET of each task τ_k with δ^k, and then applying the overhead-free analysis (Section 4.4) on the inflated taskset.

Theorem 4.11. *A taskset $\tau = \{\tau_1, ..., \tau_n\}$, where $\tau_k = (p_k, e_k, d_k, A_k)$, is schedulable under* gFPca *in the presence of cache overhead if $\tau' = \{\tau'_1, ..., \tau'_n\}$ satisfies Theorem 4.2, where $\tau'_k = (p_k, e'_k, d_k, A_k)$ and $e'_k = e_k + \delta^k$ for all $1 \leq k \leq n$.*

Proof. We will prove the theorem by contradiction. Suppose τ is unschedulable, i.e., there exists τ_k that misses its deadline, in the presence of overhead. Then, the maximum length of the busy interval of τ_k in the presence of overhead, denoted by B^{ca}_k, must be larger than $d_k - e_k$. Since e'_i is a safe upper bound of the worst-case execution time of τ_i in the presence of cache overhead for all $1 \leq i \leq n$, the worst-case demand of task τ'_i within a period of τ'_k in the absence of cache overhead is greater than or equal to the worst-case demand of task τ_i within a period of τ_k in the presence of cache overhead. Similar to the proof of Theorem 4.3, we can therefore establish that the maximum length of the busy interval of τ'_k in the absence of cache overhead – i.e., the optimal solution $\widehat{B}'_k$ of the LP formulation for the task set τ' (c.f. Section 4.4) – is a safe upper bound of the length of the busy interval of τ_k in the presence of overhead. Thus, $B'_k \geq B^{ca}_k > d_k - e_k$, which in turn implies that τ' does not satisfy the schedulability conditions given by Theorem 4.2. This proves the theorem. $\square$

4.5.7 Extension to other overhead types

Real-time tasks typically experience six major sources of overhead [22]: release, scheduling, context-switching, IPI overhead, cache related preemption and migration (CRPMD), and tick overheads. We specify the cost of each of these six overheads as $\Delta^{rel}, \Delta^{sched}, \Delta^{cxs}, \Delta^{ipi}, \Delta^{crpmd}$, and Δ^{tick}. Since the tick overhead is quite small ($< 11\mu$s for 450 tasks on our board) and does not involve any scheduling-related logic under all three (event-driven) schedulers (gFPca, nFPca, and gFP), we exclude it from the analysis and focus on the other five types of overhead. (Our analysis does not consider blocking overhead.) We first analyze the overhead when a task executes alone, and then account for all types of preemption-related overhead. We then perform WCET inflation, and apply the overhead-free schedulability analysis on the inflated taskset. The overhead values of each scheduler are measured based on our implementation.

Overhead accounting when a task executes alone. We observe that a task τ_k always incurs one release overhead, one IPI delay overhead, one scheduling overhead, and one context switch overhead, when it executes alone in the system under any of the three schedulers. Therefore, the execution time $\bar{e_k} = e_k + \Delta^{rel} + \Delta^{ipi} + \Delta^{sched} + \Delta^{cxs}$ is a safe bound on the execution time e_k of τ_k in the presence of the overhead when the task executes alone.

Overhead accounting under gFPca. Fig. 4.5 illustrates the preemption-related

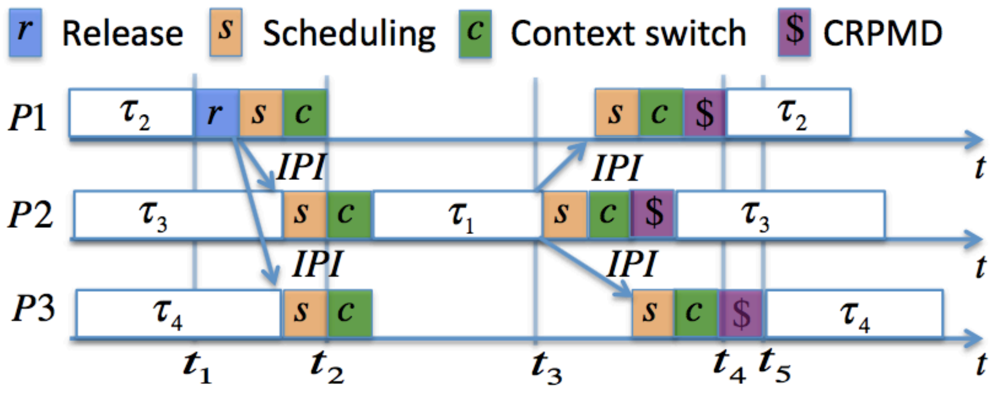


Figure 4.5: Overhead scenario when four tasks, $\tau_1 \succ \tau_2 \succ \tau_3 \succ \tau_4$, are scheduled under gFPca on three cores. Task τ_1, which requires all system's cache partitions, releases a job and preempts τ_2, τ_3 and τ_4 at t_1. When τ_1 finishes execution at t_3, the other three tasks resume. Note that the cost of the context switch overhead and the CRPMD overhead depends on the task that releases a new job or that resumes.

overhead under gFPca. We observe that at each task-resumption event of τ_k, τ_k experiences all three types of overhead, CPRMD, scheduling, and context switch once. Hence, we can account for preemption-related scheduling and context switch overheads using the same approach as the CRPMD overhead accounting in Section 4.5. Specifically, the number of task-resumption events of a task τ_k in each of its period is bounded by $NR_k = \sum_{i=1}^{k-1}(\lceil \frac{d_k}{p_i} \rceil + 2)$. The total preemption-related scheduling and context switch overhead is thus at most $\gamma_k = NR_k \times (\Delta^{sched} + \Delta^{cxs})$. Hence, the execution time of τ_i with all five overhead types is bounded by

$$e'_k = e_k + \Delta^{rel} + \Delta^{ipi} + \Delta^{sched} + \Delta^{cxs} + \delta^k + \gamma_k. \tag{4.9}$$

Overhead accounting under gFP. When a preemption event of τ_k occurs under gFP, τ_k incurs one scheduling overhead, one context switch overhead, and one CRPMD overhead, similar to the preemption-related overhead scenario under $gEDF$ shown in [22]. Since gFP does not provide cache isolation, concurrently running tasks may still evict out the cache content of each other. Since it is difficult to predict or analyze which cache content of a task may be evicted out by another currently running task, we assume all cache accesses incur cache misses to safely account for the shared-cache overhead under gFP. Let α_k be the fraction of the WCET of a task τ_k that is spent on cache hit without the shared-cache interference, and $hit\_latency$ and $miss\_latency$ be the cache hit and miss latency of the shared cache, then the shared-cache overhead of τ_k under gFP is

$$\delta_k = \lceil (\alpha_k \times e_k)/hit\_latency \rceil \times (miss\_latency - hit\_latency)$$

Therefore, the inflated execution time of τ_k that accounts for five types of overhead is bounded by

$$e'_k = e_k + \Delta^{rel} + \Delta^{ipi} + 2 \times \Delta^{sched} + 2 \times \Delta^{cxs} + \Delta^{crpmd} + \delta_k \tag{4.10}$$

Overhead accounting under nFPca. Because no preemption occurs under nFPca, the WCET of each task τ_k that accounts for all five types of overhead under nFPca is bounded by $e'_k = e_k + \Delta^{rel} + \Delta^{ipi} + \Delta^{sched} + \Delta^{cxs}$.

Overhead-aware analysis. For each scheduler (i.e., gFPca, nFPca and gFP), the overhead-aware analysis can now be achieved by applying its overhead-free analysis to the inflated taskset with the inflated WCET computed above.

4.6 Numerical evaluation

Our evaluation was based on randomly generated real-time workloads and our implementation platform, which has four cores and a 1MB shared cache that is partitioned into 16 equal partitions. We had two main objectives: (1) Evaluate the accuracy of the overhead-aware analysis for gFPca, by comparing to the overhead-free analysis and a baseline overhead-aware analysis; intuitively, the closer the overhead-aware schedulability results are to the overhead-free schedulability results, the closer the overhead accounting is to an optimal overhead accounting method. (2) Investigate the performance of gFPca in comparison to gFP and nFPca.

4.6.1 Baseline Analysis

For the baseline, since no existing overhead-aware analysis can be directly applied to gFPca, we used an extension of existing approach that works as follows: first inflates the WCET of each task τ_i ($i > 1$) with the total overhead it experiences during an entire execution of a job and then applies gFPca's overhead-free analysis.

This baseline method performs WCET inflation based on the cache overhead Δ_i that each task τ_i incurs upon resuming from a preemption. However, instead of inflating the WCET of each high-priority task with the maximum of one cache overhead of its lower-priority tasks (which is unsafe), it inflates the WCET of each τ_i with its total cache overhead (i.e., the overhead it experiences during the entire execution of a job).

Computing the total overhead of τ_i: The cache overhead that τ_i experiences when it resumes from a preemption is upper bounded by $\Delta_i \leq \mathsf{PRT} \times |\mathsf{UCP}_i|$. Since a cache partition of τ_i may be evicted from the cache only when another task τ_j uses the same cache partition, we can tighten Δ_i by considering the cache partitions used by other tasks:

Lemma 4.12. *The cache overhead a task τ_i experiences when it resumes from one preemption is upper bounded by* $\Delta_i = \mathsf{PRT} \times |\mathsf{UCP}_i \cap \cup_{j \neq i} \mathsf{ECP}_j| \leq \mathsf{PRT} \times \min\{|\mathsf{UCP}_i|, \sum_{j \neq i} |\mathsf{ECP}_j|\}$.

Proof. When a task τ_i is preempted, any other executing task τ_j may access and pollute the cache partitions in ECP_j that may be used later by τ_i.

If $| \cup_{j \neq i} \mathsf{ECP}_j| \geq A$, then the other tasks may collectively evict out all cache partitions of the cache. Therefore, $|\mathsf{UCP}_i \cap \cup_{j \neq i} \mathsf{ECP}_j| = |\mathsf{UCP}_i|$. Because $\Delta_i \leq PRT \times |UCP_i|$, the lemma holds.

If $| \cup_{j \neq i} \mathsf{ECP}_j| < A$, then the cache partitions of all tasks except for τ_i can be accommodated in the cache. Each job of τ_j will always use the cache partitions in ECP_j, because the latter job of τ_j may benefit from the cache partitions loaded by the previous job of τ_j. Therefore, the set of cache partitions of the task set $\cup_{j \neq i}\{\tau_j\}$ is $\cup_{j \neq i}\mathsf{ECP}_j$. Because a useful cache partition of τ_i will not be evicted unless the same cache partition is used by other tasks when τ_i is not executing, the maximum number of useful cache partitions of τ_i that may be evicted during the preemption of τ_i is upper bounded by $|\mathsf{UCP}_i \cap \cup_{j \neq i}\mathsf{ECP}_j|$. The maximum cache overhead τ_i experiences when it resumes is $\mathsf{PRT} \times |\mathsf{UCP}_i \cap \cup_{j \neq i}\mathsf{ECP}_j|$.

Given two sets A and B, we have $|A \cup B| \leq |A| + |B|$ and $|A \cap B| \leq \min\{|A|, |B|\}$. Therefore, it is easy to derive that $\Delta_i \leq \mathsf{PRT} \times \min\{|\mathsf{UCP}_i|, \sum_{j \neq i} |ECP_j|\}$. □

To bound the total cache overhead of τ_i, we next derive the maximum number of times that τ_i resumes (i.e., number of resumption events of τ_i) in each job's execution.

Lemma 4.13. *A task τ_i resumes only when one of the following two events happens: a higher-priority task of τ_i finishes its execution, or a higher-priority task of τ_i releases a new job.*

Proof. A task τ_i resumes only when τ_i can acquire the CPU or cache resource that were preempted by higher-priority tasks. A higher-priority task τ_l can release the CPU and/or cache resource to τ_i when it finishes its execution or when it releases a new job and preempts a medium-priority task, which will then release the resources that τ_i needs. Although there exist other two cache-related task events in the system, i.e., the task-preemption event and the task-migration event, neither of them is the source event that may release the CPU or cache resource to τ_i. Hence, the lemma holds. □

Lemma 4.14. *The maximum number of task-resumption events of τ_i during each period is at most $NS_i = \sum_{j<i} 2\lceil \frac{d_i}{p_j} \rceil + 2$.*

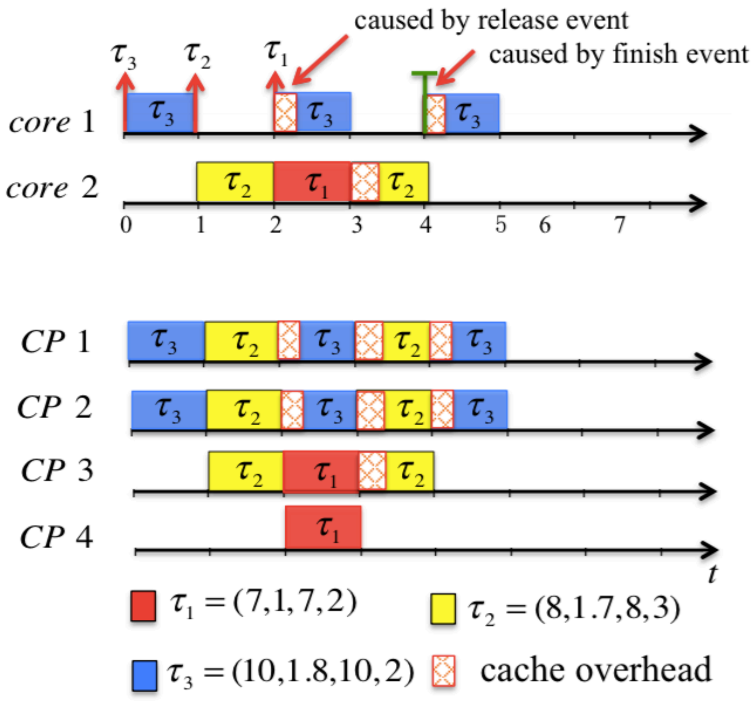


Figure 4.6: Task resumption event is caused by task release event or task finish event

Proof. Suppose a taskset $\tau = \{\tau_1 = (7, 1, 7, 2), \tau_2 = (8, 1.7, 8.3), \tau_3 = (10, 1.8, 10, 2)\}$ is scheduled by the gFPca algorithm on a platform with two cores and four cache partitions. The release and scheduling patterns of the three tasks are illustrated in Fig. 4.6. Under gFPca, the priority order of the three tasks is $\tau_1 > \tau_2 > \tau_3$.

As illustrated in the figure, task τ_3 resumes at $t = 2$ when the higher-priority task τ_1 releases a new job and at $t = 4$ when the higher-priority task τ_2 finishes its execution. We observe that both the task-release event and the task-finish event of a higher-priority task may cause τ_i to resume from a preemption.

Based on this observation, the number of task-resumption events of τ_i will be no more than the total number of task-release events and task-finish events of its higher-priority tasks. Because a higher-priority task τ_j has at most $\lceil \frac{d_i}{p_j} \rceil$ jobs whose release time and *finish time* are in the problem window of τ_i, one carry-in job whose finish time is in the problem window of τ_i, and one carry-out job whose release time is

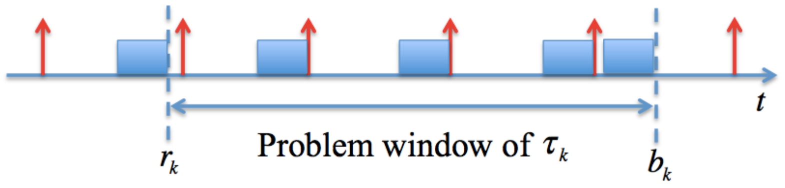


Figure 4.7: Number of task resumption event in worst case

in the problem window of τ_i (based on the worst-case scenario in Fig. 4.7), the total number of the task-resumption events of τ_i in each of its periods is upper bounded by $2\lceil \frac{d_i}{p_j} \rceil + 2$. By combining the number of task-resumption events of τ_i that are caused by each higher-priority task in the problem window of τ_i, we obtain the lemma. $\square$

Since τ_i only incurs (extrinsic) cache overhead whenever it resumes, the total overhead of τ_i is therefore at most $NS_i \times \Delta_i$.

Overhead-aware analysis: Since the total overhead of τ_i is at most $NS_i \times \Delta_i$, the WCET of τ_i in the presence of cache overhead is at most $e'_i = e_i + NS_i \times \Delta_i$. As a result, the overhead-aware analysis can be established by applying the overhead-free analysis on the inflated workload.

4.6.2 Experiment setup

Workload. Each workload contained a set of randomly generated implicit-deadline sporadic task sets. The tasks' utilizations followed the uniform distribution within the range [0.5, 0.9] as used in [66] [74]. The number of ECPs of a task was uniformly distributed in [1, 8] by default. The number of UCPs was set equal to the number of ECPs (i.e., we considered the conservative case of our theory, where the UCPs and ECPs of a task are the same).

Overhead values. For the CRPMD overhead, the latency of reloading one cache line measured on our board was 90.89ns. The size of each cache line is 32B, and thus each cache partition has $\frac{1MB}{32B \times 16} = 2048$ cache lines. Hence, it takes at most

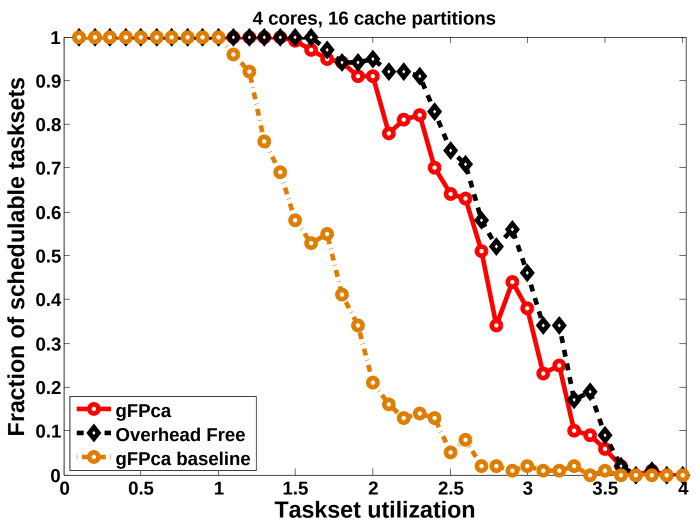


Figure 4.8: Analysis accuracy

90.89ns $\times$ 2048 $\leq$ 0.19ms to reload one cache partition. Hence, we set the cache partition reloading time $\mathsf{PRT} = 0.19$ms.

We measured the remaining overheads for each scheduler (gFPca, nFPca, gFP), and used monotonic piece-wise linear interpolation to derive the upper-bounds of each overhead under each scheduler as a function of the taskset size. For gFPca, the context switch overhead also includes the overhead for (re)assigning cache partitions, which we derived from the measured maximum latency of flushing one cache partition. (Details of the overhead values are available in [71]).

4.6.3 Evaluation of the overhead-aware analysis

We generated 4000 tasksets with taskset utilization ranging from 0.1 to 4, with a step of 0.1. For each taskset utilization, there were 100 independently generated tasksets; the task utilizations were uniformly distributed in $[0.5, 0.9]$; the task periods were uniformly distributed in $[10, 40]$ms. (These parameters followed existing work such as [66] [74].) Fig. 4.8 shows the fraction of schedulable tasksets under each analysis.

The results show that our overhead-aware analysis (shown as gFPca) is substantially tighter than the baseline; for example, when the taskset utilization is 2.5, the

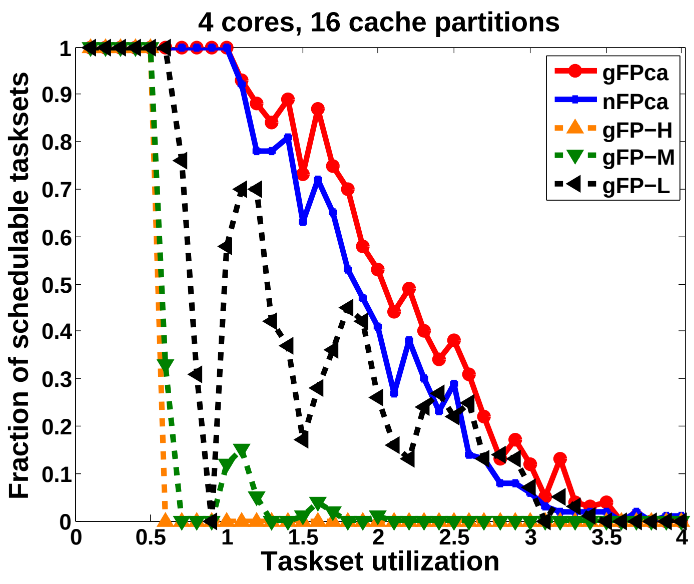


Figure 4.9: Generic.

baseline analysis claimed that only 5% of the tasksets are schedulable, even though 64% of the tasksets are schedulable under our overhead-aware analysis.

The results also show that the fractions of schedulable tasksets under our overhead-aware analysis and the overhead-free analysis are very close across all taskset utilizations. This means that our overhead-accounting technique is very close to an optimal overhead-accounting technique, which can be explained from its novel strategies for bounding the overhead.

We also evaluated the impacts of core and cache configurations, and the results further confirm these observations.

4.6.4 Evaluation of gFPca's performance.

We generated 4000 tasksets as before. The number of cache partitions of each task was uniformly distributed in $[1, 12]$. The period range that each task chooses was uniformly distributed in $[550, 650]$ (this was chosen based on [41]). We analyzed the schedulability of each taskset under gFPca, nFPca, and gFP.

Cache access information for gFP *analysis.* The overhead-aware analysis for gFP needs to consider the shared cache interference among concurrent tasks (which are eliminated in gFPca and nFPca). We derived the overhead that a task experiences from the cache hit latency (55.77ns), miss latency (146.66ns), and the *hit_time_ratio* of the task (i.e., the ratio of the time it spends on cache hit accesses to its execution time when executing alone). To generate different cache access scenarios, the *hit_time_ratio* of tasks was uniformly distributed in [0.1, 0.3] (cache light), (0.3, 0.6] (cache medium), and (0.6, 0.9] (cache heavy). The generated *hit_time_ratio* values were then used for the analysis under gFP.

Fig. 4.9 shows the fractions of schedulable tasksets under each algorithm. The lines with the labels gFP-H, gFP-M and gFP-L represent the results under gFP for the cache light, cache medium, and cache heavy scenarios, respectively.

Benefits of cache-aware scheduling: As Fig. 4.9 shows, both gFPca and nFPca perform much better than the cache-agnostic gFP under the cache medium and cache heavy configurations, and for most taskset utilizations under the cache light configuration. This is expected, because gFP does not protect concurrently running tasks from cache interference, which is more obvious for more cache-intensive workloads. On the contrary, both gFPca and nFPca mitigate such interference via cache partitioning and cache-aware scheduling, and thus they can significantly improve the schedulability of the tasksets.

Comparing the fractions of schedulable tasksets under gFP when the generated *hit_time_ratio* of tasks is in the cache light, cache medium and cache heavy scenarios, we observe that as the *hit_time_ratio* of tasks increases, the performance of gFP decreases. One reason for this trend is that tasks with a larger *hit_time_ratio* have more cache hit accesses when they execute alone, and hence they are more sensitive to the shared cache interference under gFP. Note that under gFP, we had to assume every cache hit access when it executes alone may be polluted by tasks running concurrently on other cores when it is scheduled with other tasks; therefore,

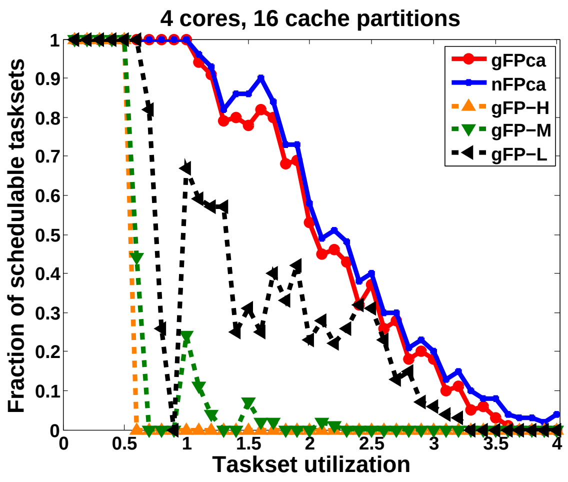


Figure 4.10: nFPca-favor.

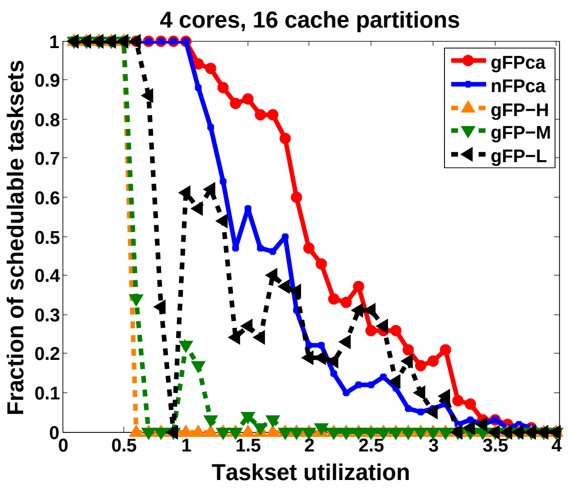


Figure 4.11: nFPca-oppose.

a higher number of cache hit accesses leads to a larger extrinsic cache overhead.

Benefits of gFPca over nFPca: We observe in Fig. 4.9 that gFPca outperforms nFPca in terms of the fraction of schedulable tasksets across all but one taskset utilizations. This is because gFPca avoids undesirable priority inversions and allows low-priority tasks to execute if high-priority tasks are unable to, and thus it utilizes the system's resources better.

The number of cache partitions and task priority relation: Because nFPca does not allow lower-priority tasks to execute when any higher-priority task is blocked by cache resource, it performs better on tasksets in which higher-priority tasks require a smaller number of cache partitions and worse on tasksets in which higher-priority tasks require a higher number of cache partitions. Recall that the maximum number of partitions a task can have is 12. To investigate the impact of the relation between the number of cache partitions and the task priority on the performance of the algorithms, we generated two kinds of tasksets: (1) the so-called nFPca-favor tasksets (i.e., tasksets that favor nFPca in comparison to gFPca), which have $|A_i| = \lfloor \frac{p_i - min\_period}{max\_period - min\_period} \cdot 12 \rfloor$ for each τ_i, and (2) the so-called nFPca-oppose tasksets, in which $|A_i| = \lfloor 12 - \frac{(p_i - min\_period) \cdot 12}{max\_period - min\_period} \rfloor$ for each τ_i. Other parameters of the tasks were generated in the same manner as above.

The fractions of schedulable tasksets are shown in Fig. 4.10 and 4.11. On the nFPca-favor tasksets, nFPca performs better than gFPca but only slightly, although the tasksets favor nFPca. We attributed this to the work-conserving nature of gFPca, which allows it to better utilize the system's resource. In contrast, the results in Fig. 4.11 show that gFPca can schedule many more tasksets than nFPca does on the nFPca-oppose tasksets. We also observe that the performance improvement that gFPca achieves over nFPca increases as the tasksets move from the nFPca-favor to the nFPca-oppose, i.e., as the number of cache partitions used by the higher-priority tasks increases.

4.7 Empirical evaluation

We used synthetic workloads to illustrate the applicability and benefits of gFPca based on our implementation platform (with four cores, 16 cache partitions). We focused on tasks that are sensitive to shared cache interferences (for which cache isolation is critical), and evaluated four algorithms: gFP (cache-agnostic global scheduling), pFP (partitioned scheduling with static core-level cache allocation), nFPca (cache-aware non-preemptive global scheduling with dynamic task-level cache allocation), and gFPca (cache-aware preemptive global scheduling with dynamic job-level cache allocation).

Workload generation. We first constructed two real-time programs in our implementation: the first randomly accesses every 32 bytes (the size of a cache line) in a 960KB array for 200 times, which was used for the highest-priority task; and the second randomly accesses every 32 bytes in a 192KB array for 2000 times, which was used for each lower-priority task. We separately measured the WCET of each program under the gFPca scheduler when it was allocated different numbers of cache partitions; the results are shown in Fig. 4.12.

We then constructed a reference taskset τ_{ref} with $n = 5$ tasks, with $\tau_1 \succ \tau_2 \succ \cdots \succ \tau_n$, where $\tau_1 = (p_1 = 5000, d_1 = 500)$ and $\tau_i = (p_i = 5000, d_i = 1550)$ for all $1 < i \leq n$. (We observed similar results when varying the number of tasks.)

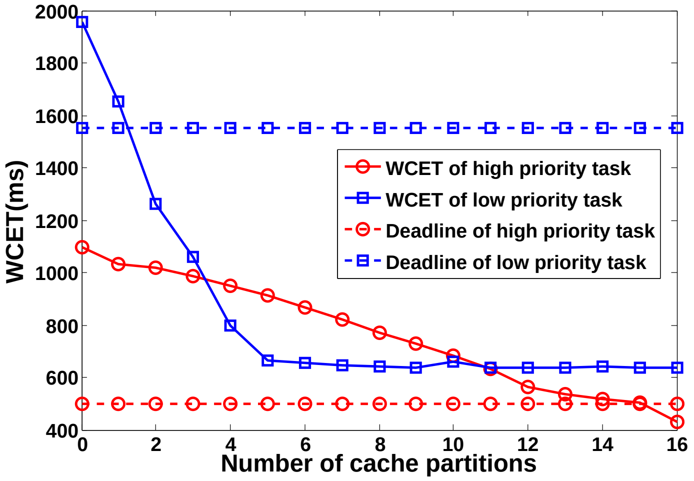


Figure 4.12: Measured WCET vs. Number of cache partitions.

Analysis of WCET and the number of cache partitions. Fig. 4.12 shows that the WCET of τ_1 is 430ms with 16 cache partitions and 501ms with 15 cache partitions. Since its deadline is 500ms, τ_1 needs all 16 cache partitions to meet its deadline. Each lower-priority task has a WCET of 800ms with 4 cache partitions, a WCET of 1059ms with 3 cache partitions and a WCET of 1958ms with 0 cache partition.

From the above analysis, we could feasibly assign the number of partitions of each task under gFPca and nFPca, i.e., $A_1 = 16$ and $A_i = 4$ $(i > 1)$. We set the WCET of each task to be an upper bound of the WCET measured under the assigned number of partitions[19], i.e., $e_1 = 500$ and $e_i = 1050$; this was used in our experiment investigating the impact of task density. (Note that, these WCETs are safe under gFP as well, since gFP allows every task to access the entire cache.)

[19]The upper bound is to account for potential sources of interference, such as TLB overhead, and variable actual program execution time.

Observation: No feasible static partitioning strategy exists. Under pFP, tasks are statically assigned to cores (e.g., as done in [36, 65]) and shared-cache isolation is achieved among tasks on different cores via static cache partitioning. However, this static approach cannot schedule the example workload. Specifically, since τ_1 requires all of 16 cache partitions to meet its deadline, if we allocate less than 16 partitions to its core, then it will miss its deadline. If we allocate all 16 cache partitions to τ_1's core, then either (i) some lower-priority task will have zero cache partition (if it is assigned to a different core) and will miss its deadline, or (ii) all tasks must be packed onto the same core as τ_1's, in which case the taskset is unschedulable (since the core utilization is more than 1). In other words, no partitioning strategy exists for the workload.

Experiment. The reference taskset illustrates the scenario where the high-priority task has a very high density (ratio of WCET to deadline) and thus is extremely sensitive to interference. To investigate the impact of task density on the performance of the algorithms, we varied the density of τ_1 from 1 to 0.1 by increasing its deadline (while keeping all the other parameters unchanged), which produced 10 tasksets. The number of cache partitions were assigned for gFPca and nFPca as above ($A_1 = 16$ and $A_i = 4$, with $i > 1$). Although our analysis shows that no feasible partitioning strategy exists for pFP, for validation we evenly distributed four low-priority tasks and 16 cache partitions to the four cores, and assigned τ_1 to any of the four cores. We ran each generated taskset for one minute under each of the four schedulers (gFPca, nFPca, gFP, pFP) schedulers, collected their scheduling traces, and derived the observed schedulability under each scheduler.

Table 4.2: Impact of task density on schedulability.

Density	≥ 0.8	0.7	0.6	0.5	0.4	0.3	0.2	0.1
gFPca	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
gFP	No	No	Yes	Yes	Yes	Yes	Yes	Yes
nFPca	No	No	No	No	No	No	No	Yes
pFP	No	No	No	No	No	No	No	No

Results. Table 4.2 shows the observed schedulability of each taskset under each scheduler. The results show that the gFPca scheduler performed best: it was able to schedule all tasksets. The gFP scheduler performed well when the high-priority task's density is low; however, as the task's deadline becomes tighter, its tolerance to cache interference from other tasks is decreased, and thus it began to miss its deadline. The results also show that the nFPca scheduler performed very poorly – it was able to schedule only one taskset; we attribute this to its poor utilization of cache and CPU resources due to its non-preemptive nature. As predicted in our analysis, the pFP scheduler could not schedule any tasksets.

4.8 Conclusion

We have presented the design, implementation and analysis of gFPca, a cache-aware global preemptive fixed-priority scheduling algorithm with dynamic cache allocation. Our implementation has reasonable run-time overhead, and our overhead analysis integrates several novel ideas that enable highly accurate analysis results. Our numerical evaluation, using overhead data from real measurements on our implementation, shows that gFP improves schedulability substantially compared to the cache-agnostic gFP, and it outperforms the existing cache-aware nFPca in most cases. Through our empirical evaluation, we illustrated the applicability and benefits of gFPca. For future work, we plan to enhance both gFPca and its implementation to improve their efficiency and performance.

Chapter 5

Dynamic shared cache management for virtualization systems by virtualizing Intel CAT

We have developed the shared-cache management and analysis solution for non-virtualization systems to allocate non-overlapped cache partitions to tasks; we now explore the solution for virtualization systems. The natural question one may ask is: can we simply apply the shared cache management solution developed for non-virtualized systems in Chapter 4 to the hypervisor for mitigating the shared-cache interference in virtualization systems? The shared cache management solution can be applied to the hypervisor to allocate the shared cache partitions to VMs, but tasks within the same VM still use the same cache area allocated to the VM and will still suffer from the shared-cache interference.

In order to mitigate the shared-cache interference, concurrently running tasks must be allocated with non-overlapped cache areas. Recall that resources are distributed hierarchically in virtualization systems: a type of hardware resource (say CPU resource) is first distributed to VMs by the hypervisor and then redistributed to tasks by OS in VMs. Observing that cache is not managed in virtualization systems, we need to establish a hierarchical cache allocation framework in order to allocate non-overlapped cache areas for tasks in virtualization systems.

Recent work has developed a hierarchical cache allocation framework for allocating non-overlapped cache areas for tasks in virtualization systems using page coloring (e.g., [75, 37]); however, it is restricted to static cache partitioning, where a fixed set of partitions is statically assigned to each task at initialization. While this approach is simple and easy to implement, it can substantially under-utilize the cache and CPU resources, and it does not work well for systems where the tasks' timing constraints and CPU/cache demands vary dynamically at run time, such as in multi-mode systems (as we shall illustrate in Section 5.4.4).

To bridge this gap, we present a new approach to cache management of real-time virtualization systems that can deliver strong (shared) cache isolation at both VM and task levels, and that can be configured for both static and dynamic allocations. Unlike existing work, which is software-based, our approach takes advantage of the Cache Allocation Technology (CAT), a hardware feature recently added in Intel multicore hardware for achieving core-level cache partitioning; therefore, it is much more efficient than software-based techniques. Since CAT only provides core-level cache isolation, we introduce vCAT, a novel design for CAT virtualization that can be used to achieve hypervisor- and VM-level cache allocations. Our approach to virtualizing cache partitions is analogous to memory virtualization: as the hardware provides a number of (indistinguishable) physical partitions, we can expose some number of "virtual partitions" to each VM and then transparently map them to physical partitions in the hypervisor; each VM can then allocate its virtual partitions to its tasks statically or dynamically at runtime.

5.1 Experimental study of Intel Cache Allocation Technology (CAT)

The Intel's CAT is a new hardware feature that allows the OS or hypervisor to control the allocation of the shared last-level cache to the physical cores. In this section, we present a study of its behavior in the current hardware, and highlight its implications on the design of CAT virtualization. Our study was performed using the Intel MSR tool [3] on an Intel Xeon E5-2618L v3 processor, which has a 20MB shared cache.

5.1.1 Background on CAT

The CAT divides the shared cache into N non-overlapped equal-size cache partitions; for instance, $N = 20$ for our experimental platform. A set of such cache partitions (specified as an N-bit mask) can be allocated to a CPU (core) by programming two model-specific registers: (1) The Class of Service (COS) register, which has an N-bit Capacity Bitmask (CBM) field to specify a particular cache partition set, and (2) the CPU's IA32_PQR_ASSOC (PQR) register, which has a COS field for linking a particular COS to the CPU; when this field is set to the ID of a COS register, CAT enforces that all cache allocation requests from the CPU will only happen in the cache partitions specified by the CBM of that COS register. For example, to allocate partitions 0 to 3 to a CPU, we set 1's for the bits 0 to 3 (and zeroing the remaining) of the CBM field of the associated COS register.

We conducted a series of experiments to validate the operation of the Intel's CAT. Our experiments confirmed that the Intel's CAT specification is correct in stating the following constraints and in the way CAT works as advertised: (1) The current CAT implementations only support an allocation with at least two partitions; (2) the number of cache partitions per CPU should not exceed the number of available partitions (which varies across processors); and (3) the partition set of a CPU can only be made of contiguous cache partitions.

5.1.2 Effects of cache partition configuration on WCET

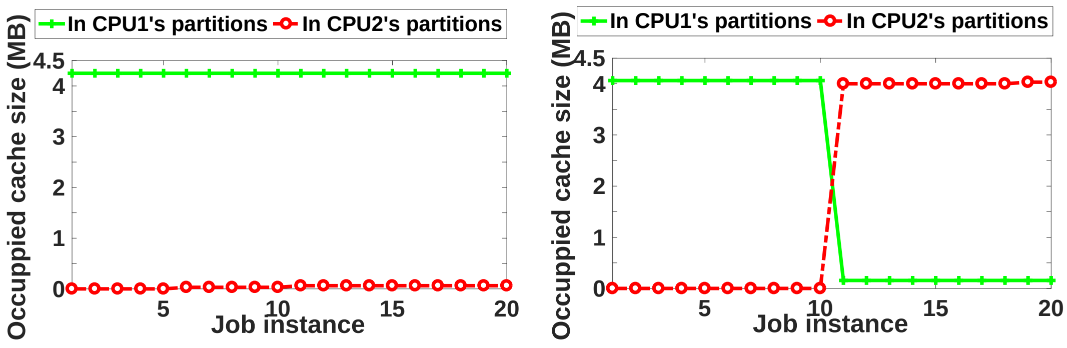
To validate that any combination of contiguous partitions with the same number of partitions has the same effect on the task's worst-case execution time (WCET), we constructed a task that sequentially accesses every 64 bytes in a 1MB array for 100 times, and executed the task alone on a CPU. We enumerated all possible combinations of two contiguous partitions; for each combination, we allocated the corresponding partitions to the CPU, and measured the WCET of the task across 25 runs. The results show the same WCET for the task with the same array across all combinations.

Finding 5.1. *Any set of contiguous partitions with the same number of partitions have the same effect on WCET.*

5.1.3 Cache lookup control

Under dynamic cache allocations, the partitions allocated to a task can change over time. When this happens, the task should only be allowed to access the cache lines in the newly assigned partitions and not the old ones. The CAT ensures that the task's new cache allocations (which happen in cases of cache misses) will happen in the new partitions, but the SDM does not specify the CAT behavior for cache lookup requests (which happen in cases of cache hits), which suggests that a task may still be able to read from the old partitions. If so, the task can interfere with another task that is currently using the old partitions. To examine whether CAT controls the cache lookup requests, we performed the following experiment using the Intel MSR tool on Linux 3.10.31 on our implementation platform, which has 20 cache partitions of size 1MB each.

Experiment. We reserved cache partitions 0–7 (CBM bitmask 0×000FF) to CPU1



(a) Without cache flush (b) With cache flush

Figure 5.1: No cache lookup control in CAT.

and partitions 8–15 (CBM bitmask 0×0FF00) to CPU2. We flushed the entire cache initially, and mitigated potential interference to CPU1 and CPU2 by moving all system services to the remaining cores and assigning to them the remaining partitions (partitions 16–19). We created a periodic task that sequentially accesses a 4MB array. We executed the first 10 jobs of the task on CPU1; upon completion, we migrated it to CPU2 and continued its execution until completing the next 10 jobs. Using the Intel Cache Monitoring Technology [1], we measured the occupied cache size in each CPU's cache partition set when each job finished.

Results. As shown in Fig. 5.1(a), the size of the occupied cache in CPU1's partitions is always approximately the same as the array size (4MB), whereas the size of the occupied cache in CPU2's partitions is close to zero, even when the task executed on CPU2. This can be explained as follows. When the first job accessed the array, it experienced compulsory cache misses and thus was allocated cache lines in CPU1's partitions (as enforced by CAT). However, since the entire task's array (4MB) fits within CPU1's partitions (8MB), the subsequent jobs would experience cache hits and access the array directly from these partitions. Our experimental results show that this happened even when the task was already migrated to CPU2 (and should no longer use CPU1's partitions), which shows that CAT does *not* control the cache lookup.

Finding 5.2. *The CAT does not control cache lookup requests, and thus does not guarantee that cache accesses happen* only *in the currently assigned partitions.*

Challenge. Due to the lack of cache lookup control, when a partition that was owned by a task A is re-assigned to a task B, the previous cached items of A in this partition are simply looked up as before. As a result, if B (the current owner) does not happen to evict these cached items of A from the partition, then A will continue to reference its cached items in B's partition.

To ensure that tasks have complete control of their partitions, in certain situations it is necessary to flush the content of a task in its old partitions when the task's partitions are changed. Our CAT virtualization uses this approach for real-time tasks, thus providing strong isolation among them. Our design also supports *shared* partitions (disjoint from those of real-time tasks) for best-effort tasks, where tasks can share the same set of partitions and no flushing is necessary.

Validation. To validate the effect of flushing, we performed the same experiment as above, except that we flushed the cache immediately after migrating the task from CPU1 to CPU2. As shown in Fig. 5.1(b), the size of the occupied cache in CPU1's partitions is dropped to nearly zero as soon as the task migrates to CPU2, whereas the size of occupied cache in CPU2's partitions increased to 4MB. This confirms that, with flushing, the task only accesses its newly assigned partitions.

5.2 CAT virtualization design

In this section, we describe the design of vCAT, as well as the necessary changes to the guest kernel and the hypervisor.

5.2.1 Overview and roadmap

At a high level, our approach to virtualizing cache partitions is similar to classical virtual memory; however, there are also several important differences. We begin

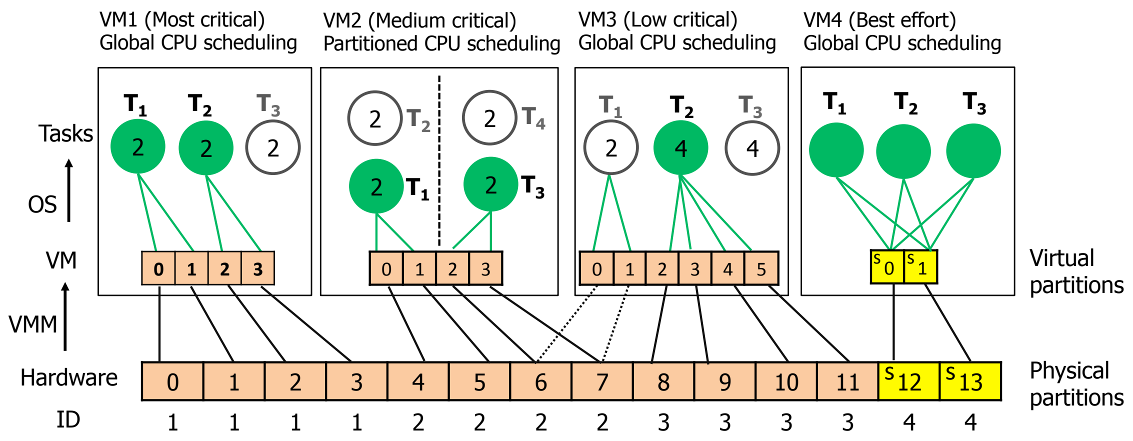


Figure 5.2: Dynamic cache management with CAT virtualization. Tasks in green (white) are currently running (waiting). Partitions in orange (yellow) are isolated (shared) partitions.

with the similarities: the hardware provides a fixed number of physical cache partitions that can be allocated to tasks, just like it provides a fixed number of physical memory pages, and—just like physical memory pages—the individual partitions are indistinguishable from each other, so it should not matter to a task which specific partitions it is using. Thus, we can simply expose some number of "virtual partitions" to each VM (Section 5.2.2) and then transparently map them to physical partitions in the hypervisor, using a data structure that somewhat resembles a page table (Section 5.2.3), and each VM can then allocate its virtual partitions to tasks dynamically at run time (Section 5.2.4). Fig. 5.2 shows an example of a system with CAT virtualization.

However, there are also two key differences. First, although cache partitions can be "preempted" just like physical pages the hypervisor needs not – and, indeed, cannot – save the contents of the partition it is preempting. Instead, it can rely on the tasks to repopulate the partitions they are being assigned. Second, the CAT specification contains a requirement that allocations are contiguous. This needs to be taken into account when allocating partitions, and it requires a procedure for handling partition fragmentation (Section 5.2.5).

The technical approach is similar to virtual memory: allocations are enforced at a per-core level, using the COS registers, just like each core has a separate page-directory base register (CR3), and the hypervisor is able to request traps on accesses to these registers to perform a "partition context switch" (Section 5.2.6). When the hypervisor or guest kernel changes the partition allocations to the VMs or tasks, it may need to flush the partitions if necessary (Section 5.2.7).

5.2.2 API changes

In order to implement a virtual CAT, we need to make four changes to the API: (1) the VMM must be able to tell the guest kernel how many partitions are available; (2) tasks must be able to request partitions from the guest kernel; (3) the guest must have a way to report the allocation, as well as any changes, to the VMM; and (4) the operator must have a way to control how partitions are divided up between the various VMs and to set/modify the mapping from virtual to physical partitions for each VM. We describe each in turn.

Since the hardware already contains a mechanism for reporting the number of available partitions (via the `cpuid` instruction), we can simply repurpose this mechanism to achieve the first goal: the hypervisor can trap on the `cpuid` instruction – which Xen already does – and change the relevant value. We do not see a good reason for reporting more partitions than are physically available, but there may be good reasons to report fewer, e.g., if the operator has divided up the available partitions between multiple VMs. If the guest kernel were to allocate more virtual partitions than the hypervisor is willing to give it, this would lead to many expensive preemptions, so it may be preferable to report the smaller number right away.

The current Linux API does not contain a system call for requesting cache partitions, so we added a call of our own that simply takes a requested number of partitions as its argument. Taking a COS-style bitmask seemed unnecessary because a task should not need to know which specific partitions it is being given – much like a task normally should not need to know which physical memory pages it is using.

We achieve the third goal by providing virtual COS registers. Thus, the guest kernel can use the same procedure to allocate partitions, whether it is running in a VM or on bare hardware. A hypercall could be added if the guest needs to communicate richer information to the VMM, e.g., to request a temporary increase in the number of partitions it can use.

To achieve the fourth goal, we added several hypercalls that can influence the partition-to-VM allocation (which we describe next), and we provided a small command-line utility for the operator to use.

5.2.3 Hypervisor-level partition allocation

When allocating partitions to VMs, the hypervisor can take three basic approaches: first, it can divide up the available physical partitions, which guarantees each VM that its partitions will not need to be preempted; second, it can allow the partitions to become oversubscribed, which can lead to preemptions; or, third, it can allow partitions to be transparently shared between VMs. The first two options are similar to physical memory, whereas the third is unique to the cache.

The first approach is clearly preferable for tasks and VMs with strict real-time requirements, since it achieves very good isolation; however, given the very small number of partitions that are available on current CPUs, it seems practical for only the most critical tasks and VMs (e.g., VM1 in Fig. 5.2). We expect the second approach to be the default choice (e.g., VM2 and VM3 in Fig. 5.2). The third approach could be used for best-effort tasks: for instance, the operator could reserve 15 of the 20 partitions for hard real-time tasks and share the remaining five among all the non-real-time tasks. This would prevent the latter from interfering with the former. In Fig. 5.2, this approach was used for tasks in VM4.

Internally, the hypervisor requires only two data structures to implement these policies: (1) for each VM i, a mapping from virtual partition numbers v to physical partition numbers $P_i(v)$, and (2) a flag for each physical partition to indicate whether the partition is shared. For example, the system in Fig. 5.2 set the shared flags (denoted as S in the figure) for partitions 12 and 13. In Section 5.2.6, we describe how these data structures are used during a partition context switch.

To meet the CAT specification, we enforce that the number of partitions allocated to each VM i must be at least two, and the partition numbers $P_i(v)$ must be contiguous. In our current prototype, these data structures must be configured manually by the operator. (The operator can use the provided utility to modify these data structures at run time, e.g., when a new VM is created or an existing VM is destroyed.) However, we note that there is a rich literature on working-set estimation [79, 28] and on memory management for real-time tasks [34, 50], which can be adapted for use with cache partitions.

5.2.4 Guest-level partition allocation

Just like the hypervisor, the guest kernel must allocate the available partitions to its tasks, based on the requests they have made. However, unlike the partition-to-VM allocation which does not change frequently, the partition-to-task allocation is done dynamically as tasks are scheduled. In our prototype, we simply allocate the partitions to real-time tasks based on either a first-come-first-served basis or criticality, and we share any unallocated partitions among all the best effort tasks. Since allocating zero partitions would effectively disable the cache, which would lead to an enormous slowdown, the kernel reserves a small number of partitions for these tasks and does not allow these partitions to be reserved by the real-time tasks. The kernel always allocates at least two, and always contiguous, virtual partitions to a task.

5.2.5 Partition defragmentation

Although future hardware may no longer need the contiguous partition allocations, current hardware does. This raises the possibility of "partition fragmentation": it could be that there are k total partitions available but not with contiguous partition numbers, which would prevent a request for k partitions from being satisfied at that point. This problem can appear both in the hypervisor and in the guest kernel.

However, there is an easy way to fix this problem when it appears: the kernel or hypervisor can "defragment" the partitions by preempting some allocations and by replacing them with others, so that the unallocated partition numbers are again contiguous. The caveat is that this can cause a temporary loss of performance as the tasks are repopulating their preempted partitions, which can lead to deadline misses. This can be alleviated somewhat by moving the partitions of less critical tasks first, or by carefully configuring the virtual-to-physical mappings. In our prototype, we disable automatic defragmentation in the highly critical VMs and at the hypervisor (since reallocating partitions to VMs requires flushing the addresses of some VMs); the operator can trigger defragmentation manually when she considers it to be safe.

5.2.6 Partition context switch

In order to enforce the partition allocation at the VM level, the hypervisor must update the COS registers whenever it performs a partition context switch. To this end, the hypervisor maintains, for each physical partition n, the ID $I(n)$ of the VM that is currently using that partition.

A partition context switch from a VCPU of VM i to a VCPU vcpu_j of VM j is done as follows: the hypervisor first iterates over all of the target's virtual partition numbers $n = 0 \ldots k$; if the nth bit of vcpu_j's virtual COS is set, the hypervisor looks up the corresponding physical partition number $P_j(n)$ and checks whether (1) $I(P_j(n)) \neq j$, and (2) the partition $P_j(n)$ is not shared. If the preemption-based strategy is set and VM j has higher criticality than every VM $I(P_j(n))$ for which both conditions (1) and (2) hold, then the hypervisor preempts the VCPU that is currently using $P_j(n)$ and clearing all bits of the COS register of the core on which that VCPU is running. Next, the hypervisor updates the physical COS register of the core on which $\texttt{vcpu}_j$ is scheduled (by setting only $P_j(n)$, $n = 0 \ldots k$ and clears all the other bits), setting the ID $I(P_j(n)) = j$ for all $n = 0 \ldots k$. In addition, if it did preempt a VCPU, it also then invokes a rescheduling event to the scheduler. Notice that a preemption happens only in cases where a partition is assigned to more than one VM, but is not shared. In Fig. 5.2, physical partitions 6 and 7 are oversubscribed by both VM2 and VM3; since VM2 has higher criticality than VM3, its VCPUs can preempt VM3's VCPUs. Here, the hypervisor preempts the VCPU currently executing T_1 of VM3, and switches the partitions' owner to the VCPU on which T_3 of VM2 will execute.

If the guest kernel is not CAT-aware, it will not modify its virtual COS from the default value (all partitions active), and the above process is sufficient. If the guest does modify the virtual COS (e.g., during a guest-level partition context switch), the kernel must intercept these accesses and modify the physical COS register and the ID of the physical partitions. Fortunately, the COS registers are machine-specific registers; they are updated with the `wrmsr` instruction, which is privileged and causes a guest exit when invoked. When the hypervisor intercepts an access, the procedure is analogous to an inter-VM partition context switch. Notice that the hypervisor cannot know whether the guest kernel is reassigning a partition from one task to another; hence, the guest must keep ownership information for the virtual partitions similar to the hypervisor's $I(n)$.

5.2.7 Flushing

At first glance, it may seem that, when a cache partition is reassigned from one VM or task to another, updating the COS register is all that is required. However, as discussed in Section 5.1.3, if the new owner does not happen to evict all cached content of the previous owner from the partition, the previous owner will continue to reference its cached items and prevent the new owner from gaining full control over the partition. To reliably avoid this, it is necessary to flush the previous owner's content from the cache when it is assigned a new set of partitions that is *not* a superset of its previous partitions, if the previously-assigned partitions are not shared.[20]

For this purpose, we maintain for each task τ_i its currently assigned set of virtual partition numbers S_i. A flushing is initiated when τ_i is scheduled to run and if it is assigned a new set of partition numbers S_i' such that $S_i' \not\supseteq S_i$ and there exists $v \in S_i - (S_i \cap S_i')$ where the shared flag of v is 0. Consider VM1 in Fig. 5.2, for instance, which has two VCPUs. Suppose T_3 was previously assigned partitions $S_3 = \{0, 1\}$, but it is preempted by T_1. Suppose later, T_2 finishes, then the kernel will assign partitions $S_3' = \{2, 3\} \not\supseteq S_3$ to T_3. Since T_3 may still access its old partitions 0 and 1 via cache hits, which are now owned by T_1, we need to flush the content of T_3 in these partitions.

Notice that partitions are only re-allocated at the hypervisor level when a mapping of virtual-to-physical partition numbers changes; therefore, flushing at the hypervisor level happens only very infrequently (i.e., during defragmentation or triggered by the operator when a VM joins or leaves the system, or when some VMs request more partitions).

Ideally, we would like to simply flush the specific partition whose ownership is changing (e.g., partitions 0 and 1 in the above example). However, the current CAT does not provide a way to do this, so our only option is to flush the cache contents of the *entire* VM or task that is being replaced. The Intel CPUs offer two ways to do this: the `clflush` instruction, which flushes the cache line that contains a specific linear address, and the `wbinvd` instruction, which writes back any modified data in the cache and then invalidates the *entire* shared cache. (A third option, the `invd`

[20]If the previous owner's new partitions include all of its old partitions, it experiences cache hits only in its own (old/new) partitions, and thus cannot access the partitions currently assigned to another running task.

instruction, would simply discard modified cache lines, so it is not an option here.) When the `clflush` instruction is used to flush the cache content of a task, it is issued on all valid linear addresses (not the entire virtual address space) of the task, with a step of a cache line (i.e., 64B). The linear addresses of a task can be found in the task's control block.

Neither option is strictly better than the other: `clflush` can avoid side effects on other tasks by flushing specific content, and it is potentially faster than `wbinvd` if the previous owner's working set is small; however, it can also be slower if there are a lot of addresses to be flushed. For simplicity, our implementation uses `wbinvd` for the hypervisor-level flushing. At the guest level, it uses a simple heuristic to choose the option to use: if the previous owner's working set is smaller than a threshold Thresh, it uses `clflush`, otherwise `wbinvd`. In Section 5.3, we will discuss in more detail how this threshold can be chosen.

5.3 Implementation

Next, we describe a prototype of vCAT that we have built for our experiments. Our prototype extends the Xen hypervisor (version 4.6) and LITMUS^{RT} 2015.1 guest kernel, running on top of the Intel Xeon CPU E5-2618L v3 processor.

5.3.1 Extended data structures and API

We extended the task structure to include a field for specifying the number of partitions a task requests, a `execOnFewer` flag that is set when the task can execute even if it receives fewer (non-zero) partitions than the requested number, and a set of currently allocated partition numbers. By default, a real-time task can only execute if it is allocated partitions, and concurrently running real-time tasks do not share partitions to ensure isolation. We added a system call that allows a task (or the operator) to request a different number of partitions from the guest kernel at run time.

A VCPU's virtual COS register in a VM has the same format (bit mask) and operation as that of a physical COS register, except that it specifies the virtual partitions allocated to the VCPU's currently running task. Like physical partitions, each virtual partition has a shared flag that is set if the partition can be shared among concurrent tasks; this is useful for allocating a shared set of virtual partitions to concurrent tasks (e.g., the standard global EDF scheduling without cache allocation within a VM).

5.3.2 Partition allocation and partition context switch

Hypervisor-level allocation: We implemented a command-line utility for the operator to configure the virtual-to-physical mappings P_i and the shared flags of the physical/virtual partitions.[21] For simplicity, we require the operator to configure these data structures when a new VM is created; she can also modify them at run time if desired. To fully utilize the cache, our prototype allows the physical partitions oversubscribed by VMs (and performs a VM partition context switch, if needed).

We also implemented a hypercall that allows a guest to release some unused partitions or request more partitions at run time. In our prototype, the hypervisor simply puts the released partitions in an unused pool and later allocates them to any VM that requests additional partitions. Internally, whenever there is a change in the virtual-to-physical mappings, the hypervisor invokes the mapping procedure, which updates the mapping P_i for each (relevant) VM and the physical COS registers, as well as performs a VM partition context switch and/or flushes the cache, if necessary (c.f. Sections 5.2.6 and 5.2.7, respectively).

Guest-level allocation: The kernel allocates the VM's available virtual partitions to tasks based on their requests. It reserves a small (configurable) number of par-

[21] Determining the best number of partitions to reserve for each VM is an interesting but orthogonal research question; one promising direction here is to extend the cache-aware compositional analysis in [70].

titions to be shared among all best-effort tasks, and uses all the rest for real-time tasks. We implemented two strategies for allocation: the first allocates partitions to tasks in a first-come-first-served basis, as they are scheduled on the VCPUs; and the second gives priority to a more critical task, i.e., allows it to preempt lower-criticality tasks to acquire sufficient partitions, similar to the approach used in [70]. In both cases, if the task's `execOnFewer` flag is set, and if the VM has some but fewer than the requested number, the kernel simply allocates the available partitions to the task (and let it execute) to maximize core utilization and to minimize preemption overhead.

Whenever the kernel of VM i (re-)allocates partitions to a task, it would update the relevant data structures (the task's assigned partition set, the ID $I(v)$ of each allocated virtual partition v the task is assigned), and flush the task's content in the old partition sets if required (c.f. Section 5.2.7). If necessary, it would also modify COS registers of its VCPU and the VCPUs of the preempted tasks (if any) by executing the `wrmsr` instruction. We extended the hypervisor to trap on this instruction and modify the physical COS registers of these VCPUs' cores (based on the mapping P_i). Notice that, when the physical partitions are oversubscribed, it is possible that VM i might set a virtual COS bit representing a virtual partition that is mapped to a physical partition currently used by another VM j. If the partition is not shared, the hypervisor simply returns failure to VM i by default, thus allocating the oversubscribed partitions in a first-come-first-served basis. However, we also implemented a preemption-based mechanism, where the hypervisor preempts VM j and reassigns the partition to VM i, if VM i has higher priority than VM j, according to some algorithm. Our prototype uses static priority when this choice is configured, but it can easily be extended to include other algorithms for deciding the priority. When a preemption occurs, the hypervisor will perform a VM-level partition context switch (as described in Section 5.2.6).

5.3.3 Flushing heuristics

For simplicity, our prototype always uses the `wbinvd` instruction for the hypervisor-level flushing, since this operation often involves flushing the working sets of several tasks in one or more VMs and thus `clflush` can take a long time. At the guest level, we implemented a simple heuristics that uses `clflush` if the working set size (WSS) of the task is smaller than a threshold Thresh, and uses `wbinvd` otherwise. Intuitively, Thresh is the smallest WSS for which the overhead when using `clflush` is larger than that when using `wbinvd`. (If the strong isolation requirement flag is set, we always use `clflush` at the guest level. Otherwise, we use the heuristic.)

At a high level, the overhead of each approach includes (1) the latency of the cache flush operations, and (2) the extra latency when tasks access the content that was but is no longer in the cache because of flushing. For `clflush`, our empirical evaluation shows that the overhead of cache flush operation is approximately linear to the task's WSS, and the cache reload overhead is linear to the WSS but converges to $\mathsf{D}_{\mathsf{reloadLLC}}$ (the overhead of reloading the entire cache) once the WSS exceeds the cache size. Thus, the estimated overhead is

$$\begin{aligned}\mathsf{Overhead(clflush)} &= \mathsf{D}_{\mathsf{clflush}} + \mathsf{D}_{\mathsf{reloadLLC}} \\ &\approx \mathsf{k}_1 \cdot \mathsf{WSS} + \min\{\mathsf{k}_2 \cdot \mathsf{WSS},\ \mathsf{D}_{\mathsf{loadLLC}}\}.\end{aligned}$$

where $\mathsf{k}_1 \approx 1.58$ (ms/MB), $\mathsf{k}_2 = 1.65$ (ms/MB), and $\mathsf{D}_{\mathsf{loadLLC}} = 26.63$ (ms) on our platform.

For `wbinvd`, the cache flush operation overhead depends on the status of the cache when the instruction is invoked, and our evaluation shows that it is upper bounded by $\mathsf{D}_{\mathsf{wbFflush}} = 0.7$ (ms). Since `wbinvd` flushes the entire cache, and without knowledge of which data need to be reloaded, we assume the worst-case scenario where we need to reload the entire cache; thus, the overhead is at most $\mathsf{D}_{\mathsf{loadLLC}}$. In other words, the overhead when using `wbinvd` is approximately $\mathsf{Overhead(wbinvd)} \approx \mathsf{D}_{\mathsf{wbFlush}} + \mathsf{D}_{\mathsf{loadLLC}}$.

Based on the above analysis, we can derive Thresh as the smallest WSS such that $\mathsf{Overhead(clflush)} > \mathsf{Overhead(wbinvd)}$, i.e.,

$$\mathsf{Thresh} \approx \max\{(\mathsf{D}_{\mathsf{wbFlush}} + \mathsf{D}_{\mathsf{loadLLC}})/(\mathsf{k1} + \mathsf{k2}),\ \mathsf{D}_{\mathsf{wbFlush}}/\mathsf{k1}\}.$$

On our experimental platform, $\mathsf{Thresh} \approx 8.46$ (MB).

5.3.4 Overhead introduced by CAT virtualization

We ran a series of micro benchmarks to evaluate the extra overhead introduced by CAT virtualization based on our prototype. The results show that our design introduces only minimal overhead in terms of partition context switch and partition allocations (within a few microseconds), and the overhead caused by flushing and defragmentation in general depends on the tasks' WSS but is always less than 27.35ms on our experimental platform (which has a 20MB shared cache).

We consider five different (but intertwined) types of overhead that our design introduces: cache flush, cache reload, context switch, partition allocations, and defragmentation. We describe each in turn.

Cache flush operation latency. Recall that Intel CPUs offer two ways for cache flushing: the `clflush` instruction, which flushes the cache line that contains a specific linear address; and the `wbinvd` instruction, which writes back any modified data in the cache and then invalidates the entire shared cache. We measured the latency for each operation, as follows.

Latency of the `clfush` approach. We created a synthetic task that sequentially accessed (i.e., either read or write) an array. We varied the task's array size from 1MB to 40MB with a step of 1MB. The task first accesses its array, and then the system flushes the task out of the cache by using the `clflush` instruction. We achieved this by enumerating all linear addresses of the task, and invoked the `clflush` instruction on all these addresses. We measured the latency of the cache flush operation when

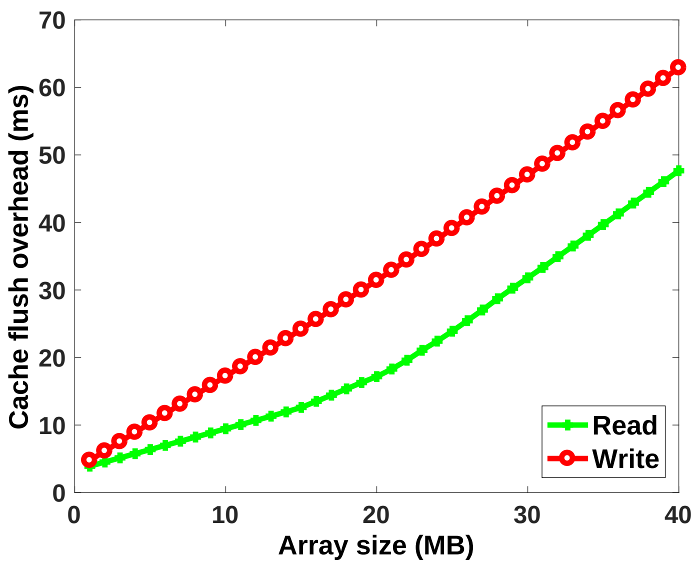


Figure 5.3: Cache flush overhead.

the task either reads from or writes to its array. The result is shown in Fig. 5.3.

The measured result shows that the latency of the cache flush operation with the clflush approach, denoted as $\mathsf{D_{clflush}}$, is proportional to the task's working set size (WSS), i.e.,

$$\mathsf{D_{clflush}} = \mathsf{k_1} \cdot \mathsf{WSS}$$

where $\mathsf{k_1} = \mathsf{D_{clflush}}/\mathsf{WSS} \le \mathsf{D_{clflush}}/\mathsf{array\_size\_i} \le 62.89\mathsf{ms}/40\mathsf{MB} \le 1.58\mathsf{ms}/\mathsf{MB}$.

Latency of the `wbinvd` approach. We repeated the same experiment as above, but we used wbinvd (instead of clflush) to flush the task. The results show that the latency of the cache flush operation with the wbinvd approach, denoted as $\mathsf{D_{wbFflush}}$, is not affected by the task's WSS, and $\mathsf{D_{wbFflush}} \le 0.7ms$.

Cache reload latency. The cache reload latency is determined by the size of the content that was but is no longer in the cache because of flushing. The size of the content to reload is upper bounded by the shared cache size.

We created a synthetic task that uses a *linked list* to access (i.e., read or write) every 64 bytes in an array for three times. The task does the following steps se-

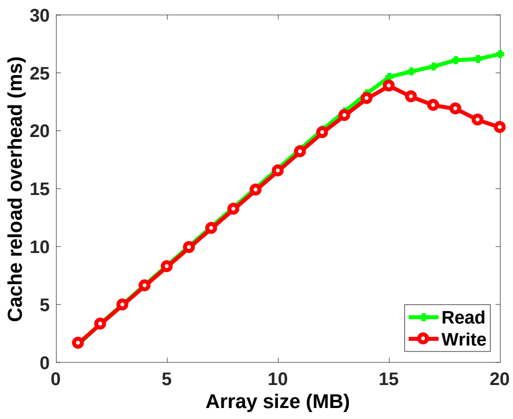


Figure 5.4: Cache reload overhead without MLP (Access via linked list).

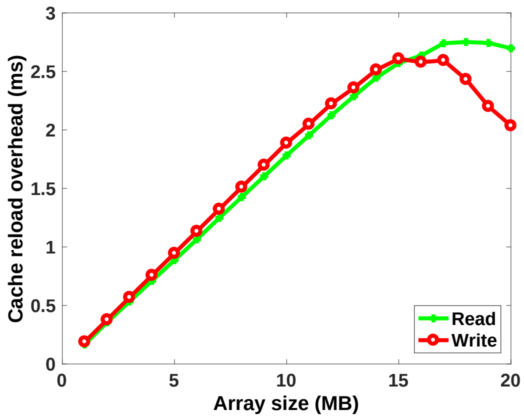


Figure 5.5: Cache reload overhead with MLP (Access via array index)

quentially: (1) access the entire array for two consecutive times; (2) flush the task's content out of the cache; and (3) access the entire array for the third time. We measured the latency of accessing the array at the second time (i.e., when the array is already cached) and at the third time (i.e., when the array is not cached). Then, the time difference between the two measured latencies is the cache reload latency because of flushing. We varied the size of the array from 1MB to 20MB (i.e., the shared cache size) with a step of 1MB, and we measured the cache reload latency under each array size. The result is shown in the Fig. 5.4.

We also repeated the same experiment but changed the synthetic task to use *array index* to iterate the same array. The result is illustrated in Fig. 5.5.

We observed that the cache reload latency is proportional to the size of the content to reload. When a task is flushed, the cache reload latency for the task, denoted as $\mathsf{D}_{\mathsf{reloadLLC}}$, is upper bounded by

$$\mathsf{D}_{\mathsf{reloadLLC}} \leq \min\{k_2 \cdot WSS,\ D_{loadLLC}\}$$

Where $k_2 = \mathsf{D}_{\mathsf{reloadLLC}}/\mathsf{WSS} \leq \mathsf{D}_{\mathsf{reloadLLC}}/\mathsf{array\_size\_i} \leq \mathsf{24.64ms/15MB} \leq \mathsf{1.65ms/MB}$, and $\mathsf{D}_{\mathsf{loadLLC}} = \mathsf{26.63ms}$ is the maximum latency of reloading the entire LLC.

Table 5.1: Partition context switch overhead (μs).

	Taskset size: 50			Taskset size: 450		
	Vanilla	vCAT	Overhead	Vanilla	vCAT	Overhead
VM	5.49	5.74	0.25	5.02	5.17	0.15
VMM	0.8	0.81	0.01	0.7	0.73	0.03

We also observe, by comparing Fig. 5.4 and Fig. 5.5, that the cache reload overhead drops by 89.67% (from 26.63 ms to 2.75 ms) when the task changed the way of accessing its array from *linked list* to *array index*. This is because changing from *linked list* to *array index* for the task eliminates the data dependence in accessing each element of the task's array. Therefore, the task can benefit from the Memory Level Parallelism (MLP) in accessing or reloading its array.

Partition context switch overhead. We measured the partition context switch overhead both in the vCAT and in the vanilla LITMUS^{RT}/Xen system. The overhead difference is the extra context switch overhead the vCAT introduces in managing the partition context.

We boot 4 guests, each with 4 full-capacity VCPUs. We randomly generated periodic task sets whose size is 50 or 450 tasks, for each domain. We generated 10 task sets per task set size. Under each environment, we used the feather-trace tool [25] to measure the context switch overhead in a VM (running LITMUS^{RT}), as in earlier LITMUS^{RT}-based studies [37] [17]. We used the Xentrace tool to measure the context switch overhead in the VMM (i.e., Xen), as in earlier RT-Xen study [69]. The result is shown in Table 5.1.

We observe the extra context switch overhead incurred by our vCAT prototype is very small (upper bounded by $0.25\mu s$).

For completeness, we also measured other types of scheduling-related overhead in VM and VMM. Table 5.2 shows the task release overhead (REL) and the scheduling overhead (SCH1) within a VM, as well as the scheduling overhead (SCH2) in the VMM. The results show that vCAT incurs negligible extra overhead for all these

Table 5.2: Average scheduling-related overhead (μs).

	Taskset size: 50			Taskset size: 450		
	Vanilla	vCAT	Overhead	Vanilla	vCAT	Overhead
REL	1.77	1.96	0.19	1.13	1.31	0.18
SCH1	2.74	2.80	0.06	3.23	3.33	0.10
SCH2	0.37	0.39	0.02	0.32	0.33	0.01

three types.

Partition allocation and deallocation overhead. In order to measure the partition allocation and deallocation overhead, we extended the feather-trace tool by adding these two overhead events in LITMUS^{RT}.

We booted one VM with 4 full-capacity VCPUs pinned to 4 cores. We randomly generated periodic task sets whose size is 50 or 450 tasks. We generated 10 task sets per task set size. We measured the average and the maximum latency the vCAT takes to allocate or deallocate cache partitions for tasks. The result is shown in Table. 5.3.

We observed that the partition allocation and deallocation overheads are negligible. The cache allocation and deallocation overheads are respectively upper bounded by $550ns$ and $318ns$.

Table 5.3: Partition allocation and deallocation overhead (ns).

	Taskset size: 50		Taskset size: 450	
	Average	Maximum	Average	Maximum
Allocation	175	550	178	463
Deallocation	102	318	96	301

Defragmentation overhead. When the defragmentation procedure happens, it involves two operations: (1) Reallocating partitions for tasks, which involves deallocating old partitions and then allocating new partitions for tasks; this overhead is upper bounded by the sum of the maximum allocation and deallocation overheads, i.e., $550ns + 318ns = 868ns$. (2) Flushing the entire cache, which has an overhead of at most $\mathsf{D}_{\mathsf{wbFflush}} \leq 0.7ms$.

The defragmentation overhead is the sum of the overhead of reallocating partitions for tasks and the overhead of flushing the entire cache. Therefore, it is upper bounded by $868ns + 0.7ms \leq 0.701ms$.

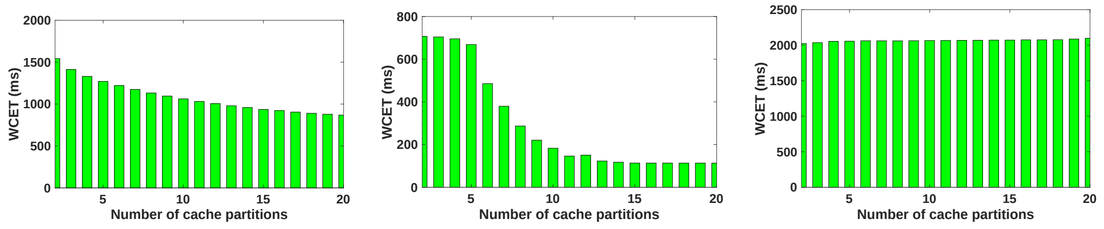
5.4 Performance evaluation

To illustrate the applicability and benefits of CAT virtualization, we conducted an extensive set of experiments on our prototype using the PARSEC benchmarks [21] and synthetic workloads. Our goal is to evaluate (i) how well task-level cache isolation using CAT virtualization can protect a task's WCET from other concurrently running tasks, and (ii) how much CAT virtualization can improve the system's real-time performance in two use cases (static and dynamic cache allocations).

5.4.1 Experimental setup

Hardware. Our prototype ran on a CAT-capable Intel Xeon CPU E5-2618L v3 processor, which has a 20MB 20-way set-associative L3 shared cache (divided into 20 partitions of 1MB each) and 32GB main memory, and with four cores enabled. Like in most existing real-time research [37], we disabled hyper-threading, SpeedStep, and hardware cache prefetcher features to avoid non-deterministic timing behavior. To minimize interference with the experimental workload, we shut down all non-essential system services during our experiments.

System configuration. We booted the hypervisor with the RTDS scheduler and the VMs with LITMUS^{RT} as the guest kernel, which uses the *PSN-EDF* scheduler. We created two user VMs, benchVM and polluteVM, which execute the tasks under evaluation and the interfering tasks, respectively. benchVM's tasks are statically partitioned into two full-capacity VCPUs, each of which is pinned to a dedicated



(a) *canneal* benchmark. (b) *cache-bench* program. (c) *cache-bomb* program.

Figure 5.6: WCET vs. Number of allocated cache partitions.

core. Similarly, pollute VM's are also statically assigned into two full-capacity VCPUs, which are pinned to two remaining cores. To minimize interference to benchVM, we allocated two VCPUs to the high-privilege VM (Domain 0) and directly pinned them to the two cores used by polluteVM. (Further necessary details will be described in the relevant evaluation.)

Workload. We considered two types of workload: the PARSEC benchmark suite [21] and synthetic workload. For the PARSEC benchmarks, we used *simsmall* as the default input for our WCET-related evaluation. For our real-time performance evaluation, for each benchmark, we first explored the influence of different input sets provided by the benchmark suite (i.e., test, simdev, simsmall, simmedium, and simlarge) on the WCET performance, and then selected the one that most influences the WCET performance for using in the schedulability evaluation.

The synthetic workload consists of two types of programs (similar to the ones used in [64]): (1) *cache-bench*, which uses a linked list to sequentially access every 64 bytes (i.e., cache line size) of an 8MB array for 50 times; and (2) *cache-bomb*, which uses the array index to sequentially access every 64 bytes of a 40MB array for 240 times.

To evaluate the relationship between the number of partitions and WCET, we measured the WCET of each workload program across 25 runs when the number of partitions it is allocated varies. The results are shown in Fig. 5.6.

As expected, as the number of allocated partitions increases, task's WCET also tends to decrease, which is the case for the *canneal* benchmark and the *cache-bench* program. Note, however, that the WCET of the *cache-bomb* program is relatively stable regardless of the number of partitions; this is because its array size is twice the entire cache's size, and thus all accesses to its array elements are cache misses even if it is allocated the entire cache. This observed relationship between WCET and the allocated number of partitions provides useful information for selecting, or dynamically modifying, the number of partitions allocated to each task to optimize the overall system's performance (e.g., schedulability).

5.4.2 Benefits of task-level cache isolation on WCET

Experiment. This experiment aims to evaluate how well task-level cache isolation with CAT virtualization can protect a task's WCET from being affected by concurrent accesses to the cache by other co-running tasks. For this, we executed the task-under-test (a PARSEC benchmark or a *cache-bench* task) alone on one VCPU of benchVM, and we executed a *cache-bomb* task in the second VCPU of benchVM and in each of polluteVM's VCPUs. (Recall that these four VCPUs are pinned to four different cores.) We configured the cache allocation data structures in our prototype to statically allocate 14 exclusive partitions for the task-under-test and 2 exclusive partitions for each of the three *cache-bomb* tasks. We then measured the WCET of the task-under-test across 25 runs, which we refer to as WCET under the PolluteCAT setting.

For comparison, we conducted the same experiment for (i) the Alone setting, where we disabled all three *cache-bomb* tasks; and (ii) the Pollute, where we ran the tasks in vanilla LITMUS^{RT}/Xen, which does not support cache allocation and thus all tasks share the entire cache.

Results for PARSEC benchmarks. Fig. 5.7 shows the slowdown factor of each PARSEC benchmark task for the three settings, where the slowdown factor for a

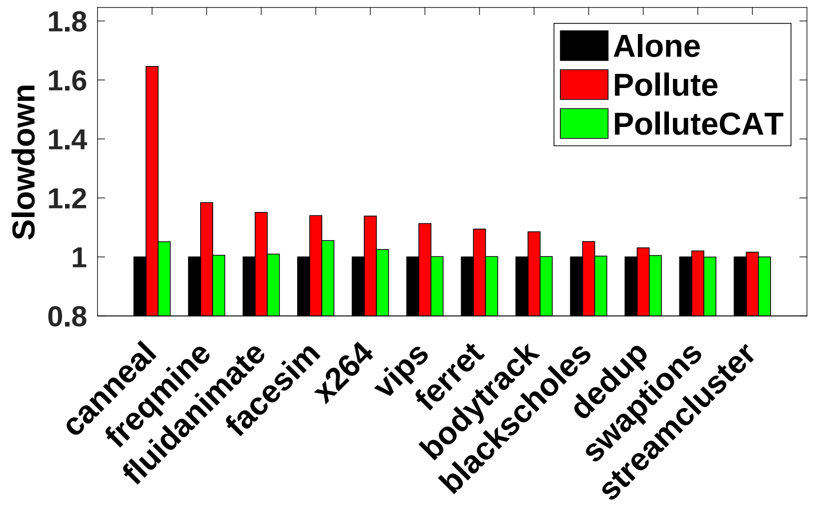


Figure 5.7: Measured WCETs of PARSEC benchmarks.

setting is the ratio of the task's WCET obtained in that setting to that was obtained in the *Alone* setting. The results show that the WCET of the benchmark task can increase substantially (up to 1.65×) in the *Pollute* setting; this is because there is no cache management in this setting and thus other co-running tasks may interfere with the benchmark task by accessing the cache. It is also worth noting that the obtained slowdown factor is with respect to a default input and not the worst-case slowdown. In contrast, the benchmark task has approximately the same or only slightly increased WCET in the PolluteCAT setting as in the Alone setting across most benchmarks. (One reason for the slight increase in WCET could be because we did not isolate the main memory in our experiments and thus, tasks may still interfere with one another due to memory bus or bank contention.) In summary, the results demonstrate that cache isolation with CAT virtualization can effectively avoid the WCET slowdown caused by cache interference.

Results for the synthetic workload. Fig. 5.8 shows the WCET slowdown of the *cache-bench* task under each setting when we varied the task's array size from 1MB to 40MB. The results further confirm that, without cache management, the shared cache interference can increase the task's WCET by a significant factor, e.g.,

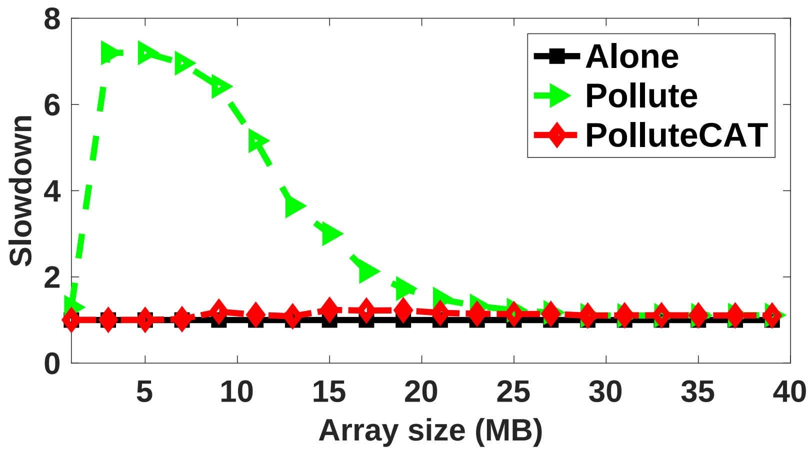


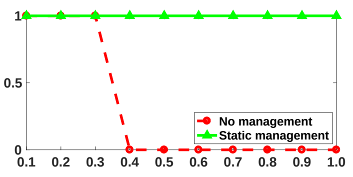
Figure 5.8: Measured WCETs of the *cache-bench* workload.

up to 7.2× (when the array size is between 3MB and 5MB). On the contrary, CAT virtualization can effectively mitigate this problem, as evident by the slowdown factor of close to 1. Notice that when the array size is larger than the cache size (20MB), the task begins to experience cache misses even when it executes alone; as a result, the WCET in the Alone setting begins to increase, leading to a decrease in the slowdown in the Pollute setting.

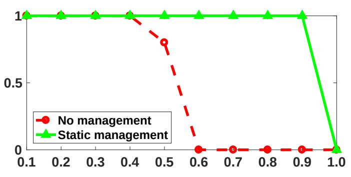
5.4.3 Real-time performance: static cache management

Next, we evaluate how much CAT virtualization can help improve the system's schedulability compared to the cache-agnostic vanilla LITMUS^{RT}/Xen system. To this end, we consider two use cases of CAT virtualization: one for static cache management, and the other for dynamic cache management. We focus on the former in this section.

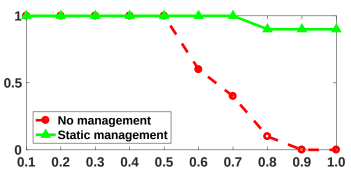
Allocation configuration. Our experiments used task sets that each consist of two workload types: (1) either the PARSEC benchmark or the *cache-bench* program, and (2) the *cache-bomb* program. We used the same configuration as in the preceding experiment: the benchmark (*cache-bench*) tasks are scheduled on one VCPU (pinned to a dedicated core) with 14 partitions; the *cache-bomb* tasks are statically parti-



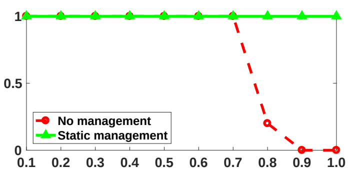
(a) *streamcluster.*



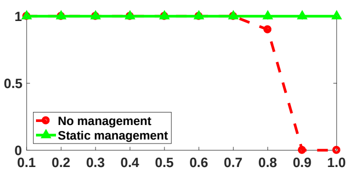
(b) *canneal.*



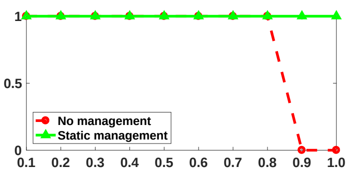
(c) *x264.*



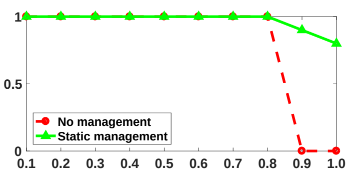
(d) *vips.*



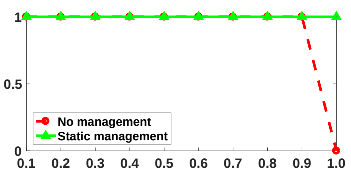
(e) *bodytrack.*



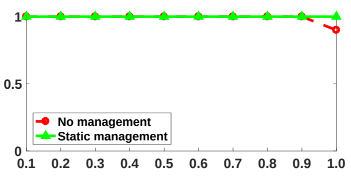
(f) *ferret.*



(g) *fluidanimate.*



(h) *blackscholes.*



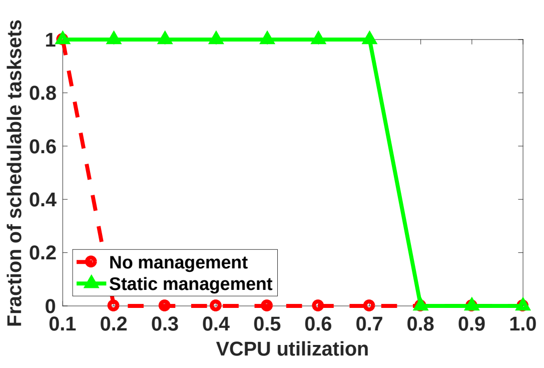
(i) *swaptions.*

Figure 5.9: Schedulability of PARSEC benchmarks. The x-axis shows the VCPU utilization, and the y-axis shows the fraction of schedulable tasksets.

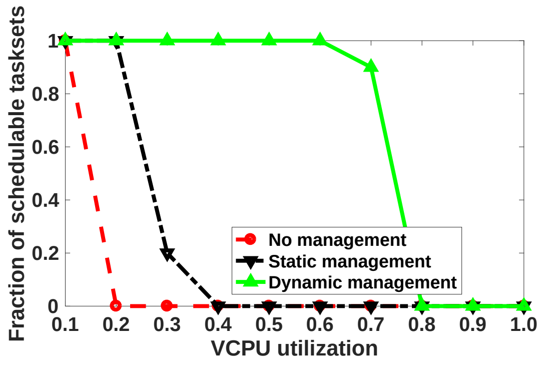
tioned into three VCPUs, each of which is allocated 2 partitions. We measured the WCET of each PARSEC benchmark, *cache-bench*, or *cache-bomb* task under this cache allocation.

Task set creation. We first converted the PARSEC benchmarks into LITMUS^{RT}-compatible real-time tasks. While doing so, we found that three benchmarks (*facesim*, *vips* and *freqmine*) contained memory leak bugs; unfortunately, we could not fix the bug in the *freqmine* benchmark and thus could not use it for our schedulability experiments. In addition, the *facesim* benchmark took too long to complete; we omitted it due to time constraints. We conducted the schedulability experiments for all the remaining ten PARSEC benchmarks.

To generate a real-time task τ_i, we first randomly generated a harmonic period p_i, and then computed the task's utilization u_i based on both p_i and its WCET (determined above).



(a) Static.



(b) Dynamic.

Figure 5.10: Schedulability of synthetic benchmarks.

A task set for the benchmark VCPU was created based on a chosen target VCPU utilization U_{vcpu}. Specifically, we randomly generated real-time tasks for the benchmark VCPU until the total utilization of the generated tasks reaches U_{vcpu}. We repeated this generation 10 times to create 10 task sets per U_{vcpu}, where U_{vcpu} ranges from 0.1 to 1.0, with a step of 0.1; this led to a total of $10 \times 10 = 100$ task sets. For each task set, we executed it for two minutes both on the vanilla LITMUS^{RT}/Xen and on our prototype, and we measured the schedulability of the task set in each setting.

Benchmark results. Fig. 5.9 shows the fraction of schedulable task sets of the PARSEC benchmarks when varying the target VCPU utilization.[22] The results across all benchmarks show that our vCAT cache management can substantially improve the system's schedulability. It can also be observed from Fig. 5.9(a) that, for the *streamcluster* benchmark, on the vanilla LITMUS^{RT}/Xen, the fraction of schedulable task sets begins to decrease quickly once the target VCPU utilization U_{vcpu} is more than 0.3, and all task sets become unschedulable when $U_{vcpu} \geq 0.4$. In contrast, with static cache allocation, all task sets remain schedulable even when each VCPU's utilization is at 1.0. The static management in vCAT can increase

[22]We omit the results of the *dedup* benchmark, since all task sets are schedulable across all utilizations for both techniques.

system utilization by up to $\frac{1.0}{0.3} = 3.3\times$.

Synthetic results. Fig. 5.10(a) shows the schedulability results for task sets with the *cache-bench* workload, which further highlights the performance benefits of and the needs for static cache allocation. Without cache management, tasks begin to miss deadlines as soon as $U_{vcpu} > 0.1$, whereas a task is only become unschedulable when $U_{vcpu} > 0.7$ under cache allocation. In other words, the static cache allocation enabled by our CAT virtualization can help increase schedulable utilization by up to 7 times.

5.4.4 Real-time performance: dynamic cache management

In the previous use case, cache allocation is performed statically, where each task is always allocated a fixed number and a fixed set of partitions (and thus has a fixed WCET). While this approach is a preferred and more efficient choice in systems with relatively static timing behavior, it may substantially underutilize the cache when the task's timing constraints (such as deadline, period, and cache demand) vary dynamically at run time. In this section, we investigate the performance benefits of our dynamic cache management enabled by CAT virtualization, using a multi-mode system use case.

Dual-mode task sets. We constructed multi-mode tasks based on unimodal *cache-bench* tasks as follows. We first generated a unimodal *cache-bench* task as in the static use case, and then created two dual-mode versions: the *cache-bench-mm1* version uses the same unimodal task parameters for both modes, except that the task period (deadline) in Mode 2 is K times the unimodal period; and the *cache-bench-mm2* version also uses the unimodal parameters for both modes, except that the period in Mode 1 is K times the unimodal period. Intuitively, K captures the degree of dynamism in the task's WCET when varying the number of allocated cache partitions. We set K to be the ratio of the WCET of *cache-bench* when requesting two (the minimum possible) partitions to its WCET when requesting 20 (the maximum possible) partitions; in our experiments, $K = \frac{707}{114} \approx 6.202$. In addition to multi-mode *cache-bench* tasks, we also used the unimodal *cache-bomb* tasks generated as in the static use case.

All *cache-bench-mm1* tasks and all *cache-bench-mm2* tasks are executed on the first VCPUs of benchVM and polluteVM, respectively. We statically partitioned the *cache-bomb* tasks into the second VCPUs of both VMs. We generated 10 task sets for each target VCPU utilization, using the same procedure as in the static use case.

Experiment. We ran each task set for two minutes on our prototype with dynamic cache allocation and measured its schedulability. The cache allocation was configured dynamically as follows. Each VCPU running *cache-bomb* tasks is always allocated two partitions. We configured each multi-mode task to execute in Mode 1 during the first minute, but in Mode 2 during the second minute. The VCPU running *cache-bench-mm1* tasks is allocated 14 partitions in Mode 1 and 2 partitions in Mode 2, whereas the VCPU running *cache-bench-mm2* tasks is allocated 2 partitions in Mode 1 and 14 partitions in Mode 2. This configuration was chosen to balance the VCPU utilization across the two modes.

For comparison, we also ran each task set on Vanilla LITMUS^{RT}/Xen and on our prototype with static allocation, where we statically allocated 8 partitions to each VCPU running multi-mode tasks, and 2 partitions to each VCPU running *cache-bomb* tasks.

Results. Fig. 5.10(b) shows the fraction of schedulable task sets per VCPU utilization for each of the three settings. As expected, both static and dynamic cache management can help improve the schedulability of the task sets substantially. The results also show that dynamic cache management outperforms static management by a substantial factor in terms of improving schedulable utilization (3×), which is expected since it is much more effective in handling workloads with dynamic timing constraints.

5.5 Conclusion

We have presented a novel approach to shared cache management in multicore virtualization systems, through an integration of Intel CAT and cache partition virtualization. Our CAT virtualization design is highly general: it can be configured to provide strong isolation among tasks and/or VMs, to support both real-time tasks—potentially with different criticality levels – and best-effort tasks, and to achieve both static and dynamic cache allocations. We implemented a prototype of the design on top of Xen and LITMUS^{RT}. Experimental results using both PARSEC benchmarks and synthetic workloads show that our prototype introduces only a small overhead while improving both the WCET and the schedulability of the system significantly. The results also show that dynamic allocation is much more effective in improving schedulability than static allocation, especially under dynamic task sets. In future work, we plan to apply our design to several other settings, as well as develop new compositional analysis techniques for cache-aware schedulability test and VM interfaces' computation.

Chapter 6

Holistic resource allocation and analysis

We have developed the shared cache management technique for mitigating the shared cache interference on the multicore virtualization platform. While the shared cache management technique in Chapter 5 brings us closer to achieving timing isolation, it does not consider the memory bandwidth interference, which ultimately introduces unaccountable extra latency. In addition, it does not address the allocation policy and analysis questions, such as what is the right number of cache partitions to allocate to a task (or a core), or how to formally analyze the schedulability of the system.

The problem of memory bandwidth interference has been addressed in non-virtualization systems [77, 64, 10]. For instance, MemGuard [77] provides a way to regulate the memory bandwidth that each core (or task) can access in Linux, and thus it can ensure that each core is guaranteed to receive the allocated amount of bandwidth. In principle, this idea should work in the virtualization setting as well; however, the existing regulation mechanisms cannot be directly applied here, due to inherent differences between the two settings. For instance, software-based approaches such as [77] [64] rely on the `perf` monitoring tool provided by Linux and they are implemented as device drivers, neither of which is supported in production hypervisors such as Xen.

To bridge the above gap, we propose vC^{2}M, a holistic solution towards timing isolation in multicore virtualization systems. On the system angle, vC^{2}M integrates both the shared cache and memory bandwidth management to provide better isolation among tasks and VMs; this is done by leveraging the existing cache allocation system vCAT [72] and a new memory bandwidth regulation mechanism for virtualization.[23] On the theory side, vC^{2}M provides an efficient resource allocation policy for tasks and VMs that can minimize resources while guaranteeing schedulability. Specifically, given a set of tasks on the VMs and a given hardware configuration, vC^{2}M will compute both (i) the assignment of tasks to virtual CPUs (VCPUs) and VCPUs to cores, and (ii) the amount of CPU, cache, and bandwidth resources for each task and each VCPU, to guarantee schedulability while minimizing resource use.

To the best of our knowledge, vC^{2}M is the first to consider CPU, cache, and memory bandwidth allocation in a holistic manner for resource allocation in real-time multicore virtualization systems.

6.1 Design

6.1.1 System architecture and overview of vC^{2}M

The platform consists of multiple VMs running on a shared multicore processor by a hypervisor, such as Xen. Each VM executes a number of real-time tasks using some real-time OS, such as LITMUS^{RT} [27, 24]. Tasks within each VM are scheduled on a set of virtual CPUs (VCPUs) by the VM's scheduler, and all VCPUs are scheduled on the physical cores by the hypervisor's scheduler. We assume that partitioned

[23]We note that the memory bank interference is another non-negligible source of overhead, which we do not consider in this work; however, there exists prior result on this topic [76], which can be incorporated into vC^{2}M.

scheduling is used at both the VM and hypervisor levels.

To achieve cache and memory bandwidth isolation among concurrently running tasks, vC^{2}M provides cache and memory bandwidth allocation at the core level. Specifically, it divides the shared cache into multiple partitions, and allocates a disjoint subset of partitions to each core; this is done by leveraging vCAT [72], an existing cache allocation system based on Intel's Cache Allocation Technology. To provide bandwidth allocation, vC^{2}M introduces a new memory bandwidth regulation mechanism that enables the run-time monitoring of memory requests and the enforcement of a given bandwidth to a core in the virtualization setting. In the following, we discuss the latter new component in detail.

6.1.2 Memory bandwidth regulation

Approach. Our memory bandwidth regulation relies on hardware performance counters to monitor the number of memory requests from each core in each regulation period (a small configurable interval, e.g., 1ms). Whenever a core exceeds a configured number of memory requests (i.e., its bandwidth budget), we throttle the core by notifying the hypervisor to leave the core idle for the remaining time of the regulation period. When a new period begins, we un-throttle the core by triggering the hypervisor to execute a VCPU on the core. With this mechanism, each core is always guaranteed to receive its configured budget in each period, and it never is allowed to use more bandwidth than it is allocated.

Relation to existing work. Conceptually, the idea is similar to that of MemGuard [77], a bandwidth regulation mechanism in the non-virtualization setting. However, our approach differs in several aspects: First, our bandwidth regulation is a built-in feature of the hypervisor instead of a loadable module, which is not always supported by a hypervisor such as Xen. Second, unlike MemGuard which relies on Linux's `perf` tool for monitoring and notification, ours works directly with the low-level hardware components to monitor the memory requests and to configure the interrupt that notifies a core when it runs out of its bandwidth budget; hence, it can reduce the extra overhead introduced by a performance monitoring tool. (We note also that, due to security concerns, `perf` is not available in production Xen virtualization system [7].) Finally, MemGuard keeps the throttled core busy by running a CPU-intensive dummy task on it, which consumes energy unnecessarily. In contrast, by modifying the hypervisor to be aware of the throttled cores and to stop scheduling VCPUs on them, these cores are always kept idle in our system.

Core components. Modern Intel processors come with several general performance counters (PC) on each core for monitoring cache misses. We use an unused PC counter to monitor the number of last-level cache misses, which can be treated as the number of memory requests [77, 46]. Each core has a Local Advanced Programmable Interrupt Controller (LAPIC), which can be configured to deliver the PC counter overflow interrupt to the core. All cores can access an overflow status register that specifies which PC counters overflowed and an overflow control register that can clear the overflow status register.

A high-level architecture of our bandwidth regulator is shown in Fig. 6.1.

Setup. The setup component is responsible for configuring the system upon initialization, including (ii) configuring an unused general PC counter on each core to monitor the number of memory requests from the core, and preseting its value so that it will overflow when the core runs out of its memory bandwidth budget; (i) configuring the LAPIC on each core to deliver the performance counter overflow interrupt to the core when its PC counter overflows; (iii) creating a periodic timer to periodically replenish each core's memory bandwidth budget; and (iv) clearing the overflow status register that indicates which PC counters overflow.

Regulation. Once the regulator has been initialized and enabled, the PC counter will begin counting the number of memory requests from each core. When a core's PC counter overflows, the LAPIC delivers the performance counter overflow interrupt to the *BW enforcer* handler running on the core (Steps ① and ② in Fig. 6.1). Upon

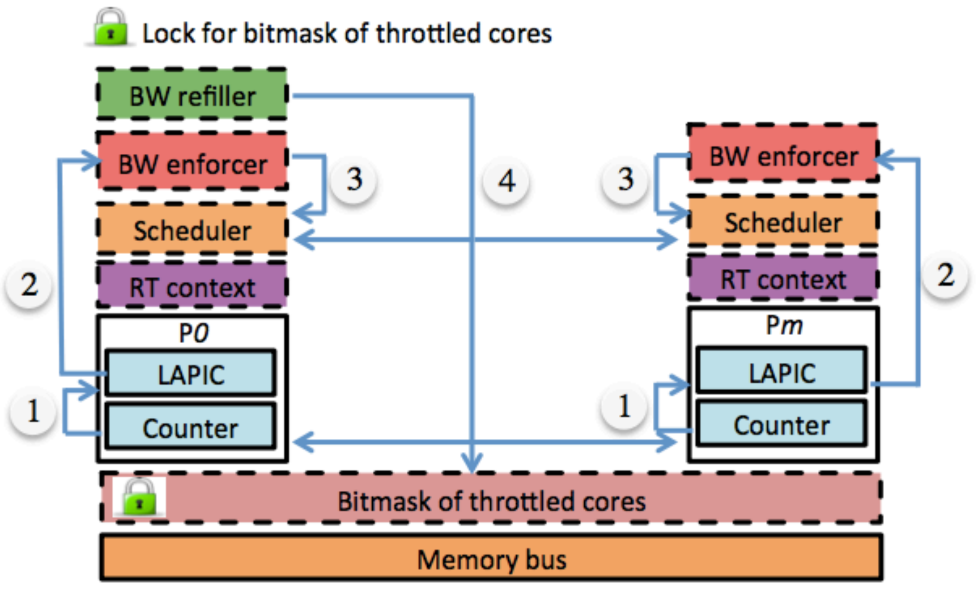


Figure 6.1: Architecture of memory bandwidth regulation in vC^{2}M.

receiving the interrupt, the *BW enforcer* handler invokes the hypervisor's scheduler to de-schedule its currently running VCPU (Step ③). The hypervisor's scheduler was modified to be aware of the throttled cores; this is necessary to ensure that it will never schedule a VCPU onto a throttled core. In addition, the bandwidth replenishment handler (*BW refiller*) periodically replenishes the budget for each core and invokes the scheduler on each throttled core to schedule a VCPU onto the core at the beginning of each regulation period (Step ④).

User-level administration tool. vC^{2}M also includes a user-level tool for system operators (in the privilege VM) to configure the memory bandwidth allocated to each core.

6.2 Implementation

We now describe a prototype of our design which will be used for our experiments. Our prototype extends the Xen hypervisor (version 4.8.0) and LITMUS^{RT} 2015.1

guest OS, running on top of the Intel Xeon CPU E5-2618L v3 processor. The Xen hypervisor has a real-time scheduler called RTDS, which we extended for our implementation.

6.2.1 Implementation of the memory bandwidth regulation

We leveraged the various hardware components (PC counters, LAPIC controllers, overflow status register, and overflow control register) for the memory request monitoring and notification as described earlier. We created two software data structures for allocation purpose: (1) a per-core *rt_ context*, which records the maximum allocated memory bandwidth budget and remaining bandwidth budget of the core in the current regulation period before it is throttled, and (2) a *bitmask of throttled cores*, which specifies the cores that have used up their allocated budgets; this bitmask is accessible by all cores, and it is protected by a spinlock (see Fig. 6.1). We modified Xen's RTDS scheduler to consider the *bitmask of throttled cores* in making scheduling decisions.

At initialization, the setup component configures the LAPIC controllers, sets up the PC counter 3 (which is unused by the system) of each core to count the cache miss events, and clears the PC counter 3's bit in the overflow status register. It also creates on core 0 a periodic timer to invoke the *BW replenish handler* in every regulation period (e.g., 1 *ms*).

We implemented two interrupt handlers for the regulation: *BW enforcer handler*, which handles the performance counter overflow interrupt; and *BW refiller handler*, which performs budget replenishments. When triggered, the *BW enforcer handler* checks whether the interrupt was raised by the PC counter 3 by reading the *overflow status register*. (This is necessary, as the handler is also triggered when other PC counters overflow.) If so, it clears the corresponding bit in the register, sets the core's bit in the *bitmask of throttled cores*, and invokes the scheduler on the core to reschedule. When the (modified) RTDS scheduler is invoked by the *BW enforcer* *handler*, it de-schedules the currently running VCPU on the core and leaves the core idle. Whenever the scheduler is invoked by a scheduling event, it checks the *bitmask of throttled cores* and leaves the core idle until the core is cleared in the bitmask.

The *BW refiller handler* is invoked every configurable regulation period by the periodic timer created at setup. Whenever the handler is invoked, it refills the memory bandwidth budget for each core by resetting the PC counter 3's value, and then invokes the scheduler on each throttled core to schedule a VCPU onto the core.

6.3 Empirical evaluation

To evaluate the implementation overhead and benefits of vC^{2}M, we performed a series of experiments on our prototype using both PARSEC benchmarks [21] and synthetic workloads. Our objectives are to evaluate (i) the impact of disabling cache on WCET, (ii) the effectiveness of vC^{2}M in mitigating cache and memory bandwidth interferences, (iii) the impact of resource allocation on WCET, and (iv) the overhead introduced by vC^{2}M.

6.3.1 Experimental setup

Hardware. Our prototype ran on a machine with a CAT-capable Intel Xeon E5-2618L v3 processor, with a 20MB 20-way set-associative L3 shared cache and an 8GB PC-2133 DDR4 DRAM. The cache can be divided into 20 equal partitions (using vCAT [72]), and a core must be allocated at least 2 partitions (due to hardware constraints). The maximum guaranteed bandwidth was $1.4GB/s$ (obtained using the same method as in [78]. For our experiments, we divided the bandwidth into 20 partitions of 70MB/s each, and the maximum bandwidth budget allocated to a core was always equal to (the size of) one or multiple partitions.

System configuration. We booted the hypervisor with the RTDS scheduler, and two guest VMs with LITMUS^{RT}. Each VM has two full-capacity VCPUs, each of which is pinned to a dedicated core. The first VM, benchVM, runs the tasks under evaluation on one VCPU and interference tasks on its other VCPU. The second VM, polluteVM, runs interference tasks on both of its VCPUs.[24]

Workload. We considered two types of workloads: the PARSEC benchmark suite [21] and a synthetic workload. For the PARSEC benchmarks, we use *simsmall* as the default input. For the synthetic workload, we have two types of programs (similar to the ones used in [68] and [72, 64], respectively): (1) *cpu-bench*, which performs a specified amount of add operations on a full-capacity VCPU; and (2) *cache-bomb*, which uses an array index to sequentially access every 64 bytes of a 40MB array until it is terminated.

6.3.2 Impact of disabling cache on WCET

Experiment. Without cache and memory bandwidth resource management, we can still avoid cache interference by disabling cache and upper bound the impact of the memory bandwidth interference by assuming the worst-case memory bandwidth. This no-resource-management approach is intuitive and straightforward, but it may lead to very pessimistic WCETs for tasks, which are significantly larger than tasks' WCETs with resource management.

To evaluate the impact of the no-resource management approach on a task's WCET, we need to measure the task's WCET with and without resource management supports. The ideal approach to get the task's WCET under the no-resource-management approach is to disable all three levels of caches on the Intel processor by setting the 30^{th} bit in the CR0 register [2]. However, this approach is not practical because system becomes extremely slow after its cache is disabled. Instead, we choose to estimate a task's WCET under the no-resource-management approach by measuring the task's cache hit and miss requests and calculating the extra de-

[24]To minimize the interference from the administration VM (domain 0), we allocated to it one VCPU, which was pinned to a separate core.

lay when all of its cache hit requests become cache misses and when the memory bandwidth is always the worst-case bandwidth.

Given a task τ_i's measured WCET with cache support and its numbers of cache hit requests, we calculate the task's WCET e'_i without cache support by

$$e'_i = e_i + N_i^{l2hit} \cdot (lat_{l3miss} - lat_{l2hit}) + N_i^{l3hit} \cdot (lat_{l3miss} - lat_{l3hit}) \tag{6.1}$$

where e_i is the task's measured WCET with cache support; N_i^{l2hit} and N_i^{l3hit} respectively are the numbers of cache hit requests on L2 and L3 caches for τ_i; and lat_{l3miss}, lat_{l2hit}, and lat_{l3hit} are the L3 cache miss latency, the L2 cache hit latency, the L3 cache hit latency respectively.

Given a task τ_i's calculated WCET e'_i without cache support and the task's total number of cache misses N_i^{miss}, we calculate the task's WCET e''_i under the worst-case memory bandwidth as

$$e''_i = e'_i + N_i^{miss} \cdot (1/BW_{worst} - 1/BW_{best}) \tag{6.2}$$

where BW_{worst} and BW_{best} respectively are the worst-case and best-case memory bandwidths supported by the hardware, and $1/BW_{worst}$ and $1/BW_{best}$ respectively are the average worst-case and best-case cache miss latencies, whose difference is the extra latency each cache miss takes in the worst-case scenario.

The above method of estimating a task's WCET under the no-resource-management approach favors the no-resource-management approach – the calculated WCET of a task is likely smaller than the actual WCET without cache support. This is because the calculated WCET does not include the following extra latencies that are included in the actual WCET: (i) the latency of extra cache misses when L1 cache hit requests become cache misses; [25] and (2) the extra latency the CPU pipeline experiences when it waits for the data.

[25]We cannot obtain the number of L1 cache hit requests due to the hardware limitation.

We run a PARSEC benchmark task with simlarge input on one core without disabling cache and measured the task's WCET across 25 runs. In each run, we measured the task's numbers of cache hit requests on L2 and L3 caches by using Intel hardware performance counter. We calculated the task's WCET under the no-resource-management approach by using Eq. 6.2.

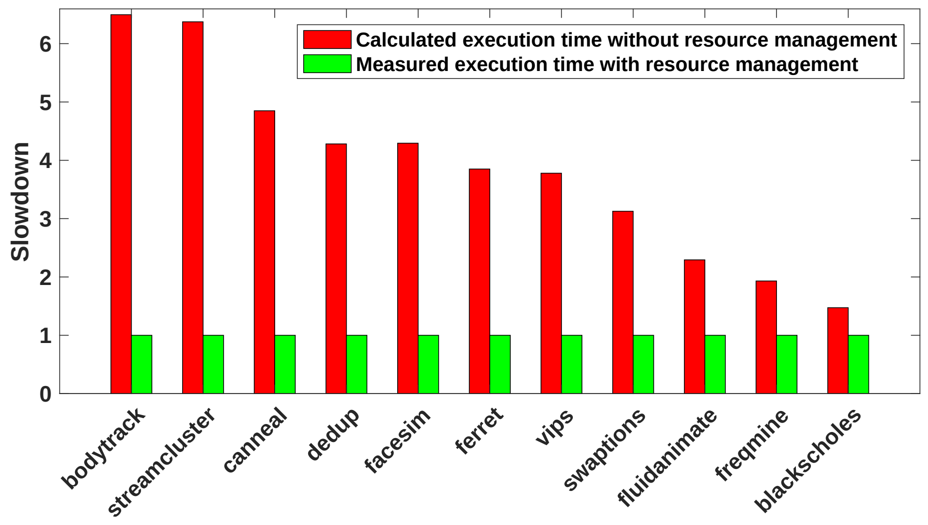


Figure 6.2: Slowdown of PARSEC benchmarks without resource management.

Results. Fig. 6.2 shows the slowdown factor of each PARSEC benchmark task with and without resource management. The slowdown factor in a setting is the ratio of the benchmark task's calculated execution time with Eq. 6.2 to the task's measured WCET with cache enabled. The results show that the execution time of the benchmark task without resource management support is significantly larger than that with resource management support (up to 6.50×). This demonstrates that disabling cache to avoid cache interference can introduce significant pessimism in estimating a task's WCET, impeding the system's schedulability. This also motivates us to manage the cache-related resources to achieve resource isolation and to better utilize the resources to improve the system's schedulability.

6.3.3 Benefits of resource isolation with vC^{2}M

Experiment. To evaluate how well vC^{2}M can protect a task's WCET from being affected by concurrent running tasks, we ran a PARSEC benchmark task on one VCPU in benchVM, and ran a *cache-bomb* task on each of the other three VCPUs. We allocated 14 cache partitions and 17 bandwidth partitions to the benchmark core (i.e., which executed the benchmark task), and allocated 2 cache partitions and 1 bandwidth partition to the other three cores. We measured the WCET of the benchmark task across 25 runs, which we refer to as PolluteCAM. For comparison, we conducted the same experiment for two additional settings: (i) the PolluteCA setting, where we only managed the cache; and (ii) the Pollute setting, where all tasks shared the entire cache and memory bandwidth without any management.

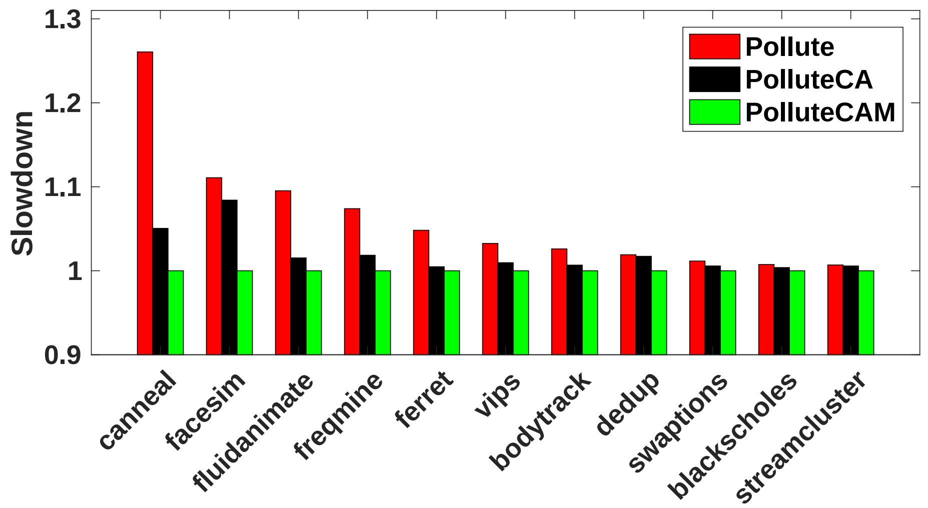


Figure 6.3: Measured WCETs of PARSEC benchmarks.

Results. Fig. 6.3 shows the slowdown factor of each PARSEC benchmark task for the three settings. The slowdown factor in a setting is the ratio of the benchmark task's WCET to its WCET obtained in the PolluteCAM setting. The results show that the WCET of the benchmark task in the Pollute setting is substantially larger than that of the PolluteCAM setting (up to 1.26×). This demonstrates that by managing both cache and memory bandwidth, we can effectively mitigate the inter-

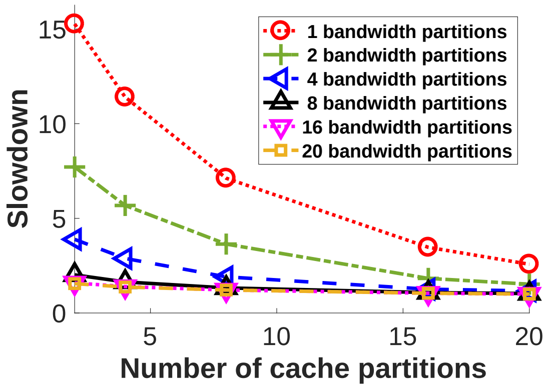


Figure 6.4: Memory bandwidth impact.

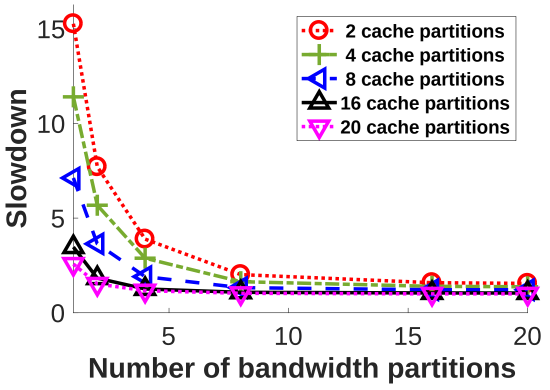


Figure 6.5: Cache impact.

ferences and improves the WCET. Further, the WCET of the benchmark task in the PolluteCA setting can increase by up to 1.11× (for *facesim* benchmark) compared to the PolluteCAM setting, which suggests that it is important to manage both cache and memory bandwidth resources to achieve better timing isolation.

6.3.4 Impact of cache and bandwidth allocation on WCET

Experiment. To evaluate the impact of cache and memory allocation on WCET, we ran the *canneal* benchmark on one full-capacity VCPU in benchVM, and we configured the corresponding core with different numbers of cache partitions and bandwidth partitions. We measured the benchmark task's WCET across 25 runs, and calculated its *resource slowdown* factor under a cache and bandwidth allocation configuration (as the ratio of the measured task's WCET to its WCET when it is allocated all cache and bandwidth partitions).

Results. Figs. 6.4 and 6.5 show the impact of bandwidth and cache resource allocation on the task's WCET, respectively. Fig. 6.4 shows that the *canneal* benchmark task's slowdown varies from 15× to 2.57× when the task is allocated 1 bandwidth partitions; in contrast, the slowdown does not change substantially when the task is allocated 20 memory bandwidth partitions. A similar trend can also be observed in Fig. 6.5. In general, we can make the following observation:

Observation 6.1. *The relation between a task's WCET and the amount of cache (resp. memory bandwidth) resource it receives is highly dependent on the amount of memory bandwidth (resp. cache) it receives. In particular, a task's WCET is more sensitive to the cache allocation when it is allocated a smaller amount of memory bandwidth, and vice versa.*

This behavior is expected, as the more cache space a task receives, the fewer cache misses it incurs, and thus the frequency that it is throttled also decreases. Similarly, when a task receives less memory bandwidth, it runs out of budget more quickly and becomes throttled more frequently, which in turn makes it more sensitive to its allocated cache space.

We repeated the experiment with each PARSEC benchmark to examine the effect of the workload characteristics. Our results show that the above observed pattern varies across benchmarks.

Observation 6.2. *The relations between a task's WCET and its allocated cache and memory bandwidth resources vary across different benchmark tasks. Some tasks (e.g.,*canneal *benchmark) are sensitive to both cache and bandwidth resources, whereas others are sensitive to only one (e.g.,* facesim *benchmark) or none (e.g.,* swaptions *benchmark) of the resources.*

These results motivate the need for considering the relations between CPU, cache and memory bandwidth resources in allocation to achieve better utilization and schedulability.

6.3.5 Overhead

We measured the overhead of vC^{2}M using the same approach as in [69].

Memory bandwidth regulator overhead. The regulator introduces two types of overhead: (i) throttle overhead, *throttle_oh*, when a core is throttled; and (ii) memory bandwidth budget replenish overhead, *mem_repl_oh*, when the bandwidth budgets are refilled for all cores. To measured these, we booted a VM with 4 full-capacity VCPUs, each of which is pinned to a dedicated core. We allocated a bandwidth budget of 140 MB/s to each core. We ran the *cache-bomb* on each core to trigger the memory regulator as frequently as possible, and measured the two types of overhead. As shown in Table 6.1, our memory bandwidth regulator introduces only a very small overhead.

Table 6.1: Memory bandwidth regulator overhead (μs).

throttle_oh			*mem_repl_oh*		
min	average	max	min	average	max
0.33	0.37	1.15	8.81	52.22	108.65

Scheduler overhead. The RTDS scheduler in Xen 4.8.0 is an event-driven scheduler [4]. The modified RTDS scheduler has three types of overheads: (i) budget replenishment overhead, *repl_oh*, for replenishing a VCPU's budget; (ii) scheduling overhead, *sched_oh*, for de-scheduling a VCPU that runs out of budget; and (iii) context switch overhead, *cxs_oh*, for switching the currently running VCPU on a core with another. Using a similar overhead measurement method as in [69], we measured the scheduler overheads when the system has 24 VCPUs and 96 VCPUs, respectively. The results are shown in Table 6.2. We can observe that the maximum scheduling-related overhead is minimal, and it increases slowly as the number of VCPUs increases; for example, when the number of VCPUs increases by $96/24 = 4\times$, the overhead increases by only up to $3.73/2.95 \approx 1.26\times$.

Table 6.2: Scheduler overhead (μs).

	VCPU set size: 24			VCPU set size: 96		
	min	average	max	min	average	max
repl_oh	0.29	0.74	2.95	0.34	1.26	3.73
sched_oh	0.13	0.57	1.73	0.13	0.55	2.03
cxs_oh	0.04	0.23	32.07	0.04	0.27	24.67

In the next two sections, we present a formal model of the system and a resource allocation algorithm for vC^{2}M.

6.4 Theoretical modeling and goal

The system we presented so far provides mechanisms for resource allocation in virtualization systems. However, one important question remains: How to compute the exact allocation of CPU, cache, and memory bandwidth for tasks and VCPUs to maximize schedulability? To solve this problem, we focus on a concrete setting of vC^{2}M, which we now formalize.

Platform model. The platform consists of M identical cores, with a shared cache and a shared memory bus that are accessible by all cores. The cache is divided into N_{cp} equal-size cache partitions, and the memory bandwidth is divided into N_{bw} equal-size memory bandwidth partitions. Cache and bandwidth allocation is done at the core level: each core is allocated a distinct set of cache partitions and a certain number of bandwidth partitions, all of which will be available to any task (VCPU) currently running on the core. To accommodate hardware constraint, we denote by N_{cp}^{min} and N_{bw}^{min} the minimum numbers of cache partitions and bandwidth partitions that a core is allocated, respectively. The hypervisor's scheduler schedules VCPUs on the cores using the partitioned Earliest Deadline First (pEDF) algorithm, and the VM's scheduler schedules its tasks on the VCPUs also according to pEDF.

VCPU model. We assume that a VCPU is implemented as a periodic server, which is the case for Xen's RTDS scheduler when we run a background CPU-intensive task on each VCPU [69]. The VCPU j of the VM i is specified as $VP_i^j = (\Pi_i^j, \Theta_i^j(vcp_i^j, vbw_i^j))$, where Π_i^j is the VCPU's period, $\Theta_i^j(vcp_i^j, vbw_i^j)$ is the VCPU's execution time budget when it is allocated vcp_i^j cache partitions and vbw_i^j memory bandwidth partitions. With this model, each VP_i^j always provides $\Theta_i^j(vcp_i^j, vbw_i^j)$ CPU time in every period of Π_i^j time units, and it guarantees that any task running on it is allocated vcp_i^j cache partitions and vbw_i^j memory bandwidth partitions. The CPU bandwidth of VP_i^j is defined as Θ_i^j/Π_i^j.

We assume tasks and VCPUs partitioned on a core with cp_i cache partitions and bw_i memory bandwidth partitions can always get cp_i cache partitions and bw_i

memory bandwidth partitions. This is the case, e.g., when tasks and VCPUs are scheduled at the boundary of the regulation period.

Application model. The workload consists of multiple components, each of which is executed in a VM and contains a set of real-time tasks.

We consider independent periodic tasks with implicit deadlines[26], but extend it to capture the relationship between the task's WCET and its cache and memory bandwidth allocation. Specifically, a task τ_i is modeled as $\tau_i = \{p_i, d_i, e_i(cp_i, bw_i) \mid N_{cp}^{min} \leq cp_i \leq N_{cp} \wedge N_{bw}^{min} \leq bw_i \leq N_{bw}\}$, where p_i is the period, $d_i(= p_i)$ is the deadline, and $e_i(cp_i, bw_i)$ is the task's WCET when it is assigned cp_i cache partitions and bw_i memory bandwidth partitions. We assume that $e_i(cp_i, bw_i)$ is known a priori (which can be obtained by analysis or measurements), and that it is monotonically decreasing with cp_i and bw_i (a common assumption in existing research [19, 63]). We consider harmonic task sets. A task set $\tau = (\tau_1, ...\tau_n)$ is harmonic *iff* for any two tasks τ_i and τ_j, where $p_i \leq p_j$, $p_j \bmod p_i = 0$.

For analysis purpose, we refer to $re_i = e_i(N_{cp}, N_{bw})$ as the *reference WCET* of τ_i (i.e., the task's WCET when it is allocated all cache and memory partitions in the system). In addition, we define τ_i's *reference utilization* to be $ru_i = re_i/p_i$. By abuse of notation, we use the term *assigned WCET* and *assigned utilization* to denote the WCET and utilization of a task, respectively, when it is already assigned a fixed number of cache partitions and a fixed number of bandwidth partitions.

The set of tasks in a VM i is given as $\tau^i = \{\tau_1^i, ..., \tau_n^i\}$, where n is the number of tasks in the VM. Further, we denote by $\tau^{i,j} = \{\tau_1^{i,j}, ..., \tau_l^{i,j}\}$ the set of tasks that run on VCPU j of VM i, where l is the number of tasks on this VCPU. We require that any task in a VM i must be assigned to one of the VM's VCPUs (i.e., $\tau^i = \cup_{1 \leq j \leq N_C^i} \tau^{i,j}$ and $\tau^{i,j} \cap \tau^{i,j'} = \emptyset$) and any VCPU must be assigned to a core.

As usual, we say that a task is schedulable *iff* it always finishes execution before its deadline, and the system is schedulable if all tasks in all VMs are schedulable.

[26]We follow this model for simplicity; it should be straightforward to extend the algorithm to constrained deadline tasks.

Problem statement. Given the above model, our goal is to develop an algorithm and associated schedulability analysis for computing (i) a mapping of tasks to VCPUs and VCPUs to cores, and (ii) the number of cache partitions, the number of memory bandwidth partitions (per regulation period), and the CPU budget for each VCPU in the system, so that the system is schedulable while minimizing the total utilization of all tasks. Note that the number of cache (memory bandwidth) partitions to be allocated to a core can be computed trivially as the maximum among that of its VCPUs.

Challenge. Achieving both effectiveness and efficiency is challenging in our setting due to two reasons: (1) the abstraction overhead in compositional analysis may cause inefficient use of CPU resource and negatively affect the cache and memory bandwidth resource allocation; (2) there exists inter-dependence between WCET, cache allocation, and memory bandwidth allocation, which also varies across tasks. The resource allocation problem we consider is, in fact, more general than the traditional packing of tasks to cores, which is known to be NP-hard. In the next two sections, we present an abstraction-free compositional analysis for harmonic tasks and a novel resource allocation approach that uses a combination of clustering and bin-packing heuristics.

6.5 Analysis

Given a resource allocation of vC^{2}M on a platform, we need an analysis technique to tell if the system is schedulable. In this section, we first review a Periodic Resource Model (PRM) based compositional analysis technique that is used for analyzing real-time virtualization systems [41]; we then propose an improved analysis for harmonic tasks that removes abstraction overhead (which will be defined later) and introduce a cache-aware analysis that considers the cache overhead among tasks on the same core.

6.5.1 Background on PRM-based compositional analysis

Recall that a PRM-based resource interface is represented as $\Omega = (\Pi, \Theta)$, specifying that the resource interface provides Θ time units for every Π time units. A PRM-based resource interface can be naturally and directly transformed into a VCPU's parameters, where the VCPU's period and budget are Π and Θ, respectively.

We can use the PRM-based compositional analysis to analyze the schedulability of a vC^{2}M system as previous work did in [41]: we first compute each VCPU's parameters by abstracting resource demand of tasks on the VCPU into a PRM-based resource interface; we then check the system's schedulability by testing if the total utilization of VCPUs on *each* core is no larger than 1.

To compute a VCPU's parameters, we need the resource demand bound function (dbf) of tasks, which represents the maximum resource demand of these tasks in a time interval, and the resource supply bound function (sbf) of the VCPU, which specifies the minimum resource supply of the VCPU in a time interval. The dbf of a set of implicit-deadline tasks $\tau_i = (\tau_i^1, ...\tau_i^n)$—each task's period is equal to its deadline—is Eq. 6.3.

$$dbf_{\tau_i}(t) = \sum_{\tau_i^j \in \tau_i} \lfloor \frac{t}{p_i^j} \rfloor \cdot e_i^j \tag{6.3}$$

The sbf of a PRM-based VCPU $VP = (\Pi, \Theta)$ is Eq. 6.4.

$$sbf_{\Omega}(t) = \begin{cases} x \cdot \Theta + \max\{0, t - 2y - x \cdot \Pi\}, & t \geq y \\ 0, & otherwise \end{cases} \tag{6.4}$$

where $x = \lfloor \frac{t-(\Pi-\Theta)}{\Pi} \rfloor$, and $y = \Pi - \Theta$.

The schedulability of tasks on a VCPU can be checked with the following Theorem [57]:

Theorem 6.1. *A task set τ_i is schedulable under EDF on a VCPU with a PRM* *model* Ω iff $dbf_{\tau_i}(t) \leq sbf_{\Omega}(t)$ *for all* $0 < t \leq LCM$*, where* LCM *is the least common multiplier of* p_i^j *for all* τ_i^j *in* τ_i*.*

When a VCPU VP_i's period Π_i is given, we can compute the VCPU's minimum budget to guarantee the schedulability of its tasks by searching each budget in $(0, \Pi_i]$ that satisfies the Theorem 6.1 and choosing the minimum one.

Discussion on abstraction overhead. Abstraction overhead of a VCPU VP_i is the difference between the VCPU's bandwidth and the total utilization of the VCPU's tasks, which is calculated as $\Delta^{abs}_{VP_i} = \frac{\Theta_i}{\Pi_i} - \sum_{\tau_i^j \in \tau_i} \frac{e_i^j}{p_i^j}$.

The abstraction overhead for a PRM-based VCPU can be very high. For example, we have one task $\tau_i^1 = (10, 1)$ scheduled under EDF on a VCPU VP_i with period equal to 10. To schedule the task, the VCPU's minimum budget is 5.5 – calculated by CARTS tool [53] – incurring the abstraction overhead $\Delta^{abs}_{VP_i} = 5.5/10 - 1/10 = 0.45$, which is $0.45/0.1 = 4.5\times$ of the task workload.

The high abstraction overhead is caused by the fact that we do not know the exact resource supply of a PRM-based VCPU and that we have to assume the worst-case resource supply pattern which rarely or even never happens.

We observe that VCPUs with harmonic periods may have well-regulated resource supply patterns, which enables us to remove the abstraction overhead for VCPUs. [27]

6.5.2 Removal of abstraction overhead in PRM-based compositional analysis

We first introduce well-regulated VCPUs before we discuss how to remove abstraction overhead in compositional analysis for the vC^{2}M systems.

[27] The abstraction-free compositional analysis is inspired from the discussion with Jin Hyun Kim.

Well-regulated VCPUs

Definition 6.1. *A VCPU has well-regulated resource supply pattern* iff *the resource supply patterns in each of its periods are the same. We call such a VCPU as a well-regulated VCPU.*

In other words, if a well-regulated VCPU VP_i does (or does not) execute at t_1, the VCPU will (or will not) execute at $t_1 + k \cdot \Pi_i$, where Π_i is the VCPU's period and $k \in N$.

Theorem 6.2. *VCPUs are well-regulated VCPUs if they satisfy the following conditions: (1) they use periodic server mechanism to manage their budgets; (2) their periods are harmonic and their release offsets are the same; (3) they are scheduled under EDF scheduling and are schedulable; (4) the scheduler uses a deterministic priority-tie breaking policy: for two VCPUs with the same deadline, the VCPU with a smaller period has higher priority; if they still have the same priority, the VCPU with a smaller index has higher priority.*

Proof. We will prove each VCPU satisfying the conditions has a well-regulated resource supply.

Because of the condition (1) and (2), the EDF scheduling becomes the fixed-priority scheduling: a VCPU with a smaller period and a smaller index always has higher priority.

For the highest-priority VCPU VP_1, it always executes immediately when it starts a new period. So it has a well-regulated resource supply pattern.

For the second-priority VCPU VP_2, we will prove it has a well-regulated resource supply pattern by contradiction. Suppose t_1 is the first time when the VCPU supplies differently at t_1 and $t_2 = t_1 + k \cdot \Pi_2$, where $k \geq 1$. It has two cases: the VCPU executes at t_1 but does not executes at t_2 and vice versa. In the first case, because the VCPU still has budget at t_1 and t_1 is the first time when the VCPU's supply pattern changes, the VCPU should also have budget at t_2. So the fact that the VCPU

VP_2 does not execute at t_2 is because the highest-priority VCPU VP_1 executes at t_2. The fact that the VCPU VP_2 executes at t_1 indicates that the highest-priority VCPU VP_1 does not execute at t_1. Because VCPUs are harmonic and VP_2 has larger period than VP_1, we have $\Pi_2 = m \cdot \Pi_1$, where $m \in N$ and $m \geq 1$. Since $t_2 = t_1 + k \cdot \Pi_2 = t_1 + k \cdot m \cdot \Pi_1$, the VCPU VP_1 should have the same resource supply at t_1 and t_2, which contradicts to the observation that the VP_1 supplies differently at these two time points. So the first case does not occur. Similarly, we can prove that the second case does not happen either. So it is proved.

By induction, we can prove each VCPU has a well-regulated resource supply. □

VCPUs in vC^{2}M can be configured to well-regulated VCPUs with minor modifications to Xen RTDS scheduler.

The Xen RTDS scheduler, which is used in vC^{2}M for scheduling VCPUs, does not satisfy the condition (4) in Theorem 6.2: the scheduler breaks the priority-tie in an arbitrary order. We add the deterministic priority-tie breaking policy into the Xen RTDS scheduler with 20 lines of change. With the modified RTDS scheduler, the vC^{2}M system satisfies the condition (4).

The Xen RTDS scheduler uses EDF scheduling. By ensuring the total utilization of VCPUs on each core is no larger than 1, vC^{2}M satisfies the condition (3).

The release offsets of VCPUs under the Xen RTDS scheduler are always 0. By configuring VCPUs' periods to be harmonic, vC^{2}M satisfies the condition (2).

Although VCPUs are implemented as deferrable server under the Xen RTDS scheduler, vC^{2}M can run a background CPU-intensive task on each VCPU to turn them to periodic servers as in [69]. The condition (1) is satisfied.

Abstraction overhead-free analysis

We use properties of well-regulated VCPUs to remove the abstraction overhead for the analysis of vC^{2}M.

Lemma 6.3. *A well-regulated VCPU VP_i with a PRM model $\Omega_i = (\Pi_i, \Theta_i)$ always provides Θ_i time in any time interval with length Π_i.*

Proof. We consider a time interval $[t_1, t_1 + \Pi_i]$ for the VCPU VP_i. If t_1 is the start of a period of the VCPU VP_i, the VCPU provides Θ_i time in the time interval, according to the VCPU's definition.

If t_1 is in the middle of a period, we define t_0 as the start of the period where t_i resides and let $t_1 = t_0 + \eta$. We split the time interval $[t_1, t_1 + \Pi_i]$ to two intervals $[t_0+\eta, t_0+\Pi_i]$ and $(t_0+\Pi_i, t_0+\eta+\Pi_i]$. Because the VCPU's resource supply repeats in each period, the resource supply in the second interval $(t_0 + \Pi_i, t_0 + \eta + \Pi_i]$ is the same with the interval $(t_0, t_0 + \eta]$. By combining the interval $(t_0, t_0 + \eta]$ and the interval $[t_0 + \eta, t_0 + \Pi_i]$, we get the interval $[t_0, t_0 + \Pi_i]$, which provides Θ_i time. The VCPU provides the same amount of time in the time interval $[t_0, t_0 + \Pi_i]$ and $[t_1, t_1 + \Pi_i]$. It is proved. $\square$

Lemma 6.4. *The resource supply bound function of a well-regulated VCPU VP_i with a PRM model $\Omega_i = (\Pi_i, \Theta_i)$ is*

$$sbf_{VP_i}(t) = \begin{cases} x \cdot \Theta_i + \max\{t - x \cdot \Pi_i - y, 0\}, & t \geq \Pi_i - \Theta_i \\ 0, & otherwise \end{cases}$$

where $x = \lfloor \frac{t}{\Pi_i} \rfloor$ and $y = \Pi_i - \Theta_i$.

Proof. According to Lemma 6.3, a well-regulated VCPU VP_i always provides Θ_i time for any time interval with length Π_i. Let the start time of t as the start time of a hypothetical resource supply period of VP_i, which has length Π_i but does not necessarily overlap with the period of VP_i. The worst-case resource supply occurs when the VCPU services its resource as late as possible in its hypothetical resource supply period.

When $t < \Pi_i - \Theta_i$, the VCPU provides 0 time, according to the worst-case scenario.

When $t \geq \Pi_i - \Theta_i$, there are $\lfloor \frac{t}{\Pi_i} \rfloor$ full hypothetical periods of VP_i, each of which provides Θ_i time. In the last partial hypothetical period of t, the VCPU provides no resource for $\Pi_i - \Theta_i$ time before it provides continuous resource for the rest of the period. Therefore, the VCPU provides $\max\{t - \lfloor \frac{t}{\Pi_i} \rfloor \cdot \Pi_i - (\Pi_i - \Theta_i), 0\}$ in the last hypothetical period. In total, the VCPU provides $\lfloor \frac{t}{\Pi_i} \rfloor \cdot \Theta_i + \max\{t - \lfloor \frac{t}{\Pi_i} \rfloor \cdot \Pi_i - (\Pi_i - \Theta_i), 0\}$ resource when $t > \Pi_i - \Theta_i$. $\square$

With the tighter resource supply bound function for well-regulated VCPUs, we can remove the abstraction overhead in computing a VCPU's parameters from its tasks' resource demand.

Theorem 6.5. *A harmonic task sets $\tau_i = \{\tau_i^1, ...\tau_i^n\}$, where $\tau_i^j = (p_i^j, e_i^j)$, is schedulable under EDF on a well-regulated VCPU VP_i with a PRM model $\Omega_i = (\Pi_i, \Theta_i)$ if $\Pi_i = \min_{\tau_i^j \in \tau_i} p_i^j$ and $\frac{\Theta_i}{\Pi_i} = \sum_{\tau_i^j \in \tau_i} \frac{e_i^j}{p_i^j}$.*

Proof. We first prove a widget to be used later: for any value a and $k \geq 1$, $k \cdot \lfloor \frac{a}{k} \rfloor \leq \lfloor a \rfloor$. Let $a = k \cdot m + r$, where m is the largest possible integer for which r is nonnegative. We have $\lfloor a \rfloor \geq k \cdot m$. Because $\frac{a}{k} = m + \frac{r}{k}$, we get $m = \lfloor \frac{a}{k} \rfloor$, and $k \cdot m = k \cdot \lfloor \frac{a}{k} \rfloor$. So $\lfloor a \rfloor \geq k \cdot \lfloor \frac{a}{k} \rfloor$.

Recall that the task set τ_i is schedulable if it satisfies Theorem 6.1. We now prove that the dbf of τ_i is always no larger than the sbf of the VCPU VP_i with the PRM model Ω_i.

Because the task set is harmonic and $\Pi_i = \min_{\tau_i^j \in \tau_i} p_i^j$, each task τ_i^j's period is $p_i^j = k_i^j \cdot \Pi_i$, where $k_i^j \in N$, and WCET is $e_i^j = u_i^j \cdot p_i^j = u_i^j \cdot k_i^j \cdot \Pi_i$, where $u_i^j = \frac{e_i^j}{p_i^j}$.

The dbf of the task set τ_i is $\mathsf{dbf}_{\tau_i}(t) = \sum_{\tau_i^j \in \tau_i} \lfloor \frac{t}{p_i^j} \rfloor \cdot e_i^j = \sum_{\tau_i^j \in \tau_i} \lfloor \frac{t}{k_i^j \cdot \Pi_i} \rfloor \cdot u_i^j \cdot k_i^j \cdot \Pi_i^j$. According to the proved widget above, $\mathsf{dbf}_{\tau_i}(t) \leq \sum_{\tau_i^j \in \tau_i} \lfloor \frac{t}{\Pi_i} \rfloor \cdot u_i^j \cdot \Pi_i = \lfloor \frac{t}{\Pi_i} \rfloor \cdot \sum_{\tau_i^j \in \tau_i} u_i^j \cdot \Pi_i = \lfloor \frac{t}{\Pi_i} \rfloor \cdot \frac{\Theta_i}{\Pi_i} \cdot \Pi_i = \lfloor \frac{t}{\Pi_i} \rfloor \cdot \Theta_i$.

The sbf of the well-regulated VCPU VP_i with the PRM model Ω_i satisfies $sbf_{VP_i}(t) \geq \lfloor \frac{t}{\Pi_i} \rfloor \cdot \Theta_i \geq dbf_{\tau_i}(t)$. It is proved. $\square$

Given a vC^{2}M system, which has harmonic tasks and the mapping of tasks to

VCPUs, we can compute each VCPU's parameters with Theorem 6.5. We remove the abstraction overhead because each VCPU's bandwidth is equal to its tasks' total utilization.

6.5.3 Cache-aware analysis

Under the core-level cache partitioning, each core has its own cache area in the LLC. Tasks on the same core use the same LLC area and may evict each other's cached content. Events that cause the cache overhead in the LLC are the same type of events that cause the cache overhead in the private cache discussed in Chapter 3. We first review these cache overhead-causal events before we discuss how to account for their impact on the system's schedulability.

Definition 6.2 (Task-preemption event.). *A task-preemption event of τ_i occurs when a job of another task τ_j on the same VCPU is released and this job preempts the current job of τ_i.*

Definition 6.3 (VCPU-preemption event.). *A VCPU-preemption event of VP_i occurs when VP_i is preempted by a higher-priority VCPU VP_j of another VM.*

Definition 6.4 (VCPU-completion event.). *A VCPU-completion event of VP_i happens when VP_i exhausts its budget in a period and stops its execution.*

When a task τ_i experiences a *task-preemption event*, its cached contents may be evicted by other tasks. When τ_i resumes and accesses the evicted cached contents, it experiences extra cache misses. We call the latency of reloading these evicted cached contents after each overhead-causal event (e.g., task-preemption event) as one-cache-overhead, which is denoted as $\Delta^{crpd}_{\tau_i}$ for the task τ_i. We denote the maximum one-cache-overhead τ_i causes to other tasks as $\delta^{crpd}_{\tau_i}$.

Each job of a task τ_i incurs at most one task-preemption event, which causes at most one cache overhead to another task. We can inflate τ_i's WCET with one maximum cache overhead it may cause to other tasks to safely account for the impact of the task-preemption event as in [74] [24]. The inflated WCET of τ_i is

$$e'_i = e_i + \delta^{crpd}_{\tau_i} \tag{6.5}$$

Theorem 6.6. *A set of tasks $\tau = \{\tau_1, ...\tau_n\}$, where $\tau_k = (p_k, e_k)$, is schedulable under EDF on a VCPU with PRM model in the presence of cache-related overhead if its inflated taskset $\tau' = \{\tau_1,', ...\tau'_n\}$ is schedulable under EDF on a VCPU with PRM model in the absence of cache-related overhead, where $\tau'_k = (p_k, e'_k)$ and e'_k is given by Eq. 6.5*

When a VCPU VP_i causes a *VCPU-preemption event* to another VCPU VP_j, tasks on VP_i can evict cached contents of the currently running task τ_k on VP_j, causing one-cache-overhead to τ_k on VP_j. We specify the maximum one-cache-overhead VP_i causes to other VCPUs at each VCPU-preemption event as $\delta^{crpd}_{vp_i}$.

Each job of a VCPU VP_i causes at most one VCPU-preemption event, which causes at most one cache overhead to another VCPU. Similar to the task-preemption event, we can inflate VP_i's budget to account for the impact of its VCPU-preemption event. The inflated budget of VP_i is

$$\Theta'_i = \Theta_i + \delta^{crpd}_{vp_i} \tag{6.6}$$

When a VCPU VP_i experiences a *VCPU-completion event*, the running task τ_k on VP_i stops and its cached content may be evicted by tasks on other VCPUs. When VP_i resumes execution in the next period, τ_k resumes as well and may experience one-cache-overhead. We denote the maximum one-cache-overhead of the running task on VP_i at a VCPU-completion event as $\Delta^{crpd}_{vp_i}$.

Each job of a VCPU VP_i experiences a task-completion event. When it resumes, it incurs at most $\Delta^{crpd}_{vp_i}$ extra latency, whose impact can be accounted by inflating the VCPU's budget with $\Delta_{vp_i}^{crpd}$ as in [74] [24]. The inflated budget of VP_i is

$$\Theta_i' = \Theta_i + \Delta_{vp_i}^{crpd} \tag{6.7}$$

The inflated budget of VP_i, considering the cache-overhead impact caused by VCPU-preemption and VCPU-completion events, is

$$\Theta_i'' = \Theta_i + \delta_{vp_i}^{crpd} + \Delta_{vp_i}^{crpd} \tag{6.8}$$

Theorem 6.7. *Consider a set of VCPUs $VP = \{VP_1, ...VP_n\}$ scheduled under EDF on a core, where $VP_i = (\Pi_i, \Theta_i)$. Let $VP'' = \{VP_1'', ..VP_n''\}$, where $VP_i'' = (\Pi_i'', \Theta_i'')$ and Θ_i'' is Eq. 6.8 for all $1 \leq i \leq n$. Then VP is schedulable on the core in the presence of cache-related overhead, if the set of inflated VCPUs VP'' is schedulable under EDF in the absence of overhead.*

6.6 Resource allocation algorithm

The allocation algorithm in vC^{2}M integrates the results of two schemes: a VM-level allocation scheme that determines the tasks-to-VCPUs mapping and VCPUs' parameters, and a hypervisor-level allocation scheme that determines the VCPUs-to-cores mapping and each core's allocated resources.

6.6.1 VM-level resource allocation

Basic strategies. Driven by observations for tasks in Section 6.3.4, we propose the following high-level strategies:

Strategy 6.1. *(Group by sensitivity) As tasks on the same VCPU (and on the same core) are always allocated the same amount of cache and bandwidth resources (equal to that of the core), grouping tasks with similar sensitivity to the cache and bandwidth* *resource allocations onto the same VCPU (and onto the same core) can help better utilize the cache and bandwidth resources.*

Towards this, we define a task's *resource-allocation slowdown* under cp_i cache partitions and bw_i bandwidth partitions to be $t\_slowdown_i(cp_i, bw_i) = \frac{e_i(cp_i, bw_i)}{re_i}$, where re_i is the task's reference WCET (defined in Section 6.4).

The next strategy simply aims to balance load across VCPUs. This strategy aims to avoid the pathological situation that some VCPUs are overloaded and hard to be scheduled on any core while others are underloaded.

Strategy 6.2. *(Load balancing) Given an allocation of tasks to VCPUs, evenly distributing the tasks among VCPUs based on the assigned VCPUs' utilizations can help balance the load across VCPUs and eventually avoid under-utilized cores.*

Algorithm 1 Heuristic VM-level resource allocator

```
Input: $\tau$: the set of tasks in a VM, $M$: the number of cores, $maxIterKM$: the
    maximum number of iterations for KMeans.
Output: $V$: the set of VCPUs for the VM.
1: $\tau' \leftarrow accountForTaskOh(\tau)$          ▷ Inflate each task's WCET with Eq. 6.5.
2: $m \leftarrow min\{$the number of tasks, $M\}$
3: $clusters \leftarrow clusterTasks(\tau', m, maxIterKM)$
4: Sort tasks in each cluster in decreasing order of tasks' reference utilization
5: $V \leftarrow binPackTaskClusters(clusters, m)$
6: $calcVCPUParams(V)$
```

Overview of the algorithm. Algorithm 1 shows the high-level idea of our allocation algorithm for grouping tasks to VCPUs. The algorithm works in four phases:

(1) Phase 1 (Line 1): It inflates each task's WCET by using Eq. 6.5 to account for the cache-overhead impact caused by task-preemption events.

(1) Phase 2 (Lines 2–4): It first groups tasks that have similar sensitivity to cache and memory bandwidth into the same cluster, based on Strategy 6.1. Then, it sorts tasks in each cluster in decreasing order of tasks' reference utilization – this is because it is typically harder for a task with higher utilization to find a feasible VCPU.

(2) Phase 3 (Line 5): It packs each task in each cluster onto a VCPU, such that the total reference utilization of tasks on each VCPU is similar (i.e., close to the average reference utilization of all tasks), as guided by Strategy 6.2.

(3) Phase 4 (Line 6): Next it calculates each VCPU's parameters using Theorem 6.5.

Algorithm details. We now discuss the key ideas of the main procedures using in our VM-level resource allocation algorithm: *clusterTasks()* and *binPackTaskClusters()*.

Algorithm 2 clusterTasks(τ, m, $maxIterKM$)

Input: τ: the set of tasks with inflated WCET, m: the number of clusters, $maxIterKM$: the maximum number of iterations for KMeans.
Output: m clusters of tasks

```
1:  Calculate each task's resource-allocation slowdowns
2:  Create $m$ clusters $C$ with $m$ randomly picked tasks as its centroid
3:  repeat
4:      $updated = false$
5:      $maxIterKM = maxIterKM - 1$
6:      for all $\tau_i \in \tau$ do
7:          $min\_distance \leftarrow \infty$
8:          for all $c \in C$ do
9:                                  ▷ $distance(v, c)$ is distance between $\tau_i$ and $c$.
10:             if $distance(\tau_i, c) < min\_distance$ then
11:                 $task\_cluster = c$
12:                 $min\_distance = distance(\tau_i, c)$
13:             if $\tau_i \notin task\_cluster$ then
14:                 $task\_cluster \leftarrow \tau_i$          ▷ Assign $\tau_i$ to $task\_cluster$
15:                 $updated = true$
16:     for all $c \in C$ do
17:         Calculate the mean of all tasks in $c$
18:         Update the new mean as $c$'s new centroid
19: until $updated = false$ or $maxIterKM = 0$
```

The *clusterTasks()* procedure (c.f. Algorithm 2) uses the KMeans algorithm [45] (which is widely used in machine learning for clustering data points with similar features) to cluster tasks that have similar sensitivity to cache and bandwidth resources. Each task τ_j has a N_{config} dimensional slowdown vector $\vec{sv}_j$, where N_{config} is the number of valid resource configurations. (A valid resource configuration is a pair of a valid number of cache partitions and a valid number of memory bandwidth partitions.) The i^{th} element in a task's slowdown vector is the task's resource allocation slowdown under the corresponding resource configuration. Formally, the procedure aims to divide the set of tasks τ into m clusters such that the pairwise deviation of tasks in the same cluster is minimized:

$$\arg\min_{C} \sum_{k=1}^{m} \frac{1}{2|C_k|} \sum_{\tau_i, \tau_j \in C_k} ||s\vec{v}_i - s\vec{v}_j||^2 \tag{6.9}$$

where $|C_k|$ is the number of tasks in the m cluster C_k.

The *clusterTasks()* procedure has three steps: (1) initialization, which calculates each task's slowdown vector and creates an initial set of m clusters; (2) assignment, which assigns each task to the task's closest cluster whose mean has the least square distance to the task; and (3) update, which calculates the new mean of each cluster as the new centroid of the cluster. The algorithm repeats the assignment step and the update step until all clusters' assigned tasks are no longer changed or until after $maxIterKM$ iterations.

The *binPackTaskClusters()* procedure (c.f. Algorithm 3) packs tasks of clusters into m VCPUs such that each VCPU's reference utilization (i.e., the total reference utilizations of all tasks on the VCPU) is similar. The procedure first computes the average reference utilization $meanRefU$ of m clusters, i.e., the total reference utilization of all tasks divided by m. Then it uses our modified first-fit bin-packing algorithm to pack tasks to VCPUs: for each task, it tries to pack it from core 0 to core $m-1$. It packs a task to a VCPU if the VCPU's current reference utilization is smaller than the average reference utilization $meanRefU$ *and* the VCPU's current reference utilization plus the task's reference utilization is no larger than 1. The procedure packs a task to VCPU 0 if it cannot find any VCPU that satisfies the

Algorithm 3 binPackTaskClusters(C, m)

Input: C: clusters of tasks, m: the number of VCPUs
Output: V: m VCPUs that have similar reference utilizations

```
1: Initialize $V$ as $m$ empty VCPUs
2: $sumU \leftarrow 0$
3: for all $c \in C$ do
4:     $util \leftarrow c$                              ▷ Get $c$'s reference utilization
5:     $sumU+ = uil$
6: $meanU \leftarrow sumU/m$
7: for all $c \in C$ do
8:     for all $v \in c$ do
9:         $vU \leftarrow v$                            ▷ Get $v$'s reference utilization
10:        for $i = 0; i < m; i = i + 1$ do
11:            $VP_i \leftarrow V$                      ▷ Get $i^{th}$ VCPU in $V$
12:            $pU \leftarrow VP_i$                     ▷ Get $VP_i$'s reference utilization
13:            if $pU > meanU$ or $pU + vU > 1$ then
14:                continue
15:            else if $pU + vU \leq 1$ then
16:                $chosen \leftarrow VP_i$
17:            else
18:                $chosen \leftarrow VP_0$
19:        $chosen \leftarrow v$                        ▷ Assign $v$ to chosen core
20: return $P$
```

above condition.

Complexity. The VM-level heuristic algorithm's running time is polynomial of the following parameters: n, the number of tasks; m, the number of cores, and $maxIterKM$, the maximum number of iterations in the *clusterTasks()* procedure.

6.6.2 Hypervisor-level resource allocation scheme

Basic strategies. Since tasks on a VCPU have similar resource sensitivity and the total utilization of tasks on a VCPU is always the same with the VCPU's at each possible combination of cache and memory bandwidth allocation, the observations for tasks in Section 6.3.3 also hold for VCPUs. Driven by these observations, we reuse the strategies at the VM-level resource allocation to determine the mapping of VCPUs to cores; and we also propose a new strategy to determine the cache and memory bandwidth allocation to cores:

We define a core i's *resource utility* as the average reduced utilization per newly allocated cache and bandwidth partition for the core:

$$reducedU_i = \begin{cases} (u_i - u'_i)/(cp + bw) & \text{if } u_i > 1 \\ 0 & \text{otherwise} \end{cases} \tag{6.10}$$

where u_i and u'_i are the core's utilization before and after it is allocated for extra cp cache partitions and bw bandwidth partitions, respectively. To balance the load across cores, we will find an allocation that maximizes the resource utility whenever assigning some extra partitions to a core:

Strategy 6.3. *When adding more cache and memory bandwidth resources to a core that is unschedulable under the current allocation, allocating resources to a core that results in the maximum* resource utility *can provide a more effective use of the scarce cache and bandwidth resources.*

Algorithm 4 Heuristic hypervisor-level resource allocator

Input: V: the set of VCPUs, m: the number of cores, N_{cp}: the number of cache partitions, N_{bw}: the number of bandwidth partitions, $maxIterKM$: the maximum number of iterations for KMeans, $maxIterPerm$: the maximum number of iterations for permuting VCPUs.

Output: Schedulable or Unschedulable.

```
 1: accountForVCPUOh(V)                    ▷ inflate each VCPU's budget with Eq. 6.8.
 2: $clusters \leftarrow clusterVCPUs(V, m, maxIterKM)$
 3: Sort VCPUs in clusters in decreasing order of VCPUs' reference utilizations
 4: repeat
 5:     $perm\_clusters \leftarrow permute(clusters)$   ▷ randomly pick one permutation of $clusters$
 6:     $cores \leftarrow binPackVCPUClusters(perm\_clusters, m)$
 7:     $cores \leftarrow allocResource(cores, m, N_{cp}, N_{bw})$   ▷ $cores$ specify VCPUs and resources allocated to each core
 8:     $sched \leftarrow checkSchedulability(cores)$   ▷ schedulable if each core's assigned utilization is no larger than 1
 9:     if $sched$ = schedulable then
10:         break
11:     $oldVal \leftarrow \infty$                          ▷ previous imbalance value
12:     while true do                        ▷ balance $cores$' utilizations iteratively
13:         $val \leftarrow getImbalanceValue(cores)$
14:         $cores \leftarrow balance(cores)$
15:         $cores \leftarrow allocResource(cores)$
16:         $sched \leftarrow checkSchedulability(cores)$
17:         if $sched$ = schedulable or $val > oldVal$ then
18:             break
19:         $oldVal \leftarrow val$
20:     $maxIterPerm \leftarrow maxIterPerm - 1$
21: until $maxIterPerm = 0$
22: return $sched$
23:
24: function getImbalanceValue($cores$)
25:     $imbalance = 0$
26:     for all $c \in cores$ do
27:         if $c$'s assigned utilization > 1 then
28:             $imbalance$ += $c's$ assigned utilization $- 1$
29:     return $imbalance$ rounded to 2 fractional digits
```

Overview of the algorithm. Algorithm 4 shows the high-level idea of our allocation algorithm for m cores. Initially, each core is allocated the minimum number of cache and bandwidth partitions. It then works in four phases:

(1) Phase 1 (Line 1): It inflates each VCPU's budget with Eq. 6.8 to account for the cache-overhead impact of VCPU-preemption and VCPU-completion events.

(2) Phase 2 (Lines 2–3): It packs VCPUs to cores in a similar way as the VM-level resource allocation does. It first groups VCPUs that have similar sensitivity to cache and memory bandwidth into the same cluster and packs VCPUs to cores, such that the total reference utilization of VCPUs on each core is similar.

(3) Phase 3 (Lines 5–10): It allocates the cache and memory bandwidth resources to cores while aiming to maximize the resulting resource utility, based on the Strategy 6.3. Once the resources allocated to each core are determined, it calculates the resulting utilization of each core and checks the system's schedulability. If the system is schedulable, the algorithm terminates and outputs the resource allocation policy that schedules the system; otherwise, it continues to the next phase.

(4) Phase 4 (Lines 11–19): The algorithm tries to balance the VCPU workloads across cores (Line 14). For each unschedulable core, it migrates each of its VCPUs to a schedulable core that will have the smallest utilization after the migration, until the unschedulable core becomes schedulable. After the *balance* procedure finishes, the algorithm re-runs the *allocResource()* procedure for cores and checks if the system becomes schedulable. The algorithm keeps balancing VCPUs on cores until the system becomes schedulable or there is no benefit in balancing (Line 17). Because the order of the clusters may affect the bin packing result (Line 6), which may later affect the resource allocation and balance procedure, the algorithm re-orders the clusters and repeats the bin-packing procedure in Phase 2 and procedures in Phases 3 and 4 (Lines 4–21) for a user-specified constant number (i.e., *maxIterPerm*) before it claims that the system is unschedulable.

We observe that an unschedulable system on m cores may become schedulable when the system has fewer cores. This could happen as fewer cores also means more average cache and memory bandwidth resources available to a core (since each core must have a certain minimum amount of cache and memory bandwidth resources), which could lead to smaller tasks' and VCPUs' utilizations (if the tasks are sensitive to cache and bandwidth resources), hence making the system easier to be scheduled. Based on this observation, we use Algorithm 4 to check the system's schedulability on each valid number of cores m, where $1 \leq m \leq M$, and M is the maximum number of cores supported by the hardware.

Algorithm details. The hypervisor-level resource allocation algorithm (c.f. Algorithm 4 has four main procedures: *clusterVCPUs(), binPackVCPUClusters(), allocResource(), and balance().*

The *clusterVCPUs() and binPackVCPUClusters()* procedures use the same algorithms to map VCPUs to cores at the hypervisor level as the the *clusterTasks() and binPackTaskClusters()* procedures do at the VM level. We get the *clusterVCPUs() and binPackVCPUClusters()* procedures by respectively replacing task(s) with VCPU(s) in the *clusterTasks()* procedure (c.f. Algorithm 2) and the *binPackTaskClusters()* procedure (c.f. Algorithm 3).

We now discuss the key ideas of the other two main procedures: *allocResource()* and *balance()*.

The *allocResource()* procedure (c.f. Algorithm 5) allocates cache and memory bandwidth resources to cores for making the system schedulable with less cache and memory bandwidth resources. The procedure first allocates the minimum number of cache and memory bandwidth partitions to each core. Then the procedure always allocates some or all remaining cache and memory bandwidth partitions to the core that has the maximum *resource utility*, until all cores become schedulable or there is no benefit in reducing an unschedulable core's utilization.

The *balance()* procedure (c.f. Algorithm 6) migrates VCPUs from unschedulable cores to schedulable cores so that cores' utilizations are balanced, which makes it

Algorithm 5 allocResource(C, m, N_{cp}, N_{bw})

```
Input: C: m cores, N_cp: the number of available cache partitions, N_bw: the number of available bandwidth partitions
Output: The number of cache and bandwidth partitions allocated for each core in C that maximizes the reduced resource utilization
1: for all $c \in C$ do
2:     $c \leftarrow (N_{cp}^{min}, N_{bw}^{min})$    ▷ Assign the minimum number of cache partitions and bandwidth partitions to $c$
3: $remainCP = N_{cp} - m \cdot N_{cp}^{min}$
4: $remainBW = N_{bw} - m \cdot N_{bw}^{min}$
5: while $remainCP > 0$ or $remainBW > 0$ do
6:     $maxRU = 0$    ▷ Maximum resource utilization
7:     for all $c \in C$ do
8:         $(cMaxRU, cChosenCP, cChosenBW) \leftarrow getMaxResUtil(c, remainBW, remainCP)$
9:         if $cMaxRU > maxRU$ then
10:            $maxRU = cMaxRU$; $chosenCP \leftarrow cChosenCP$
11:            $chosen \leftarrow c$; $chosenBW \leftarrow cChosenBW$
12:    if $chosenCP = 0$ and $chosenBW = 0$ then
13:        break    ▷ Stop due to no schedulability benefit
14:    /* Allocate found resource to $chosen$ cluster */
15:    $chosen \leftarrow chosenCP$; $chosen \leftarrow chosenBW$
16:    $remainCP -= chosenCP$; $remainBW -= chosenBW$
17: return $C$
18:
19: function getMaxResUtil($c$, $remainBW$, $remainCP$)
20:     for $bw \leftarrow 0$ to $remainBW$ do
21:         for $cp \leftarrow 0$ to $remainCP$ do
22:             $curU \leftarrow c$    ▷ $c$'s utilization under currently allocated resource
23:             $newU \leftarrow c$    ▷ $c$'s utilization if $c$ is allocated for extra $cp$ cache and $bw$ bandwidth partitions
24:             if $curU \leq 1$ or $cp + bw = 0$ then
25:                 $resU = 0$    ▷ No benefit for schedulability
26:             else
27:                 $resU = (curU - newU)/(cp + bw)$
28:             if $resU \geq cMaxRU$ then
29:                 $cMaxRU \leftarrow resU$
30:                 $cChosenCP \leftarrow cp$; $cChosenBW \leftarrow bw$
31:     return (cMaxRU, cChosenCP, cChosenBW)
```

Algorithm 6 balance(C)

```
Input: $C$: $m$ cores whose number of cache and bandwidth partitions is determined
Output: Cores whose assigned utilizations are similar
 1: Create $uC$ which holds all unschedulable cores in $C$
 2: Sort $uC$ in decreasing order of cores' assigned utilizations
 3: Sort VCPUs in each core in increasing order of VCPU's assigned slowdown =
    assigned utilization / reference utilization
 4: for all $uc \in uC$ do
 5:     for all $v \in uc$ do
 6:         $curU \leftarrow v$                          ▷ $v$'s assigned utilization in $uc$
 7:         /* Find a core with smallest assigned utilization after $v$ is moved to the
    core */
 8:         for $c \in C$ do
 9:             if  $c = c$ then
10:                 $continue$
11:             $util \leftarrow c \cup v$               ▷ $c$'s assigned utilization if $v$ moves to $c$
12:             if $util < min\_util$ then
13:                 $min\_util \leftarrow util$; $dst \leftarrow c$
14:         $dst \leftarrow v$                               ▷ Move $v$ to $dst$ core
15:         Update assigned utilization for $uc$ and $dst$ core
16:         if  $uc$'s assigned utilization $\leq 1$ then
17:             break
18: return $C$                           ▷ Cores' assigned utilization are balanced
```

easier to schedule the system. The input of the procedure is m cores whose VCPUs and the number of cache and memory bandwidth partitions have been determined. The procedure first sorts VCPUs on unschedulable cores in increasing order of VCPUs' assigned slowdowns, each of which is a VCPU's assigned utilization divided by the VCPU's reference utilization. For each unschedulable core, the procedure migrates each of its sorted VCPUs to the core that has the smallest assigned utilization after the VCPU is migrated, until the unschedulable core becomes schedulable. The procedure terminates after all unschedulable cores in the input become schedulable. Note that the schedulable cores in the input may become unschedulable after this procedure, in which case the heuristic algorithm will call the *allocResource()* procedure to re-allocate resources to cores (c.f. Algorithm 4).

Complexity. The hypervisor-level heuristic algorithm's running time is polynomial of the following parameters: n, the number of VCPUs; m, the maximum number of cores; N_{cp}, the maximum of cache partitions; N_{bw}, the maximum number of memory bandwidth partitions; $\lceil U \rceil$, where U is the total utilization of all VCPUs under the minimum number of cache and memory bandwidth resources; $maxIterKM$, the maximum number of iterations in the *clusterVCPUs()* procedure; and $maxIterPerm$, the maximum number of iterations in the VMM-level heuristic algorithm.

6.7 Performance evaluation

To evaluate the effectiveness and efficiency of our resource allocation algorithm, we conducted an extensive set of experiments using randomly generated real-time workloads. We had three main objectives: (i) to evaluate the performance of our algorithm in terms of schedulability; (ii) to investigate the impact of platforms' and tasks' parameters on the schedulability performance; and (iii) to evaluate the efficiency of our algorithm.

For comparison, we also performed the same set of experiments for four other solutions: (i) a baseline algorithm that does not use cache to avoid cache-related overhead; (ii) an evenly-partitioned algorithm that evenly distributes the cache and memory bandwidth to cores and uses the abstract overhead-free analysis; (iii) a strawman algorithm that use our proposed heuristic resource allocation algorithm but uses the original compositional analysis that has abstraction overhead, and (iv) a flattened algorithm that directly manage the resources for tasks and has close-to-optimal schedulability performance in native environment [73].

6.7.1 Experimental setup

Workload. Each workload contained a number of randomly generated periodic tasksets. The tasks' periods were harmonic and uniformly distributed in [100, 1100] [41]. A task has a Provisioned Execution Time (PET) [38], which is the task's estimated WCET when the task does not use cache and uses the worst-case memory bandwidth. A task's *normalized utilization* is the task's PET divided by its period. The tasks' normalized utilizations followed one of four distributions: a uniform distribution within the range $[0.1, 0.4]$ and three bimodal distributions, where the utilizations were distributed uniformly over either $[0.1, 0.4]$ or $[0.5, 0.9]$, with respective probabilities of 8/9 and 1/9 (light), 6/9 and 3/9 (medium), and 4/9 and 5/9 (heavy). [28] Without further specification, the tasks' normalized utilizations followed the uniform distribution within the range $[0.1, 0.4]$.

The tasks' workloads were randomly selected from one of the resource-sensitive PARSEC benchmarks (e.g., canneal and streamcluster). A task's *reference WCET* is the task's PET divided by the task's no-cache slowdown, which was obtained by profiling the PARSEC benchmarks in Section 6.3.2. A task's *reference utilization* is the task's reference WCET divided by its task. The task's slowdown values under different cache and bandwidth configurations were assigned to be the same with the

[28]The bimodal distribution probabilities are similar to the ones used in [24].

slowdown values of the corresponding benchmark, which were obtained by profiling on our prototype. The task's WCET values under different cache and bandwidth configurations were computed as the product of the task's reference utilization and the task's corresponding slowdown.

We profiled the execution time of different PARSEC benchmarks with simlarge input under different cache and bandwidth configurations using our prototype on our empirical evaluation machine (c.f. Section 6.3). For each PARSEC benchmark, we dedicated one core for the benchmark, configured the core with a valid cache and bandwidth configuration, and measured the execution time of the benchmark for 25 runs. The valid number of cache partitions was ranged from 2 to 20, with a step of 1; the valid number of bandwidth partitions was ranged from 1 to 20, with a step of 1. The set of valid cache and bandwidth configurations is a cartesian product of the valid number of cache partitions and the valid number of bandwidth partitions. For each PARSEC benchmark, we measured its execution time under $19 \times 20 = 380$ valid cache and memory bandwidth configurations and calculated its slowdowns. The obtained slowdowns were used for the tasks, as explained above.

Platform configurations. We analyzed the above generated workloads for three platform configurations (based on the Intel Xeon 2618v3, Intel Xeon D-1528, and Intel Xeon D-1518 processors, respectively): Platform A has 4 cores and 20 cache partitions; Platform B has 6 cores and 20 cache partitions; and Platform C has 4 cores and 12 cache partitions. The number of memory bandwidth partitions is the same as the number of cache partitions on each platform.

Baseline algorithm. The baseline algorithm uses tasks' calculated execution time without cache – the execution time we calculated by assuming the task has no cache and uses the worst-case memory bandwidth as we did in Seciton 6.3.2–for the resource allocation. At the VM level, the algorithm uses the best-fit bin-packing algorithm to pack tasks to VCPUs and then computes each VCPU's budget and period by using the original compositional analysis in [57] with the CARTS tool [53]. At the hypervisor level, the algorithm uses the best-fit bin-packing algorithm to pack VCPUs to cores. If the total assigned utilization of VCPUs on each core is no larger than 1, the system is deemed schedulable; otherwise, it is deemed unschedulable.

Evenly-partitioned algorithm. The evenly-partitioned algorithm evenly distributes cache and memory bandwidth resources to cores. Each core has $\lfloor N_{cp}/m \rfloor$ cache partitions and $\lfloor N_{bw}/m \rfloor$ memory bandwidth partitions, where N_{cp} is the total number of cache partitions, N_{bw} is the total number of bandwidth partitions, and m is the number of cores on the platform. The WCET of a task is the execution time of the generated task under $\lfloor N_{cp}/m \rfloor$ cache partitions and $\lfloor N_{bw}/m \rfloor$ bandwidth partitions. If a task's assigned WCET is larger than its period, the taskset is immediately deemed unschedulable. Similar to the baseline algorithm, the algorithm uses the best-fit bin-packing algorithm to pack tasks to VCPUs at the VM level and to pack VCPUs to cores at the hypervisor level. Different from the baseline algorithm, the algorithm computes the VCPUs' budgets and periods using the abstraction overhead-free compositional analysis in Section 6.5.

By comparing our proposed algorithm against this evenly-partitioned algorithm, we can understand the schedulability benefit of our proposed heuristic resource allocation algorithm.

Strawman algorithm. The strawman algorithm uses the same heuristic resource allocation algorithm in Section 6.6 to determine the resource allocation. Different from our proposed algorithm, the algorithm uses the original compositional analysis in [57], instead of the improved analysis in Section 6.5, to compute the VCPUs' budgets. A VCPU's period is set to half of the minimum period of tasks on the VCPU. [29]

By comparing our proposed algorithm against this strawman algorithm, we can understand the benefit of our improved abstraction-free analysis.

Flattened algorithm. The flattened algorithm removes the hypervisor layer and

[29]An interesting future work is to explore the optimal period for each VCPU under the strawman solution.

directly manages the resources for tasks. It treats each task as a VCPU and uses the hypervisor-level resource allocation scheme (Section 6.6.2) to allocate the CPU, cache and memory bandwidth resources directly to tasks. As shown in our previous study [73], which compared the flattened algorithm with an optimal mixed-integer programming based solution, the flattened algorithm has close-to-optimal performance in terms of system schedulability.

Analysis. We analyzed the same set of tasksets for each of the five algorithms: our algorithm (Heuristic), the baseline algorithm (Baseline), the evenly-partitioned algorithm (Evenly-partition), the strawman algorithm (Strawman), and the flattened algorithm (Flattened). Our analyses were performed on an Intel Xeon E5-2683 v4 processor, which has 32 cores (with hyper threading enabled) operating at 2.10GHz.

6.7.2 Schedulability performance

We generated tasksets with taskset's reference utilization (defined in Section 6.7.1) ranging from 0.1 to 2, with a step of 0.05. For each taskset reference utilization, we generated 50 independent tasksets (i.e., 1950 tasksets in total), with tasks' normalized utilizations uniformly distributed in $[0.1, 0.4]$. We analyzed the tasksets for Platform A using the five algorithms. Fig. 6.6 shows the fraction of schedulable tasksets under each solution.

The results show that the fraction of schedulable tasksets under our algorithm is very close to that of the flattened algorithm. During the taskset's reference utilization range $[0.1, 1.3]$, all tasksets that are schedulable under the flattened algorithm are schedulable under our algorithm. Among all generated tasksets, only 100 out of 1950 tasksets (5%) are schedulable under the flattened algorithm but are unschedulable under our algorithm. The results also show that our algorithm significantly outperforms the baseline algorithm. The tasksets' reference utilization after which tasksets start to become unschedulable is 0.5 under the baseline algorithm, while it is 1.3 under our algorithm. This shows that our algorithm can increase system's

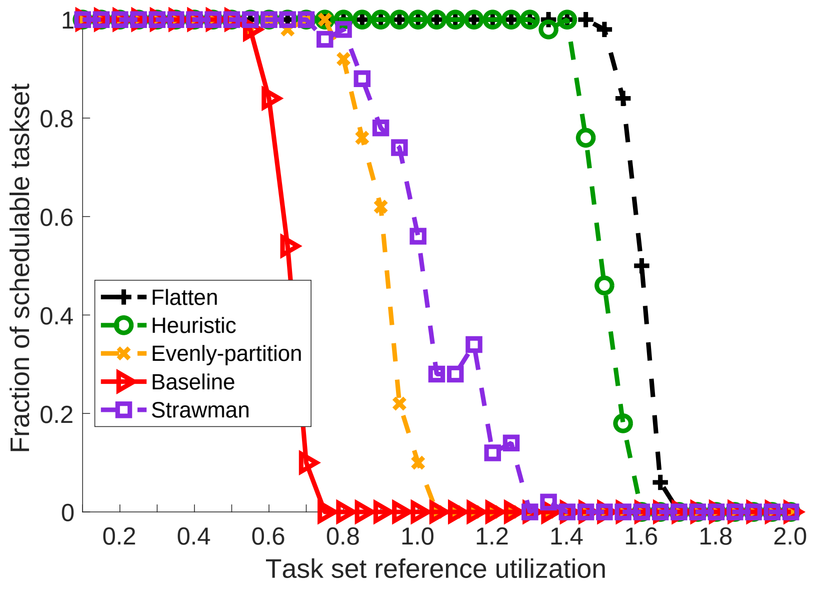
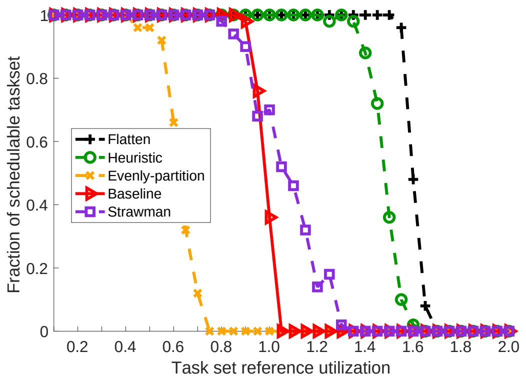


Figure 6.6: Performance on Platform A.

workload by 1.3/0.5 = 2.6× without sacrificing the system's schedulability.

We also observe that we must combine the abstraction-free analysis and the heuristic resource allocation algorithm together to get the satisfied performance. If we only use the heuristic resource allocation algorithm, whose result is shown as the strawman algorithm in purple line in Fig. 6.6, the abstraction overhead of computing VCPUs' parameters is too high, making VCPUs hardly schedulable at the hypervisor level even when the taskset's reference utilization is very small (i.e., 0.7). If we only use the abstraction-free analysis to remove the abstraction overhead, whose result is shown as the evenly-partition algorithm in yellow line in Fig. 6.6, the cache and memory bandwidth resource are used ineffectively, making the system become unschedulable when the taskset's reference utilization is larger than 0.5. By combining both proposed techniques, our algorithm significantly increases the taskset's reference utilization up to 1.3, which is 1.3/0.5 = 2.6× over the evenly-partition algorithm ,without sacrificing the system's schedulability.


(a) *Platform B*

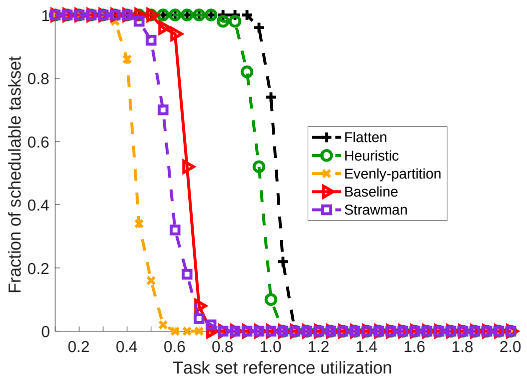
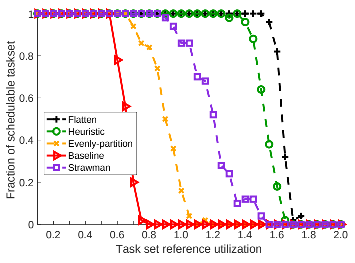
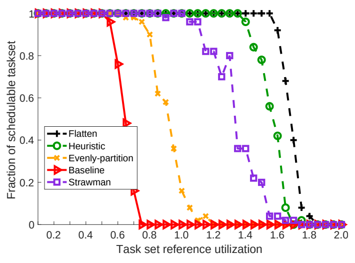

(b) *Platform C*

Figure 6.7: Performance for different platforms.


(a) *Bimodal light*


(b) *Bimodal medium*

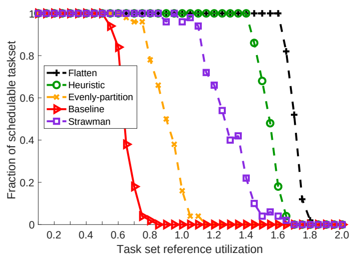

(c) *Bimodal heavy*

Figure 6.8: Performance for different taskset utilization distributions.

6.7.3 Impact of platform configurations and task parameters

We investigated the impact of the platform configurations and the tasks' parameters on the fraction of schedulable tasksets for all three algorithms. For this, we repeated the above experiment on the remaining two platforms (i.e., Platforms B and C), as well as using tasksets with the bimodal-light, bimodal-medium and bimodal-heavy utilization distributions.

The results for Platform B and Platform C are shown in Fig. 6.7. We observe that our algorithm performs close to the flattened algorithm on different platforms. We also observe that the more powerful (e.g., more cores) the platform is, the more performance benefit our algorithm is over the other three algorithms (i.e., Baseline, Evenly-partitioned, and Strawman).

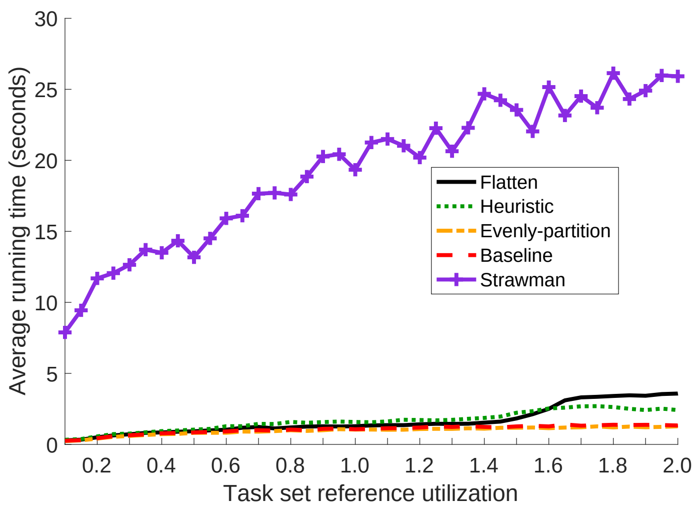


Figure 6.9: Average computation time.

The results for tasksets with the three bimodal distributions are illustrated in Fig. 6.8. Our observation that our heuristic solution always performs very close to the flattened algorithm while significantly outperforming the other three algorithms still hold across different taskset utilization distributions.

6.7.4 Running time efficiency

We measured the computation time of all five algorithms in the evaluation in Section 6.7.2. We observed that our algorithm can efficiently analyze the schedulability of a system: its maximum average running time is less than 3 seconds. We also observed that our algorithm needed at most 0.40 extra second to determine the resource allocation for a taskset than the flattened algorithm did. This is because our algorithm needs to compute resource allocation at two levels while the flattened algorithm needs to compute only one level. When the tasksets became hardly schedulable under large taskset reference utilization (i.e., 1.65), our algorithm took less time in average – and up to 1.38 second – to determine a taskset is unschedulable than the flattened algorithm did.

We also observed that both our algorithm and the flattened algorithm took more time than the baseline algorithm and the evenly-partition algorithm did because the former two algorithms tried different resource allocations. The strawman algorithm took considerable amount of computation time because the running time of the original compositional analysis used by the algorithm was known as pseudo-polynomial to the least common multiplier of tasks' periods, which could be very large. This also demonstrates that our improved abstraction overhead-free analysis not only improves the schedulability but also reduces the computation time for harmonic tasks.

6.8 Conclusion

We have presented a holistic framework called $\mathsf{vC^2M}$ for the co-allocation of CPU, cache, and memory bandwidth resources on multicore virtualization systems. $\mathsf{vC^2M}$ provides a mechanism for memory bandwidth regulation with minimal run-time overhead, as well as an effective and efficient resource allocation algorithm. We have shown through extensive evaluations on our prototype that $\mathsf{vC^2M}$ can effectively mitigate interference among concurrent running tasks and thus substantially improve tasks' WCETs. In addition, by proposing an abstraction-free compositional analysis and considering the interdependence among multiple resource types, its allocation solution offers close-to-optimal schedulability performance while being highly efficient, and it outperforms a baseline approach in both metrics.

Chapter 7

Conclusions

7.1 Conclusion

In this thesis, we have presented novel system and analysis approaches to address predictable performance challenges associated with cache-related resources for multicore virtualization systems. We have answered two questions fundamental to providing the predictable performance to tasks on cache-based multicore virtualization systems: (i) can the timing requirements be satisfied under the cache interference; (ii) how to manage the cache to mitigate the cache interference.

In Chapter 3, we present a cache-aware compositional analysis to analyze the impact of private cache on systems' predictable performance. We characterize different types of events that cause cache misses in the presence of virtualization. We have developed two approaches, task-centric and model-centric, for analyzing the cache-related overhead and for testing the schedulability of components in the presence of cache overhead. Our evaluation on synthetic workloads shows that the model-centric approach achieves significant resource savings compared to the task-centric approach (which is based on WCET inflation).

In Chapter 4, we present $\mathsf{gFP}_{\mathsf{ca}}$, a cache-aware variant of the global preemptive fixed-priority (gFP) algorithm to mitigate the shared cache interference among tasks inside the same OS. The $\mathsf{gFP_{ca}}$ algorithm dynamically allocates non-overlapped shared cache areas to running tasks to mitigate the interference of running tasks and to improve the cache utilization for better system performance. We have developed a cache-aware analysis to reason about the real-time performance of tasks under the $\mathsf{gFP_{ca}}$ algorithm. We have also implemented the algorithm in LITMUS^{RT}, a Linux-based operating systems. Our evaluations, using overhead data from real measurements on our implementation, show that $\mathsf{gFP_{ca}}$ improves schedulability substantially compared to the cache-agnostic gFP, and it outperforms the existing cache-aware nFPca in most cases.

In Chapter 5, we present vCAT, a dynamic cache management framework for virtualization systems that can deliver strong shared cache isolation at both VM and task levels, and that can be configured for both static and dynamic allocations. vCAT virtualizes the Intel CAT in software for achieving hypervisor- and VM-level cache allocations. To illustrate the feasibility of our approach, we provide a proof-of-concept prototype of vCAT on top of Xen and LITMUS^{RT}. We conduct extensive evaluations to demonstrate that vCAT incurs reasonably small overhead and that vCAT significantly improve systems' real-time performance compared to no cache management and static cache management.

In Chapter 6, we propose $\mathsf{vC^2M}$, a holistic solution towards timing isolation in multicore virtualization systems. $\mathsf{vC^2M}$ develops an abstraction-overhead free compositional analysis for multicore virtualization systems and proposes a novel heuristic resource allocation algorithm that allocates CPU, shared cache, and memory bandwidth in a holistic manner to tasks. We have implemented a Xen-based prototype of $\mathsf{vC^2M}$. Our evaluation shows that $\mathsf{vC^2M}$ can be implemented with minimal overhead and that $\mathsf{vC^2M}$ can significantly improve systems' real-time performance compared to approaches that consider only one type of resources.

7.2 Future research directions

Timing is critical for cyber-physical systems (CPS). We believe that the work in this dissertation can be used to achieve the predictable performance for CPS. Yet, there still exists a plenty of future work in this area.

Better real-time virtualization. Shared hardware resources introduce the interference among tasks that impedes systems' predictable performance. This work focuses on solving the challenges introduced by cache-related resources, i.e., private cache, shared cache and memory bus. Yet, systems' real-time performance may still be affected by the interference from other shared hardware resources. For example, the resource contention on cache Miss Status Holding Registers (MSHR) can significantly increase a task's execution time on some hardware types [64]. To provide better predictable performance, we need to regulate tasks' access to these shared hardware resources, such as MSHR and GPU, and provide analysis to account for the extra delay caused by the resource regulation. Regulating these resources can be challenging because system software may not have direct control of these hardware resources. One solution to this challenge would be controlling tasks' execution progress to control tasks' accesses to shared resources, similar to how our vC^{2}M controls memory bandwidth for each core in Chapter 6.

Shared software resources, such as shared memory and shared I/O buffer, also introduces potential contention among tasks that can hurt systems' predictable performance. This work only focuses on independent tasks that have no shared software resources. To push this work applicable to a broader range of applications that may have shared software resources, we need to develop resource-sharing protocols and corresponding analysis techniques for tasks in virtualization systems.

Real-time edge computing. Edge computing is a distributed and localized cloud computing system that performs computation at the edge of network, near the source of data. It is a promising technique to provide more computation power, higher network throughput, and lower response latency for safety-critical CPS, such as connected cars [9]. The service scenarios of edge computing, such as intelligent driving and vehicle-to-cloud cruise control, share the following properties: multiple tenants share the same edge and require real-time performance for their safety-critical applications.

Real-time virtualization is a promising technique to provide real-time performance to tasks on a single edge. Yet, to provide real-time performance to edge-based services, we must extend the resource management and analysis framework to a distributed setting. One preliminary solution would be applying the resource management techniques for real-time cloud [67] to edge computing. However, this solution is not good enough. Achieving real-time performance in edge computing is more challenging than in cloud because edges–which is similar to a localized cloud–are distributed geographically and users of edges (such as vehicles) move across edges. Solving these challenges would help safety-critical CPS benefit from edge computing.

Bibliography

[1] Cache monitoring technology and cache allocation technology. `https://github.com/01org/intel-cmt-cat`. Accessed: 2015-11-19.

[2] Intel 64 and ia-32 architectures software developer's manuals. `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`. Accessed: 2015-10-27.

[3] MSR-tools. `https://01.org/msr-tools`. Accessed: 2016-02-01.

[4] RTDS-based-scheduler. `http://xenbits.xen.org/docs/unstable/misc/xl-psr.html`. Accessed: 2017-10-5.

[5] Tracing with LITMUS^{RT}. `http://www.cs.unc.edu/~anderson/litmus-rt/doc/tracing.html`. Accessed: 2015-10-15.

[6] x86: intel cache allocation technology support. `http://lwn.net/Articles/622893/`. Accessed: 2015-01-09.

[7] Xen security advisory xsa-163: virtual pmu is unsupported. `http://xenbits.xen.org/xsa/advisory-163.html`. Accessed: 2017-10-01.

[8] Restraint device : Supplemental restraint system (srs) airbag. `http://www.toyota-global.com/innovation/safety_technology/safety_technology/technology_file/passive/airbag.html`, 2017. Accessed: 2017-11-01.

[9] Automotive edge computing consortium white paper. https://aecc.org/, 2018. Accessed: 2018-04-16.

[10] A. Agrawal, G. Fohler, J. Freitag, J. Nowotsch, S. Uhrig, and M. Paulitsch. Contention-Aware Dynamic Memory Bandwidth Isolation with Predictability in COTS Multicores: An Avionics Case Study. In *ECRTS*, 2017.

[11] S. Altmeyer, R. I. Davis, and C. Maiza. Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. *Real-Time Systems*, 48(5):499–526, 2012.

[12] S. Altmeyer, R. Douma, W. Lunniss, and R. Davis. Evaluation of cache partitioning for hard real-time systems. In *ECRTS*, 2014.

[13] S. Altmeyer and C. Maiza Burguière. Cache-related preemption delay via useful cache blocks: Survey and redefinition. *Journal of Systems Architecture*, 57(7):707–719, Aug. 2011.

[14] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, 2003.

[15] S. Baruah and T. Baker. Schedulability analysis of global EDF. *Real-Time Systems*, 38(3):223–235, 2008.

[16] S. Baruah and N. Fisher. Component-based design in multiprocessor real-time systems. In *ICESS*, 2009.

[17] A. Bastoni, B. B. Brandenburg, and J. H. Anderson. Cache-Related Preemption and Migration Delays: Empirical Approximation and Impact on Schedulability. In *OSPERT 2010*, Brussels, Belgium, 2010.

[18] S. Basumallick and K. Nilsen. Cache issues in real-time systems, 1994.

[19] N. Beckmann and D. Sanchez. Jigsaw: Scalable software-defined caches. In *PACT*, 2013.

[20] M. Bertogna, M. Cirinei, and G. Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *Parallel and Distributed Systems, IEEE Transactions on*, 20(4):553–566, 2009.

[21] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, 2008.

[22] B. Brandenburg and J. Anderson. On the implementation of global real-time schedulers. In *RTSS*, 2009.

[23] B. Brandenburg, J. Calandrino, and J. Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *RTSS*, 2008.

[24] B. B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.

[25] B. B. Brandenburg and J. H. Anderson. Feather-trace: A light-weight event tracing toolkit, 2007.

[26] J. Busquets-Mataix, J. Serrano, R. Ors, P. Gil, and A. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *RTAS*, 1996.

[27] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson. LITMUS^{RT}: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In *RTSS*, 2006.

[28] A. M. Dani, B. Amrutur, and Y. N. Srikant. Toward a scalable working set size estimation method and its application for chip multiprocessors. *IEEE Transactions on Computers*, 63(6):1567–1579, June 2014.

[29] F. M. David, J. C. Carlyle, and R. H. Campbell. Context switch overheads for linux on arm platforms. In *ExpCS*, 2007.

[30] A. Easwaran, M. Anand, and I. Lee. Compositional analysis framework using edp resource models. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, RTSS '07, 2007.

[31] A. Easwaran, I. Shin, and I. Lee. Optimal virtual cluster-based multiprocessor scheduling. *Real-Time Syst.*, 43(1):25–59, Sept. 2009.

[32] N. Guan, M. Stigge, W. Yi, and G. Yu. Cache-aware scheduling and analysis for multicores. In *EMSOFT*, 2009.

[33] L. Ju, S. Chakraborty, and A. Roychoudhury. Accounting for cache-related preemption delay in dynamic priority schedulability analysis. In *DATE*, 2007.

[34] S. Kato, Y. Ishikawa, and R. R. Rajkumar. CPU scheduling and memory management for interactive real-time applications. *Real-Time Syst.*, 47(5):454–488, Sept. 2011.

[35] R. E. Kessler and M. D. Hill. Page placement algorithms for large real-indexed caches. *ACM TOCS*, 10(4):338–359, Nov. 1992.

[36] H. Kim, A. Kandhalu, and R. R. Rajkumar. A coordinated approach for practical OS-level cache management in multi-core real-time systems. In *ECRTS*, 2013.

[37] H. Kim and R. R. Rajkumar. Real-time cache management for multi-core virtualization. In *EMSOFT*, 2016.

[38] N. Kim, B. C. Ward, M. Chisholm, C. Y. Fu, J. H. Anderson, and F. D. Smith. Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. In *RTAS*, 2016.

[39] T. Kim, M. Peinado, and G. Mainar-Ruiz. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *USENIX Security*, 2012.

[40] C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Trans. Comput.*, 47(6):700–713, June 1998.

[41] J. Lee, S. Xi, S. Chen, L. T. X. Phan, C. Gill, I. Lee, C. Lu, and O. Sokolsky. Realizing compositional scheduling through virtualization. In *RTAS*, 2012.

[42] C. Li, C. Ding, and K. Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, 2007.

[43] G. Lipari and E. Bini. A framework for hierarchical scheduling on multiprocessors: From application requirements to run-time allocation. In *RTSS*, 2010.

[44] 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.

[45] S. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–137, March 1982.

[46] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving resource efficiency at scale. In *ISCA*, 2015.

[47] W. Lunniss, S. Altmeyer, C. Maiza, and R. Davis. Integrating cache related pre-emption delay analysis into edf scheduling. In *RTAS*, April 2013.

[48] W. Lunniss, R. I. Davis, C. Maiza, and S. Altmeyer. Integrating cache related pre-emption delay analysis into edf scheduling. In *RTAS*, 2013.

[49] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *RTAS*, 2013.

[50] A. Marchand, P. Balbastre, I. Ripoll, M. Masmano, and A. Crespo. Memory resource management for real-time systems. In *ECRTS*, 2007.

[51] J. C. Mogul and A. Borg. The effect of context switches on cache performance. In *ASPLOS*, 1991.

[52] F. Mueller. Compiler support for software-based cache partitioning. In *LCTES*, 1995.

[53] L. T. X. Phan, J. Lee, A. Easwaran, V. Ramaswamy, S. Chen, I. Lee, and O. Sokolsky. Carts: A tool for compositional analysis of real-time systems. *SIGBED Rev.*, 8(1):62–63, Mar. 2011.

[54] L. T. X. Phan, M. Xu, J. Lee, I. Lee, and O. Sokolsky. Overhead-aware compositional analysis of real-time systems. In *RTAS*, 2013.

[55] L. Sha, J. P. Lehoczky, and R. Rajkumar. Solutions for Some Practical Problems in Prioritized Preemptive Scheduling. In *RTSS*, 1986.

[56] I. Shin, A. Easwaran, and I. Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *ECRTS*, 2008.

[57] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *RTSS*, 2003.

[58] J. Stärner and L. Asplund. Measuring the cache interference cost in preemptive real-time systems. In *LCTES*, 2004.

[59] J. Staschulat, S. Schliecker, and R. Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *ECRTS*, 2005.

[60] Y. Tan and V. Mooney. Timing analysis for preemptive multitasking real-time systems with caches. *ACM TECS*, 6(1), Feb. 2007.

[61] H. Tomiyama and N. Dutt. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. In *CODES*, 2000.

[62] D. Tsafrir. The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In *Experimental Computer Science on Experimental Computer Science*, 2007.

[63] P.-A. Tsai, N. Beckmann, and D. Sanchez. Nexus: A new approach to replication in distributed shared caches. In *PACT*, 2017.

[64] P. K. Valsan, H. Yun, and F. Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *RTAS*, 2016.

[65] B. Ward, J. Herman, C. Kenna, and J. Anderson. Making shared caches more predictable on multicore platforms. In *ECRTS*, 2013.

[66] B. C. Ward, A. Thekkilakattil, and J. H. Anderson. Optimizing preemption-overhead accounting in multiprocessor real-time systems. In *RTNS*, 2014.

[67] S. Xi, C. Li, C. Lu, C. D. Gill, M. Xu, L. T. X. Phan, I. Lee, and O. Sokolsky. Rt-open stack: Cpu resource management for real-time cloud computing. In *IEEE Cloud*, 2015.

[68] S. Xi, J. Wilson, C. Lu, and C. Gill. RT-Xen: towards real-time hypervisor scheduling in xen. In *EMSOFT*, 2011.

[69] S. Xi, M. Xu, C. Lu, L. T. X. Phan, C. Gill, O. Sokolsky, and I. Lee. Real-time multi-core virtual machine scheduling in xen. In *EMSOFT*, 2014.

[70] 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 *RTAS*, 2016.

[71] 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 *Technical Report*, 2016.

[72] M. Xu, L. T. X. Phan, H.-Y. Choi, and I. Lee. vcat: Dynamic cache management using cat virtualization. In *RTAS*, 2017.

[73] M. Xu, L. T. X. Phan, H.-Y. Choi, and Y. Lin. Holistic resource allocation for multicore real-time systems. In *Technical Report*, 2018.

[74] M. Xu, L. T. X. Phan, I. Lee, O. Sokolsky, S. Xi, C. Lu, and C. Gill. Cache-aware compositional analysis of real-time multicore virtualization platforms. In *RTSS*, 2013.

[75] Y. Ye, R. West, J. Zhang, and Z. Cheng. Maracas: A real-time multicore vcpu scheduling framework. In *RTSS*, 2016.

[76] H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni. PALLOC: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *RTAS*, 2014.

[77] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *RTAS*, 2013.

[78] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory bandwidth management for efficient performance isolation in multi-core platforms. *IEEE Transactions on Computers*, 65(2):562–576, Feb 2016.

[79] W. Zhao, X. Jin, Z. Wang, X. Wang, Y. Luo, and X. Li. Low cost working set size tracking. In *USENIX ATC*, 2011.