Summer 9-1-2014

Real-Time Virtualization and Cloud Computing

Sisu Xi
Washington University in St. Louis

Washington University in St. Louis

School of Engineering and Applied Science

Department of Computer Science and Engineering

Dissertation Examination Committee:
Chenyang Lu, Chair
Christopher D. Gill, Co-Chair
Kunal Agrawal
Mark J. Jakiela
Insup Lee
Jonathan S. Turner

Real-Time Virtualization and Cloud Computing

by

Sisu Xi

A dissertation presented to the Graduate School of Arts and Sciences
of Washington University in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

August 2014
Saint Louis, Missouri

Contents

List of Figures

List of Tables

Acknowledgments

My greatest thanks go to my advisors: Chenyang Lu and Christopher D. Gill. Professor Lu helped me get into this wonderful Ph.D program in Washington University in St. Louis, and together with Professor Gill and many other people, helped me finish. Without them, none of the work described in this dissertation would have been possible. Professor Lu and Professor Gill have been amazing advisors, who introduced me to the real-time area, encouraged me to become a systems researcher, inspired me with invaluable insights, and guided me in finishing this dissertation. Their enthusiasm for exploring the unlimited possibilities in research has led me to discover the insights presented in this dissertation.

I would like to thank my colleagues, who provided invaluable input and contributions to this work. They are: Meng Xu, Chong Li, Jaewoo Lee, Linh T.X. Phan, Oleg Sokolsky, and Insup Lee. Meng, Chong, Linh and Jaewoo, every research meeting with you in particular brought new ideas and knowledge to me. Insup also kindly served as my external committee member and provided me with useful comments. I would also like to thank my other committee members, Kunal Agrawal, Mark J. Jakiela, and Jonathan S. Turner, for their advice.

I would like to thank all the great members in Cyber-Physical Systems Lab, Mo Sha, Chengjie Wu, Jing Li, Rahav Dor, and many more for the invaluable discussion we have on every lab meeting. I will miss the time we spent together working on polishing presentations and papers, and discussing research ideas. I would also like to thank all my friends in St. Louis: you guys made the past five years here the most memorable years in my life.

I would like to thank my parents for their encouragement and unconditional love and support. I would also like to thank my girl friend Elva Lu for sharing with me all the joys and supporting me through the Ph.D. journey.

Sisu Xi

Washington University in Saint Louis
August 2014

To my parents, Baiyao and Yueming

ABSTRACT OF THE DISSERTATION

Real-Time Virtualization and Cloud Computing

by

Sisu Xi

Doctor of Philosophy in Computer Science

Washington University in St. Louis, August 2014

Professor Chenyang Lu, Chair

Professor Christopher D. Gill, Co-Chair

In recent years, we have observed three major trends in the development of complex real-time embedded systems. First, to reduce cost and enhance flexibility, multiple systems are sharing common computing platforms via virtualization technology, instead of being deployed separately on physically isolated hosts. Second, multi-core processors are increasingly being used in real-time systems. Third, developers are exploring the possibilities of deploying real-time applications as virtual machines in a public cloud. The integration of real-time systems as virtual machines (VMs) atop common multi-core platforms in a public cloud raises significant new research challenges in meeting the real-time latency requirements of applications.

In order to address the challenges of running real-time VMs in the cloud, we first present RT-Xen, a novel real-time scheduling framework within the popular Xen hypervisor. We start with single-core scheduling in RT-Xen, and present the first work that empirically studies and compares different real-time scheduling schemes on a same platform. We then introduce

RT-Xen 2.0, which focuses on multi-core scheduling and spanning multiple design spaces, including priority schemes, server schemes, and scheduling policies. Experimental results demonstrate that when combined with compositional scheduling theory, RT-Xen can deliver real-time performance to an application running in a VM, while the default credit scheduler cannot. After that, we present RT-OpenStack, a cloud management system designed to support co-hosting real-time and non-real-time VMs in a cloud. RT-OpenStack studies the problem of running real-time VMs together with non-real-time VMs in a public cloud. Leveraging the resource interface and real-time scheduling provided by RT-Xen, RT-OpenStack provides real-time performance guarantees to real-time VMs, while achieving high resource utilization by allowing non-real-time VMs to share the remaining CPU resources through a novel VM-to-host mapping scheme. Finally, we present RTCA, a real-time communication architecture for VMs sharing a same host, which maintains low latency for high priority inter-domain communication (IDC) traffic in the face of low priority IDC traffic.

Chapter 1

Introduction

With the recent advent of virtualization technology, complex systems are being deployed as virtual machines (VMs) running simultaneously on a single host. A virtual machine monitor (VMM) maintains each VM's resource isolation and reduces the whole system's size, weight, and power consumption. Building on virtualization, cloud computing and other techniques allow developers to deploy VMs in a public cloud for flexible management. Furthermore, with more advanced VM capabilities, such as cloning, template-based deployment, check-pointing, and live migration, developers can easily scale their applications in the cloud according to the demand. As a result, real-time applications – that is, applications whose performance not only depends on the correctness of the results, but also on latency – are increasingly being deployed in a virtualized environment or even in the cloud.

One reason to combine real-time applications and virtualization is system consolidation. Real-time applications are widely used in embedded systems, in which the system's power consumption, size, and weight are critical. Embracing virtualization technology can potentially reduce all these, and make the VMs easy to operate and maintain. As a result, avionics [4], shipboard computing [26], and automotive computing [5] are all developing standards to integrate existing systems with virtualization.

Another important motivation for running real-time applications in a virtualized environment is the computing power provided by the cloud, which makes cloud computing an attractive choice for hosting computation-intensive real-time tasks, such as object recognition and tracking, high-definition video and audio stream processing, and feedback control loops in general. For example, a GPS device can offload its computation to the cloud to boost its battery life [58], and a gaming console can use the computing power in the cloud to provide

better image quality to the end users [17]. Another example is the Firefly feature in the Amazon Fire Phone [2], which sends images to the cloud and recognizes over 70 million products including books, DVDs and more. Prolonged latency for such applications often leads to frustrating or unacceptable experience to end users.

Many real-time applications can be modeled as a collection of real-time tasks, and each real-time task is a sequence of jobs that are released periodically. Each job is associated with a deadline and needs to finish before it. Scheduling real-time tasks on a single host without virtualization has been studied extensively both theoretically and empirically, while scheduling them in a virtualized environment remains an open question. The goal of this dissertation is to determine how such tasks should be supported in a virtualized environment, or even in a public cloud. In particular, the research presented here focuses on four questions fundamental to combining real-time applications and virtualization:

Q1: What is an appropriate interface to provide resource guarantees for a real-time VM?

Q2: How do various scheduling algorithms perform in practice?

Q3: How to integrate real-time virtualization with cloud computing?

Q4: How to support real-time communication between different VMs?

To motivate the research, we begin by illustrating why current virtualization technologies and cloud computing are not suitable for running real-time applications.

1.1 The Challenge: Combining Real-Time and Virtualization

By nature, virtualization allows multiple VMs to run simultaneously on the same hardware, and needs to maintain resource isolation between VMs by providing a resource interface. In the virtual machine monitor (VMM), there is a scheduler responsible for scheduling multiple VMs. For example, the default credit scheduler in Xen (a widely used open-source VMM) allows a VM to configure only its proportional share relative to other VMs. While this is

suitable for throughput-oriented applications, real-time applications usually have a timing requirement of milliseconds, which cannot be satisfied by current virtualization technology. In cloud gaming for example, which offloads computation to servers in the cloud, a 50 fps (frame-per-second) rate is required for high quality videos, which means the server VM needs to get the CPU resources every 20 ms to process a frame. For a first-person shooter game, the delay between server and client must be less than 100 ms [41]. If a VM can configure only its relative share, it can be blocked for a long time. An ad-hoc solution is to dedicate a subset of cores to a real-time VM, so it can get the computation resources whenever it wants them. However, this solution sacrifices the benefit of system consolidation.

When it comes to public cloud computing, the resource interfaces are even more limited. Most of them allow users to specify only the number of Virtual CPUS (VCPUs) associated with a VM, and provide sparse information about the VCPU. For example, the CPU resources in the Amazon EC2 [1] are described in numbers of ECUs (Elastic Compute Units), simply explained as "one ECU has the equivalent CPU capacity of a 1.0 - 1.2 GHz 2007 Opteron or 2007 Xeon processor". Furthermore, most cloud management systems oversubscribe the system to better utilize the resources. As a result, as long as one of the co-locating VMs consumes lots of resources, all other VMs on that host suffer performance degradation, known as the "noisy neighbor" [23] problem in cloud computing. The lack of system support for latency guarantees has forced cloud providers and users to develop proprietary application-level solutions to cope with the resource uncertainty. For example, Netflix, which runs its services in Amazon EC2, constantly monitors the resources used by each VM. If a VM cannot meet its performance requirement (usually due to a co-located noisy neighbor), Netflix shuts down the VM and restarts it on another host, hoping that the newly located host is less crowded [27]. Moreover, Netflix developed a tool called "chaos monkey" [6], which introduces artificial delays to simulate service degradation and then measures if the application can respond appropriately. An alternative solution is to pay for dedicated hosts for running real-time applications, which usually results in resource under-utilization and is not cost-effective.

The lack of appropriate resource interfaces and the underlying real-time scheduling services lead to poor real-time performance in a virtualized environment, and over-subscribing resources in cloud computing makes it worse. The integration of real-time systems as virtual machines on a common computing platform brings significant challenges in simultaneously

meeting the real-time performance requirements of multiple systems. This in turn requires fundamental advances in the underlying VM scheduling framework at the VMM level and also changes to the cloud management system.

1.2 Contributions

My research bridges the gap between real-time applications and virtualization through three projects: (1) RT-Xen, a real-time scheduling framework for the popular Xen hypervisor; (2) RT-OpenStack, a cloud management system designed to support co-hosting real-time and non-real-time VMs in a public cloud; and (3) RTCA, a real-time communication architecture for VMs sharing a same host.

1.2.1 RT-Xen

The key component of this dissertation is RT-Xen [21], an open-source real-time VM scheduling framework in Xen [33], a VMM that has been widely adopted in both embedded systems [25] and cloud computing [24]. Built on compositional real-time scheduling theory [45, 75], RT-Xen realizes a suite of real-time schedulers spanning the design space, including global and partitioned multi-core scheduling, fixed and dynamic priority policies, and different budget management schemes. RT-Xen provides a platform for researchers and integrators to develop and evaluate real-time scheduling techniques, which to date have been studied predominately via analysis and simulation. Work is underway to incorporate RT-Xen into the Xen mainstream distribution.

1.2.2 RT-OpenStack

While RT-Xen focuses on providing real-time performance guarantees on a single host, RT-OpenStack integrates RT-Xen with a popular cloud management system, OpenStack [19]. In particular, we focus on the problem of co-hosting real-time (RT) VMs with non-real-time VMs problem in a cloud. The salient feature of RT-OpenStack is to provide real-time

performance to real-time VMs, while allowing non-real-time VMs to share the remaining CPU resources without interfering with the real-time performance of RT VMs. Specifically, the RT-OpenStack System entails three contributions: (1) integration of RT-Xen and a cloud management system through real-time resource interfaces; (2) extension of RT-Xen to allow non-real-time VMs to share hosts without reducing the real-time performance of RT VMs; and (3) a VM-to-host mapping strategy that provides real-time performance to RT VMs while allowing effective resource sharing by non-real-time VMs. RT-OpenStack represents a promising step towards real-time cloud computing for latency-sensitive real-time applications.

1.2.3 RTCA

Both RT-Xen and RT-OpenStack focus on the CPU resources, while RTCA (real-time communication architecture) focuses on providing low latency for inter-domain communications (IDC) between high-priority domains sharing a same host. We first studied the limitations of the existing Xen communication architecture, then found that both the VMM scheduler and the manager domain can significantly reduce IDC performance under different conditions. Experimental results show that improving the VMM scheduler alone via RT-Xen cannot effectively prevent priority inversion for local IDC. To address these limitations, we have developed RTCA to maintain low latency between high priority domains in the face of interference from low priority domains. By combining RTCA with RT-Xen, our experimental results show that the latency between high priority domains can be improved dramatically, from ms to μs, in the presence of heavy low priority inter-domain communication traffic.

1.3 Organization

The remainder of this dissertation is organized as follows. In Chapter 2, we present RT-Xen 1.0, which focuses on single-core scheduling. Chapter 3 details RT-Xen 2.0, a multi-core real-time scheduling framework for Xen. Chapter 4 introduces our work on RT-OpenStack, along with its integration with RT-Xen. Chapter 5 presents RTCA, a real-time communication

architecture for Xen. Finally, Chapter 6 summarizes our results, raises open questions, and discusses future work.

Chapter 2

RT-Xen 1.0: Single-core Real-Time Virtualization

In this chapter, we present RT-Xen 1.0, which focuses on single-core, fixed priority scheduling. In Section 2.1, we review the Xen virtualization architecture, its scheduling framework, and its default schedulers. In Section 2.2, we discuss our task model and our guest OS configuration. Thereafter, in Section 2.3, we summarize real-time hierarchical scheduling theories, which provide guidelines for designing and implementing RT-Xen. In Section 2.4, we present the design and implementation of RT-Xen 1.0, followed by its extensive evaluation in Section 2.5. After that, we introduce two enhanced periodic servers in Section 2.6, and compare them with the original periodic server in Section 2.7.

2.1 Xen Architecture

Xen [33] is a popular open-source virtualization platform that has been developed over the past decade. It provides a layer called the virtual machine monitor (VMM) that allows multiple domains (VMs) to run different operating systems and to execute concurrently on shared hardware. In virtualization, a virtual machine is referred to as a domain, and from a scheduling perspective, it contains multiple virtual CPUs. In the rest of this dissertation, we use domain to refer to a virtual machine, *virtual CPU (VCPU)* to refer to a virtual core in a domain, and use *physical CPU (PCPU)* and *core* interchangeably to refer to an actual physical core. In order to achieve close to native performance, Xen adopts *para-virtualization* technology, where a guest domain knows that it runs on a virtualized environment and

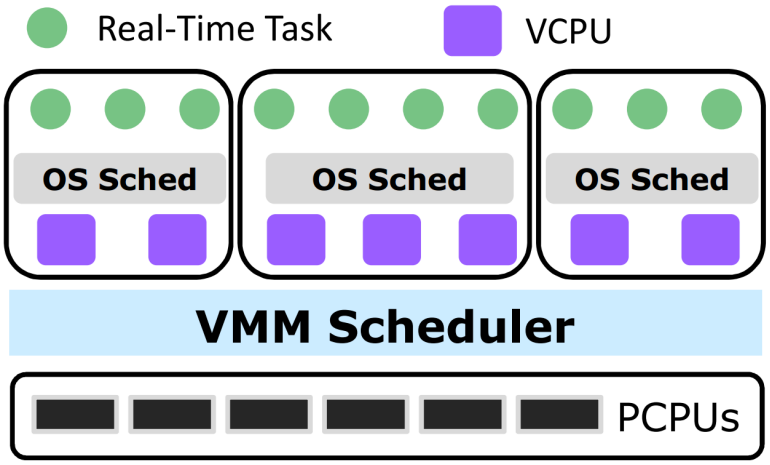


Figure 2.1: Xen Architecture

optimizes itself for better performance. At boot time, Xen creates a special domain called domain 0, which is responsible for managing the other guest domains. Figure 2.1 illustrates the architecture for Xen with three domains running, each with multiple VCPUs on a multi-core platform.

Due to the clear separation between VMM and domain 0, Xen is also referred to as a type-1 standalone hypervisor. Another virtualization technology is KVM [14], which integrates the VMM into the host OS. This dissertation focuses on using Xen as the underlying VMM, but its insights and ideas can be easily applied to other virtualization technologies as well.

2.1.1 Xen Scheduling Framework

Xen schedules *VCPUs* just like the Linux scheduler schedules tasks. A VCPU can have two scheduling states: *runnable* (has tasks running) and *non-runnable* (blocked by IO or just idle). When there are no qualified VCPUs to run, Xen schedules an idle VCPU, which works like the idle task in Linux. By default, Xen boots one idle VCPU per PCPU.

Xen also provides a well-defined scheduling framework, where different VMM schedulers can focus on implementing scheduler-dependent functions and share other parts. Among these functions, the most important ones for real-time performance are

- *do_schedule*:This function decides which VCPU should be running next, then returns its identity along with the amount of time for which to run it.
- *wake*: When a domain receives a task to run, the *wake* function is called; usually it will insert the VCPU into the CPU's RunQ (a queue containing all the runnable VCPUs). If it has higher priority than the currently running one, an interrupt triggers the *do_schedule* function to perform the context switch.
- *pick_cpu*: According to the domain's VCPU settings, this function chooses on which physical core the VCPU should be running. If it is different from the current one, a VCPU migration is triggered.
- *sleep*: This function is called when any guest OS is paused, and removes the VCPU from the RunQ.

Additional functions exist for initializing and terminating domains and VCPUs, querying and setting parameters, logging, etc. Another scheduler-independent function is the *context_switch*, which is triggered after the *do_schedule* to switch the context.

2.1.2 Xen Default Schedulers

Three schedulers are distributed with Xen: credit, credit2, and simple earliest deadline first (SEDF). The Credit scheduler is used by default from Xen 3.0 onward, and provides a form of proportional share scheduling. In the Credit scheduler, every physical core has one *Run Queue* (RunQ), which holds all the *runnable* VCPUs (VCPU with a task to run). Each domain contains two parameters: *weight* and *cap*. *Weight* defines the domain's proportional share, and *cap* defines the upper limit of its received CPU resources. At the beginning of an accounting period, each domain is given *credit* according to its *weight*, and the domain distributes the *credit* to its VCPUs. VCPUs consume *credit* as they run, and are divided into three categories when on the RunQ: BOOST, UNDER, and OVER. A VCPU is put into the BOOST category when it performs I/O, UNDER if it has remaining *credit*, and OVER if runs out of *credit*. The scheduler picks VCPUs in the order of BOOST, UNDER, and OVER. Within each category, it is important to note that VCPUs are scheduled in a round robin fashion. By default, when picking a VCPU, the scheduler allows it to run for

30 ms, and then triggers the *do_schedule* function again to pick the next one. This quantum involves trade offs between real-time performance and throughput. A large quantum may lead to poor real-time performance due to coarse-grained scheduling.

The credit2 scheduler presents the same interface to the system integrator as the credit scheduler – each domain is given a *weight* that determines its proportional share – but it differs from the credit scheduler in that it uses a global run queue per CPU chip (instead of one run queue per core in *credit*) to optimize system performance. However, the credit2 scheduler does not support *caps* and is not CPU-mask aware. As a result, it cannot limit each domain's CPU resource, nor can it dedicate cores to domains.

Xen also ships with a SEDF scheduler, in which every VCPU has three parameters: *slice* (equals *budget* in our RT-Xen scheduler), *period*, and *extra time* (whether or not a VCPU can continue to run after it runs out of its *slice*). The SEDF scheduler consumes a VCPU's *slice* when it is running, preserves the *slice* when not running, and sets the *slice* to full when the next accounting period comes. Every physical core also has one RunQ containing all the runnable VCPUs with positive *slice* values. VCPUs are sorted by their relative deadlines, which are equal to the ends of their current *periods*. Right now, SEDF supports only single-core scheduling; it is no longer in active development and will be phased out in the near future [8]. We are actively working with Xen developers to integrate RT-Xen into the Xen mainstream to replace the legacy SEDF scheduler.

2.2 Task Model and Guest Scheduler Configuration

As depicted in Figure 2.1, from a scheduling perspective, a virtualized system has at least a two-level hierarchy, where the VMM scheduler schedules guest operating systems, and each guest OS in turn schedules tasks. We now focus on a typical real-time task model, and how to configure it in Linux as the guest OS. Each guest OS runs a set of periodic real-time tasks. Every task has a *period*, which denotes the job release interval, and a *cost*, which indicates the worst case execution time to finish a job. Each task has a relative deadline that is equal to its *period*. In this work, we focus on *soft real-time* applications, in which a job continues to execute until it finishes, even if its deadline has passed, because deadline misses represent degradation in quality of service instead of failure. As a starting

point for demonstrating the feasibility and efficacy of real-time virtualization in Xen, we assume a relatively simple task model, where tasks are independent and CPU-bound, with no blocking or resource sharing between jobs. Such task models are also consistent with existing hierarchical real-time scheduling algorithms and analysis [43, 55, 65].

Each guest OS is responsible for scheduling its task sets. To be consistent with existing hierarchical scheduling analysis [43], in this chapter we use the preemptive fixed-priority scheduling class in Linux to schedule the tasks. We focus on single-core scheduling here, and allocate one VCPU for each guest OS. To minimize interferences from domain 0, we allocate a dedicated core to it, and run all guest OS on another separate core. Multi-core scheduling will be introduced in Chapter 3.

2.3 Single-Core Hierarchical Scheduling Theories

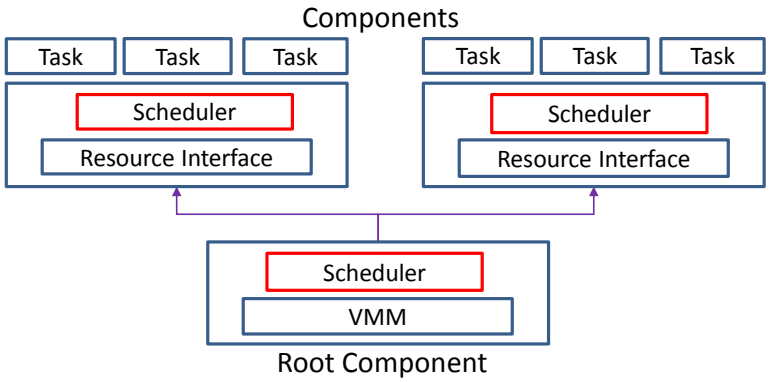


Figure 2.2: Compositional Scheduling Architecture

We observe that there is a natural mapping from a virtualized system to a two-level compositional scheduling model [65, 66]. For example, a virtual machine corresponds to a unit of composition, i.e., an elementary component. The root component corresponds to a composition of multiple elementary components (VMs) scheduled by a single virtual machine monitor. Figure 2.2 demonstrates a two-level compositional scheduling architecture: the system consists of a set of *components*, where each component is composed of either a set of subcomponents or a set of tasks. Each component is defined by $C = (W, \Gamma, A)$, where

W is a workload, i.e., a set of tasks or components; Γ is a resource interface; and A is a scheduling policy used to schedule W. All tasks are periodic, where each task T_i is defined by a period (and deadline) p_i and a worst-case execution time e_i, with $p_i \geq e_i > 0$ and $p_i, e_i \in Integers$. Interface Γ is a periodic resource model, where the scheduling policy can be either static priority (rate monotonic, RM) or dynamic priority (earliest deadline first, EDF). There are competing theories for single-core hierarchical scheduling [43, 44, 55, 57]. They all share the same CPU resource interface as compositional scheduling theory, but differ in the resource models (also referred to as server mechanisms). We will introduce different server mechanisms in Section 2.4.

2.4 RT-Xen 1.0: Design and Implementation

This section presents the design and implementation of RT-Xen, which is shaped by both theoretical and practical concerns. Section 2.4.1 describes the four fixed-priority schedulers in RT-Xen, and section 2.4.2 describes the VMM scheduling framework within which different root schedulers can be configured for scheduling guest operating systems.

2.4.1 VMM Scheduling Strategies

In this chapter, we consider four servers: deferrable server [69], sporadic server [67], periodic server, and polling server [64]. These server schemes have all been studied in the recent literature on hierarchical fixed-priority real-time scheduling [43, 55, 65]. For all of these schedulers, a server corresponds to a VCPU, which in turn appears as a physical core in the guest OS. Each VCPU has three parameters: *budget*, *period*, and *priority*. As Davis and Burns showed in [42], server parameter selection is a holistic problem, and RM does not necessary provide the best performance. Thus we allow developers to assign arbitrary priorities to the server, giving them more flexibility. When a guest OS executes, it consumes its *budget*. A VCPU is eligible to run if and only if it has positive *budget*. Different server algorithms differ in the way the *budget* is consumed and replenished, but each schedules eligible VCPUs based on preemptive fixed-priority scheduling.

- A deferrable server is invoked with a fixed period. If the VCPU has tasks ready, it executes them either until the tasks complete or the budget is exhausted. When the guest OS is idle, its budget is preserved until the start of its next period, when its budget is replenished.
- A periodic server is also invoked with a fixed period. In contrast to a deferrable server, when a VCPU has no task to run, its budget idles away, as if it had an idle task that consumed its budget. Details about how to simulate this feature are discussed in Section 2.4.2.
- A polling server is also referred to as a *discarding periodic server* [43]. Its only difference from a periodic server is that a polling server discards its remaining budget immediately when it has no tasks to run.
- A sporadic server differs from the other servers in that it is not invoked with a fixed period, but rather its budget is continuously replenished as it is used. We implement the enhanced sporadic server algorithm proposed in [68]. Implementation details again can be found in Section 2.4.2.

2.4.2 VMM Scheduling Framework

As we described in Section 2.1.1, to add a new scheduler in Xen, a developer must implement several important functions, including *do_schedule*, *wake*, and *sleep*. We now describe how the four RT-Xen schedulers (deferrable server, periodic server, polling server, and sporadic server) are implemented.

We assume that every guest OS is equipped with one VCPU, and all the guest OS are pinned on one specific physical core. Since the deferrable, periodic, and polling servers all share the same replenishment rules, we can implement them as one *subscheduler*, and have developed a tool to switch between them on the fly. The sporadic server is more complicated and is implemented individually.

In all four schedulers in RT-Xen, every physical core is equipped with three queues: a *Run Queue* (RunQ), a *Ready Queue* (RdyQ), and a *Replenishment Queue* (RepQ). The RunQ and RdyQ are used to store active VCPUs. Recall that RunQ always contains the IDLE

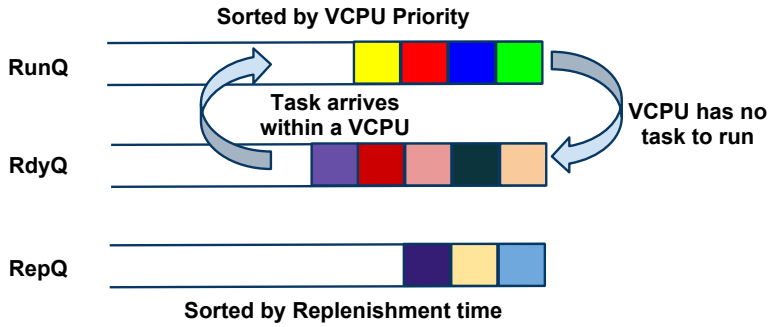


Figure 2.3: Scheduler Queues in RT-Xen 1.0

VCPU, which always has the lowest priority and is put at the end of the RunQ. Figure 2.3 illustrates the three different queues, as well as how a VCPU migrates between the RunQ and the RdyQ.

- The RunQ holds VCPUs that have tasks to run (regardless of budget), sorted by priority. Every time *do_schedule* is triggered, it inserts the currently running VCPU back into the RunQ or RdyQ, then picks the highest priority VCPU with a positive budget from the RunQ, and runs it for one quantum (we choose the quantum to be 1ms, based on our evaluation in Section 2.5).

- The RdyQ holds all VCPUs that have no task to run. It is designed especially for periodic server to mimic "as if budgets are idled away" behavior. When the highest VCPU becomes IDLE and still has budget to run, we schedule the IDLE VCPU on the RunQ and consume the VCPU's budget. This requires us to store VCPUs even if they have no task to run, and to compare their priority with the ones on RunQ to decide whether to schedule the IDLE VCPU or not.

- The RepQ stores replenishment information for all the VCPUs on that physical core. Every entry in RepQ contains three elements: the VCPU to replenish, the replenishment time, and the replenishment amount to perform. A *tick* function is triggered every scheduling quantum to check the RepQ, and if necessary, perform the corresponding replenishment. If the replenished VCPU has higher priority than the currently running one, an interrupt is raised to trigger the *do_schedule* function, which stops the current VCPU and picks the next appropriate one to run. Note here that in sporadic server, a VCPU can have multiple replenishments pending, so the RepQ is necessary.

Since the four different scheduling strategies share common features, we first describe how to implement deferrable server, and then describe additional extensions for the other three schedulers.

Algorithm 1 Scheduler Function For Deferrable Server

```
1: consume current running VCPU's budget
2: if current VCPU has tasks to run then
3:     insert it into the RunQ according to its priority
4: else
5:     insert it into the RdyQ according to its priority
6: end if
7: pick highest priority VCPU with budget from RunQ
8: remove the VCPU from RunQ and return it along with one quantum of time to run
```

As is shown in line 5 of Algorithm 1, when the VCPU is no longer runnable, its *budget* is preserved and the VCPU is inserted into the RdyQ. The polling server differs from the deferrable server in that in line 5, the VCPU's *budget* is set to 0. For the periodic server, in line 1, if the current running VCPU is the IDLE VCPU, it would consume some of the *budget* of the highest priority VCPU with a positive *budget* on the RdyQ; in line 7, it would compare the VCPUs with a positive *budget* on both RunQ and RdyQ: if RunQ one had higher priority, it would return it to run; otherwise, it would return the IDLE VCPU to run.

Sporadic server is more complicated in its replenishment rules. We use the corrected version of sporadic server described in [68], which showed that the POSIX sporadic server specification may suffer from three defects: *Budget Amplification*, *Premature Replenishments*, and *Unreliable Temporal Isolation*. Since we are implementing the sporadic server in the VMM level, the *Budget Amplification* and *Unreliable Temporal Isolation* problems do not apply because we allow each VCPU to run only up to its *budget* time, and we do not have to set a *sched_ss_low_priority* for each VCPU. To address the *Premature Replenishments* problem, we split the replenishment as described in [68]. Our sporadic server implementation works as follows: each time the *do_schedule* function is called, if the chosen VCPU is different from the currently running one, the scheduler records the current VCPU's *budget* consumed since its last run, and registers a replenishment in the RdyQ. In this way, the replenishment is correctly split and a higher priority VCPU will not affect the lower priority ones. Interested readers are directed to [68] for details.

For all four schedulers, whenever the *wake* function is called and the target VCPU is on the RdyQ, it is migrated to the RunQ within the same physical core. If its priority is higher than the currently running VCPU, a scheduling interrupt is raised.

2.5 RT-Xen 1.0: Evaluation

This section describes our evaluation of the RT-Xen 1.0 scheduling framework. First, we measured real-time performance with different scheduling quanta, ranging from 1 millisecond down to 10 microseconds. Based on the results, we chose 1 millisecond as the scheduling quantum. Second, a detailed overhead measurement was performed for each of the four schedulers. Third, we studied the impact of an overloaded domain on both higher and lower priority ones. Finally, we empirically evaluated the *soft real-time* performance under different system loads.

2.5.1 Experiment Setup

Platform

We performed our experiments on a Dell Q9400 quad-core machine without hyper-threading. SpeedStep was disabled by default and each core ran at 2.66 GHz. The 64-bit version of Fedora 13 with para-virtualized kernel 2.6.32.25 was used in domain 0 and all guest operating systems. The most up-to-date Xen version 4.0 was used. Domain 0 was pinned to core 0 with 1 GB memory, while the guest operating systems were pinned to core 1 with 256 MB memory each. Data were collected from the guest operating systems after the experiments were completed. During the experiments, the network service and other inessential applications were shut down to avoid interference.

Implementation of Tasks on Linux

We now describe how we implemented real time tasks atop the guest operating systems. The implementations in the hypervisor and Linux are separate and independent from each other. The modifications to the hypervisor included the server-based scheduling algorithms. We did not make any changes to the Linux kernel (other than the standard paravirtualization patch required by Xen), but used its existing APIs to trigger periodic tasks and assign thread priorities (based on the rate monotonic scheme) at the user level. Currently, the scheduling tick (jiffy) in Linux distributions can be configured at a millisecond level. This quantum was used as a lower bound for our tasks. We first calibrated the amount of work that requires exactly 1 ms on one core (using native Linux), and then scaled it to generate any workload specified in millisecond resolution. As we noted in Section 2.4, the workload is independent and CPU intensive. Using the well-supported POSIX interfaces on Linux, every task was scheduled using SCHED_FIFO, and the priority was set inversely to its deadline: the shorter the deadline, the higher the priority. With this setting, the Linux scheduler performs as a rate monotonic scheduler. We used POSIX real time clocks to generate interrupts to release each job of a task, and recorded the first job release time. Recall that we assume we are dealing with *soft real time* systems, so that even if a job misses a deadline, it still continues executing, and the subsequent jobs will queue up until their predecessors complete. When each job finished, its finish time was recorded using the RDTSC instruction, which provides 1 nano-second precision with minimal overhead. After all tasks finished, we used the first job's release time to calculate every job's release time and deadline, and compared each deadline with the corresponding job finish time. In this way, we could count the deadline miss ratio for each individual task. All the information was stored in locked memory to avoid memory paging overhead. Based on the collected data, we calculated the total number of jobs that missed their deadlines within each OS. Dividing by the total number of jobs, we obtained the deadline miss ratio for each domain.

2.5.2 Impact of the Scheduling Quantum

In this experiment our goal was to find an appropriately fine-grained scheduling quantum involving acceptable overhead. We defined the scheduling quantum to be the time interval

at which *do_schedule* is triggered, which represents the precision of the scheduler. While a finer grained quantum allows more precise scheduling, it also may incur larger overhead. We defer a more detailed overhead measurement to Section 2.5.3.

We varied the scheduling quantum from 1 millisecond down to 10 microseconds to measure its effects. Two domains were configured to run with different priorities. The high priority one, configured as domain 1, was set up with a *budget* of 1 quantum and a *period* of 2 quanta (a share of 50 %). To minimize the guest OS scheduling overhead, domain 1 ran a single real time task with a deadline of 100 ms, and its cost varied from 1ms to 50ms. For each utilization, we ran the task with 600 jobs, and calculated how many deadlines are missed. The low priority domain was configured as domain 2, with a *budget* of 2 quanta and *period* of 4 quanta. It ran a busy loop to generate the most possible interference for domain 1. Note that under this setting, whenever domain 1 had a task to run, it would encounter a context switch every scheduling quantum, generating the worst case interference for it. In real world settings, a domain would have larger budgets and would not suffer this much interference. Since we ran only a single task within domain 1, and the task's deadline was far larger than the domain's *period*, the choice of scheduler did not matter, so we used deferrable server as the scheduling scheme.

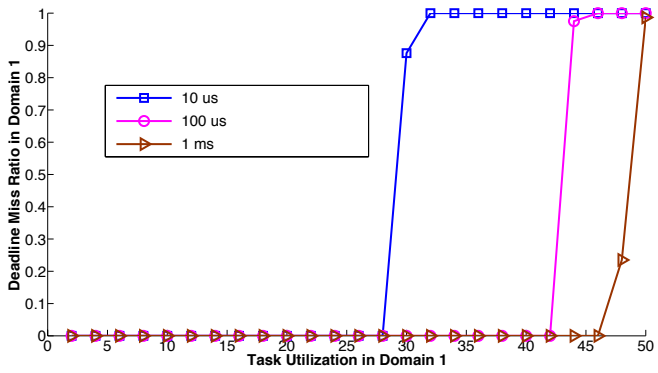


Figure 2.4: Performance under Different Scheduling Quanta

Figure 2.4 shows the results for scheduling quanta varying from 1 ms to 10 μs. From this figure, we see a deadline miss starting at 48% for 1 ms, 44% for 100 μs, and 30% for 10 μs. When 1 μs was chosen, the overhead was so large that guest OS cannot even be booted. Based on these results, we chose 1 ms as our scheduling quantum since it suffers only 4% loss ($\frac{50\%-48\%}{50\%}$), and provides enough precision for the upper level tasks. Recall that this is

the worst case interference. Under the schedulability test below, we apply a more realistic setting, in which the interference is much less.

2.5.3 Overhead Measurement

The focus of this work is fixed-priority preemptive hierarchical scheduling, within which we can compare different server schemes. Therefore we consider the forms of overhead which are most relevant to fixed-priority scheduling schemes: scheduling latency and context switches.

- *scheduling latency*: the time spent in the *do_schedule* function, which inserts the current VCPU back into the RunQ or the RdyQ, picks the next VCPU to run, and updates the corresponding status.
- *context switch*: the time required to save the context for the currently running VCPU and to switch to the next one.

The scheduler first decides which VCPU to run next, and if necessary, performs a context switch. Other sources of overhead such as migration, cache effects and bus contention, are independent of different server schemes, and therefore we defer investigation of their effects to future work.

Five domains were configured to run under the four schedulers in RT-Xen, using the "even share" configuration as in Section 2.5.5, Table 2.3. The total system load was set to 70%, and each domain ran five tasks. For completeness, we ran the same workload under the credit and SEDF schedulers and measured their overheads as well. For the credit scheduler, we kept *weight* the same for all the domains (because they have the same share ($\frac{budget}{period}$)), and set *cap* to 0 by default. Recall that we changed the quantum to 1 ms resolution to give a fair comparison (the original setting was 30ms). For the SEDF scheduler, the same (*slice*, *period*) pair was configured as (*budget*, *period*) for each domain, and *extratime* was disabled.

Each experiment ran for 10 seconds. To trigger recording when adjusting parameters for domain 0, a timer in *scheduler.c* was set to fire 10 seconds later (giving the system time to return to a normal running state). When it fired, the experiment began to record the

time spent in the *do_schedule* function, and the time spent in each *context switch*. After 10 seconds, the recording finished and the results were collected.

Table 2.1: Overhead Measurement for 10 Seconds

	Deferrable	Periodic	Polling	Sporadic	Credit	SEDF
total time in *do_schedule*	1,435 μs	1,767 μs	1,430 μs	1,701 μs	216 μs	519 μs
total time in *context switch*	19,886 μs	20,253 μs	19,592 μs	22,263 μs	4,507 μs	8,565 μs
total time combined	21,321 μs	22,020 μs	21,022 μs	23,964 μs	4,722 μs	9,084 μs
percentage of time loss in **10 seconds**	**0.21%**	**0.22%**	**0.21%**	**0.23%**	**0.04%**	**0.09%**
do_schedule overhead (max)	5,642 ns	461 ns	370 ns	469 ns	382 ns	322 ns
do_schedule overhead (median)	121 ns	159 ns	121 ns	150 ns	108 ns	130 ns
99% quantile values in *do_schedule*	250 ns	328 ns	252 ns	303 ns	328 ns	192 ns
number of *do_schedule* called	10,914	10,560	10,807	10,884	1,665	4,126
context switches overhead (max)	12,456 ns	13,528 ns	8,557 ns	11,239 ns	8,174 ns	8,177 ns
context switches overhead (median)	1,498 ns	1,555 ns	1,513 ns	1,569 ns	2,896 ns	2,370 ns
99% quantile values in *context switches*	3,807 ns	3,972 ns	3,840 ns	3,881 ns	3,503 ns	3,089 ns
number of *context switches* performed	3,254	3,422	2,979	4,286	1,665	3,699

We make the following observations from the results shown in Table 2.1:

- The four fixed-priority schedulers do encounter more overhead than the default credit and SEDF ones. This can be attributed to their more complex RunQ, RdyQ, and RepQ management. However, the scheduling and context switch overheads of all the servers remain moderate (totaling 0.21 - 0.23% of the CPU time in our tests). These results demonstrate the feasibility and efficiency of supporting fixed-priority servers in a VMM.

- Context switch overhead dominates the scheduling latency overhead, as a context switch is much more expensive than an invocation of the scheduler function. Context switch overhead therefore should be the focus of future optimization and improvements.

- The different server schemes do have different overheads. For example, as expected, sporadic server has more overhead than the others due to its more complex budget management mechanisms. However, the differences in their overheads are insignificant (ranging from 0.21% to 0.23% of the CPU time).

We observed an occasional spike in the duration measured for the deferrable server, which may have been by an interrupt or cache miss. It occurred very rarely, as the 99% quantile value shows, which may be acceptable for many *soft real-time* systems. The credit and SEDF schedulers return a VCPU to run for up to its available *credits* or *slices*, and when an IDLE VCPU is selected, the scheduler will return it to run forever until interrupted by others. As a result, the number of times that the *do_schedule* function is triggered is significantly fewer than in the other four schedulers.

2.5.4 Impact of an Overloaded Domain

To deliver desired real-time performance, RT-Xen also must be able to provide fine grained controllable isolation between guest operating systems. Even if a system developer misconfigures tasks in one guest OS, that should not affect other guest operating systems. In this experiment, we studied the impact of an overloaded domain under the four fixed-priority schedulers and the default ones. [1]

The settings introduced in Section 2.5.3 were used with only one difference: we overloaded domain 3 by "misconfiguring" the highest priority task to have a utilization of 10%. Domain 3's priority is intermediate, so we can study the impact on both higher and lower priority domains. We also ran the experiment with the original workload, which is depicted as the normal case. The performance of the credit and SEDF schedulers are also reported for the same configuration described in Section 2.5.3. Every experiment ran for two minutes, and based on the recorded task information, we calculated the deadline miss ratio, which is the percentage of jobs that miss their deadlines, for each domain.

Table 2.2: Isolation with RT-Xen, Credit, and SEDF

Domain		1	2	3	4	5
Normal	Sporadic	0	0	0	0	0
	Periodic	0	0	0	0	0
	Polling	0	0	0	0	0
	Deferrable	0	0	0	0	0
	Credit	96%	0.1%	0	0	0
	SEDF	0	0	0	0	0
Overloaded	Sporadic	0	0	49.8%	0	0
	Periodic	0	0	48.9%	0	0
	Polling	0.08%	0	49.7%	0.28%	0
	Deferrable	0	0	48%	0	0
	Credit	100%	0	1.6%	0	0
	SEDF	0	0	0	0.08%	0

Table 2.2 shows the results: under the *normal* case, all four fixed-priority schedulers and SEDF meet all deadlines, while in the credit scheduler, domain 1 misses nearly all deadlines. There are two reasons for this.

[1] The default credit scheduler also provides isolation for longer periods, but not shorter ones.

- All five VCPUs are treated equally, so the credit scheduler picks them in a round robin fashion, causing domain 1 to miss deadlines. However, in the fixed-priority schedulers it has the highest priority, and would always be scheduled first until its *budget* was exhausted.
- Domain 1 has the smallest period, and the generated tasks also have the relatively tightest deadlines, which makes it more susceptible to deadline misses.

Under the *overloaded* case, the sporadic, periodic, and deferrable servers provided good isolation of the other domains from the overloaded domain 3. For polling server and SEDF, we see deadline misses in domain 1 and domain 4, but only in less than 0.3 % of all cases. We think this is tolerable for *soft real-time* systems running atop an off-the-shelf guest Linux on top of the VMM, since interrupts, bus contention, and cache misses may cause such occasional deadline misses. Although credit scheduler met most of its deadlines in the overloaded domain 3 (benefiting from system idle time with a total load of 70%), domain 1 again was severely impacted, missing all deadlines. These results illustrate that due to a lack of finer grained scheduling control, the default credit scheduler is obviously not suitable for delivering real time performance, while all four fixed-priority scheduler implementations in RT-Xen are suitable.

2.5.5 Soft Real-Time Performance

Table 2.3: *Budget*, *Period* and *Priority* for Five Domains

Domain		1	2	3	4	5
Priority		1	2	3	4	5
Budget		2	4	6	8	10
Period	*Decreasing*	4	20	40	80	200
	Even	10	20	30	40	50
	Increasing	40	40	40	40	20

This set of experiments compared the *soft real-time* performance of different servers. Note that our study differs from and complements previous theoretical comparisons which focus on the capability to provide hard real-time guarantees. To assess the pessimism of the analysis,

we also compared the actual real-time performance against an existing response time analysis for fixed-priority servers.

The experiments were set up as follows: five domains were configured to run, with *budget* and *priority* fixed, but *period* varied to represent three different cases: decreasing, even, and increasing share (share is defined as the ratio of *budget* to *period*). All five domains' shares add up to 100%, as shown in Table 2.3. Note that the shares do not represent the real system load on the domain.

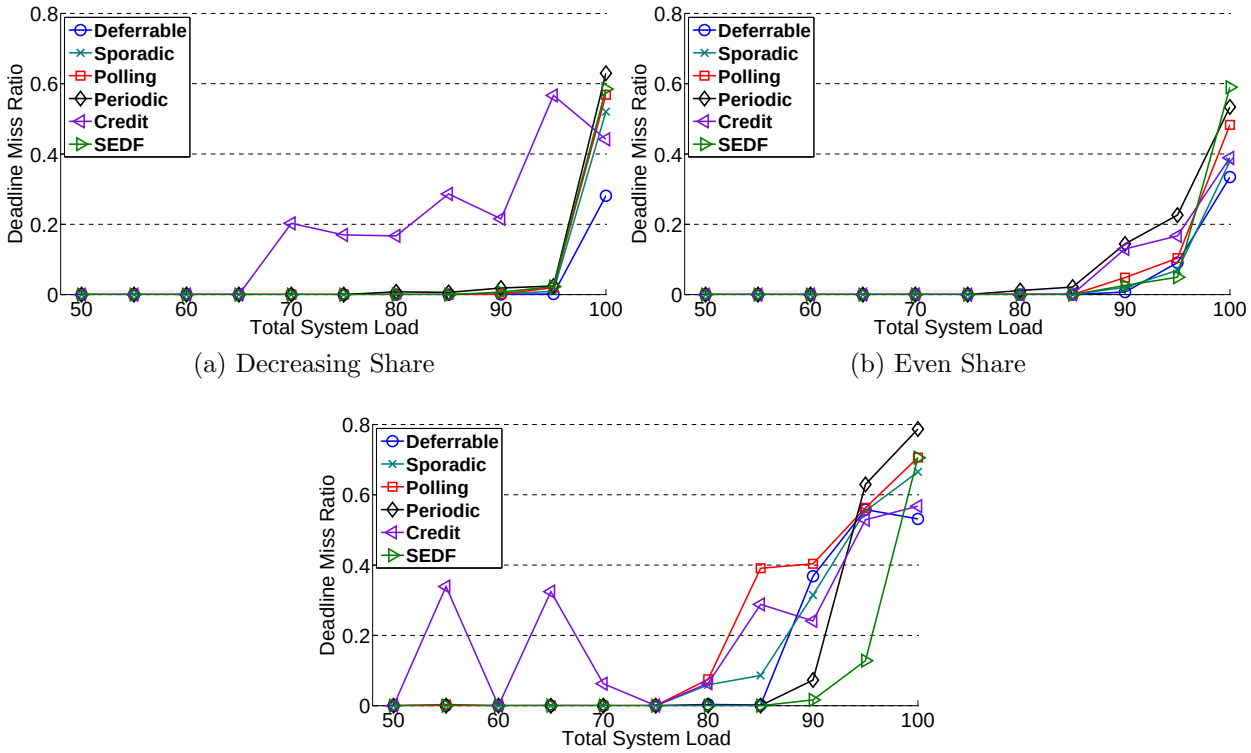
Task sets were randomly generated following the steps below. A global variable α was defined as the total system load. It varied from 30% to 100%, with steps of 5%. For each α, we generated five tasks per domain, making 25 in total. Within each domain, we first randomly generated a *cost* between 5 ms and 10 ms for each of the five tasks (using α as a random seed), then randomly distributed the domain's share times α (which represents the real domain load) among the five tasks. Using every task's *cost* and *utilization*, we could easily calculate its *deadline*. Note that all *costs* and *deadlines* were integers, so there was some reasonable margin between the real generated system load and the α value.

We can see that the task's *period* is highly related to the domain's *period* and *share*. The decreasing share case is the "easiest" one to schedule, where domain 1 has the largest share and highest priority, so a large number of tasks are scheduled first. Even share is the "common" case, where every domain has the same share and we can see the effects of different priorities and periods. Increasing share is the "hardest" case, where the lowest priority domain holds the largest number of tasks. Also note that the increasing share case is the only one that does not correspond to RM scheduling theory at the VMM level.

For completeness, we again include results for the same workload running under the credit and SEDF schedulers as well. For the credit scheduler, the scheduling quantum was configured at 1 ms. The *weight* was assigned according to the domain's relative share. For example, if a domain's share was 20%, its weight took 20% of the total weight. The *cap* was set to 0 as in the default setting, so each domain would take advantage of the extra time. For the SEDF scheduler, we configured the same (*slice*, *period*) pair as (*budget*, *period*) for each domain, and again disabled *extratime*.

Each experiment ran for five minutes. Figure 2.5 shows the results for all three cases. When the system load was between 30% and 50%, all deadline miss ratios were 0%. We omitted these results for a better view of the remaining data. Note that the Y axis ranges from 0% to 80%. The four solid lines represent our four fixed-priority schedulers, and the two dashed lines represent the default credit and SEDF schedulers.

We evaluated different schedulers based on two criteria: (1) At what load does the scheduler see a "significant" deadline miss ratio? Since we are dealing with *soft real-time* systems, we consider a 5% miss ratio as significant, and define the maximum system load without significant deadline miss to be the *soft real-time capacity* of the scheduler. (2) What is the scheduler's performance under the *overloaded* situation (e.g., 100%)?



(a) Decreasing Share

(b) Even Share

(c) Increasing Share

Figure 2.5: Deadline Miss Ratio under Different Shares

From Figure 2.5, we can see several things:

- The default credit scheduler performs poorly in terms of *capacity*, even when configured at a 1ms resolution.

- The SEDF scheduler maintains a good capacity of almost 90%. With respect to its *overload* behavior, it is comparatively worse than the fixed-priority schedulers in most cases.

- The deferrable server scheduler generally performs well among RT-Xen schedulers. It has equally good capacity, and the best *overload* behavior under all three cases, indicating that its budget preservation strategy is effective in delivering good *soft real-time* performance in a VMM. Note that while it is well known that deferrable server can suffer from the "back-to-back" preemption effect in terms of worst-case guarantees, such effects rarely happen in real environments.

- Among RT-Xen schedulers, the periodic server scheduler performs worst in the *overloaded* situation. As we discussed in Section 2.4, to mimic the "as if *budget* is idled away" behavior, when a high priority VCPU has *budget* to spend even if it has no work to do, periodic server must run the IDLE VCPU and burn the high priority VCPU's *budget*. During this time, if a low priority VCPU with a positive *budget* has work to do, it must wait until the high priority VCPU exhausts its *budget*. While this does not hurt the *hard real-time* guarantees, the *soft real-time* performance is heavily impacted, especially under the *overloaded* situation, due to the non-work-conserving nature of the periodic server.

Table 2.4: Theoretical Guaranteed Schedulable System Load (Percentage)

	deferrable server	periodic server
Decreasing	30-45%	30-50%, 60-75%
Even	30-45%	30-50%, 60-75%
Increasing	30-45%	30-50%, 60-75%

Since we used the same settings as in [43], we also applied that analysis to the task parameters for comparison. Note that all the tasks were considered "unbound" because the task *periods* were generated randomly, and we assumed the overhead was 0. Table 2.4 shows the results, where under deferrable and periodic server the task set should be schedulable. Clearly, when theory guarantees the tasks are schedulable, they are indeed schedulable using those

schedulers in RT-Xen. These results also show the pessimism of the theory, where with deferrable server, for all three cases, theory guarantees it is schedulable only if total system load is under 45 %, while in reality it is schedulable up to nearly 85 %.

2.6 Improving Periodic Servers

A key observation from the evaluation results in Section 2.5 is that due to non-work-conserving behavior, the periodic server performs worst among RT-Xen servers in an overloaded situation. Specifically, when a higher-priority component has no work to do, it simply idles away its *budget* while lower-priority components are not allowed to run. Our implementation emulates this feature by scheduling the idle VCPU to run while a high-priority domain idles away its *budget*. This scenario arises when a high-priority domain under-utilizes its *budget*, e.g., due to an interface overhead or an over-estimation of task execution times when configuring the domains' *budgets*. We refer to this approach as the *purely time-drive periodic server*. While the non-work-conserving approach does not affect the worst-case guarantees, it wastes CPU cycles and increases the response times of low-priority domains. This is particularly undesirable for soft real-time systems, as well as many hard real-time systems where short response times are also beneficial.

Based on this observation, we present two enhanced variations of the *purely time-driven periodic server* to optimize run-time performance and resource-use efficiency, namely the *work-conserving periodic server* and the *capacity-reclaiming periodic server*. These variations differ in how a server *budget* changes when the server has remaining *budget* but is idle (i.e., has no unfinished jobs), or when it is non-idle but has no *budget* left. Recall that in the classical purely time-driven periodic server, a server's *budget* is replenished to full capacity every *period*. The server is eligible for execution only when it has non-empty *budget*, and its *budget* is always consumed at the rate of one execution unit per time unit, even if the server is idle. In the work-conserving periodic server variant, whenever the currently scheduled server is idle, the VMM's scheduler lets another lower-priority non-idle server run; thus, the system is never left idle if there are unfinished jobs in a lower-priority domain. Finally, the capacity-reclaiming periodic server variant further utilizes the unused resource *budget* of an idle server to execute jobs of any other non-idle servers, effectively adding extra *budget*

to the non-idle servers. Both servers preserve the conservative compositional schedulability analysis, while yielding substantial improvements in observed response times and resource utilization, which are desirable for not only soft real-time applications but also many classes of hard real-time applications.

Purely Time-driven Periodic Server (PTPS). As mentioned above, the *budget* of a PTPS is replenished at every *period*, and its *budget* is always consumed whenever it is executed. As Xen is an event-triggered virtual platform, we introduce a mechanism to allow this time-driven *budget* replenishment and scheduling approach. Note that the PTPS approach is not work-conserving since the system resource is always left unused if the currently scheduled server (Xen domain) is idle.

Work-Conserving Periodic Server (WCPS). The *budget* of a WCPS is replenished in the same fashion as that of a PTPS. However, if the currently scheduled server (C_H) is idle, the scheduler picks a lower-priority non-idle server to execute, according to the following work conserving rules:

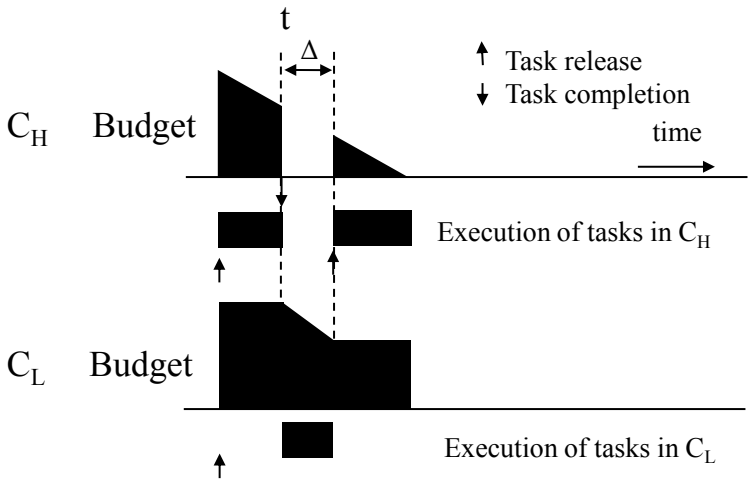


Figure 2.6: Execution of Servers in the WCPS Approach

(1) Choose a lower-priority server, C_L, with the highest priority among all non-idle lower-priority servers.

(2) Start executing C_L and consuming the budgets of both C_L and C_H, each at the rate of one unit per time unit.

(3) Continue running C_L until one of the following occurs:

- (a) C_L has no more jobs to execute;
- (b) C_L has no more budget;
- (c) Some jobs in C_H become ready and C_H has remaining budget;
- (d) C_H has no more budget.

In the case of (a) or (b), the scheduler goes back to step 1 where it selects another lower-priority non-idle server. In the case of (c), C_L immediately stops its execution and *budget* consumption, whereas C_H resumes its execution. In the case of (d), C_L immediately stops its execution and *budget* consumption; a new server will be chosen for execution by the scheduler.

Figure 2.6 illustrates a general scenario under the work conserving rule. In this scenario, C_H becomes idle at time t, and thus a lower-priority server C_L is selected for execution. At time $t + \Delta$, some jobs in C_H become ready (i.e., case (c) in step 3); therefore C_H preempts C_L and resumes its execution. By allowing C_L to run (if C_H is idle) and maintaining the same execution for C_H, the WCPS achieves shorter overall response times for tasks than PTPS, while preserving conservative compositional scheduling analysis.

Capacity Reclaiming Periodic Server (CRPS). Like the WCPS, the CRPS is also work conserving, and the *budget* of a server is replenished to full capacity every *period*. However, the CRPS improves task response times by allowing the idle time of the currently running server to be utilized by *any* other server (including higher-priority ones). Specifically, we define the *residual capacity* of a server to be the time interval during which the server consumes its *budget* but is idle (e.g., C_H has a residual capacity of $[t, t+\Delta]$ in Figure 2.6). At run time, the server *budget* is modified using the following capacity-reclaiming rule: during a residual capacity interval of a server C_H, the resource *budget* of C_H is re-assigned to any other non-idle server C_L, and only this *budget* is consumed (e.g., the *budget* of C_L remains intact).

2.7 Evaluation of Improved Periodic Servers

This section presents our evaluation of the PTPS, WCPS, and CRPS approaches that are implemented in RT-Xen. We focus on the run-time performance of real-time tasks, considering the following two evaluation criteria: (1) responsiveness, which is the ratio of a job's response time to its relative deadline; and (2) deadline miss ratio. Our evaluation consists synthetic workloads.

2.7.1 Experiment Setup

We assume all tasks are CPU intensive and independent of each other. Every task is characterized by three parameters: *worst case execution time (WCET)*, *period* (equals *deadline*), and *execution time factor (ETF)*. Here, the *ETF* represents the variance of each job's actual execution time (uniformly distributed in the interval $(WCET * ETF, WCET)$). An *ETF* of 100% indicates that every job of the task takes exactly *WCET* units of time to finish. The task model fits typical soft real-time applications (e.g., multimedia decoding applications where frame processing times are varied but are always below an upper limit).

In the rest of the chapter, U_W denotes the total utilization of all tasks in the system (utilization of the workload), U_{RM} denotes the total bandwidth of interfaces (utilization of resource models), $U_{RM} - U_W$ denotes the interface overhead.

Real-time scheduling of domains. We first determined the domains' resource needs by computing an optimal PRM interface for each domain. These interfaces were implemented as PTPS, WCPS, or CRPS variants of periodic servers, which were then scheduled by the VMM. For the workloads, we applied the compositional scheduling theory to compute the optimal integer-valued PRM interfaces for the domains. We computed optimal rational-valued interfaces, and then rounded up the *budgets* to the closest integer values. Although the integer-valued interfaces may have interface overheads of zero, rounding may introduce additional overheads, effectively allocating extra *budget* to the corresponding domains. For each workload and corresponding interface obtained above, we repeated the experiment and evaluated the respective performances of the system when setting the hypervisor scheduler to be WCPS, CRPS, and the baseline PTPS.

This set of experiments compared the *soft real-time* performance of the three different periodic servers. The PTPS, WCPS, and CRPS servers differ primarily in how idle time is utilized within the system. The idle time comes from two main sources: the interface overhead due to theoretical pessimism, and over-estimation of tasks' execution times (also called *slack*). Hence, we designed two sets of experiments to show the effect of different idle times: (1) The range for the workload periods was varied to create different interface overheads, and (2) The *ETF* for the jobs was varied so that if a job executes in less than its *WCET*, it could potentially give some *slack* to other domains.

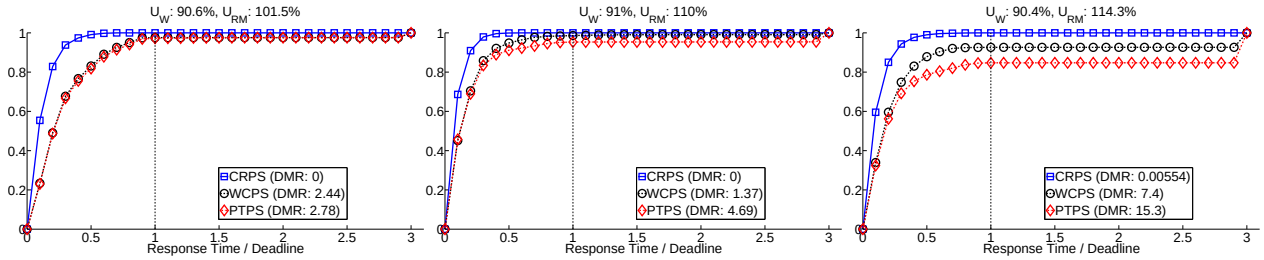
For *soft real-time* systems, we are interested not only in schedulable situations but also in overloaded situations. As a result, we ranged the U_W from 0.7 to 1.0, with steps of 0.1, to create different U_W conditions.

All the experiments were conducted as follows. We first defined a particular U_W, and then generated tasks (with utilization uniformly distributed between 0.2% and 5%) until the U_W was reached. The distributions of execution times were typically application dependent; here, we used the uniform distribution, which has been commonly used in the real-time scheduling literature [31, 38]. Using this generation method, the generated U_W is usually larger than the desired one, but would only be 0.05 more in the worst case. After all the tasks were generated, we randomly distributed the tasks among the five domains.

We ran each experiment for 5 minutes, and then calculated the $\frac{ResponseTime}{Deadline}$ for all the task sets within each domain of the experiment. For clarity of presentation, any job whose $\frac{ResponseTime}{Deadline}$ was greater than 3 was clipped at 3.

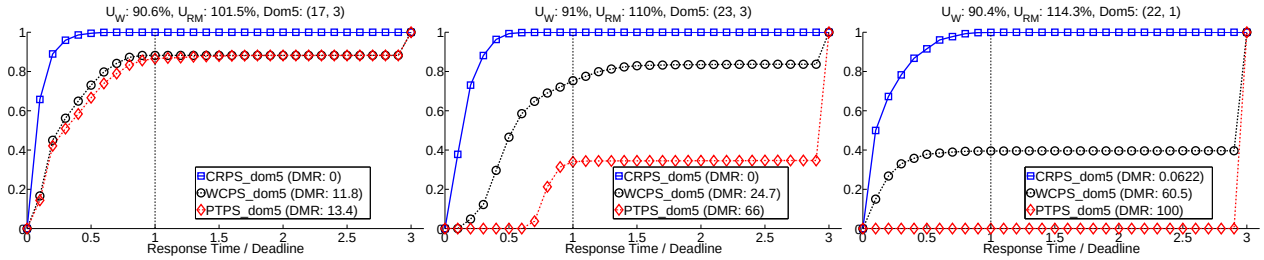
Impact of Task Period. We varied the task period range in this experiment to create different interface overheads, and then evaluated the three schedulers for the generated task sets. For each different U_W (from 0.7 to 1.0), we generated three different task sets whose periods were uniformly distributed between 550-650 ms, 350-850 ms, and 100-1100 ms, respectively. From the calculated interfaces, the 350-850 ms task period range gives the most interface overhead, followed by 100-1100 ms, and then 550-650 ms. For all the experiments, the *ETF* value was set to 100%. In other words, we let all jobs execute at their worst case execution times, so that the idle time came only from the interface overheads. Note that when the U_W was the same, we scheduled different task sets under different task periods.

Figure 2.7 shows the results for all domains under $U_W = 0.9$, where DMR means Deadline Miss Ratio. This $U_W(= 0.9)$ represents a typical heavily *overloaded* situation; other cases include those either guaranteed to be schedulable theoretically and incurring only negligible deadline miss ($U_W = 0.7$), not heavily *overloaded* ($U_W = 0.8$), or too *overloaded* to be schedulable ($U_W = 1.0$).



(a) 550-650 ms, Overhead: 10.9% (b) 100-1100 ms, Overhead: 19.1% (c) 350-850 ms, Overhead: 24%

Figure 2.7: CDF Plot of $\frac{ResponseTime}{Deadline}$ for All Tasks in Five Domains($U_W = 0.9$)



(a) 550-650 ms, Overhead: 10.9% (b) 100-1100 ms, Overhead: 19.1% (c) 350-850 ms, Overhead: 24%

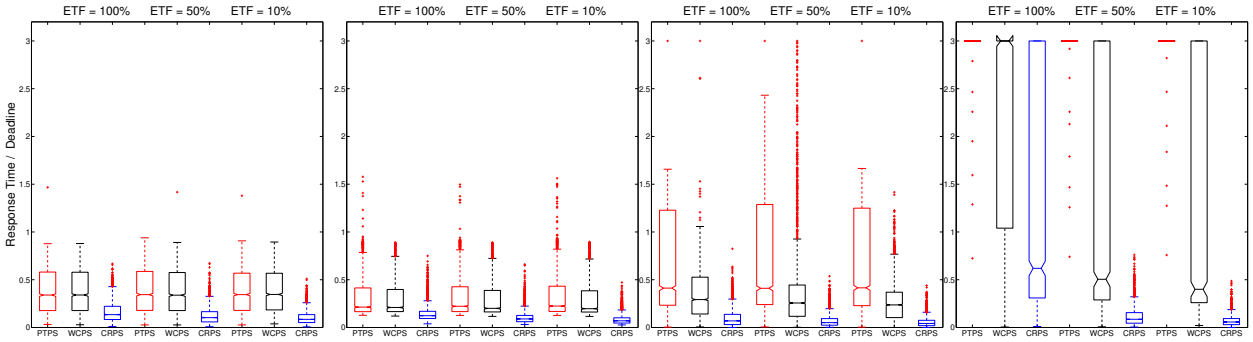
Figure 2.8: CDF Plot of $\frac{ResponseTime}{Deadline}$ for Tasks in the Lowest Priority Domain ($U_W = 0.9$)

Since we are using rate monotonic scheduling, the higher priority domains have shorter periods, and thus have a larger number of jobs. The data in Figure 2.7 are therefore dominated by the results for higher priority domains. Lower priority domains, though having fewer jobs, suffer most from the *overloaded* situation. Thus, we plot the data for the lowest priority domain (domain 5) in Figure 2.8 with the interface parameters given in the format of (*period*, *budget*). Figure 2.7 and Figure 2.8 clearly show that the CRPS outperforms the WCPS, which in turn outperforms the PTPS. Notably, with an interface overhead of 24% (Figure 2.8c), while all jobs miss their deadlines under the PTPS ($\frac{ResponseTime}{Deadline} > 1$), 60.5% and 6.2% of the jobs in domain 5 miss their deadlines under the WCPS and CRPS, respectively. These results demonstrate the effectiveness of the work-conserving and capacity-reclaiming

mechanisms in exploiting the interface overhead to improve the performance of low-priority domains. The CRPS is the most effective of these approaches for implementing the resource interfaces calculated by compositional scheduling analysis.

Impact of the *Execution Time Factor (ETF).* In *real-time* applications such as multimedia frame decoding, every frame may take a different amount of time to finish. Traditionally, the *WCET* is used to represent every task's execution time. This usually results in a relatively large interface, giving more idle time for the domain.

In this set of experiments, the same U_W ranging from 0.7 to 1.0 were used. Under each U_W, we generated only one task set. Then, for each particular task set, three *ETF* values (100%, 50%, and 10%) were configured for the three highest priority domains, while leaving the two low priority ones with an *ETF* of 100%. A lower *ETF* value means a lower "actual" U_W for that domain; for example, if an *ETF* of 10% is applied, all jobs' execution time is uniformly distributed between 10% and 100% of *WCET*. On average, the actual U_W is 55% ($\frac{100\%+10\%}{2}$). All task periods were uniformly distributed between 550 ms and 650 ms. We note that the idle time came not only from the interface overhead but also from the over-estimation of job execution times.



(a) $U_W = 0.703$ (b) $U_W = 0.832$ (c) $U_W = 0.91$ (d) $U_W = 1.021$

Figure 2.9: Box Plot of $\frac{ResponseTime}{Deadline}$ for Tasks in the Lowest Priority Domain under Different U_W and *ETF* Values

Figure 2.9 shows box plot results for all U_W for the lowest priority domain. Results for all domains exhibit the same behavior. On each box, the central mark represents the median value, whereas the upper and lower box edges show the 25th and 75th percentiles separately. If the data values are larger than $q_3 + 1.5 * (q_3 - q_1)$ or smaller than $q_1 - 1.5 * (q_3 - q_1)$ (where

q_3 and q_1 are the 75th and 25th percentiles, respectively), they are considered outliers and plotted via individual markers. Within each subfigure, the boxes are divided into three sets, from left to right, corresponding to the results under the *ETFs* of 100%, 50%, and 10%, respectively.

As shown in Figure 2.9, the CRPS again outperforms the WCPS and PTPS. In Figure 2.9c, the deadline miss ratio under the PTPS stays constant when the *ETF* is varied (26.9%, 27.3%, and 27.3% respectively), while performance improvement is seen under the WCPS (11.7%, 8.51%, and 0.49%) and CRPS (0.02%, 0%, and 0%). In an extremely *overloaded* situation (Figure 2.9d), all jobs missed their deadlines under the PTPS, whereas (75.6%, 32.7%, and 31.3%) of jobs missed their deadlines under the WCPS, and (36.1%, 0%, and 0%) of jobs missed their deadlines under the CRPS. This again demonstrates that the WCPS and the CRPS benefit from the idle time introduced by interface overheads and over-estimations of jobs' execution times.

2.8 Summary

RT-Xen 1.0 represents the first hierarchical real-time scheduling framework for Xen, a widely used open-source virtual machine monitor. RT-Xen bridges the gap between real-time scheduling theory and Xen, whose wide-spread adoption makes it an attractive virtualization platform for soft real-time and embedded systems. RT-Xen also provides an open-source platform for researchers to develop and evaluate real-time scheduling techniques. Extensive experimental results demonstrate the feasibility, efficiency, and efficacy of fixed-priority hierarchical real-time scheduling in the Xen VMM.

RT-Xen differs from prior efforts in real-time virtualization in several important aspects. A key technical contribution of RT-Xen is the instantiation and empirical study of a suite of fixed-priority servers (Deferrable Server, Periodic Server, Polling Server, and Sporadic Server) within a VMM.

Our empirical study represents the first comprehensive experimental comparison of these algorithms in the same virtualization platform. Our study shows that while more complex

algorithms do incur higher overhead, the overhead differences among different server algorithms are insignificant. However, in terms of *soft real-time* performance, deferrable server generally performs well, while periodic server performs worst under *overloaded* situations. Based on this observation, we present two enhanced variations of the *purely time-driven periodic server* to optimize run-time performance and resource-use efficiency, namely the *work-conserving periodic server* and the *capacity-reclaiming periodic server*. Both servers preserve the conservative compositional schedulability analysis, while yielding substantial improvements in observed response times and resource utilization, which are desirable for not only soft real-time applications but also many classes of hard real-time applications.

Chapter 3

RT-Xen 2.0: Multi-Core Real-Time Virtualization

Chapter 2 studied different server schemes under single-core scheduling with fixed priority. For this chapter, we further explore the effect of different priority schemes and scheduling policies on multi-core platform. We first review the multi-core hierarchical scheduling theories in Section 3.1, then introduce RT-Xen 2.0 in Section 3.2. In Section 3.3 we present an extensive evaluation both theoretically and experimentally, and we summarize in Section 3.4.

3.1 Multi-Core Hierarchical Scheduling Theories

In a single-core hierarchical scheduling theory, the VCPUs' resource interface is captured by *budget* and *period*, and different theories applied in several resource models (server mechanisms). For multi-core hierarchical scheduling theory, besides the resource model, there are also different ways to distribute the *budget* among multiple VCPUs. In RT-Xen 2.0, we focus on using compositional scheduling analysis [45], which represents the resource requirements of each domain as a multiprocessor periodic resource (MPR) interface, $\mu = \langle \Pi, \Theta, m' \rangle$, which specifies a resource allocation that provides a total of Θ execution time units in each period of Π time units, with a maximum level of parallelism m'.

Other theories differ in the distribution of the budget among multiple VCPUs, and also in whether to use a uniform *period* or not. We designed the interface of RT-Xen 2.0 to be compatible with most of them, which is discussed in the next Section.

3.2 RT-Xen 2.0: Design and Implementation

In this section, we first describe the design principles behind the multi-core real-time scheduling framework of RT-Xen 2.0, and then we discuss our implementation in detail.

3.2.1 Design Principles

To leverage multi-core platforms effectively in real-time virtualization, we designed RT-Xen 2.0 to cover three dimensions of the design space: global and partitioned scheduling, dynamic and static priority schemes, and two server schemes (deferrable and periodic) for running the VMs. In summary, RT-Xen 2.0 supports:

- a scheduling interface that is compatible with a range of resource interfaces used in compositional schedulability theory (e.g., [36, 45, 75]);
- both global and partitioned schedulers, called rt-global and rt-partition, respectively;
- EDF and DM priority schemes for both schedulers; and
- for each scheduler, a choice of either a work-conserving deferrable server or a periodic server.

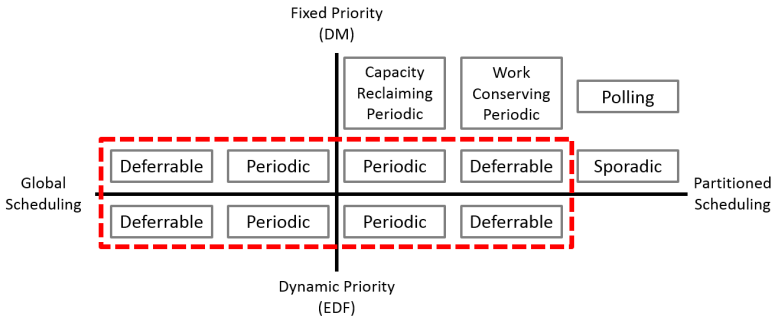


Figure 3.1: Design Space of RT-Xen Scheduling Framework.

We next discuss each dimension of the design space, focusing on how theory and platform considerations influenced our design decisions.

Scheduling interface. In RT-Xen 2.0, the scheduling interface of a domain specifies the amount of resource allocated to the domain by the VMM scheduler. In a single-core virtualization setting, each domain has only one VCPU, and thus its scheduling interface can be defined by a *budget* and a *period* [56]. In contrast, each domain in a multi-core virtualization setting can have multiple VCPUs. As a result, the scheduling interface needs to be sufficiently expressive to enable resource allocation for different VCPUs at the VMM level. At the same time, it should be highly flexible to support a broad range of resource interfaces, as well as different distributions of interface bandwidth to VCPUs (to be scheduled by the VMM scheduler), according to the compositional scheduling theory.

Accordingly, RT-Xen 2.0 defines the scheduling interface of a domain to be a set of VCPU interfaces, where each VCPU interface is represented by a *budget*, a *period*, and a *cpu_mask* (which gives the subset of PCPUs on which the VCPU is allowed to run), all of which can be set independently of other VCPUs. This scheduling interface has several benefits: (1) it can be used directly by the VMM scheduler to schedule the VCPUs of the domains; (2) it can be configured to support different resource interfaces from the compositional scheduling analysis literature, such as MPR interfaces [45], deterministic MPR interfaces [75], and multi-supply function interfaces [36]; (3) it is compatible with different distributions of interface bandwidth to VCPU budgets, such as one that distributes budget equally among the VCPUs [45], or one that provides the maximum bandwidth (equal to 1) to all but one VCPU [75]; and finally (4) it enables the use of CPU-mask-aware scheduling strategies, such as one that dedicates a subsets of PCPUs to some VCPUs and schedules the rest of the VCPUs on the remaining PCPUs [75].

Global vs. partitioned schedulers. Different multi-core schedulers require different implementation strategies and provide different performance benefits. A partitioned scheduler schedules VCPUs only in its own core's run queue and hence is simple to implement; in contrast, a global scheduler schedules all VCPUs in the system and thus is more complex but can provide better resource utilization. We support both by implementing two schedulers in RT-Xen 2.0: rt-global and rt-partition.[2] The rt-partition scheduler uses a partitioned queue scheme, whereas the rt-global scheduler uses a global shared run queue that is protected

[2]In our current platform, all cores share an L3 cache, thus limiting the potential benefits of cluster-based schedulers; however, we plan to consider cluster-based schedulers in our future work on new platforms with multiple multi-core sockets.

by a spinlock. An alternative approach to approximating a global scheduling policy is to employ partitioned queues that can push/pull threads from each other [34] (as adopted in the Linux Kernel). We opt for a simple global queue design because the locking overhead for the shared global queue is typically small in practice, since each host usually runs only relatively few VMs. For each scheduler, users can switch between dynamic priority and static priority schemes on the fly.

Server mechanisms. Each VCPU in RT-Xen 2.0 is associated with a *period* and a *budget*, and is implemented as a server: the VCPU is released periodically and its *budget* is replenished at the beginning of every *period*. It consumes its *budget* when running, and it stops running when its *budget* is exhausted. Different server mechanisms provide different ways to schedule the VCPUs when the current highest priority VCPU is not runnable (i.e., has no jobs to execute) but still has unused *budget*. For instance, when implemented as a deferrable server, the current VCPU defers its unused *budget* to be used at a later time within its current *period* if it becomes runnable, and the highest-priority VCPU among the runnable VCPUs is scheduled to run. In contrast, when implemented as a periodic server, the current VCPU continues to run and consume its *budget* (as if it had a background task executing within it). As shown by our experimental results in Section 3.3.5, results can be quite different when different servers are used, even if the scheduler is the same. We implemented both rt-global and rt-partition as deferrable servers, and can configure them as periodic servers by running a lowest priority CPU-intensive task in a guest VCPU.

3.2.2 Implementation

We first introduce the run queue structure of the rt-global scheduler, followed by that of the rt-partition scheduler, which has a simpler run queue structure. We then describe the key scheduling functions in both schedulers.

Run queue structure. Figure 3.2 shows the structure of the global run queue (RunQ) of the rt-global scheduler, which is shared by all physical cores. The RunQ holds all the runnable VCPUs, and is protected by a global spin-lock. Within this queue, the VCPUs are divided into two sections: the first consists of the VCPUs with a nonzero remaining budget, and the second consists of VCPUs that have no remaining budget. Within each section,

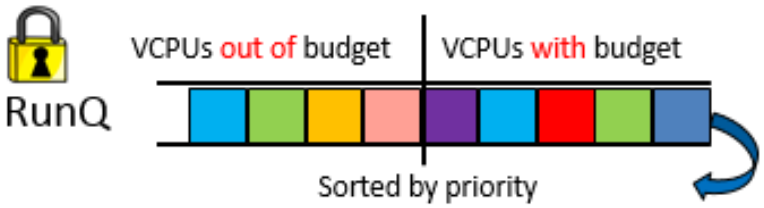


Figure 3.2: rt-global run queue structure

the VCPUs are sorted based on their priorities (determined by a chosen priority assignment scheme). We implemented both EDF and DM priority schemes in RT-Xen 2.0. A RunQ of the rt-partition scheduler follows the same structure as the RunQ in rt-global, except that it is not protected by a spinlock, and the rt-partition scheduler maintains a separate run queue for each core.

Scheduling functions: The scheduling procedure consists of two steps: first, the scheduler triggers the scheduler-specific *do_schedule* function to make scheduling decisions; then, if necessary, it triggers the *context_switch* function to switch the VCPUs, and after that, a *context_saved* function to put the currently running VCPU back into the run queue (only in shared run queue schedulers).

Algorithm 2 shows the pseudo-code of the *do_schedule* function of the rt-global scheduler under an EDF priority scheme (i.e., gEDF). In the first `for` loop (Lines 5–10), it replenishes the budgets of the VCPUs and rearranges the VCPUs in the RunQ appropriately. In the second `for` loop (Lines 11–18), it selects the highest-priority runnable VCPU (snext) that can be executed. Finally, it compares the deadline of the selected VCPU (snext) with that of the currently running VCPU (scurr), and returns the VCPU to be executed next (Lines 19–24).

There are two key differences between RT-Xen 2.0 and the single-core scheduling algorithm in RT-Xen [73]: (1) the second `for loop` (Lines 11–18) guarantees that the scheduler is CPU-mask aware; and (2) if the scheduler decides to switch VCPUs (Lines 22–24), the currently running VCPU (scurr) is **not** inserted back into the run queue; otherwise, it could be grabbed by another physical core before its context is saved (since the run queue is shared among all cores), which would then make the VCPU's state inconsistent. For this reason, Xen adds another scheduler-dependent function named *context_saved*, which is invoked at

the end of a *context_switch* to insert scurr back into the run queue if it is still runnable. Note that both *do_schedule* and *context_saved* need to grab the spin-lock before running; since this is done in the Xen scheduling framework, we do not show this in Algorithm 2.

Algorithm 2 *do_schedule* function for rt-global under EDF.

```
 1: scurr ← the currently running VCPU on this PCPU
 2: idleVCPU ← the idle VCPU on this PCPU
 3: snext ← idleVCPU
 4: burn_budget(scurr)
 5: for all VCPUs in the RunQ do
 6:    if VCPU's new period starts then
 7:       reset VCPU.deadline, replenish VCPU.budget
 8:       move VCPU to the appropriate place in the RunQ
 9:    end if
10: end for
11: for all VCPUs in the RunQ do
12:    if VCPU.cpu_mask & this PCPU ≠ 0 then
13:       if VCPU.budget > 0 then
14:          snext ← VCPU
15:          break
16:       end if
17:    end if
18: end for
19: if (snext = idleVCPU or snext.deadline > scurr.deadline)
      and (scurr ≠ idleVCPU) and (scurr.budget > 0)
      and vcpu_runnable(scurr) then
20:    snext ← scurr
21: end if
22: if snext ≠ scurr then
23:    remove snext from the RunQ
24: end if
25: return snext to run for 1 ms
```

Another essential function of the scheduler is the *wake_up* function, which is called when a domain receives a packet or a timer fires within it. In the *wake_up* function of the rt-global scheduler, we issue only an interrupt if there is a currently running VCPU with a lower priority than the domain's VCPUs, so as to reduce overhead and to avoid priority inversions. We also implemented a simple heuristic to minimize the cache miss penalty due to VCPU migrations: whenever there are multiple cores available, we assign the previously scheduled core first.

The *do_schedule* function of the rt-partition scheduler is similar to that of the rt-global scheduler, except that (1) it does not need to consider the CPU mask when operating on a local run queue (because VCPUs have already been partitioned and allocated to PCPUs based on the CPU mask), and (2) if the scheduler decides to switch VCPUs, the currently running VCPU scurr will be immediately inserted back into the run queue. In addition, in the *wake_up* function, we compare only the priority of the VCPU that is being woken up to the priority of the currently running VCPU, and we perform a switch if necessary.

We implemented both rt-global and rt-partition schedulers in C. We also patched the Xen tool for adjusting the parameters of a VCPU on the fly. Our modifications were done solely within Xen. The source code of RT-Xen 2.0 and the data used in our experiments are both available on-line via the RT-Xen website: *https://sites.google.com/site/realtimexen*.

3.3 RT-Xen 2.0: Evaluation

In this section, we present our experimental evaluation of RT-Xen. We have five objectives for our evaluation:

- (1) to evaluate the scheduling overhead of the rt-global and rt-partition schedulers compared to the default Xen credit scheduler;
- (2) to experimentally evaluate the schedulability of the system under different combinations of schedulers at the guest OS and VMM levels;
- (3) to evaluate the real-time system performance under RT-Xen schedulers in overload situations;
- (4) to compare the performance of the deferrable server scheme and the periodic server scheme;
- (5) to evaluate the impact of cache on global and partitioned schedulers.

3.3.1 Experiment Setup

We performed our experiments on an Intel i7 x980 machine, with six cores (PCPUs) running at 3.33 GHz. We disabled hyper-threading and SpeedStep to ensure constant CPU speed, and we shut down all other non-essential processes during our experiments to minimize interference. The scheduling quantum for RT-Xen was set to 1 ms. Xen 4.3 was patched with RT-Xen and installed with a 64-bit Linux 3.9 kernel as domain 0, and a 64-bit Ubuntu image with a para-virtualized Linux kernel as the guest domain. For all experiments, we booted domain 0 with one VCPU and pinned this VCPU to one core; the remaining five cores were used to run the guest domains. In addition, we patched the guest OS with LITMUS$^{\mathrm{RT}}$ [16] to support EDF scheduling.

For the partitioned scheduling policy at the guest OS level and the VMM level, we used a variant of the best-fit bin-packing algorithm for assigning tasks to VCPUs and VCPUs to cores, respectively. Specifically, for each domain, we assigned a task to the VCPU with the *largest* current bandwidth[3] among all existing VCPUs of the domain that could feasibly schedule the task. Since the number of VCPUs of the domain was unknown, we started with one VCPU for the domain, and added a new VCPU when the current task could not be packed into any existing VCPU. At the VMM level, we assigned VCPUs to the available cores in the same manner, except that (1) in order to maximize the amount of parallelism available to each domain, we tried to avoid assigning VCPUs from the same domain to the same core, and (2) under an overload condition, when the scheduler determined that it was not feasible to schedule the current VCPU on any core, we assigned that VCPU to the core with the *smallest* current bandwidth, so as to balance the load among cores.

We performed the same experiments as above using the credit scheduler, with both the *weight* and the *cap* of each domain configured to be the total bandwidth of its VCPUs. (Recall that the bandwidth of a VCPU is the ratio of its *budget* to its *period*.) The CPU-mask of each VCPU was configured to be 1-5 (the same as in the rt-global scheduler).

[3]The maximum bandwidth of a VCPU is 1, since we assume that it can execute on only one core at a time.

3.3.2 Workloads

In our experiments, tasks were created based on the *base_task* provided by LITMUS^{RT}. To emulate a desirable execution time for a task in RT-Xen, we first calibrated a CPU-intensive job to take 1 ms in the guest OS (when running without any interference), then scaled it to the desirable execution time. For each task set, we ran each experiment for 60 seconds, and recorded the deadline miss ratio for each task using the *st_trace* tool provided by LITMUS^{RT}.

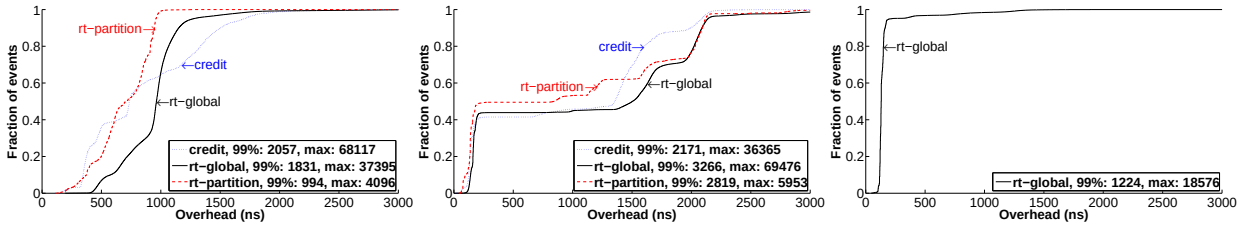
Following the methodology used in [35] to generate real-time tasks, our evaluation used synthetic real-time task sets. The tasks' periods were chosen uniformly at random between 350-850 ms, and the tasks' deadlines were set equal to their *periods*. The tasks' utilizations followed the medium bimodal distribution, where the utilizations are distributed uniformly over [0.0001, 0.5) with a probability of 2/3, or [0.5, 0.9] with a probability of 1/3. Since there were five cores for running the guest domains, we generated task sets with total utilizations ranging from 1.1 to 4.9, with a step of 0.2. For a specific utilization, we first generated tasks until we exceeded the specified total task utilization, then we discarded the last generated task and used a "pad" task to make the task set utilization exactly match the specified utilization. For each of the 20 task set utilizations, we used different random seeds to generate 25 task sets. In total, there were 20 (utilization values) $\times$ 25 (random seeds) = 500 task sets in our experiments.

Each generated task was then distributed into four different domains in a round robin fashion. We applied compositional scheduling analysis to compute the interface of each domain, and to map the computed interface into a set of VCPUs to be scheduled by the VMM scheduler. In our evaluation, we used harmonic periods for all VCPUs. We first evaluated the real-time schedulers using CPU-intensive tasks in the experiments, followed by a study on the impacts of cache on the different real-time schedulers using cache-intensive tasks with large memory footprints (Section 3.3.6).

3.3.3 Scheduling Overhead

In order to measure the overheads for different schedulers, we booted four domains, each with four VCPUs. We set each VCPU's bandwidth to 20%, and distributed the VCPUs to five

PCPUs for the rt-partition scheduler in a round robin fashion; for the rt-global and credit schedulers, we allowed all guest VCPUs to run on all five PCPUs. We ran a CPU-intensive workload with a total utilization of 3.10. We used the EDF scheme in both rt-global and rt-partition schedulers, as the different priority schemes differ only in their placement of a VCPU in the RunQ. In the Xen scheduling framework, there are three key functions related to schedulers, as described in Section 3.2.2. We measured the overheads of the time spent in the *do_schedule* function as *scheduling latency*, the time spent in the *context_switch*, and the time spent in the *context_saved*. Note that *context_saved* is necessary only in rt-global schedulers, as they have shared queues. For rt-partition and credit schedulers, the running VCPU is inserted back to run queue in *do_schedule* function. To record these overheads, we modified *xentrace* [50] and used it to record data for 30 seconds.



(a) *do_schedule* (b) *context_switch* (c) *context_saved*

Figure 3.3: CDF Plot for Scheduling Overhead for Different Schedulers over 30 Seconds

Figure 3.3 shows CDF plots of the time spent in the three functions for different schedulers. Since 99% of the values are smaller than 3 microseconds (except for rt-global in the *context_switch* function, which is 3.266 microseconds), we cut the X-axis at 3 microseconds for a clearer view, and included the 99% and maximum values in the legend for each scheduler. We observe the following:

First, as is shown in Figure 3.3a, the rt-global scheduler incurred a higher scheduling latency than the rt-partition scheduler. The rt-global scheduler experienced the overhead to grab the spinlock, and it had a run queue that was five times longer than that of the rt-partition scheduler. The credit scheduler performed better than the rt-global scheduler in the lower 60%, but performed worse in the higher 40% of our measurements. We attribute this to the load balancing scheme in the credit scheduler, which must check all other PCPUs' RunQs to "steal" VCPUs.

Second, Figure 3.3b shows that the context switch overheads for all three schedulers were largely divided into two phases: approximately 50% of the overhead was around 200 nanoseconds, and the remaining half was more than 1500 nanoseconds. We find that the first 50% (with lower overhead) ran without actually performing context switches, since Xen defers the actual context switch until necessary: when the scheduler switches from a guest VCPU to the IDLE VCPU, or from the IDLE VCPU to a guest VCPU with its context still intact, the time spent in the *context_switch* function is much shorter than a context switch between two different guest VCPUs. We can also observe that *context_switch* in rt-global has a higher overhead. We attribute this to the global scheduling policy, where the VMM scheduler moves VCPUs around all PCPUs, which would cause more preemptions than a partitioned scheduling policy like rt-partition.

Third, Figure 3.3c shows the time spent in the *context_saved* function for the rt-global scheduler. Recall that this function is NULL in the rt-partition and credit schedulers, since the current VCPU is already inserted back into the run queue by the *do_schedule* function. We observe that, for the rt-global scheduler, around 90% of the overhead was 200 nanoseconds or less, and the 99% value was only 1224 nanoseconds. We attribute this to the extra overhead of grabbing the spinlock to access the shared run queue in the rt-global scheduler.

Overall, in 99% of the cases, the overhead of all three functions (do_schedule, context_switch, and context_saved) for all schedulers was smaller than four microseconds. Since we use a 1 ms scheduling quantum for both the rt-global and the rt-partition schedulers, an overhead of four microseconds corresponds to a resource loss of only 1.2% per scheduling quantum. Notably, in contrast to an OS scheduler – which is expected to handle a large number of tasks – the VMM scheduler usually runs fewer than 100 VCPUs, as each VCPU typically demands much more resources than a single task. As a result, the run queue is typically much shorter, and the overhead for grabbing the lock in a shared run queue is typically smaller than in an OS scheduler.

Summary: Both rt-global and rt-partitioned schedulers incur moderate overhead.

3.3.4 Multi-Core Real-Time Performance of Credit Scheduler

We conducted experiments to compare the real-time performance of the default credit scheduler in Xen and our RT-Xen schedulers. All guest domains ran the pEDF scheduler in the guest OS. For the credit scheduler, we configured each domain's *weight* and *cap* as the sum of all its VCPU's bandwidths, as described in Section 3.3.1. For comparison, we also plotted the results for the gEDF and gDM schedulers in RT-Xen using a periodic server.

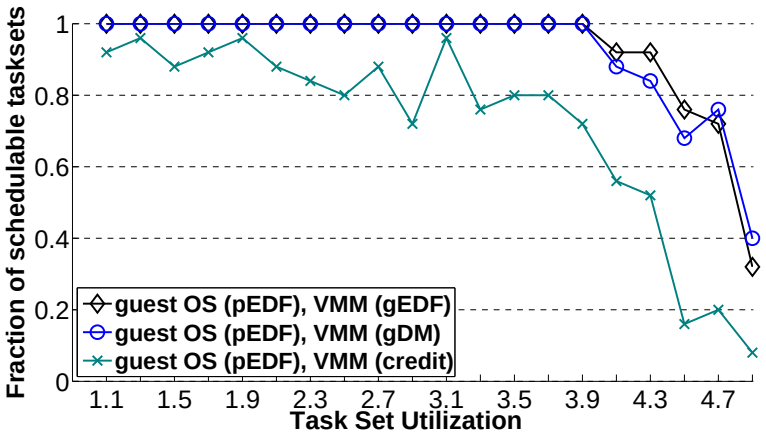


Figure 3.4: Credit vs. RT-Xen schedulers

Figure 3.4 shows the results. With the credit scheduler, even when the total workload utilization is as low as 1.1 (22% of the CPU capacity), 10 % of the task sets experienced deadline misses, which clearly demonstrates that the credit scheduler is not suitable for scheduling VMs that contain real-time applications. In contrast, our real-time VM schedulers based on the gEDF and gDM policies can meet the deadlines of all task sets at utilizations as high as 3.9 (78% of the CPU capacity). This result highlights the importance of incorporating real-time VM schedulers in multi-core hypervisors such as Xen. In the following subsections, we compare different real-time VM scheduling policies in RT-Xen.

Summary: Credit scheduler cannot deliver the real-time performance to VMs, while RT-Xen can reach a CPU capacity as high as 78%.

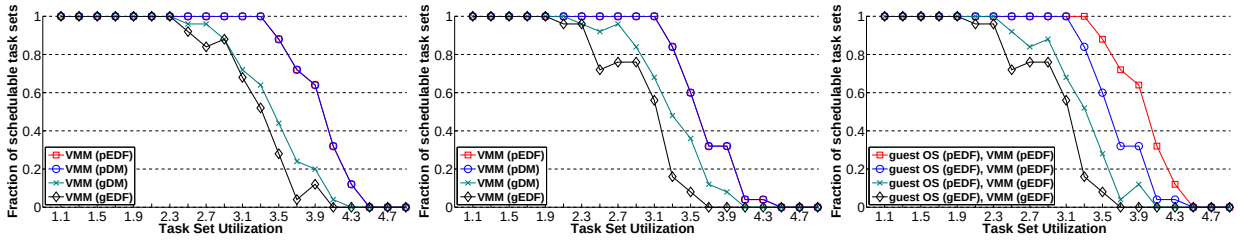
3.3.5 Comparison of Real-Time Scheduling Policies

We now evaluate different real-time VM scheduling policies supported by RT-Xen. We first compare their capability to provide theoretical schedulability guarantees based on compositional scheduling analysis. We then experimentally evaluate their capability to meet the deadlines of real-time tasks in VMs on a real multi-core machine. This approach allows us to compare the theoretical guarantees and experimental performance of real-time VM scheduling, as well as the real-time performance of different combinations of real-time scheduling policies at the VMM and guest OS levels. In both theoretical and experimental evaluations, we used the medium-bimodal distribution, and we performed the experiments for all 25 task sets per utilization under the rt-global and rt-partition schedulers.

Theoretical guarantees. To evaluate the four scheduling policies at the VMM level, we fixed the guest OS scheduler to be either pEDF or gEDF, and we varied the VMM scheduler among the four schedulers (pEDF, gEDF, pDM and gDM). For each configuration, we performed the schedulability test for every task set.

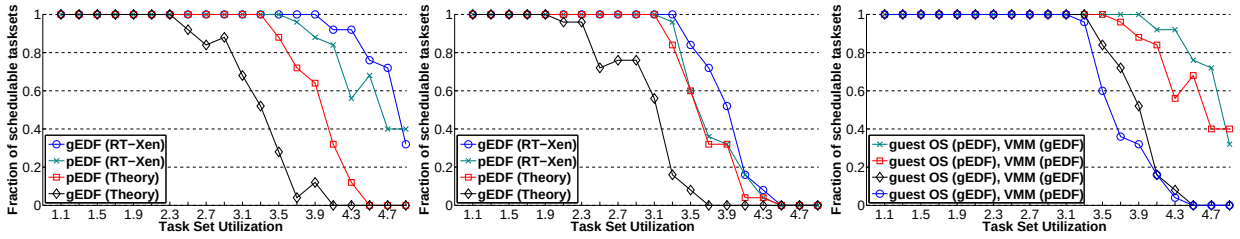
Performance of the four schedulers at the VMM level: Figures 3.5(a) and 3.5(b) show the fraction of schedulable task sets for the four schedulers at the VMM level with respect to the task set utilization when fixing pEDF or gEDF as the guest OS scheduler, respectively. The results show that, when we fix the guest OS scheduler, the pEDF and pDM schedulers at the VMM level can provide theoretical schedulability guarantees for more task sets than the gDM scheduler, which in turn outperforms the gEDF scheduler, for all utilizations. Note that the fraction of schedulable task sets of the pEDF scheduler is the same as that of the pDM scheduler. This is because the set of VCPUs to be scheduled by the VMM is the same for both pDM and pEDF schedulers (since we fixed the guest OS scheduler), and these VCPUs have harmonic periods; as a result, the utilization bounds under both schedulers are both equal to 1. The results also show that the partitioned schedulers usually outperformed the global schedulers in terms of theoretical schedulability.

Combination of EDF schedulers at both levels: Figure 3.5(c) shows the fraction of schedulable task sets for each task set utilization under four different combinations of the EDF priority assignment at the guest OS and the VMM levels. The results show a consistent order among



(a) Guest OS with pEDF (b) Guest OS wiht gEDF (c) EDF for both Layers

Figure 3.5: Theoretical Results: Schedulability of Different Schedulers.



(a) Guest OS with pEDF (b) Guest OS with gEDF (c) Experimental results

Figure 3.6: Experimental vs. Theoretical Results: Schedulability of Different Schedulers.

the four combinations in terms of theoretical schedulability (from best to worst): (pEDF, pEDF), (gEDF, pEDF), (pEDF, gEDF), and (gEDF, gEDF).

Experimental evaluation on RT-Xen. From the above theoretical results, we observed that pEDF and gEDF have the best and the worst theoretical performance at both levels. Henceforth, we focus on EDF results in the experimental evaluation. We have not observed statistically distinguishable differences between DM and EDF scheduling in their empirical performance, and the DM results follows similar trends to EDF scheduling. We include the DM results in the appendix for completeness.

Experimental vs. theoretical results: Figures 3.6(a) and 3.6(b) show the fractions of schedulable task sets that were predicted by the compositional scheduling analysis and that were observed on RT-Xen for the two EDF schedulers at the VMM level, when fixing pEDF or gEDF as the guest OS scheduler, respectively. We examine all 25 task sets for each level of system, and we find that whenever a task set used in our evaluation is schedulable according to the theoretical analysis, it is also schedulable under the corresponding scheduler on RT-Xen in our experiments. In addition, for both pEDF and gEDF schedulers, the fractions of

schedulable task sets observed on RT-Xen are always larger than or equal to those predicted by the theoretical analysis. The results also show that, in contrast to the trend predicted in theory, the gEDF scheduler at the VMM level can often schedule more task sets empirically than the pEDF scheduler. We attribute this to the pessimism of the gEDF schedulability analysis when applied to the VMM level, but gEDF is an effective real-time scheduling policy in practice due to its flexibility to migrate VMs among cores.

Combination of EDF schedulers at both levels: Figure 3.6(c) shows the fraction of empirically schedulable task sets at different levels of system utilization under four different combinations of EDF policies at the guest OS and VMM levels. The results show that, at the guest OS level, the pEDF scheduler always outperform the gEDF scheduler. Further, if we fix pEDF (gEDF) for the guest OS scheduler, the gEDF scheduler at the VMM level can often schedule more task sets than the pEDF scheduler.

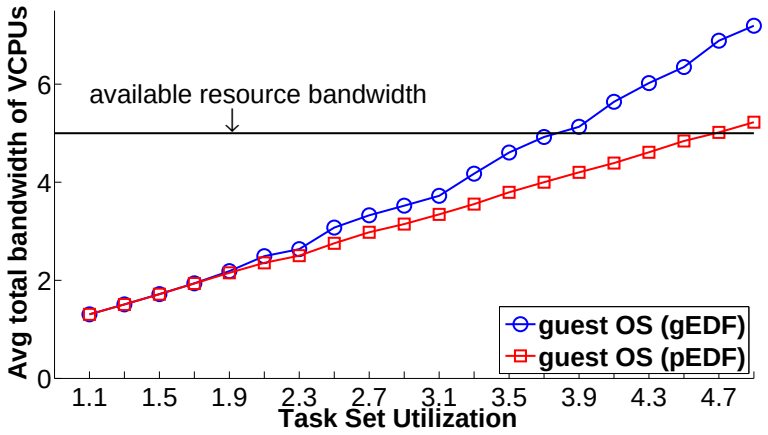


Figure 3.7: Average Total VCPU Bandwidth Comparison.

To explain the relative performance of pEDF and gEDF in a two-level scheduling hierarchy, we investigated the corresponding set of VCPUs that are scheduled by the VMM when varying the guest OS scheduler. For the same task set, the VCPUs of a domain under the pEDF and gEDF schedulers can be different; hence, the set of VCPUs to be scheduled by the VMM can also be different. Figure 3.7 shows the total bandwidth of all the VCPUs that are scheduled by the VMM – averaged across all 25 task sets – at each level of system utilization for the pEDF and gEDF schedulers at the guest-OS level. The horizontal line represents the total available resource bandwidth (with 5 cores).

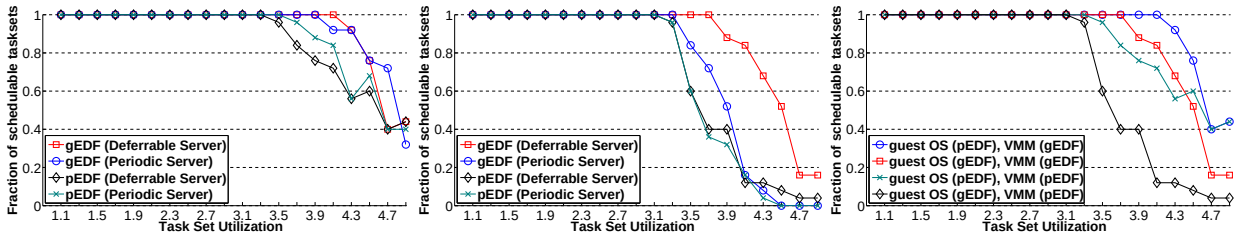
The figure shows that gEDF as the guest OS scheduler results in a higher average total VCPU bandwidth than pEDF; therefore, the extra resource that the VMM allocates to the VCPUs (compared to that actually required by their tasks) is much higher under gEDF. Since the resource that is unused by tasks of a higher priority VCPU cannot be used by tasks of lower-priority VCPUs when VCPUs are implemented as periodic servers, more resources were wasted under gEDF. In an overloaded situation, where the underlying platform cannot provide enough resources at the VMM level, the lower-priority VCPUs will likely miss deadlines. Therefore the poor performance of gEDF at the guest OS level results from the combination of pessimistic resource interfaces based on the compositional scheduling analysis and the non-work-conserving nature of periodic server. We also study deferrable server, a work-conserving mechanism for implementing VCPUs.

In contrast, when we fix the guest OS scheduler to be either pEDF or gEDF, the set of VCPUs that is scheduled by the VMM is also fixed. As a result, we observe that more VCPUs are schedulable on RT-Xen under the gEDF scheduler than under the pEDF scheduler at the VMM level (c.f., Figure 3.6(c)). This is consistent with our earlier observation, that the gEDF scheduler can often schedule more task sets than the pEDF scheduler empirically because of the flexibility to migrate VMs among cores.

Comparison between periodic server and deferrable server: As observed in the last set of experiments, realizing VMs as periodic servers suffers from the non-work-conserving nature of the periodic server algorithm. A deferrable server, on the other hand, implements VMs in a work-conserving fashion.[4] Thus, we repeat the experiments in Section 3.3.5 with a deferrable server configuration.

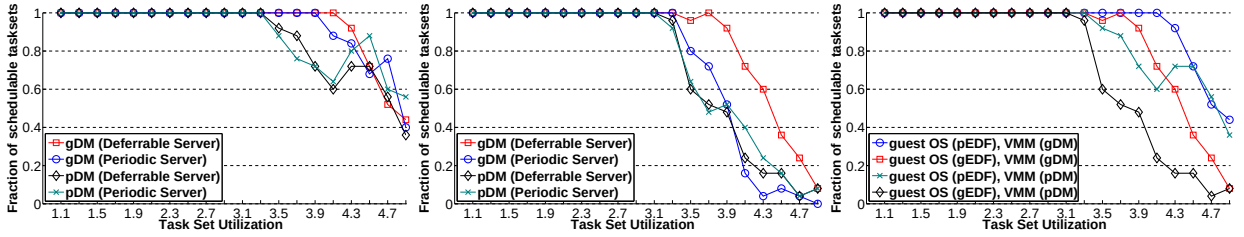
Figure 3.8a and Figure 3.8b show the fraction of schedulable task sets with the periodic server and with the deferrable server, respectively. It can be observed that, when pEDF is used in the guest OS (Figure 3.8a), the two servers are incomparable in terms of the fraction of schedulable task sets. There is little slack time in each VCPU's schedule (recall from Figure 3.7 that the total VCPU bandwidth for pEDF in the guest OS is close to the actual

[4]Theoretically, it is well known that the deferrable server scheme can suffer from back-to-back effects, in which a higher-priority server executes back to back, causing lower-priority servers to miss deadlines. While the back-to-back effect affects deferrable server's capability to provide theoretical schedulability guarantees, in practice the back-to-back effect happens infrequently, and its negative impacts are often offset by the benefits of the work-conserving property of deferrable server, as shown in our experimental results.



(a) Guest OS with pEDF (b) Guest OS with gEDF (c) Deferrable Server's Performance

Figure 3.8: Fraction of Schedulable Task Sets (EDF in RT-Xen)



(a) Guest OS with pEDF (b) Guest OS with gEDF (c) Deferrable Server's Performance

Figure 3.9: Fraction of Schedulable Task Sets (DM in RT-Xen)

task set utilization) and thus a deferrable server behaves almost like a periodic server. In contrast, when the guest OS is using gEDF (Figure 3.8b), using gEDF in the VMM with a deferrable server clearly outperforms the other three combinations. We attribute this to the work-conserving behavior of the deferrable server, which can take advantage of the available slack time at runtime to improve the system schedulability. We also plotted the results for DM priority schemes in RT-Xen in Figure 3.9, and the conclusions are similar to the ones using EDF priority scheme.

Another interesting observation is that, when pEDF is used in RT-Xen, the difference between the performance of the two servers is not obvious. We attribute this to the VCPU parameters calculated based on compositional scheduling analysis: The computed bandwidth of a VCPU is often larger than half of the available bandwidth of a PCPU. As a result, when a partitioned scheduler is used in RT-Xen, every PCPU is either able to feasibly schedule all tasks (if it executes only one VCPU) or is heavily overloaded (if it executes two or more VCPUs). In the former case, there is no deadline miss on the PCPU under either server; in the latter, using deferrable server cannot help improve the deadline miss ratio much, since there is often no slack available when the PCPU is heavily overloaded.

Finally, Figure 3.8c shows four configurations of the gEDF and pEDF scheduling policies with a deferrable server. We can observe that generally global scheduling in VMM outperforms partitioned scheduling empirically. Further, for the same VMM scheduler, using pEDF in the guest OS results in better performance than using gEDF.

Summary: gEDF combined with deferrable server delivers the best real-time performance among all RT-Xen schedulers.

3.3.6 Experimental Results for Cache Intensive Workloads

Our previous experiments use CPU-intensive workloads with small memory footprints. In comparison, due to cache penalty, a memory-intensive workload may be more affected by VCPU migrations caused by a global VM scheduler. To study the impacts of cache effects, we conducted a new empirical comparison between rt-global and rt-partition schedulers using a memory-intensive workload. The Intel i7 processor used in this set of experiments contained 6 cores. Each core owned dedicated L1 (32KB data, 32KB instruction) and L2 (256KB unified) caches, while all cores shared a unified 12MB L3 cache. The last-level cache is inclusive [12], which means the data that is in a PCPU's L2 cache must also be in the shared L3 cache. Therefore, the cache penalty of a VCPU migration is usually associated with latency difference between core-specific private caches (L1 or L2) and the shared L3 cache. On the i7 processor, the latency difference between the L2 and L3 cache is 18 cycles [13], about 5 nano-seconds per cache line (64B). The local L2 cache size is 256 KB (4096 cache lines), therefore, a VCPU migration may result in a cache penalty as high as 4096 x 5 ns = 20 μs. However, due to the widely used cache pre-fetch technology, the observed migration penalty is usually much less than the worst case. In comparison to a VMM scheduling quantum of 1 ms, we hypothesize that the VCPU migration would not incur a significant performance penalty. [5]

To create a significant cache penalty from VCPU migrations, we designed the memory access pattern of our tasks as follows. We allowed each task to access a fixed sized array within the L2 cache. The access pattern was one element per cache line, and we stored the next element's

[5]This analysis is valid only for processors with a shared last-level cache. For platforms where the last-level cache is not shared, global scheduler can cause last-level cache miss and result a higher penalty, as shown in an earlier study on global scheduling at the OS level [35].

index in the current element, so that it was data dependent and the improvement from cache pre-fetch could be mitigated. Recent work in compositional scheduling theory also considers cache impact [75], but assumes there is no shared cache. Therefore, we kept the other parameters of the task sets the same as in our previously generated workload. The impact of cache has received significant attention in the context of one level scheduling [30, 59, 76]; we defer integrating these insights into a two-level hierarchal scheduling to future work.

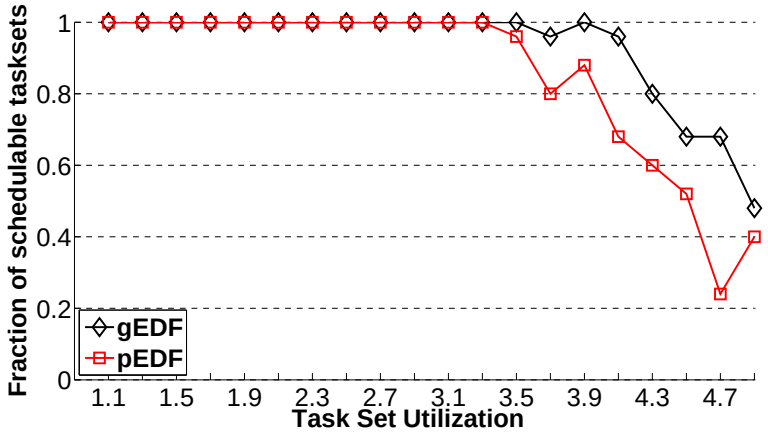


Figure 3.10: Cache-Intensive Workloads (guest OS with pEDF)

We used pEDF in the guest OS so that the cache penalties were attributed only to the VMM-level schedulers. We compared the real-time performance of the gEDF and pEDF VMM schedulers. As shown in Figure 3.10, gEDF again outperforms pEDF despite the cache penalty. This confirms that the benefits of a global scheduling policy outweighs the cache penalty caused by the VCPU migration on a multi-core platformed with shared last-level cache.

Summary: Benefit of global scheduling dominate migration cost on a shared L3-cache platform.

3.4 Summary

RT-Xen 2.0 realizes global and partitioned VM schedulers, and each scheduler can be configured to support dynamic or static priorities, and to run VMs as periodic or deferrable

servers. Through a comprehensive experimental study, we show that both global and partitioned VM scheduling can be implemented in the VMM at moderate overhead. Moreover, at the VMM scheduler level, in compositional schedulability theory pEDF is better than gEDF in schedulability guarantees, but in our experiments their actual performances are reversed in terms of the fraction of workloads that meet their deadlines on virtualized multi-core platforms. At the guest OS level, pEDF requests a smaller total VCPU bandwidth than gEDF based on compositional schedulability analysis, and therefore using pEDF in the guest OS level leads to more schedulable workloads on a virtualized multi-core processor. The combination of pEDF in guest OS and gEDF in the VMM therefore resulted the best experimental performance. Finally, on a platform with a shared last-level cache, the benefits of global scheduling outweigh the cache penalty incurred by VM migration.

Chapter 4

RT-OpenStack: Real-Time Cloud Computing

Chapters 2 and 3 focus on real-time virtualization using RT-Xen on a dedicated host. While RT-Xen can deliver the required real-time performance to VMs, it has two drawbacks in supporting real-time VMs in a cloud. First, RT-Xen employs compositional scheduling analysis [45, 75] to compute the resource interfaces of VCPUs needed to guarantee the real-time performance of applications running in the VMs. While compositional analysis provides the theoretical foundation for providing real-time guarantees on RT-Xen, the resource interfaces computed based on the compositional analysis are often conservative. As a result, provisioning CPU resources based on the resource interfaces may lead to significant CPU underutilization (around 60% in our previous experiments, see Figure 3.5 in Chapter 3). Second, there is no differentiation between real-time and non-real-time VMs. Both real-time and non-real-time VMs are scheduled using the same type of resource interface, and the non-real-time VMs must be incorporated into the underlying compositional scheduling analysis even though they do not require any latency guarantees. Therefore, if we directly apply RT-Xen 2.0 in a cloud, the host will be underutilized, and the non-real-time VMs will further reduce the resource utilization.

This chapter presents RT-OpenStack, which provides a holistic solution for co-hosting real-time VMs with non-real-time VMs in a cloud. We first introduce background information on OpenStack and its limitations for supporting real-time VMs in Section 4.1. We then describe the design and implementation of RT-OpenStack in Section 4.2 and present our experimental evaluation in Section 4.3. Finally, we summarize this chapter in Section 4.4.

4.1 OpenStack and its Limitations

OpenStack [19] was developed in 2010 by Rackspace and NASA, and has quickly became a popular cloud management software (used in production by RackSpace [20] and HP-Cloud [11]). It adopts a centralized architecture and consists of interrelated modules that control pools of CPU, memory, networking, and storage resources of a cluster of hosts. When integrated with Xen, a special agent domain is created on each host to support these resource management functions in co-ordination with domain 0 of the host.

We now review three aspects of OpenStack that are critical for managing the real-time performance of VMs. (1) the resource interface that specifies the resource management of a VM; (2) an admission control scheme for each host to avoid overload situation; and (3) a VM allocation scheme that maps VMs to hosts.

Resource Interface: The resource interface in OpenStack is represented by a pre-set type (called a "flavor"). The cloud manager can configure the number of VCPUs, memory size, and disk size. The user can also pass other information, such as the VM-to-VM affinity, to the cloud manager.

Admission Control: The admission control in OpenStack is referred to as "filtering". OpenStack provides a framework where users can plug in different filters. By default, there are more than ten filters provided, which focus on checking for enough memory, storage, and VM image compatibility. Two of the filters are related to the CPU resources: (1) core filter, which uses a VCPU-to-PCPU ratio to limit the maximum number of VCPUs per host. By default, this ratio is set to 16:1, which means if there are 4 PCPUs in a host, the filter can accept up to 64 VCPUs; (2) max VM filter, which limits the maximum number of VMs per host. By default, this value is set to 50. Clearly, these filters cannot provide real-time performance guarantees to real-time VMs allocated to a host, given the coarse-grained nature of the heuristics used.

VM Allocation: After the filtering process, OpenStack needs to select one host for the VM. This is referred to as "weighing". By default, OpenStack uses a worst-fit algorithm based on the amount of free memory on each host.

While OpenStack is widely used in the cloud, it cannot support real-time VMs demanding latency guarantees. First, the resource interface is inadequate. Users can configure only the number of VCPUs, but they cannot specify the resource and timing granularity needed to achieve real-time performance guarantees. Second, the VM allocation heuristics ignore real-time requirements and allocate VMs based on coarse-grained metrics that are insufficient for provisioning real-time performance guarantees. In the filter stage, OpenStack uses heuristics to decide whether a host is suitable for the VM. In the weighing stage, the current scheme is based on memory and ignores the CPU resources demand for meeting the latency requirements of applications within the VMs.

4.2 RT-OpenStack: Design and Implementation

A real-time cloud management system for co-hosting real-time and non-real-time VMs should have the following characteristics:

- It should provide a real-time resource interface for the VMs that includes the resources needed to ensure timing guarantees of the applications running within the VMs, such as the number of VCPUs required by each VM and their specifications (e.g., *budget* and *period* for each VCPU).
- It should deliver the resources according to the specification to the real-time VMs. To achieve this, a real-time VMM scheduler at the host level is required.
- It should perform an appropriate VM-to-Host mapping, which maintains the schedulability of real-time VMs without overloading the hosts.
- It should be able to co-host non-real-time VMs with real-time VMs.
- It should be work-conserving and maintain a high CPU utilization at each host.

We have designed RT-OpenStack, a cloud management system designed to support co-hosting real-time and non-real-time VMs in a cloud. On a single host level, we designed RT-Xen 2.1 to support co-hosting real-time and non-real-time VMs. It allows the non-real-time VMs to share the remaining CPU resources without interfering with the real-time

performance of RT VMs. On the cloud management level, we have designed an RT-Filter that works as admission control on each host for real-time VMs, and an RT-Weigher that allocates real-time VMs based on CPU resources. We now discuss them one by one.

4.2.1 Co-Scheduling real-time and non-real-time VMs on a Host

Recall in RT-Xen that the resource interface of a real-time VM is computed using compositional scheduling analysis theory [36, 45, 75], which ensures that if the host has sufficient resources to feasibly schedule the VCPUs specified by the interfaces, then all applications running within the VMs are schedulable. We add one field called "rt" for each VM, and re-order the run queue based on this value, as shown in Figure 3.1.

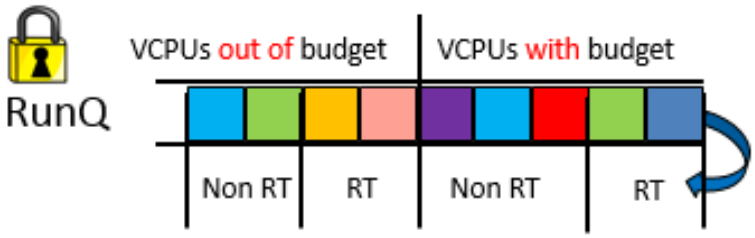


Figure 4.1: Run Queue Architecture in RT-Xen 2.1

The key difference between Figure 4.1 and Figure 3.2 in Chapter 3 is that real-time VMs' VCPUs always have higher priority then non-real-time VMs' VCPUs. Therefore, the non-real-time VMs do not affect the compositional schedulability analysis of the real-time VMs. At the same time, they can utilize the remaining CPU resources.

4.2.2 Co-Hosting real-time and non-real-time VMs in a Cloud

Recall that OpenStack lacks an adequate resource interface for VMs, and the existing VM-to-host mapping ignores real-time scheduling analysis. We now discuss these concerns one by one.

Resource Interface: Now that each host runs with the RT-Xen 2.1 scheduler, we still need a method to pass the real-time specification to the cloud manager when creating a VM. We

can create various VM template "flavors" with different pre-defined values, but that would be too rigid. In contrast, we use the existing flavors which, includes the number of VCPUs, and pass information to OpenStack via the scheduler hint: *rt*, whether this VM is a real-time VM or not; *budget*, the total *budget* for all VCPUs; and *period*, the shared *period* for all VCPUs. When there are multiple VCPUs in the flavor, we distribute the total *budget* evenly among them.

For a non-real-time VM, a system manager usually does not know the workload characteristic ahead. Since it will not affect the real-time VMs' performance, we set the *budget* to be the same as its *period* for non-real-time VCPUs, so that they can use the remaining CPU resources whenever available. We also configure the same *period* value for all non-real-time VMs' VCPUs, so they always have the same deadline in EDF scheduling. When multiple VCPUs share the same deadline, RT-Xen 2.1 uses a round robin scheduling scheme among them. As a result, the non-real-time VMs share the remaining CPU resources evenly. Note here that for a non-real-time VM, a user can also specify its *budget* to be less than its *period*, which limits the CPU resources to the non-real-time VM. Exploration of this option is left to future work. There are other approaches to integrating non-real-time tasks with real-time tasks [32, 37], but either they require the non-real-time tasks to follow the same task model as real-time tasks, or they treat non-real-time tasks as real-time ones, which further reduces the utilization bound.

RT-Filter: In addition to the existing filters in the OpenStack scheduler, we implemented an RT-Filter for RT VMs. It acts as an admission control for real-time VMs on each host. When a real-time VM creation request is submitted, the RT-Filter is triggered on each host. RT-Filter reads the already accepted real-time VMs' information on the host, and together with the new request, it performs the schedulability test to get the minimal number of PCPUs to schedule those VCPUs. If the required number of PCPUs is larger than the available PCPUs, it rejects the request; otherwise, it accepts the request. Note that the RT-Filter is applied only for real-time VM requests, and it considers only the real-time VMs' information when performing the compositional scheduling analysis. In this way, we can maintain the real-time VMs' performance by not overloading the host, while being able to accept non-real-time VMs to fully utilize the underlying CPU resources.

RT-Weigher: For the VM allocation (weighing) process, we face the problem of considering at least two resources: CPU and memory. We focus on the CPU resources for real-time VMs in this chapter, and use a worst-fit allocation scheme for real-time VMs, based on CPU resources. We have designed and implemented an RT-Weigher into the OpenStack scheduling framework. The RT-Weigher works very similar to RT-Filter, but instead of returning a value that indicates whether the VM is accepted or not, RT-Weigher returns the remaining CPU capacity on the host. It also considers only real-time VMs when performing the compositional schedulability analysis. For the non-real-time VMs, we fall back to the default worst-fit allocation scheme based on memory.

Table 4.1: RT-OpenStack for real-time and non-real-time VMs

	Resource Interface	Admission Control	VM Allocation
Real-time VM	Compositional Sched	Existing Filters + RT-Filter	RT-Weigher
Non-real-time VM	Full CPU	Existing Filters	Memory Weigher

Table 4.1 summarizes the differences in treating real-time and non-real-time VMs in RT-OpenStack. In summary, we consider only existing real-time VMs' information for RT VMs, and fall back to the default schemes for non-real-time VMs.

We implemented RT-Xen 2.1 in C. We also extended the RT-Xen tool for including the rt parameters. The RT-Filter and RT-Weigher are implemented in Python. Both RT-Xen 2.1 and RT-OpenStack are open source and can be downloaded at `https://sites.google.com/site/realtimexen`.

4.3 RT-OpenStack: Evaluation

We now present our experimental evaluation of RT-OpenStack for co-hosting real-time VMs with non-real-time VMs. We first evaluate RT-Xen 2.1 on a single host to demonstrate that non-real-time VMs cannot affect the performance of real-time VMs. We then conduct a study on a seven host cluster to demonstrate that RT-OpenStack can satisfy real-time VMs' resource requirement, while keeping hosts fully utilized.

4.3.1 Experimental Setup

Our testbed contains seven multi-core machines, named from host 0 to host 6. For the CPU resources, host 2 has 6 cores, while all other 5 hosts have 4 cores; for the memory resource, host 0 and 1 have 8 GB memory, host 2 has 12 GB memory, and hosts 3 to 6 have 16 GB memory each. Host 0 is configured as the controller, and it can also run guest VMs. XenServer 6.2 patched with RT-Xen 2.1 is installed on all machines. On each machine, domain 0 is configured with 1 VCPU, 1 GB memory, and is pinned to core 0; the agent VM is configured with 1 VCPU, 3 GB memory, and is also pinned to core 0. The XenServer takes another 200 MB extra memory on each machine. The remaining cores and memory are used to run the guest VMs. We used gEDF scheduler with deferrable server on each machine, as it was shown to work best in Chapter 3. We disabled the dynamic frequency scaling, turbo boost, and hyper-threading so that the PCPU worked at a constant speed. All other unnecessary services were turned off during the experiment.

4.3.2 Impact of non-real-time VMs on real-time VMs

RT VM's reservation on single core

We first demonstrate that the non-real-time VM cannot affect the CPU resources allocated to an RT VM. We focus on the single-core case, and set up the experiment as follows: We ran the experiment on a single host, boot one real-time VM and five competing non-real-time VM (named VM1 to VM5). They have one VCPU each, and were all pinned to a single core (through cpumask). The RT VM was configured with a *budget* of 4 and *period* of 10, and the non-real-time VM's *budget* was set to be equal to its *period*. All VMs ran a CPU busy program to take as much CPU resources as possible. We started with only one real-time VM running, then gradually enable the CPU busy program in non-real-time VMs, and record the CPU resources received them.

Table 4.2 shows the results. We observe that the RT VM's performance is not affected by non-real-time VMs, even under stress testing. We also notice that non-real-time VMs share the remaining CPU resources, as expected. Another observation is that all CPU utilizations add to at least 99.5%, which demonstrates our efficient implementation of RT-Xen 2.1.

Table 4.2: CPU Utilization Test on a Single Core

RT VM	40.3%	40.2%	40.2%	40.2%	40.2%	40.2%
VM 1	-	59.5%	29.8%	19.9%	14.9%	11.9%
VM 2	-	-	29.8%	19.8%	14.8%	11.9%
VM 3	-	-	-	19.9%	14.9&	12.0%
VM 4	-	-	-	-	14.8%	12.0%
VM 5	-	-	-	-	-	12.0%
Total	40.3%	99.7%	99.8%	99.8%	99.6%	100%

Note here that for the default credit scheduler, a system administrator can adjust each VCPU's *weight* to make the real-time VM receive a certain amount of resources. However, this requires a global knowledge of all the running domains, and also needs re-adjustment whenever there is a change in the number of VMs. In contrast, when a system administrator allocates a certain amount of CPU resources to a real-time VM in RT-Xen 2.1, it will not change, regardless of the number of non-real-time VMs.

Schedulability test for RT VM

This experiment demonstrates that RT-Xen 2.1 can provide CPU resources to the real-time VMs at the right time to meet the real-time application's deadlines. We set up this experiment as follows: Each RT VM contained two real-time tasks, with period randomly selected from 20 ms to 33 ms, and execution time randomly selected from 10 ms to 20 ms. For the underlying VCPU parameters, we iterated all periods that were less than 30 ms, then used the compositional scheduling analysis [45, 75] to generate the required *budget* for each VCPU. After getting all the combinations, we used the one with the minimal total VCPU bandwidth.

We ran the experiments with three PCPUs as the constraint, and generated two real-time VMs: the actual total task utilization was 2.03, while the total VCPU bandwidth was 2.93, and they required three full PCPU to schedule the two real-time VMs. We then booted up two non-real-time VMs, and configured the cpu test program in sysbench [22] to run in them. The program kept calculating prime numbers until a predefined threshold, then started from 2 again. It reported the number of rounds achieved during a given time. The real-time task

and the sysbench are all configured to run for 1 minutes, and we record the results. We also repeated the experiment with the credit scheduler.

Table 4.3: Schedulability Test on Multi-Core

	Deadline Miss Ratio		Number of Rounds Calculating Primes	
	RT VM 1	RT VM 2	Non-real-time VM 1	Non-real-time VM 2
RT-Xen 2.1	0%	0%	-	-
	0%	0%	1929	-
	0%	0%	1280	1266
Credit	0.01%	0.5%	-	-
	3.4%	15.3%	2596	-
	73.7%	40.7%	1941	1736

Table 4.3 shows the results. We observe that under all cases, RT-Xen 2.1 can meet the deadline requirements for real-time VMs, and evenly distribute the remaining resources for non-real-time VMs. In sharp contrast, using credit scheduler experienced light deadline misses (0.01% and 0.5%) for both real-time VMs even when there was no interference, and the deadline miss ratio grew up to 73.5% for RT VM 1 when there are were non-real-time VMs running. We also observe that when there are multiple non-real-time VMs configured with same *weight*, the CPU resources they get are not equal. We attribute this to the nature of partitioned scheduling in the credit scheduler, and to its heuristic load-balancing scheme.

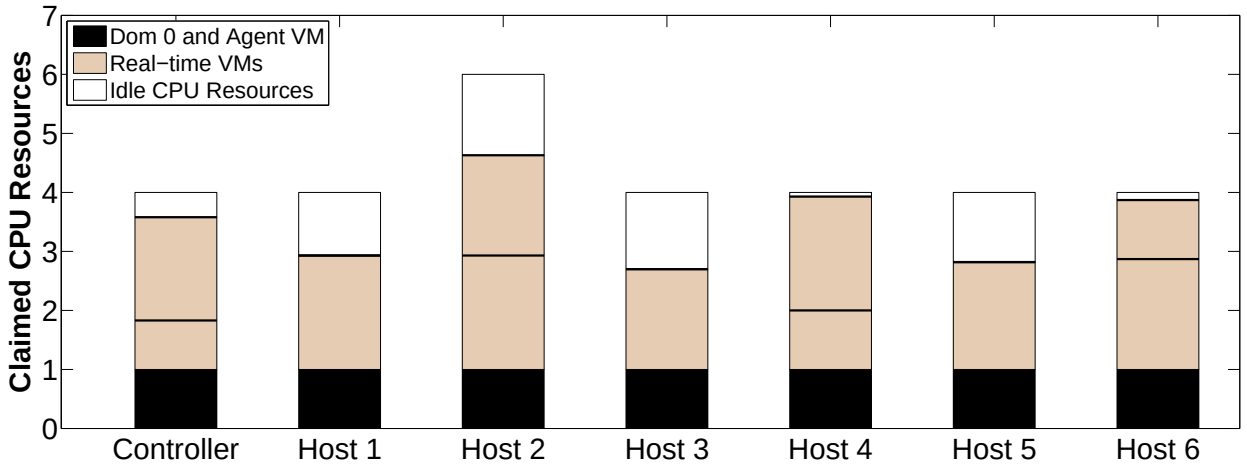
Summary: On a single host, RT-Xen 2.1 can maintain real-time VMs' performance while keeping the host utilization high by running non-real-time VMs.

4.3.3 RT-OpenStack on a Cluster

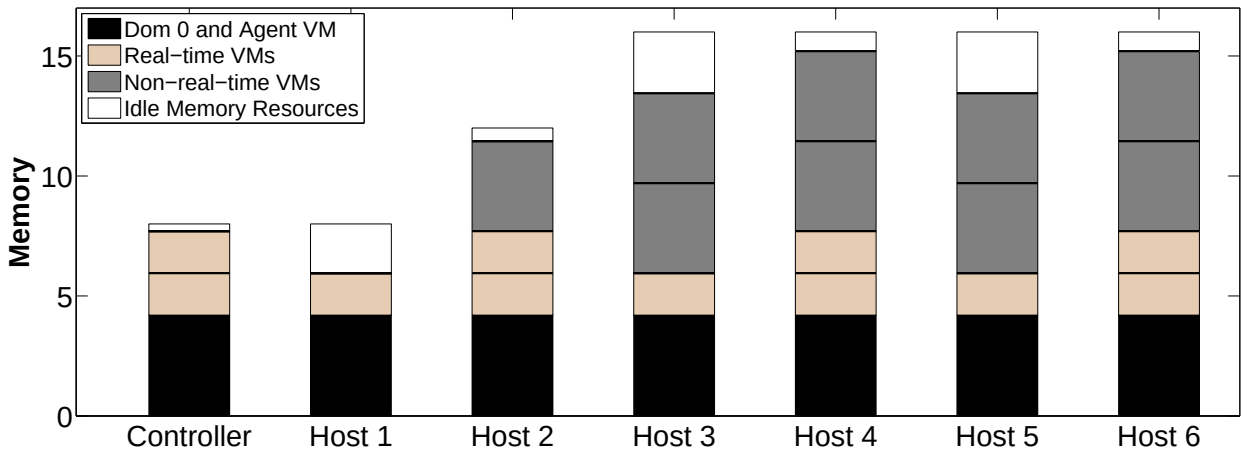
This experiment was set up to evaluate RT-OpenStack on a cluster. All seven hosts were used as a cloud. Both real-time and non-real-time VMs were running in the cloud. In each real-time VM, we emulated a cloud gaming server described in [52], where there are two real-time tasks: a video encoder and a audio encoder. We randomly chose each task's period in the range between 20 ms to 33 ms, to emulate different frame rates between 30 fps and 50 fps. Each task's execution time was randomly ranged from 10 ms to 20 ms, to represent different games, resolutions, and settings. We applied the $Litmus^{RT}$ [16] patch for the RT VM and used the gEDF scheduler to schedule real-time tasks. Compositional scheduling

theory [45,75] was used to generate the VCPU parameters for the RT VM. All the non-real-time VM were configured to be a hadoop cluster, and we ran the standard pi program to test its performance as a whole. The hadoop program requires all non-real-time VMs in the cluster to finish; as a result, if any of them does not get enough CPU resources, the total finish time will be affected.

The VM booting sequence was as follows: We first kept creating real-time VMs until rejected. Each RT VM was configured with 1.75 GB memory. After that, we kept booting non-real-time VMs with 2 VCPUs and 3.75 GB memory each until rejected. Eventually, eleven real-time VMs and nine non-real-time VMs were accepted.



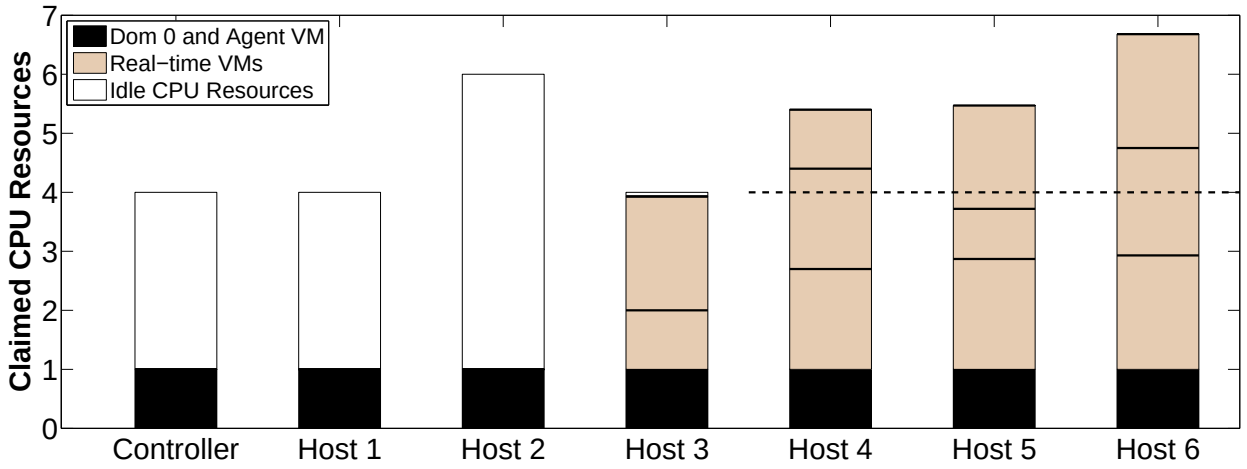
(a) CPU Resources



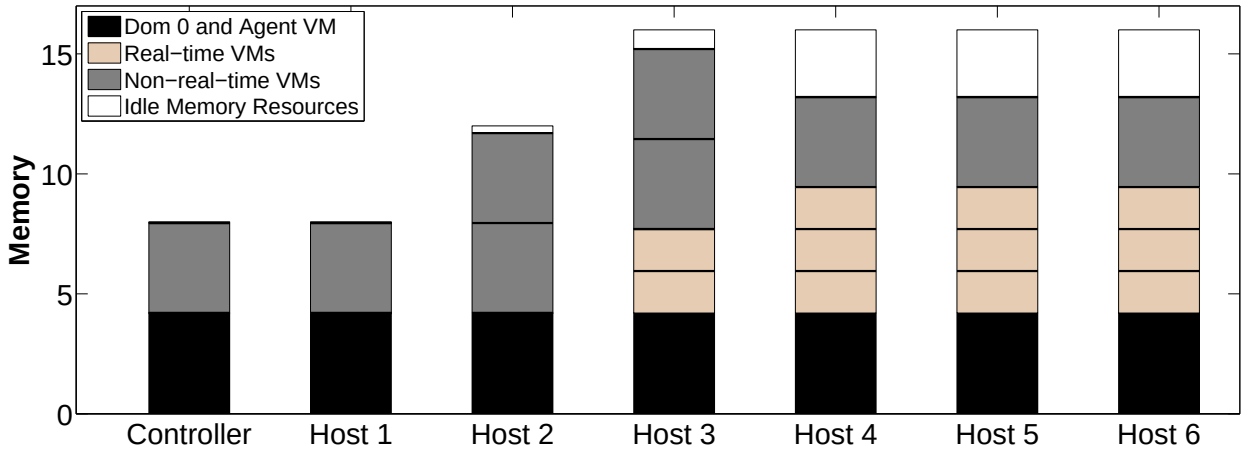
(b) Memory Resources

Figure 4.2: RT-OpenStack VM Allocation

Figure 4.2 shows the VM allocation scheme for RT-OpenStack. We can see that the RT VMs are evenly distributed among seven hosts, and the non-real-time VMs are booted on hosts with enough memory to take advantage of the remaining CPU resources. Because we configured each non-real-time VM's VCPU's *budget* to be the same as its *period* so they can fully use the CPU resources, we do not show their CPU allocation in the Figure 4.2a.



(a) CPU Resources



(b) Memory Resources

Figure 4.3: OpenStack VM Allocation

We repeated the same booting sequence using the default OpenStack, and Figure 4.3 shows the results. As expected, a worst-fit algorithm based on memory is used, and the first eleven real-time VMs are located hosts 3-6. We draw a dashed line in Figure 4.3 for the limit on

CPU resources on hosts 4 to 6. As we can see, the CPU resources are overloaded on these three hosts.

After all the VMs were ready, we ran the hadoop workload in the non-real-time VMs, and at the same time started the real-time tasks in RT VMs. When the hadoop workload finished, we manually terminated the real-time task in each RT VM and recorded its deadline miss ratio. For both RT-OpenStack and OpenStack allocation schemes, we ran the experiments with RT-Xen scheduler and credit scheduler on each host.

Table 4.4: Cluster Performance Comparison

		RT-OpenStack+RT-Xen	RT-OpenStack+Credit	OpenStack+RT-Xen	OpenStack+Credit
Deadline Miss Ratio	RT 1	0%	3%	9%	37%
	RT 2	0%	1%	0%	31%
	RT 3	0%	54%	0%	61%
	RT 4	0%	35%	0%	13%
	RT 5	0%	21%	2%	75%
	RT 6	0%	14%	0%	29%
	RT 7	0%	0%	0%	30%
	RT 8	0%	0%	0%	36%
	RT 9	0%	51%	41%	73%
	RT 10	0%	35%	11%	47%
	RT 11	0%	0%	0%	32%
Hadoop finish time		435 s	254 s	-	314 s

Table 4.4 shows the results. The RT-OpenStack + RT-Xen configuration experienced no deadline miss in all 11 RT VMs, and finished the hadoop task in 435 seconds. In contrast, using the same RT-OpenStack allocation scheme but with the credit VMM scheduler, eight out of eleven RT VMs experienced deadline misses, and two of them have deadline miss ratio larger than 50% (RT VM 3 and 9). However, the hadoop tasks finished in 254 seconds, which is 3 minutes faster than the RT-Xen scheduler. This was expected because in credit scheduler, the non-real-time VMs get more resources. When using the OpenStack allocation schemes with the RT-Xen scheduler, the hadoop made no progress at all. So we terminated the experiments at five minutes and report the deadline miss ratio in all real-time VMs. Four out of eleven RT VMs experienced deadline misses, we further examined the allocation and found three of them were allocated on the same host (host 6). This finding shows that RT-Xen can prioritize the CPU resources to RT VMs, however, due to the allocation scheme, on host 6 there are not enough CPU resources. The OpenStack + Credit combination

experienced the worst deadline miss ratio for all real-time VMs. This again demonstrated that the default allocation scheme ignores the CPU resource requirement, and the underlying credit scheduler cannot deliver real-time performance. We also observe that its hadoop task finished 1 minute later than the RT-OpenStack + credit combination, which we attribute to the CPU overloading on hosts 3-6.

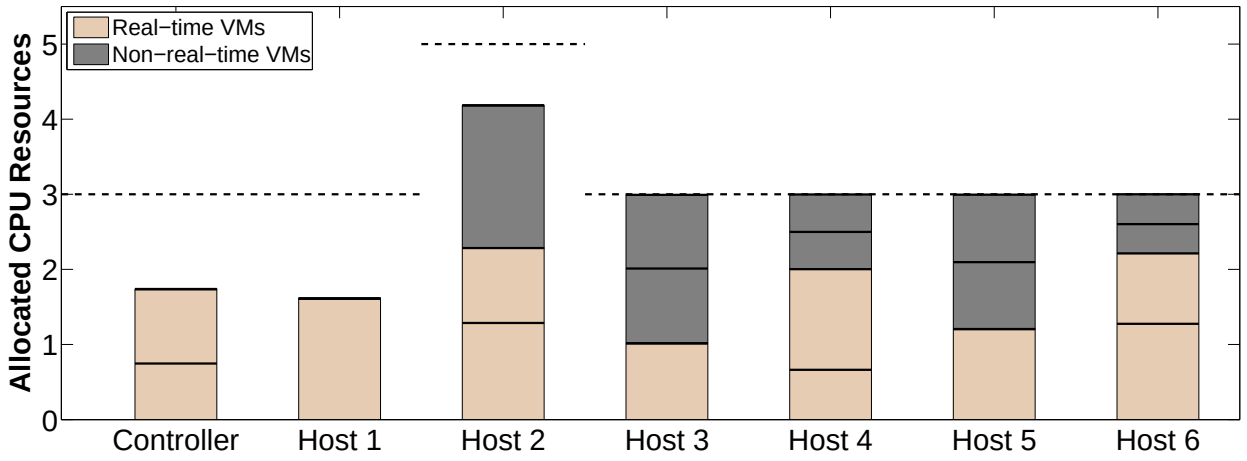


Figure 4.4: Actual CPU Resource Usage for RT-OpenStack

Since the hadoop program took the longest to finish in the RT-OpenStack + RT-Xen setup, we were also interested in whether the hosts are fully utilized or not. We repeated the experiment and recorded each domain's actual CPU consumption for 10 seconds. Figure 4.4 shows the results, we exclude core 0 which runs domain 0 and the agent VM. We also draw a dashed line to represent the CPU resource limit on each host. Comparing the actual allocation with the claimed CPU resources by real-time VM in Figure 4.2a, we have the following insights: (1) although the claimed CPU resources almost reached the limit, the actual CPU consumption by RT VM was much less than claimed, which further proves the pessimism of the hierarchical scheduling theory, and motivates us to co-host real-time VM with non-real-time VMs; (2) on hosts 3 to 6, the actual total CPU utilization has already reached the limit, which means any improvement on the hadoop program will affect the real-time performance of RT VMs. On host 2, the actual CPU allocation for non-real-time VMs reached 200%, which is the upper limit for 2 VCPUs.

Summary: RT-OpenStack can provide real-time performance to real-time VMs, while allowing non-real-time VMs to share the remaining CPU resources without interfering with the performance of real-time VMs.

4.4 Summary

This chapter presents RT-OpenStack, a cloud management system that can co-host real-time VMs with non-real-time VMs. RT-OpenStack makes three main contributions: (1) the integration of a real-time hypervisor (RT-Xen) and a cloud management system (OpenStack) through real-time resource interface; (2) RT-Xen 2.1 scheduler to allow non-real-time VMs to share hosts with real-time VMs without jeopardizing the real-time performance of RT VMs; and (3) a VM-to-host mapping strategy that provides real-time performance to RT VMs while allowing effective resource sharing among non-real-time VMs. Our experimental results demonstrate that RT-OpenStack can support latency guarantees for real-time VMs, and at the same time let the non-real-time VMs fully utilize the remaining CPU resources.

Chapter 5

RTCA: Real-Time Communication Architecture

RT-Xen and RT-OpenStack focus on CPU resources to provide real-time guarantees, which is adequate for computation-intensive applications. However, multiple network communication can also introduce a significant amount of delay. A modern virtualized systems may seat as many as 40 to 60 VMs per physical host [60], and with the increasing popularity of 32-core and 64-core machines [38], the number of VMs per host is likely to keep growing. As a result, a significant amount of network communication may become local inter-domain communication (IDC) within the same host. When multiple domains co-exist on a same host, it is important to properly schedule the processing of local communication to achieve latency differentiation among VMs with different priorities and quality of service (QoS) requirements.

This chapter presents RTCA, a novel real-time communication architecture for Xen to preserver the low inter-domain communication latency between high-priority domains in face of low priority domain's traffic. We review the background of Xen communication architecture in Section 5.1, then closely examine the latency of IDC flows in Xen and point out its key limitations that can cause significant priority inversion in IDC in Section 5.2. We show experimentally that improving the VMM scheduler along cannot achieve latency differentiation for IDC due to significant priority inversion in the domain 0. To address this problem, we have designed and implemented RTCA, which is described in detail in Section5.4 and Section 5.5. We summarize in Section 5.6.

5.1 Background

This section provides background information about the key communication architecture components in Xen.

5.1.1 Xen Communication Architecture

Figure 5.1 gives an overview of the communication architecture in Xen. For IDC, domain 0 contains a *netback* driver that coordinates with a *netfront* driver in each guest domain. For example, the upper connecting lines in Figure 5.1 show the inter-domain communication for application A from domain 1 to domain 2. Application A first sends packets to the *netfront* driver in domain 1; the *netfront* driver delivers the packets to domain 0; domain 0 examines each packet, determines it is for a local domain, delivers it to domain 2 and notifies the VMM scheduler; when domain 2 gets scheduled, its *netfront* driver send each packet to application A. Note that the applications running atop the guest domains are not aware of this para-virtualization, so no modification to them is needed. Another approach for IDC is to use shared memory to exchange data between domains [53, 54, 70, 77], thus avoiding the involvement of domain 0 to obtain better performance. However, the shared memory approach requires modifications to the guest domain besides the well supported Xen patch, and may even require modifying the applications as well. domain 0 also contains a NIC driver, and if a packet is for another host, it directs the packet to the NIC driver, which in turn sends it out via the network. Improving the real-time performance of inter-host communication is outside the scope of this chapter and will be considered as future work.

As Figure 5.1 illustrates, in IDC two mechanisms play important roles: (1) the VMM scheduler, which needs to schedule the corresponding domain when it has pending/coming packets; and (2) the *netback* driver in domain 0, which needs to process packets according to their QoS requirements.

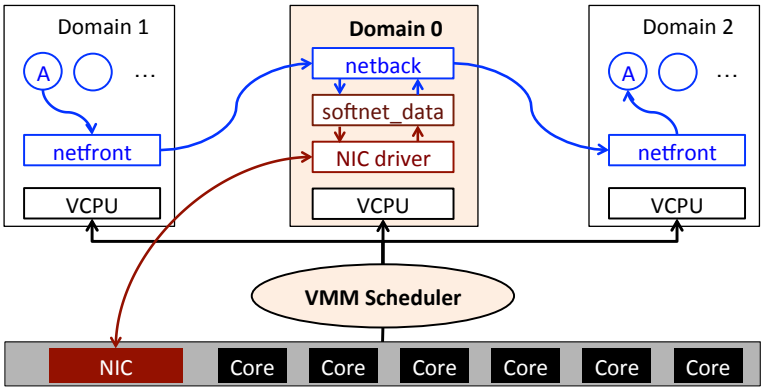


Figure 5.1: Xen Communication Architecture Overview

5.1.2 IDC in Domain 0

To explain how IDC is performed in domain 0, we now describe how Linux processes packets, how the softirqs and kernel threads behave, and show how Xen hooks its *netfront* and *netback* drivers into that execution architecture to process packets.

When a guest domain sends a packet, an interrupt is raised to notify domain 0. To reduce context switching which can produce receive livelock [61], Linux 2.6 and later versions have used the New API packet reception mechanism [63]. The idea is that only the first packet raises a NET_RX_SOFTIRQ, and after that the interrupt is disabled and all the following packets are queued without generating interrupts. The softirqs are scheduled by a per-CPU kernel thread named ksoftirq. Also, a per-CPU data structure called softnet_data is created to hold the incoming packets.

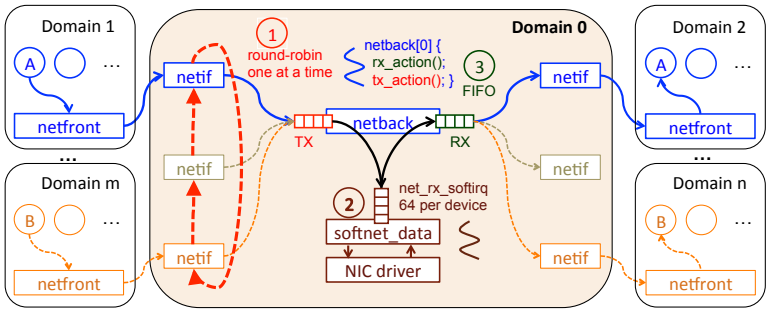


Figure 5.2: Xen Communication Architecture in Domain 0

As shown in Figure 5.1, Xen uses the *netfront* and *netback* drivers to transmit packets between guest and manager domains. Figure 5.2 illustrates in detail how domain 0 works, with the source domains on the left sending packets to the destination domains on the right. When domain 0 boots up, it creates as many *netback* devices as it has VCPUs (here we consider only the single core case, with a single *netback* device in domain 0). The *netback* device maintains two queues: a TX Queue for receiving packets from all guest domains, and an RX Queue for transmitting packets to all guest domains. They are processed by a single kernel thread in Linux 3.4. The kernel thread always performs the *net_rx_action* function first to process the RX Queue, and then performs the *net_tx_action* function to process the TX Queue. When a guest domain boots up, it creates a *netif* device in domain 0 and links it to the *netback* device. [6]

Within the domain 0 kernel, all the *netback* devices are represented by one backlog device and are treated the same as any other device (e.g., a NIC). As can be seen from Figure 5.2, when an IDC flow goes through domain 0, there are three queues involved, which we now consider in order by where the packets are processed.

Netback TX Queue: The *netback* device maintains a *schedule_list* of all the *netif* devices that have pending packets. When the *net_tx_action* function is processed, it picks the first *netif* device in the list, processes one packet, and if it still has pending packets, puts the *netif* device at the end of the list, which results in a round robin transmission order with a quantum of 1. In one round, it processes up to a certain number of packets, which is related to the page size on the machine: on our 64-bit Linux machine, that number is 238. If there are still packets pending after a round, it notifies the scheduler to schedule the kernel thread again later. Xen by default adopts a server-based algorithm [29] to achieve rate limiting for each domain within this stage; if a *netif* device has pending packets but exceeds the rate limit, Xen instead picks the next one. In this chapter, we leave the rate control default (unlimited) as it is and instead change the order of pending packets. RTCA can be seamlessly integrated with default or improved rate control mechanisms [39].

Softnet_Data Queue: All the packets dequeued from the TX Queue are enqueued into a single softnet_data queue. domain 0 processes this queue when responding to the NET_RX_SOFTIRQ.

[6] Linux (version 2.4 and after) provides a traffic control tool [15]. A priority qdisc can be used to prioritize packets. However, the priority qdisc works only within one device, while in the IDC, traffic belonging to different domains has already been delivered to different *netif* devices.

A list of all active devices (usually NIC and backlog) is maintained, and domain 0 processes up to 64 packets for the first device, puts it at the end of the list, and then processes the next one, also resulting in a round robin order with a quantum of up to 64. In one round, the function quits after either a total of 300 packets is processed or 2 jiffies have passed. If there are still pending packets at the end of a round, another NET_RX_SOFTIRQ is raised. When processing a packet, if domain 0 finds that its destination is a local domain, it bridges the packet to the RX Queue in the corresponding *netback* device; if it is processing the first packet, it also notifies the scheduler to schedule the kernel thread. Note that by default 1000 packet limit applies for the backlog device [62]. We consider only IDC in this chapter and defer integration with the NIC as future work.

Netback RX Queue: Similar to the TX Queue, the *netback* driver also has an RX Queue (associated with a function *net_rx_action*) that contains packets whose destination domain's *netif* is associated with that *netback* device. All the packets in this case are processed in FIFO order and are delivered to the corresponding *netif* device. Note that processing of this queue also has a limit (238) for one round, and after that, if there are still packets pending, it tells the scheduler to schedule them later.

5.2 Limitations of the Communication Architecture

As Figure 5.1 shows, both the VMM scheduler and domain 0 can impact IDC performance. This section describes qualitatively the limitations of both the VMM scheduler and domain 0 for prioritized IDC. The next section will present an empirical study to quantify the impacts of their limitations on the performance of prioritized IDC flows.

5.2.1 Limitations of the VMM Schedulers

The default Credit scheduler has two major problems when handling prioritized traffic: (1) it schedules VCPUs with outgoing packets in a round robin fashion, and (2) for incoming packets, it applies a general boost to a blocked VCPU regardless of its priority. Note that

boosting the priority of a low priority VCPU to receive a packet can introduce priority inversion when a high priority VCPU is running.

The RT-Xen scheduler [56, 73] applies a strict priority policy to schedule VCPUs for both outgoing and incoming packets, and thus can prevent interference from lower priority domains within the same core. However, it uses 1 ms as the scheduling quantum, and when a domain executes for less than 0.5 ms, its *budget* is not consumed. On a typical machine, however, the time for a domain to send a packet to another local domain is less than 10 μs. Consider a case where one packet is bouncing between two domains on the same core: if these two domains run no other tasks, the RT-Xen scheduler would switch rapidly between these two domains, with each executing for only about 10 μs. As a result, neither domain's *budget* will be consumed, resulting in a 50% share for each, regardless of their *budget* and *period* configuration. This clearly violates the resource isolation property of the VMM scheduler. We address this limitation by providing dual time resolutions: μs for CPU time accounting, and ms for VCPU scheduling. The dual resolution provides better resource isolation, while maintaining 1ms as an appropriate scheduling quantum for real-time applications. For all the evaluations in this chapter we use this improved version of the RT-Xen scheduler.

5.2.2 Limitations of Domain 0

Domain 0 also has major limitations in terms of real-time IDC performance. As was discussed in Section 5.1.2, the TX, *softnet_data*, and RX queues are shared by all guest domains, resulting in a round robin scheduling policy with a quantum of 1, regardless of the domain's priority. We show that even under a light interference workload from other cores (which cannot be prevented by any VMM scheduler), the IDC latency for high priority domains is severely affected. Another limitation is that the TX, *softnet_data*, and RX queues are processed in a fixed order, regardless of the priority of the current processing packets. Before Linux 3.0, TX and RX processing was executed by two TASKLETs in an arbitrary order. As a result, the "TX - *softnet_data* - RX" stage could be interrupted by the RX processing for previous packets and by the TX processing for future packets. Linux 3.0 (and later versions) switched to using one kernel thread to process both TX and RX queues, with the RX Queue always being processed first. This change introduces another problem: the higher priority packets may need to wait until a previous lower priority one has finished

transmission. Finally, the priority inversion is exacerbated by large and mismatched queue sizes. The TX and RX queues have total processing sizes of 238, with a quantum of 1 for each domain, while the *softnet_data* queue has a total processing size of 300, with a quantum of 64 for each device. These large and mismatched sizes make timing analysis difficult and may degrade performance. For example, under a heavy IDC workload where a NIC also is doing heavy communication, the *softnet_data* queue (total size of 300) is equally shared by backlog and NIC devices. Every time the TX Queue delivers 238 packets to the *softnet_data* queue, the *softnet_data* queue is able to process only 150 of them, causing the backlog queue to become full and to start dropping packets when its limit of 1000 packets is reached.

5.3 Quantifying the Effects of the VMM Scheduler and Domain 0

We ran a series of experiments to evaluate the impacts of the VMM scheduler and domain 0 on IDC performance. The experiments were performed on an Intel i7-980 six core machine with hyper-threading disabled. SpeedStep was disabled by default, and each core ran at 3.33 GHz constantly. We installed 64-bit CentOS with para-virtualized kernel 3.4.2 in both domain 0 and the guest domains, together with Xen 4.1.2 after applying the RT-Xen patch. We focused on the single-core case with every domain configured with one VCPU, and we dedicated core 0 to domain 0 with 1 GB memory. Dedicating a separate core to handle communication and interrupts is a common practice in multi-core real-time systems research [38]. It is also recommended by the Xen community to improve I/O performance [72]. During all experiments we disabled the NIC and configured all the guest domains within a local IP address, focusing on IDC only. We also shut down all other unnecessary services to minimize incidental sources of interference. Domain 0 does not itself run other tasks that might interfere with its packet processing.

5.3.1 Effect of the VMM Scheduler: Credit vs. RT-Xen

The experiment presented in this section examines the effect of the VMM scheduler when all interference is coming from the same core. We booted ten domains and pinned all of them to core 1 (domain 0 still owns core 0). Each guest domain had 10% CPU share, which was achieved via the -c parameter in the Credit scheduler, and by configuring a *budget* of 1 ms and a *period* of 10 ms in the RT-Xen scheduler. We configured domain 1 and domain 2 with highest priority and measured the round-trip time between them: domain 1 sent out 1 packet every 10 ms, and domain 2 echoed it back. The *rdtsc* command was used to measure time. For each experiment, we recorded 5,000 data points. We configured the remaining eight domains to work in four pairs and constantly bounced a packet between each pair. Note that all 10 domains were doing IDC in a blocked state, and thus they would all be boosted by the Credit scheduler. As expected, when domain 1 or domain 2 was inserted at the end of the BOOST category, the queue already had a long backlog, with eight interfering domains, thus creating a priority inversion. In contrast, the RT-Xen scheduler would always schedule domains based on their priorities.

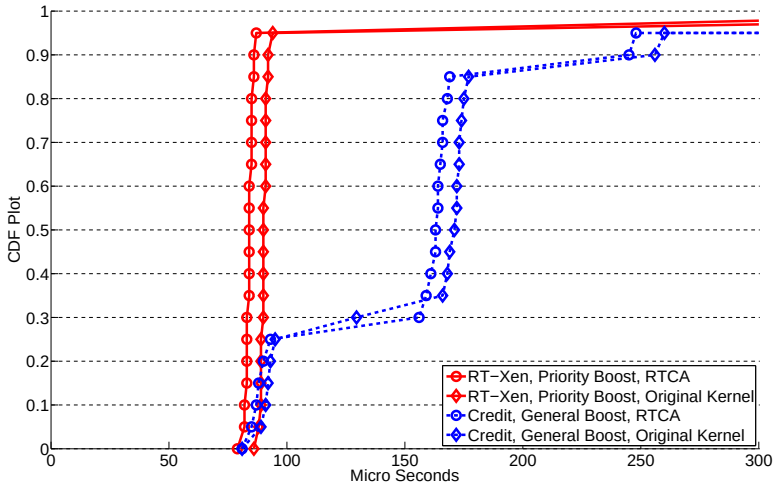


Figure 5.3: Effect of the VMM Scheduler: Credit VS. RT-Xen

Figure 5.3 shows a CDF plot of the IDC latency between the pair of high priority domains, with a percentile point every 5%. The solid lines show the results using the RT-Xen scheduler, and the dashed lines represent the Credit scheduler. The lines with diamond markers were obtained using the original kernel, and the lines with circles were obtained using a modified domain 0 with our new RTCA, which will be discussed in Sections 5.4 and 5.5.1. We can

clearly see that due to the general boost, the IDC latency between the high priority domains under the Credit scheduler is severely affected by the interfering IDC between low priority domains, growing from around 80 μs to around 160 μs at 30%, and further extending to 250 μs at 90%. In contrast, the RT-Xen scheduler can limit the latency to within 100 μs until the 95th percentile. We also noticed that when we were doing experiments, domain 0's CPU utilization stayed around 60%, indicating it was more than capable of processing the IDC load it was given.

5.3.2 The VMM Scheduler is Not Enough

We have shown that scheduling VCPUs based on priorities can deliver better IDC performance for high priority domains. The experiment presented in this subsection shows that domain 0 can also become a bottleneck when processing IDC, especially when significant contention from low priority domains exists.

In this experiment, we again pinned domain 0 to core 0, and dedicated core 1 and core 2 to domain 1 and domain 2, respectively, so the VMM scheduler would not matter. The same workload still ran between domain 1 and domain 2, and we measured the round trip times. For the remaining three cores, we booted three domains on each core, with all of them doing intensive IDC, creating a heavy load on domain 0.

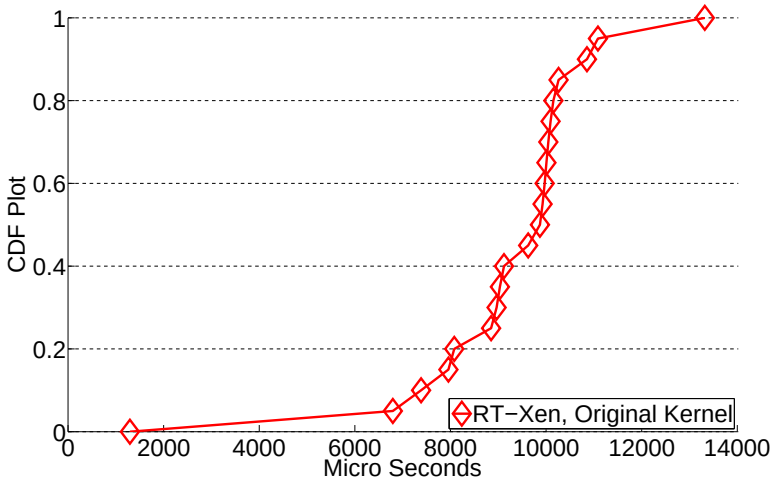


Figure 5.4: Bottleneck in Domain 0

Figure 5.4 shows a CDF plot of the results, with a sampling point every 5th percentile. Please note the larger x axis range in this figure, compared to Figure 5.3. The IDC latency between the high priority domains grows from the μs level to more than 6 ms. Since all the interference occurs within domain 0, any improvement to the VMM scheduler thus cannot help. Therefore, it is important to introduce prioritized IDC packet processing in domain 0.

5.4 RTCA: Design and Implementation

To address the limitations of domain 0, this section presents a new RTCA for domain 0. The goal of the RTCA is to support packet processing based on domain priorities, while reducing priority inversion. The priorities are based on domains instead of flows because the domain is the unit provided to the end customer.

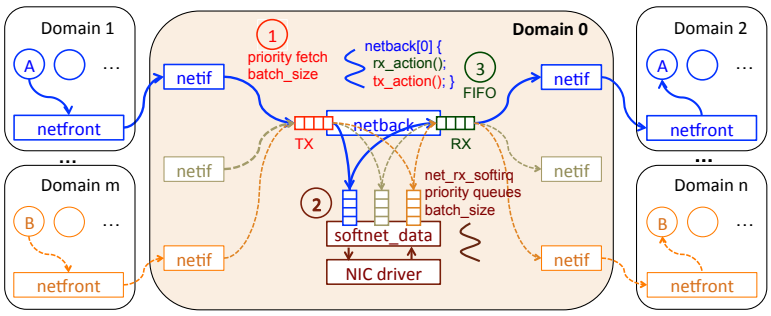


Figure 5.5: RTCA: Real-Time Communication Architecture

Figure 5.5 shows the RTCA in domain 0. We now discuss the changes made to each of the three queues.

Netback TX Queue: Algorithm 3 describes how we process the packets in the *net_tx_action* function. Instead of a round robin policy, we now fetch packets according to their priorities, one at a time. Within one round, we fetch up to a *batch size* number of packets, and we also make the batch size tunable to make IDC QoS more configurable for different system integrators. A counter is initialized to 0 and used to keep track of how many packets are processed within one round. Packets are processed one at a time because during the processing of lower priority domains, a higher priority domain may become active and dynamically add

its *netif* to the schedule list. Making a prioritized decision at each packet thus minimizes priority inversion. Note that due to other information kept separately in the *netback* driver about the packet order, neither splitting the queue nor simply reordering it is easily achievable without causing a kernel panic[7]. As a result, the TX Queue is dequeued in FIFO order. However, whenever a higher priority domain arrives in one round, we reset the counter so that the performance for higher priority domains will not be affected. Section 5.5 shows that with a batch size of 1, the system achieves suitable IDC latency and throughput for high priority domains, as a result of the longer blocking time for each packet when the batch size increases. If a batch size of 1 is used, the total size limit of 238 is unlikely to be reached, and so the total number of packets for a high priority domain is unlikely to be limited by the previously processed lower priority domains.

Algorithm 3 *net_tx_action* function

```
1: cur priority = highest active netif priority
2: total = 0
3: counter = 0
4: while schedule_list not empty &&
   counter < batch size && total < round limit do
5:    fetch the highest priority active netif device
6:    if its priority is higher than cur priority then
7:       reset counter to 0
8:       reset current priority to its priority
9:    end if
10:   enqueue one packet
11:   counter++, total++
12:   update information including packet order, total size
13:   if the netif device still has pending packets then
14:      put the netif device back into the schedule list
15:   end if
16: end while
17: dequeue from TX Queue to softnet_data queue
    raise NET_RX_SOFTIRQ for first packet
18: if schedule_list not empty then
19:   notify the scheduler
20: end if
```

[7]As future work, we plan to examine how to address this remaining limitation.

Softnet_Data Queue: Since the packets coming from the TX Queue can be from different domains, we split the queue by priorities, and process only the highest priority one within each NET_RX_SOFTIRQ. The batch size is also a tunable parameter for each queue. Moreover, under a heavy overload, the lower priority queues can easily be filled up, making the total size limit for all the *softnet_data* queues easily reached. Therefore, we eliminate the total limit of 1000 packets for all domains, and instead set an individual limit of 600 for each *softnet_data* queue. Note that this parameter is also tunable by system integrators at their discretion.

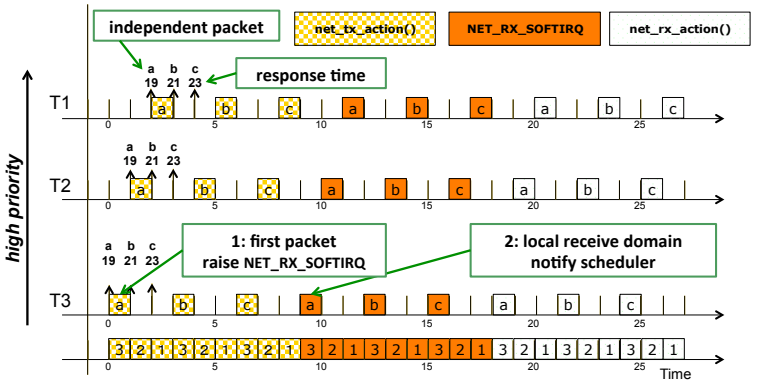
Netback RX Queue: As the packets coming from the *softnet_data* queue are only from one priority level, there is no need to split this queue. Moreover, by appropriately configuring the batch size for the *softnet_data* queue (making it less than 238), the capacity of the RX Queue will always be enough. For these reasons, we made no modification to the *net_rx_action* function. Please note that both the *softnet_data* and RX Queues are non-preemptable: even for the lower priority domains, once the kernel begins processing them, an arriving higher priority domain packet can only notify the kernel thread and has to wait until the next round to be processed.

Notably, without changing the fundamental architecture of domain 0, we keep the benefits of compatibility with the original Xen features (for example, the existing rate control mechanism can be seamlessly integrated with RTCA), while improving the IDC latency between high priority domains (as shown in Section 5.5) by an order of magnitude, resulting in μs level timing that is suitable for many soft real-time systems. Therefore, while the RTCA does not completely eliminate priority inversion, it can be highly effective in improving IDC prioritization for soft real-time applications.

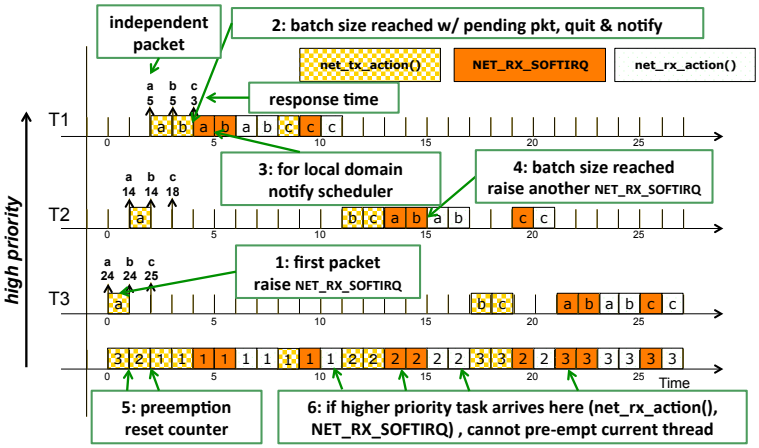
Examples for Packet Processing

To better illustrate how RTCA works, we show the packet processing order both in RTCA (Figure 5.6b) and in the original kernel (Figure 5.6a), assuming that the guest domains always get the physical CPU when they need it (e.g., via a perfect VMM scheduler). Both examples use the same task set, where three domains (T3, T2, and T1, with increasing priority) are trying to send three individual packets successively, starting from time 1, 2,

and 3. The lowest line of each figure shows the processing order for each domain, and the corresponding upper lines show the processing order for individual packets in each domain. To better illustrate preemption in the TX Queue, all three domains are configured with a batch size of 2 in the TX and *softnet_data* queues. The upper arrow shows the release of the packet, and the number above the arrow shows the response time for each packet.



(a) Original Kernel



(b) RTCA Kernel

Figure 5.6: Packet Processing Illustration

Several key observations can be made here:

- RTCA effectively reduces the IDC latency of the packets between the higher priority domains (from 19, 21, and 23 to 5, 5, and 3, respectively). Since (unmodified) Xen

processes packets in a round robin order and uses a relatively large batch size for all three queues, the response time is identical for each domain; in contrast, RTCA prioritizes the processing order and imposes a smaller batch size, resulting in faster IDC for higher priority domains.

- Whenever the batch size is reached and there are still pending packets, or when the first packet arrives, either a *softirq* is raised or the scheduler is notified (points 1, 2, 3, and 4 in Figure 5.6b; points 1 and 2 in Figure 5.6a).
- In RTCA, TX Queue processing is pre-emptive, and every time a high priority domain packet arrives, the counter is reset (point 5 in Figure 5.6b).
- The *softnet_data* and RX Queue processing is non-pre-emptive: if higher priority tasks are released during their processing, the scheduler is only notified but no preemption occurs (point 6 in Figure 5.6b).

5.5 RTCA: Evaluation

This section focuses on comparing the original domain 0 kernel and RTCA. As we discussed in Section 5.4, RTCA can be configured with different batch sizes, which we address here. We first repeat the experiments in Section 5.3.1 to see the combined effect of the VMM scheduler and domain 0 kernel. After that, we focus on domain 0 only and show the latency and throughput under four levels of interference workload. Finally, we use an end-to-end task set to evaluate the combined effect of the VMM scheduler and domain 0 on the end-to-end performance of IDC. All the experiments use the same setup as in Section 5.3.

5.5.1 Interference within the Same Core

We repeated the experiments in Section 5.3.1 with RTCA, using a batch size of 1 (which as later experiments show, gives better latency performance). For brevity and ease of comparison, we plotted the results in Figure 5.3, where the lines marked by circles show results obtained using RTCA. A key observation is that the difference between the two dashed lines

(and similarly, between the two solid lines) is small. This indicates that when domain 0 is not busy, the VMM scheduler plays a more important role, which is to be expected since the RT-Xen scheduler can effectively prevent priority inversion within the same core, and thus the interference from other VCPUs is much less.

Summary: When domain 0 is not busy, the VMM scheduler dominates the IDC performance for higher priority domains.

5.5.2 Interference from Multiple Cores

The subsequent experiments focused on showing the effect of domain 0 when it becomes the performance bottleneck. We use *Original* to represent the default communication architecture in contrast to RTCA in domain 0.

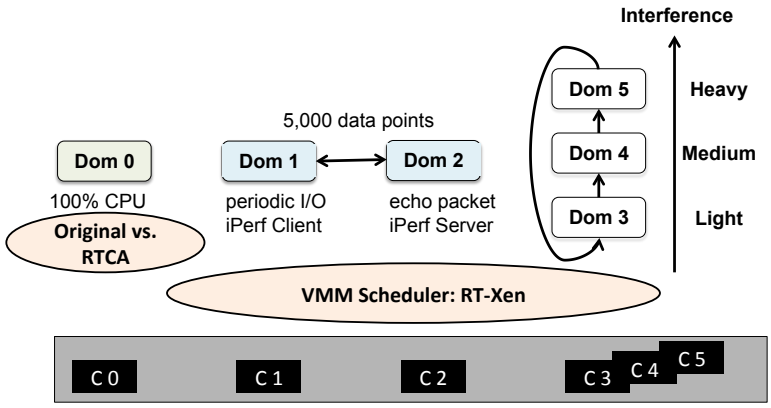


Figure 5.7: Experiment with Interference from Multiple Cores

Figure 5.7 shows the setup, with three cores dedicated to domain 0 and the two highest priority domains, respectively, so they always get the CPU when needed, thus emulating the best that a VMM scheduler can do. On each of the remaining three cores, we booted up three interference domains, and gave each domain 30% of the CPU share. They all performed intensive IDC (constantly sending UDP packets to other domains). Interference was generated at four levels in different experiments, with *Base* being no interference, *Light* being only one active domain per core, *Medium* being two active domains per core, and *Heavy* having all three of them active.

As we discussed earlier, the batch size can affect the performance of RTCA. Therefore, we examined three batch sizes: 1, as it represents the most responsive domain 0; 64, as this would be the default batch size for the *softnet_data* queue; and 238, as this is the maximum batch size for the TX and RX Queues on our hardware. For the *Original* case, we kept everything as defaulted (process 64 packets per device per time, and 300 packets per round).

Latency

Table 5.1: Effect of Interference from Multiple Cores: Latency

	Median (μs)				75th percentile (μs)				95th percentile (μs)			
domain 0	Original	RTCA			Original	RTCA			Original	RTCA		
		1	64	238		1	64	238		1	64	238
Base	68	70	71	71	69	72	72	72	71	74	74	74
Light	**5183**	60	64	64	**5803**	61	115	90	**6610**	66	261	324
Medium	**9621**	61	216	**2421**	**9780**	63	272	**2552**	**11954**	68	363	**3404**
Heavy	**9872**	69	317	**3661**	**10095**	71	347	**4427**	**11085**	76	390	**4643**

Similar to the experiments in Section 5.3.1, the same periodic workload was used to measure the round-trip time between the two high priority domains, domain 1 and domain 2. Table 5.1 shows the median, 75%, and 95% values among 5000 data points. All values larger than 1000 μs (1 ms) are made bold for ease of comparison.

From those results, several key observations can be made:

- With the *Original* kernel, even under *Light* interference, the latency increases from about 70 μs to over 5 ms.
- In contrast, RTCA performs well for soft real-time systems: except for a batch size of 238, 95% of the data points are under 500 μs. This indicates that by prioritizing packets within domain 0, we can greatly reduce the IDC latency between high priority domains under interfering IDC from low priority domains.
- The smaller the batch size, the better and less varied the results. Using a batch size of 1 results in around 70 μs round trip times for all cases; with a batch size of 64, the latency grows to around 300 μs under interference; and with a batch size of 238 it

reaches to above 3 ms. This trend is due to the increasing blocking times caused by priority inversion in all three queues, as discussed in Section 5.2. As a result, using a batch size of 1 makes the system most responsive to high priority IDC.

Summary: By reducing priority inversion in domain 0, RTCA can effectively mitigate impacts of low priority traffic on the latency of high priority IDC.

Throughput

The previous experiment shows that using a batch size of 1 results in the best latency. However, a smaller batch size also means more frequent context switches, resulting in larger overhead and potentially reduced throughput. This experiment measures throughput under the same settings.

We kept the interference workload used in Section 5.5.2, and used iperf [51] (which is widely used in networking evaluations) in domain 1 and domain 2 to measure the throughput. Domain 2 ran the iperf server, while domain 1 ran the iperf client using the default configuration for 10 seconds. For each data point, the experiments were repeated 10 times, and we plotted the mean value. For completeness, results using the original kernel are also included in.

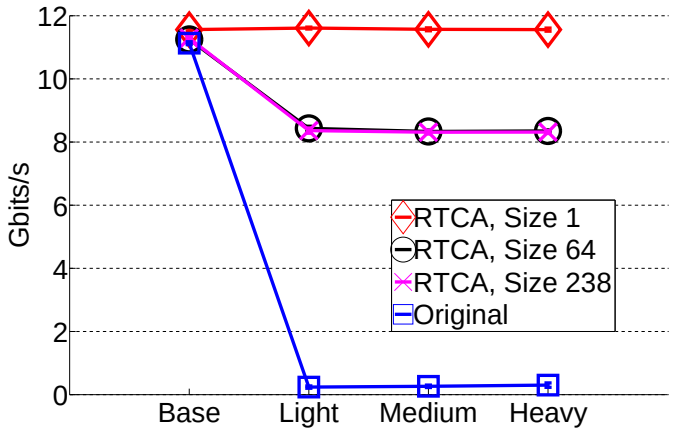


Figure 5.8: Interference from Multiple Cores: Throughput

As expected, in Figure 5.8, under the *Base* case, the original kernel and RTCA perform about the same, at 11.5 Gb/s. When there is interference, the throughput of high priority IDC

with the original kernel drops dramatically, to less than 1 Gb/s, due to priority inversions in domain 0. The RTCA with batch size 1 provides steady performance as the blocking time due to priority inversion stays relatively constant regardless of the interference level. This also indicates that in local IDC, the context switching time is insignificant. The size 64 and size 238 curves overlap each other, and all performed at about 8.3 Gb/s under interference. The reason is a larger batch size enables lower priority domains to consume more time in domain 0, making high priority IDC performance worse.

Summary: A small batch size leads to significant reduction in high priority IDC latency and improved IDC throughput under interfering traffic.

5.5.3 End-to-End Task Performance

The previous experiments used micro benchmarks to evaluate both the original domain 0 and RTCA in terms of latency and throughput. However, in typical soft real-time systems, a domain runs both computation and communication workloads. This section studies the combined effects of the VMM schedulers and the domain 0 communication architecture on end-to-end tasks.

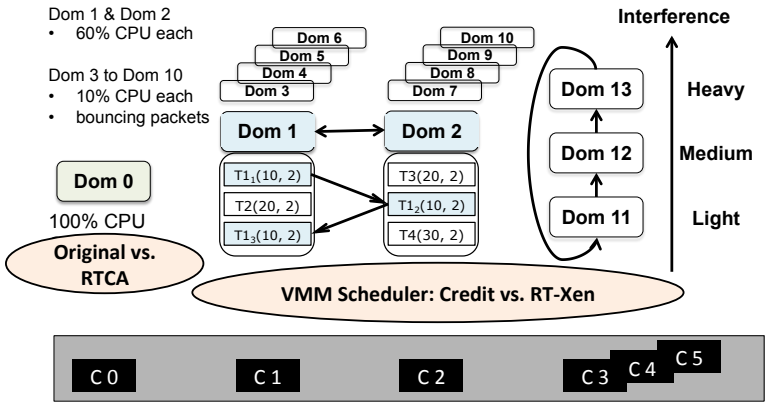


Figure 5.9: Experiment with End-to-End Tasks

Figure 5.9 shows the setup. domain 0 runs on a dedicated core. domain 1 and domain 2 are given the highest priority and are pinned to cores 1 and 2, respectively, each with 60% of the CPU share. Task $T1$ is an end-to-end task consisting of three subtasks, $T1_1$ and $T1_3$ in domain 1 and $T1_2$ in domain 2. A new instance of $T1$ is released every 10 ms. For

each instance, $T1_1$ first ran for 2 ms and sent a packet to $T1_2$ in domain 2. Once domain 2 received that packet, $T1_2$ ran for 2 ms and sent a packet back to domain 1. $T1_3$ received the packet and ran for 2 ms and completed the job of an end-to-end task. domain 1 also contains a local periodic task $T2$, and domain 2 contains two local periodic tasks, $T3$ and $T4$. To simulate interference within the same core, we booted four pairs of other domains, with each pair bouncing packets between each other. Each of the eight interfering domains was given 10% CPU share and assigned a lower priority. On the remaining three cores, a similar setup to that in Section 5.5.2 was used to generate IDC interference from multiple cores. For RTCA, since the results given in Section 5.5.2 already showed that using a batch size of 1 resulted in the best performance, we did not try other batch sizes. Each experiment ran for 10 seconds.

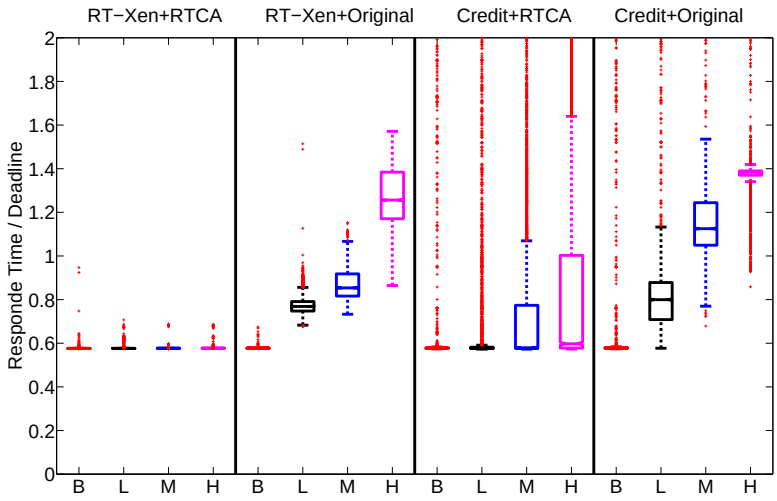


Figure 5.10: Box Plot of Normalized Latency for Task T1

We use a metric called the normalized latency, defined as the ratio between the response time of a task to its deadline. A task meets its deadline if its normalized latency is within 1; otherwise it misses its deadline. Figure 5.10 shows a box plot of the normalized latency for the end-to-end task $T1$ under different interference levels, with B indicating the *Base* case, L the *Light* case, M the *Medium* case, and H the *Heavy* case. On each box, the central mark represents the median value, whereas the upper and lower box edges show the 25th and 75th percentiles separately. If the data values are larger than $q_3 + 1.5 * (q_3 - q_1)$ or smaller than $q_1 - 1.5 * (q_3 - q_1)$ (where q_3 and q_1 are the 75th and 25th percentiles, respectively), they are considered outliers and plotted via individual markers. For clarity of presentation, any

job whose normalized latency is greater than 2 is not shown here (note here as well that if normalized latency is larger than 1, it means the job has missed its deadline).

Starting from the left, the "RT-Xen+RTCA" combination consistently meets its deadline, with the median normalized latency below 0.6. This shows that by combining the two improved subsystems, we can effectively alleviate interference both from the same core and from other cores. The "RT-Xen+Original" combination misses deadlines under heavy interference. The results confirm that when IDC is involved, domain 0 cannot be simply treated as a black box, due to the possible priority inversion within its communication subsystem. The "Credit+RTCA" combination performs slightly better than the second combination, but still incurs a large number of deadline misses (denoted as outliers in Figure 5.10) even under the *Base* case. This is due to the BOOST contention from domain 3 through domain 10. The "Credit+Original" combination performs the worst, as T1 suffers interference from all the other domains.

Summary: By combining the RT-Xen VMM scheduler and RTCA domain 0 kernel, we can deliver end-to-end real-time performance to tasks involving both computation and communication, even under interference from low-priority IDC flows.

5.6 Summary

This chapter addresses the open problem of supporting local *inter-domain communication* (IDC) within the same host. It examines the IDC performance of Xen, a widely used open-source virtual machine monitor that recently has been extended to support real-time domain scheduling. We show through both analysis and experiments that improving the VMM scheduler alone cannot achieve effective latency differentiation for IDC, due to significant priority inversion with the manager domain. To address this limitation, we have designed and implemented a *Real-Time Communication Architecture* (RTCA) within the manager domain to achieve effective prioritization among IDC flows. Empirical results demonstrate that combining RTCA and a real-time VMM scheduler can reduce the latency of high priority IDC significantly in the presence of heavy low priority traffic by effectively mitigating priority inversion within the manager domain.

Chapter 6

Conclusion

This dissertation seeks to answer the following questions, posed in Chapter 1:

Q1: What is an appropriate interface to provide resource guarantees for a real-time VM?

Q2: How do various scheduling algorithms perform in practice?

Q3: How to integrate real-time virtualization with cloud computing?

Q4: How to support real-time communication between different VMs?

With regards to Q1 and Q2, we have built RT-Xen, a real-time scheduling framework for Xen. We have conducted evaluations comparing different scheduling policies, server mechanisms, and priority schemes. For Q3, we presented RT-OpenStack, which integrates RT-Xen into the OpenStack and provides an RT-filter and RT-weigher for VM-to-host mapping. For Q4, we have developed RTCA, a real-time communication architecture which can maintain low latency between high priority domains under heavy interference traffic.

In Section 6.1, we first summarize all results. Then we address future work and open questions in Section 6.2, and conclude in Section 6.3.

6.1 Summary of Results

We briefly recap the key points of Chapters 2-5 here.

6.1.1 Real-Time Virtualization

In recent years, real-time systems have been evolving from being deployed on a single host towards running in a virtualized environment, or even in the cloud. The combination of real-time performance and virtualization raises significant research challenges in meeting the real-time performance requirement. We developed RT-Xen, which focuses on providing CPU resources to the VMs, with timing guarantees.

Resource Interfaces

We observe that there is a natural mapping between a virtualized environment and a two-level hierarchical scheduling model (described in Chapter 2); therefore, we used *budget* and *period* as the resource interface for a VCPU for a VM (in Chapter 2). Most single-core hierarchical scheduling theory assumes the same interface, but differs in the underlying different server mechanisms for *budget* management.

In a multi-core environment, where a VM can have multiple VCPUs, the interface in RT-Xen was changed a set of VCPUs, where each VCPU can have its own *budget*, *period*, and *cpumask* (in Chapter 3). As a result, RT-Xen is compatible with multiple multi-core hierarchical scheduling theories [36, 45, 75].

Empirical Comparison of Scheduling Algorithms

Three key factors decide a scheduling algorithm in a hierarchical scheduling environment: scheduling policies, server mechanisms, and priority schemes (in Chapter 3). We now recap the insights from our experiments.

Scheduling policies: We considered global scheduling and partition scheduling in this dissertation. While partitioned scheduling provides higher schedulability guarantees according to compositional scheduling analysis, in our experiments global scheduling outperformed partition scheduling in terms of the fraction of workloads that met their deadlines. We also found that on a platform with a shared last-level cache, the work-conserving benefits of global scheduling outweighed the cache penalty incurred by VM migration.

Server mechanisms: We considered six servers in the single core case (Chapter 2), and focused on comparing deferrable server and periodic server in the multi-core case (Chapter 3). On a single core platform, while in theory periodic server is superior and deferrable server performed the worst, in our experiments deferrable server generally performs well due to its work-conserving nature, while periodic server performed worst under overloaded circumstances. We then proposed two enhanced periodic servers to improve task response time while maintaining the theoretical schedulability of the original periodic server. In a multi-core environment, we found that deferrable server again outperformed periodic server due to its work-conserving nature.

Priority schemes: We considered earliest deadline first (EDF) and deadline monotonic (DM) priority schemes in this dissertation. In theory, EDF outperformed DM in partitioned scheduling, but underperformed DM in global scheduling. In our experiments, EDF outperformed DM under both partitioned and global scheduling.

Overall, our experiments study showed that the combination of gEDF with deferrable server in the VMM resulted in the best experimental performance. We are in the process of pushing this scheduling design into the mainstream distribution of Xen.

6.1.2 Real-Time Cloud Computing

We have developed RT-OpenStack, a cloud management system for co-hosting real-time VMs with non real-time VMs in a cloud computing environment. The salient feature of RT-OpenStack is to provide real-time performance to real-time VMs, while allowing non-real-time VMs to share the remaining CPU resources without interfering with the real-time performance of RT VMs.

6.1.3 RTCA

We addressed the problem of prioritizing local inter-domain communication in a virtualized host. We found that both the VMM scheduler and the manager domain can significantly impact latency under different conditions; therefore, improving the VMM scheduler along

with using RT-Xen could not effectively prevent priority inversion. To address this limitation, we developed the real-time communication architecture (RTCA) within the manager domain to achieve effective prioritization among flows. Experimental results showed that a combination of RTCA and RT-Xen could deliver end-to-end real-time performance to tasks involving both computation and communication, in the face of interfering requests for both CPU and network resources.

6.2 Open Questions and Future Work

The work presented in this dissertation represents a promising step towards real-time virtualization and cloud computing. Our work can be extended in the following aspects.

6.2.1 Real-Time Virtualization

Task model: The task sets evaluated in RT-Xen have two assumptions: sequentiality and independence. Therefore, the work can be extended by challenging these two assumptions: considering parallel tasks and tasks with dependency. In both cases, multiple VCPUs can have dependencies between each other, and the VMM scheduler needs to consider these constraints, as observed in [47]. Further research is needed to address this requirement in a virtualized environment.

Multiple last-level caches: We briefly studied the cache effect on a shared last-level cache platform in Chapter 3. When the underlying hardware platform has separate last-level caches, the cache-miss caused by VCPU migration can cause a dramatic increase in task execution time. As an earlier study on native OS-level scheduling [35] shows, a cluster scheduler achieves the best performance. In future work, it will be interesting to evaluate a cluster VMM scheduler by conducting a comprehensive study on cache effects on virtualized real-time systems under different cache architectures.

Co-locating the manager domain: In our experiments, we always dedicated one PCPU to the manager domain and ran guest VMs on the remaining cores. Co-scheduling guest VM

with the manager domain has been studied in [46] for the SEDF scheduler for a single core. Further research is needed to study this problem in a multi-core environment with RT-Xen.

Other scheduling algorithms: We covered global and partitioned scheduling policies, EDF and DM priority schemes, and multiple server mechanisms (deferrable, periodic, polling, sporadic, work-conserving periodic, and capacity-stealing periodic servers). There are other scheduling algorithms to consider: for example, cluster scheduler, job-level priority schemes, and constant bandwidth server. RT-Xen provides a scheduling framework to implement all these different algorithms, and it will be beneficial to the community to further study more of them.

6.2.2 Real-Time Cloud Computing

VM live migration: We considered the initial VM placement problem in RT-OpenStack. Another important function of the cloud management system is VM live migration, which raises significant research challenges in meeting real-time VM performance. Further research is needed to address the VM live migration problem.

Integration with real-time communication: Network latency is also important for a real-time application, especially when it is running in the cloud. It would be interesting to explore the integration of RT-OpenStack with a real-time communication architecture.

6.2.3 RTCA

Multicore manager domain: We assumed there is only one VCPU for the manager domain, and designed RTCA for the single VCPU situation. When the manager domain has multiple VCPUs, further research is needed to adapt RTCA accordingly.

NIC traffic: We considered only local inter-domain communication in this dissertation. It would be interesting to integrate NIC traffic into RTCA and still maintain the priority scheme.

6.3 Closing Remarks

Virtualization technologies are becoming ubiquitous in modern computer systems. They enable easy composition and effective isolation of independently developed applications. However, for a real-time application whose performance depends not only on the correctness of the results, but also on latency, the existing virtualization technology cannot satisfy real-time requirements, which fundamentally requires changes in both the VMM scheduler and the manager domain.

In this dissertation, we have presented three projects – RT-Xen, RT-OpenStack, and RTCA – to support real-time applications in virtualized environments. Furthermore, our work studies different scheduling algorithms and compares them in both theory and practice. Our system and experimental studies also bridge the critical gap between compositional real-time scheduling theory and practical virtualization platforms. This dissertation work therefore represents a promising step towards real-time virtualization and real-time cloud computing.

References

[1] Amazon EC2 Instance Types. http://wiki.xensource.com/xenwiki/CreditScheduler.

[2] Amazon Fire Phone. http://www.amazon.com/Fire_Phone_13MP-Camera_32GB/dp/B00E0E0WKQ#firefly.

[3] ARINC 653 and Virtualization Solutions, Architectures and Partitioning. .

[4] ARINC Standards. http://www.aviation-ia.com/standards/index.html.

[5] AUTomotive Open System ARchitecture (AUTOSAR). http://www.autosar.org/.

[6] Chaos Monkey Released Into the Wild. http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html.

[7] CloudStack: Open Source Cloud Computing. https://www.cloudstack.apache.org.

[8] Credit Scheduler. http://wiki.xen.org/wiki/Credit_Scheduler.

[9] Eucalyptus: Open Source Private Cloud Software. https://www.eucalyptus.org.

[10] Fiasco micro-kernel. http://os.inf.tu-dresden.de/fiasco/.

[11] HP Public Cloud - Enterprise Public Cloud Products. http://www.hpcloud.com/.

[12] Intel 64 and IA-32 Architectures Optimization Reference Manual. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[13] Intel Ivy Bridge 7-Zip LZMA Benchmark Results. http://www.7-cpu.com/cpu/IvyBridge.html.

[14] Kernel Based Virtual Machine. http://www.linux-kvm.org.

[15] Linux Advanced Routing and Traffic Control. http://www.lartc.org/.

[16] LITMUS-RT. http://www.litmus-rt.org/.

[17] Microsoft Shows Off "Power of the Cloud" With Azure Servers. http://www.forbes.com/sites/davidthier/2014/04/04/microsoft-shows-off-power-of-the-cloud-for-xbox-one/.

[18] OpenNebula: Flexible Enterprise Cloud Made Simple. https://www.opennebula.org.

[19] OpenStack: Open Source Software for Building Private and Public Clouds. https://www.openstack.org.

[20] Rackspace: The Leader in Hybrid Cloud. http://www.rackspace.com/.

[21] RT-Xen: Real-Time Virtualization Based on Compositional Scheduling. https://sites.google.com/site/realtimexen/.

[22] Sysbench benchmark. http://sourceforge.net/projects/sysbench.

[23] The Problem with Noisy Neighbors in the Cloud. http://allthingsd.com/20130225/the-problem-with-noisy-neighbors-in-the-cloud/.

[24] The Xen Project is Built for Cloud Computing. http://www.xenproject.org/users/cloud.html.

[25] The Xen Project's Hypervisor for the ARM architecture. http://www.xenproject.org/developers/teams/arm-hypervisor.html.

[26] Total Ship Computing Environment. http://www.raytheon.com/capabilities/products/ddg_1000/tech/tsce/index.html.

[27] Understanding CPU Steal Time - When should you be worried? http://blog.scoutapp.com/articles/2013/07/25/understanding-cpu-steal-time-when-should-you-be-worried.

[28] Xen Credit2 Scheduler. http://wiki.xen.org/wiki/Credit2_Scheduler_Development.

[29] Xen Network Configuration. http://xenbits.xen.org/docs/unstable/misc/xl-network-configuration.html.

[30] Sebastian Altmeyer, Robert I Davis, and Claire 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.

[31] Theodore P Baker. A comparison of global and partitioned edf schedulability tests for multiprocessors. In *In International Conf. on Real-Time and Network Systems*. Citeseer, 2005.

[32] Scott Banachowski, Timothy Bisson, and Scott A Brandt. Integrating best-effort scheduling into a real-time system. In *Real-Time Systems Symposium, 2004. Proceedings. 25th IEEE International*, pages 139–150. IEEE, 2004.

[33] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[34] Sanjoy Baruah and Bjorn Brandenburg. Multiprocessor feasibility analysis of recurrent task systems with specified processor affinities. In *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*, pages 160–169. IEEE, 2013.

[35] Andrea Bastoni, Björn B Brandenburg, and James H Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 14–24. IEEE, 2010.

[36] Enrico Bini, Giorgio Buttazzo, and Marko Bertogna. The multi supply function abstraction for multiprocessors. In *Embedded and Real-Time Computing Systems and Applications, 2009. RTCSA'09. 15th IEEE International Conference on*, pages 294–302. IEEE, 2009.

[37] Björn B Brandenburg and James H Anderson. Integrating hard/soft real-time tasks and best-effort jobs on multiprocessors. In *Real-Time Systems, 2007. ECRTS'07. 19th Euromicro Conference on*, pages 61–70. IEEE, 2007.

[38] Björn B Brandenburg and James H Anderson. On the implementation of global real-time schedulers. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 214–224. IEEE, 2009.

[39] Luwei Cheng, Cho-Li Wang, and Sheng Di. Defeating network jitter for virtual machines. In *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*, pages 65–72. IEEE, 2011.

[40] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

[41] Mark Claypool and Kajal Claypool. Latency and player actions in online games. *Communications of the ACM*, 49(11):40–45, 2006.

[42] Rob Davis, Alan Burns, et al. An investigation into server parameter selection for hierarchical fixed priority pre-emptive systems. In *16th International Conference on Real-Time and Network Systems (RTNS 2008)*, 2008.

[43] Robert I Davis and Alan Burns. Hierarchical fixed priority pre-emptive scheduling. In *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, pages 10–pp. IEEE, 2005.

[44] Zhong Deng and Jane W-S Liu. Scheduling real-time applications in an open environment. In *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*, pages 308–319. IEEE, 1997.

[45] Arvind Easwaran, Insik Shin, and Insup Lee. Optimal virtual cluster-based multiprocessor scheduling. *Real-Time Systems*, 43(1):25–59, 2009.

[46] Sriram Govindan, Arjun R Nath, Amitayu Das, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Xen and co.: communication-aware cpu scheduling for consolidated xen-based hosting platforms. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 126–136. ACM, 2007.

[47] Zonghua Gu and Qingling Zhao. A state-of-the-art survey on real-time issues in embedded systems virtualization. *Journal of Software Engineering & Applications*, 5(4), 2012.

[48] Ajay Gulati, Anne Holler, Minwen Ji, Ganesha Shanmuganathan, Carl Waldspurger, and Xiaoyun Zhu. Vmware distributed resource management: Design, implementation, and lessons learned. *VMware Technical Journal*, 1(1):45–64, 2012.

[49] Ajay Gulati, Ganesha Shanmuganathan, Anne Holler, and Irfan Ahmad. Cloud-scale resource management: challenges and techniques. In *Proceedings of the 3rd USENIX conference on Hot topics in cloud computing*, pages 3–3. USENIX Association, 2011.

[50] Diwaker Gupta, Rob Gardner, and Ludmila Cherkasova. Xenmon: Qos monitoring and performance profiling tool. *Hewlett-Packard Labs, Tech. Rep. HPL-2005-187*, 2005.

[51] Chung-Hsing Hsu and Ulrich Kremer. Iperf: A framework for automatic construction of performance prediction models. In *Workshop on Profile and Feedback-Directed Compilation (PFDC), Paris, France*. Citeseer, 1998.

[52] Chun-Ying Huang, Cheng-Hsin Hsu, Yu-Chun Chang, and Kuan-Ta Chen. Gaminganywhere: An open cloud gaming system. In *Proceedings of the 4th ACM multimedia systems conference*, pages 36–47. ACM, 2013.

[53] Wei Huang, Matthew J Koop, Qi Gao, and Dhabaleswar K Panda. Virtual machine aware communication libraries for high performance computing. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, page 9. ACM, 2007.

[54] Kangho Kim, Cheiyol Kim, Sung-In Jung, Hyun-Sup Shin, and Jin-Soo Kim. Inter-domain socket communications supporting high performance and full binary compatibility on xen. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 11–20. ACM, 2008.

[55] Tei-Wei Kuo and Ching-Hui Li. A fixed-priority-driven open environment for real-time applications. In *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*, pages 256–267. IEEE, 1999.

[56] Jaewoo Lee, Sisu Xi, Sanjian Chen, Linh TX Phan, Chris Gill, Insup Lee, Chenyang Lu, and Oleg Sokolsky. Realizing compositional scheduling through virtualization. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*, pages 13–22. IEEE, 2012.

[57] Giuseppe Lipari and Enrico Bini. Resource partitioning among real-time applications. In *Real-Time Systems, 2003. Proceedings. 15th Euromicro Conference on*, pages 151–158. IEEE, 2003.

[58] Jie Liu, Bodhi Priyantha, Ted Hart, Heitor S Ramos, Antonio AF Loureiro, and Qiang Wang. Energy efficient gps sensing with cloud offloading. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, pages 85–98. ACM, 2012.

[59] Will Lunniss, Sebastian Altmeyer, Claire Maiza, and Robert I Davis. Integrating cache related pre-emption delay analysis into edf scheduling. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 75–84. IEEE, 2013.

[60] Ben Pfaff, Justin Pettit, Keith Amidon, Martin Casado, Teemu Koponen, and Scott Shenker. Extending networking into the virtualization layer. In *Hotnets*, 2009.

[61] KK Ramakrishnan. Performance considerations in designing network interfaces. *Selected Areas in Communications, IEEE Journal on*, 11(2):203–219, 1993.

[62] Khaled Salah and A Qahtan. Implementation and experimental performance evaluation of a hybrid interrupt-handling scheme. *Computer Communications*, 32(1):179–188, 2009.

[63] Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov. Beyond softnet. In *Proceedings of the 5th annual Linux Showcase & Conference*, volume 5, pages 18–18, 2001.

[64] Lui Sha, John P Lehoczky, and Ragunathan Rajkumar. Solutions for some practical problems in prioritized preemptive scheduling. In *RTSS*, volume 86, pages 181–191, 1986.

[65] Insik Shin and Insup Lee. Periodic resource model for compositional real-time guarantees. In *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*, pages 2–13. IEEE, 2003.

[66] Insik Shin and Insup Lee. Compositional real-time scheduling framework. In *Real-Time Systems Symposium, 2004. Proceedings. 25th IEEE International*, pages 57–67. IEEE, 2004.

[67] Brinkley Sprunt. *Aperiodic task scheduling for real-time systems*. PhD thesis, Citeseer, 1990.

[68] Mark Stanovich, Theodore P Baker, An-I Wang, and Michael González Harbour. Defects of the posix sporadic server and how to correct them. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*, pages 35–45. IEEE, 2010.

[69] Jay K. Strosnider, John P. Lehoczky, and Lui Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *Computers, IEEE Transactions on*, 44(1):73–91, 1995.

[70] Jian Wang, Kwame-Lante Wright, and Kartik Gopalan. Xenloop: a transparent high performance inter-vm network loopback. In *Proceedings of the 17th international symposium on High performance distributed computing*, pages 109–118. ACM, 2008.

[71] Fetahi Wuhib, Rolf Stadler, and Hans Lindgren. Dynamic resource allocation with management objectivesimplementation for an openstack cloud. In *Network and service management (cnsm), 2012 8th international conference and 2012 workshop on systems virtualiztion management (svm)*, pages 309–315. IEEE, 2012.

[72] Xen Wiki. Xen common problems. `http://wiki.xen.org/wiki/Xen_Common_Problems`.

[73] Sisu Xi, Justin Wilson, Chenyang Lu, and Christopher Gill. Rt-xen: towards real-time hypervisor scheduling in xen. In *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on*, pages 39–48. ACM, 2011.

[74] Sisu Xi, Meng Xu, Chenyang Lu, Linh TX Phan, Christopher Gill, Oleg Sokolsky, and Insup Lee. Real-time multi-core virtual machine scheduling in xen. In *Embedded Software (EMSOFT), 2014 Proceedings of the International Conference on*. ACM, 2014.

[75] Meng Xu, Linh TX Phan, Insup Lee, Oleg Sokolsky, Sisu Xi, Chenyang Lu, and Christopher Gill. Cache-aware compositional analysis of real-time multi-core virtualization platforms. In *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*, pages 1–10. IEEE, 2013.

[76] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 55–64. IEEE, 2013.

[77] Xiaolan Zhang, Suzanne McIntosh, Pankaj Rohatgi, and John Linwood Griffin. Xensocket: A high-throughput interdomain transport for virtual machines. In *Middleware 2007*, pages 184–203. Springer, 2007.