# RT-Xen: Towards Real-time Hypervisor Scheduling in Xen

Sisu Xi, Justin Wilson, Chenyang Lu, and Christopher Gill
Department of Computer Science and Engineering
Washington University in St. Louis
{xis, wilsonj, lu, cdgill}@cse.wustl.edu

## ABSTRACT

As system integration becomes an increasingly important challenge for complex real-time systems, there has been a significant demand for supporting real-time systems in virtualized environments. This paper presents RT-Xen, the first real-time hypervisor scheduling framework for Xen, the most widely used open-source virtual machine monitor (VMM). RT-Xen bridges the gap between real-time scheduling theory and Xen, whose wide-spread adoption makes it an attractive platform for integrating a broad range of real-time and embedded systems. Moreover, RT-Xen provides an open-source platform for researchers and integrators to develop and evaluate real-time scheduling techniques, which to date have been studied predominantly via analysis and simulations. Extensive experimental results demonstrate the feasibility, efficiency, and efficacy of fixed-priority hierarchical real-time scheduling in RT-Xen. RT-Xen instantiates a suite of fixed-priority servers (Deferrable Server, Periodic Server, Polling Server, and Sporadic Server). While the server algorithms are not new, this empirical study represents the first comprehensive experimental comparison of these algorithms within the same virtualization platform. Our empirical evaluation shows that RT-Xen can provide effective real-time scheduling to guest Linux operating systems at a 1ms quantum, while incurring only moderate overhead for all the fixed-priority server algorithms. While more complex algorithms such as Sporadic Server do incur higher overhead, none of the overhead differences among different server algorithms are significant. Deferrable Server generally delivers better soft real-time performance than the other server algorithms, while Periodic Server incurs high deadline miss ratios in overloaded situations.

## Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application based Systems – Real-time and embedded systems; C.4 [**Computer Systems Organization**]: Performance of Systems – Performance attributes; D.4.7 [**Software**]: Operating Systems – Organization and Design – Hierarchical design

## Keywords

real-time virtualization; hierarchical scheduling; sporadic server, deferrable server, periodic server, polling server

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$10.00.

## 1. INTRODUCTION

*Virtualization* has been widely adopted in enterprise computing to integrate multiple systems on a shared platform. Virtualization breaks the one-to-one correspondence between logical systems and physical systems, while maintaining the modularity of the logical systems. Breaking this correspondence allows resource provisioning and subsystem development to be done separately, with subsystem developers stating performance demands which a system integrator must take into account. Subsystem developers provide *virtual machine images* (comprising guest operating systems and applications) that the system integrator can then load onto the appropriate physical machines. The result is an appropriately provisioned system without unnecessary cost or complexity. Recent years have seen increasing demand for supporting real-time systems in virtualized environments as system integration has become an increasingly important challenge for complex real-time systems. To support real-time and embedded systems, a *virtual machine monitor* (VMM) must deliver the desired real-time performance to the virtual machine images to ensure that the integrated subsystems meet their requirements.

This paper presents *RT-Xen*, the first hierarchical real-time scheduling framework for Xen. RT-Xen bridges the gap between hierarchical real-time scheduling theory and Xen, the most widely used open-source virtual machine monitor. On one hand, the real-time systems community has developed solid theoretical foundations for hierarchical scheduling [15, 18, 23, 25, 35]. On the other hand, the wide-spread adoption and large user base makes Xen [5] an attractive platform for integrating soft real-time and embedded systems. RT-Xen's marriage of hierarchical real-time scheduling and Xen therefore enables real-time and embedded systems developers to benefit from both solid real-time scheduling theory and a mainstream virtualization environment. Moreover, RT-Xen also provides an open-source experimental platform for the researchers and integrators to develop and evaluate hierarchical real-time scheduling techniques which to date have been studied predominantly via analysis and simulations only.

RT-Xen demonstrates the feasibility, efficiency, and efficacy of fixed-priority hierarchical real-time scheduling in virtualized platforms. A key technical contribution of RT-Xen is the instantiation and empirical evaluation of a set of fixed-priority servers (Deferrable Server, Periodic Server, Polling Server, and Sporadic Server) within a VMM. While the server algorithms are not new, our empirical study represents to our knowledge the first comprehensive experimental comparison of these algorithms in a widely used full-featured virtualization platform. Our empirical evaluation shows that while more complex algorithms such as Sporadic Server do incur higher overhead, overhead differences among different server algorithms are insignificant in RT-Xen. Furthermore, the Deferrable Server outperforms Xen's default Credit scheduler while generally delivering better soft real-time performance than the other server algorithms, but the Periodic Server performs worst in *overloaded* situations.

The rest of this paper is structured as follows. Section 2 compares RT-Xen with the state of the art in hierarchical scheduling and real-time virtualization. Section 3 provides background on the Xen scheduling framework, including its two default schedulers. In Section 4, we describe RT-Xen, a novel framework for scheduling and measuring the execution of real-time tasks, within which we demonstrate the ability to implement different real-time schedulers including Polling Server, Periodic Server, Deferrable Server, and Sporadic Server. In Section 5, we show how we can choose a suitable scheduling quantum for our scheduler, present detailed overhead measurements, and compare the performance of different schedulers to each other and to analytic predictions.

## 2. RELATED WORK

There has been a rich body of theoretical research on hierarchical real-time scheduling that covers both fixed-priority and dynamic-priority scheduling [1, 2, 4, 11, 15, 17–21, 23, 25, 26,28,29,31,35,36,43]. In this work we focus on fixed-priority scheduling due to its simplicity and efficiency. As a starting point for developing a hierarchical real-time scheduling framework in Xen, our current implementation supports only independent, periodic CPU-intensive tasks. Recent advances in hierarchical real-time scheduling theory address more sophisticated task models involving resource sharing [6, 16, 33] and multiprocessors [7–9, 27, 34], which we plan to consider in future work.

Hierarchical real-time scheduling has been implemented primarily within OS kernels [3, 30, 40]. Aswathanarayana et al. [3] show that a common hierarchical scheduling model for thread groups can be implemented at either the middleware or kernel level with precise control over timing behavior. Note that these systems implement all levels of the scheduling hierarchy within the same operating system. In sharp contrast, RT-Xen splits the scheduling hierarchy into two levels: one at the hypervisor level and the other within each guest OS.

Recent efforts closely related to our work on RT-Xen include [12, 35] which examine real-time virtualization using L4/Fiasco as the hypervisor and L4Linux as the guest OS. Shin et al. [42] use a Periodic Server at the root level, and compare it to a round robin scheduler and a fixed priority scheduler. Crespo et al. [13] propose a bare-metal hypervisor which also uses para-virtualization and dedicated device techniques as in Xen. It runs on the SPARC V8 architecture and adopts a fixed cyclic scheduling policy. Cucinotta et al. [14] use the KVM hypervisor, with a Constant Bandwidth Scheduler algorithm as the global scheduler. RT-Xen differs from these works in that it builds on the scheduling framework of Xen, whose broad user base and adoption makes it an attractive virtualization platform for soft real-time and embedded systems. Furthermore, RT-Xen instantiates a suite of four server algorithms (Polling Server, Periodic Server, Deferrable Server, and Sporadic Server) and provides to our knowledge the first comprehensive empirical comparison of these algorithms in a VMM. Our empirical studies lead to important insights about both fixed-priority hierarchical scheduling in general and the relative merits of different server algorithms in terms of soft real-time performance and efficiency in a VMM.

RT-Xen is complementary to other recent work on adding real-time capabilities to Xen. Lee et al. [24] improve the Credit scheduler for domains that run both CPU and I/O intensive applications. Govindan et al. [22] enhance the network I/O performance using a simple heuristic scheme: scheduling domains by the number of pending packets. While these approaches adopt heuristic techniques to enhance real-time performance, RT-Xen leverages hierarchical real-time scheduling algorithms based on real-time scheduling theory. Those techniques also consider I/O issues which are not addressed by RT-Xen. As future work, we plan to build on their insights to develop more rigorous real-time scheduling support for I/O in RT-Xen.

## 3. BACKGROUND

This section provides background information on the Xen hypervisor, the scheduling framework in Xen, and the two default schedulers provided with Xen. It also describes the tasks and guest OS scheduler.

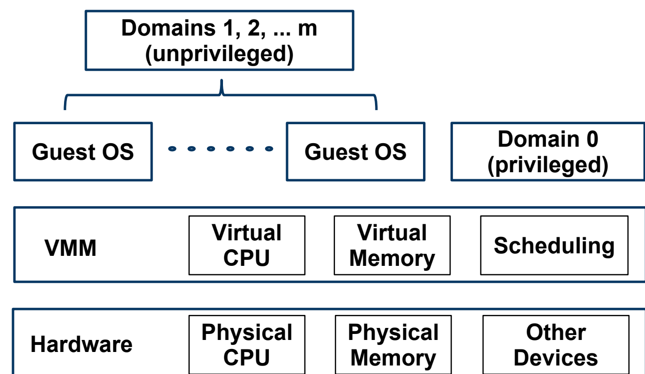



Figure 1: Architecture of a Xen System

Xen was developed by Barham et al. in 2003 [5] and has become the most widely used open-source *virtual machine monitor* (VMM). A VMM lies between the hardware and guest operating systems, allowing multiple guest operating systems to execute concurrently. The VMM controls essential processor and memory resources in a technique known as *para-virtualization*, where a specific *domain* called *domain 0* is created at boot time and is responsible for creating, starting, stopping, and destroying other domains (also known as *guest operating systems*). The guest operating systems are modified to perform I/O through virtualized drivers, e.g.,

for virtual hard disks and network cards that pass on I/O requests to the VMM. The VMM then redirects the I/O requests to *domain 0*, which contains the actual drivers for accessing the hardware. The architecture of Xen is shown in Figure 1.

## 3.1 Scheduling Framework in Xen

The VMM must ensure that every running guest OS receives an appropriate amount of CPU time. The main scheduling abstraction in Xen is the virtual CPU (VCPU), which appears as a normal CPU to each guest OS. To take advantage of symmetric multiprocessing, each domain can be configured with one or more VCPUs, up to the number of underlying physical cores. A VCPU in Xen is analogous to a process in a traditional operating system. Just as the scheduler in a traditional operating system switches among processes as they become ready or blocked, the scheduler in Xen switches among VCPUs.

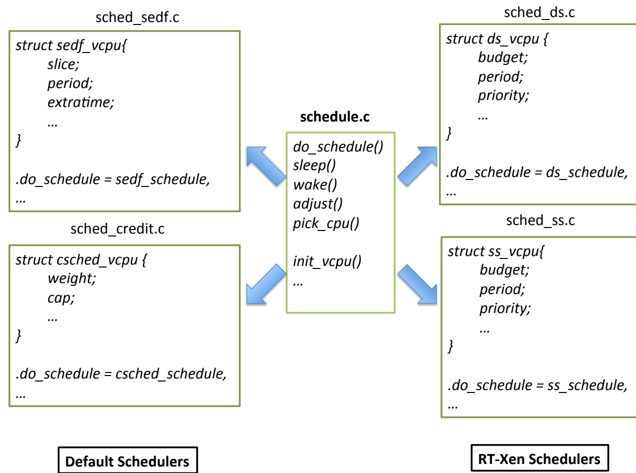



Figure 2: Xen Schedulers

Xen provides a well-defined scheduling interface, as shown in Figure 2. The left side shows the structure of the default schedulers, while the right side shows the RT-Xen schedulers discussed in detail in Section 4. A common *schedule.c* file defines the framework for the Xen scheduler, which contains several functions including *do_schedule*, *sleep*, *wake*, etc. To implement a specific scheduling algorithm, a developer needs to implement a *subscheduler* file(for example, *sched_credit.c*) which defines all these functions, and then hook them up to the Xen scheduling framework. A developer can also design specific VCPU data structures in the *subscheduler* file. Among these functions, the most important ones for real-time performance are:

- *do_schedule*:This function decides which VCPU should be running next, and 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), and if it has higher priority than the currently running one, an interrupt is raised to trigger 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.

Xen currently ships with two schedulers: the Credit scheduler and the Simple EDF (SEDF) scheduler. 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). An IDLE VCPU for each physical core is also created at boot time. It is always *runnable* and is always put at the end of the RunQ. When the IDLE VCPU is scheduled, the physical core becomes IDLE. Each domain contains two parameters: *weight* and *cap*, as shown in Figure 2. *Weight* defines its proportional share, and *cap* defines the upper limit of its execution time. 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_scheduler* function again to pick the next one. This quantum involves tradeoffs between real-time performance and throughput. A large quantum may lead to poor real-time performance due to coarse-grained scheduling. As is discussed in Sections 4 and 5, for fair comparison, we used the same quantum (1 ms) for the credit scheduler and our schedulers.

Xen also ships with a SEDF scheduler, in which every VCPU has three parameters: *slice* (equals *budget* in our RT-Xen scheduler), *period*, and *extratime* (whether or not a VCPU can continue to run after it runs out of its *slice*), as is shown in Figure 2. The SEDF scheduler works much like a Deferrable Server, where each VCPU's *slice* is consumed when running, preserved when not running, and set 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*. Although SEDF uses dynamic priorities, and the focus of this paper is on fixed-priority scheduling, we include SEDF in our evaluation for completeness. When conducting the experiments in Section 5, we configured the same SEDF *slice* and *period* as our RT-Xen schedulers' *budget* and *period*, and disabled *extratime* to make a fair comparison. Please note that SEDF is no longer in active development, and will be phased out in the near future [41].

## 3.2 Guest OS Scheduler and Tasks

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 jobs of its tasks, as depicted in Figure 3. We will describe the tasks and guest OS scheduler here, and introduce the VMM Scheduler in Section 4. Note that our hypervisor scheduler does not require any assumptions on tasks or guest OS schedulers. We implemented Rate-Monotonic Scheduling on Linux only as an example of guest OS scheduling for our experimental study. We also implemented periodic tasks, as they are required by the schedulability analysis presented in [15] which we used to assess the pessimism of the worst-case analysis at the end of Section 5.

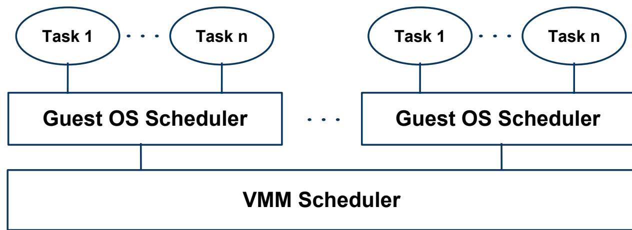



Figure 3: System Model

### *Task Model*

A set of periodic tasks runs on each guest OS. 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 [15, 23, 35]. In future work, we plan to extend RT-Xen to support more sophisticated task models.

### *Guest OS Scheduler*

Each guest OS is responsible for scheduling its tasks. The current implementation of RT-Xen supports Linux. To be consistent with existing hierarchical scheduling analysis [15], we used the pre-emptive fixed-priority scheduling class in Linux to schedule the tasks in the experiments described in Section 5. Each guest OS is allocated one VCPU. As [10] shows, using a dedicated core to deal with interrupts can greatly improve system performance, so we bind *domain 0* to a dedicated core, and bind all other guest operating systems to another core to minimize interference, as we discuss in more detail in Section 4.2.

# 4. DESIGN AND IMPLEMENTATION

This section presents the design and implementation of RT-Xen, which is shaped by both theoretical and practical concerns. Section 4.1 describes the four fixed-priority Schedulers in RT-Xen, and section 4.2 describes the VMM scheduling framework within which different root schedulers can be configured for scheduling guest operating systems.

## 4.1 VMM Scheduling Strategies

In this paper, we consider four servers: Deferrable Server [39], Sporadic Server [37], Periodic Server, and Polling Server [32]. These scheduling policies have been studied in the recent literature on hierarchical fixed-priority real-time scheduling [15, 23, 35]. 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 [17], 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 pre-emptive fixed-priority scheduling.

- A *Deferrable Server* is invoked with a fixed period. If the VCPU has tasks ready, it executes them until either 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 4.2.
- A *Polling Server* is also referred to as a *Discarding Periodic Server* [15]. 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 [38]. Implementation details again can be found in Section 4.2.

## 4.2 VMM Scheduling Framework

As we described in Section 3, 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 OSes are pinned on one specific physical core. In all four schedulers, each VCPU has three parameters: budget, period, and priority. 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, as is shown in Figure 2.

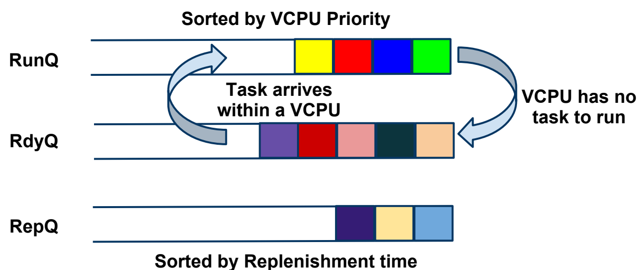



Figure 4: Three Queues within One Physical Core

**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

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 VCPU, which always has the lowest priority and is put at the end of the RunQ.

- 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 5).
- The RdyQ holds all VCPUs that have no task to run. It is designed especially for Periodic Server to mimic the "as if budgets are idled away" behavior. When the highest VCPU becomes IDLE and still has budget to run, we would 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 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.

Figure 4 illustrates the three different queues, as well as how a VCPU migrates between the RunQ and the RdyQ.

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.

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 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 has higher priority, return it to run, else, 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 [38], 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 [38]. 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 consumed budget since its last run, and registers a replenishment in the RdyQ. In this way, the replenishment is correctly split and a higher priority VCPU won't affect the lower priority ones. Interested readers are directed to [38] 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.

We implement *sched_ds.c* and *sched_ss.c* in about 1000 lines of C code each, as extensions within the framework provided by Xen. We also extend the existing XM utility for on-the-fly manual adjustment of the *budget*, *period*, and *priority* of each VCPU. All the source code for our schedulers and the tools, along with the all the test programs and generated random task sets is available as open-source software at *sites.google.com/site/realtimexen.*

## 5. EVALUATION

In this section, we evaluate the RT-Xen scheduling framework based on the following criteria. First, we measured real-time performance with different scheduling quanta, ranging from 1 millisecond down to 10 microseconds. Based on the results, 1 millisecond was chosen as our 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 present an empirical evaluation of *soft real-time* performance under different system loads.

## 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 needs exactly 1ms on one core (using native Linux), and then scaled it to generate any workload specified at a millisecond resolution. As we noted in Section 4, the work load 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.

## 5.2 Impact of 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 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 100ms, 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.

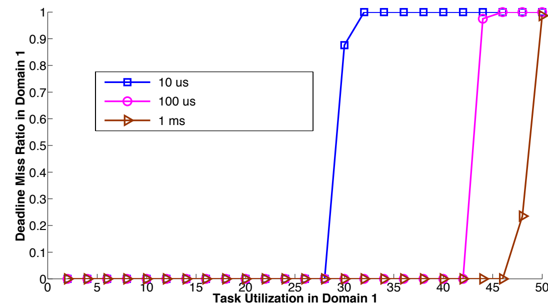



Figure 5: Performance under Different Scheduling Quanta

Figure 5 shows the results for scheduling quanta varying from 1ms to 10 $\mu$s. From this figure, we see a deadline miss starting at 48 % for 1ms, 44 % for 100 $\mu$s, and 30 % for 10 $\mu$s. When 1 $\mu$s was chosen, the overhead is 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.

## 5.3 Overhead Measurement

The focus of this work is fixed-priority pre-emptive hierarchical scheduling, within which we can compare the different server schemes. Therefore we consider the forms of overhead which are most relevant to the 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 corresponding status.
- *context switch*: the time required to save the context

Table 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 |

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 not dramatically different for the different scheduler 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 5.5, Table 3. The total system load was set to 70 %, and each domain ran 5 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 displayed via "*xm dmesg*".

We make the following observations from the results shown in Table 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 be caused 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 would 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 ours.

## 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 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 5.3. Every experiment ran for 2 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 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.

- All five VCPUs are treated equally, so the Credit scheduler picks them in a round-robin fashion, causing *do-*

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

Table 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 |

*main 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. For the Credit scheduler, although it 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.

## 5.5 Soft Real-Time Performance

Table 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 compare 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 ($\frac{budget}{period}$). All five domains' shares add up to 100 %, as shown in Table 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 $\alpha$ is defined as the total system load. It varied from 30 % to 100 %, with a step of 5 %. For each $\alpha$, we generated five tasks per domain, making 25 in total. Within each domain, we first randomly generated a *cost* between 5 and 10 for each of the five tasks (using $\alpha$ as a random seed), then randomly distributed the domain's share times $\alpha$ (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 $\alpha$ 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 in the VMM level.

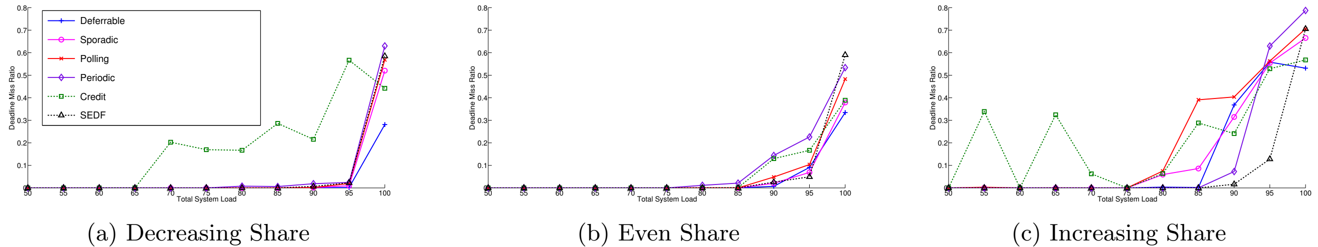
For completeness, we again include results for the same workload running under the Credit and SEDF schedulers as well. For the Credit scheduler, the scheduling quantum was configured at 1 ms. The *weight* was assigned according to the domain's relative share. For example, if a domain's share was 20 %, its weight took 20 % of the total weight. The *cap* was set to 0 as in 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 5 minutes. Figure 6 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 graph. 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 % deadline 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 %)?

From Figure 6, 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




(a) Decreasing Share (b) Even Share (c) Increasing Share

Figure 6: Deadline Miss Ratio under Different Shares

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" pre-emption effect in terms of worst-case guarantees, such effects rarely happen in real experiments.

- Among RT-Xen schedulers, the Periodic Server scheduler performs worst in the *overloaded* situation. As we discussed in Section 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 4: Theory Guaranteed Schedulable System Loads

| | 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 use the same settings as in [15], we also applied that analysis to the task parameters for comparison. Note that all the tasks are considered "unbound" because the tasks' *periods* are generated randomly, and we assume the overhead is 0. Table 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 nearly 85 %.

# 6. CONCLUSIONS

We have developed RT-Xen, the first hierarchical real-time scheduling framework for Xen, the most widely used open-source virtual machine monitor (VMM). 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 empirical studies show 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. RT-Xen represents a promising step toward real-time virtualization for real-time system integration. Building upon the initial success of RT-Xen, we plan to extend it in several important future directions including resource sharing, I/O, and the exploitation of multiprocessor and multicore architecture by leveraging recent advances in hierarchical real-time scheduling theory.

# ACKNOWLEDGEMENTS

This research was supported in part by NSF grants CNS-0821713 (MRI), CNS-0834755 (SUBSTRATE), CNS-0448554 (CAREER), and CCF-0448562 (CAREER).

# 7. REFERENCES

[1] L. Almeida and P. Pedreiras. Scheduling Within Temporal Partitions: Response-Time Analysis and Server Design. In *EMSOFT*, 2004.

[2] M. Anand, A. Easwaran, S. Fischmeister, and I. Lee. Compositional Feasibility Analysis of Conditional Real-Time Task Models. In *ISORC*, 2008.

[3] T. Aswathanarayana, D. Niehaus, V. Subramonian, and C. Gill. Design and Performance of Configurable Endsystem Scheduling Mechanisms. In *RTAS*, 2005.

[4] P. Balbastre, I. Ripoll, and A. Crespo. Exact Response Time Analysis of Hierarchical Fixed-Priority Scheduling. In *RTCSA*, 2009.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *SOSP*, 2003.

[6] M. Behnam, I. Shin, T. Nolte, and M. Nolin. SIRAP: A Synchronization Protocol for Hierarchical Resource Sharing in Real-Time Open Systems. In *EMSOFT*, 2007.

[7] E. Bini, M. Bertogna, and S. Baruah. Virtual Multiprocessor Platforms: Specification and Use. In *RTSS*, 2009.

[8] E. Bini, G. Buttazzo, and M. Bertogna. The Multi Supply Function Abstraction for Multiprocessors. In *RTAS*, 2009.

[9] E. Bini, G. Buttazzo, J. Eker, S. Schorr, R. Guerra, G. Fohler, K. Arzen, V. Romero, and C. Scordino. Resource Management on Multi-core Systems: the ACTORS Approach. *IEEE Micro*, 2011.

[10] B. Brandenburg and J. Anderson. On the Implementation of Global Real-Time Schedulers. In *RTSS*, 2009.

[11] R. Bril, W. Verhaegh, and C. Wust. A Cognac-glass Algorithm for Conditionally Guaranteed Budgets. In *RTSS*, 2006.

[12] F. Bruns, S. Traboulsi, D. Szczesny, E. Gonzalez, Y. Xu, and A. Bilgic. An Evaluation of Microkernel-Based Virtualization for Embedded Real-Time Systems. In *ECRTS*, 2010.

[13] A. Crespo, I. Ripoll, and M. Masmano. Partitioned Embedded Architecture Based on Hypervisor: the XtratuM Approach. In *EDCC*, 2010.

[14] T. Cucinotta, G. Anastasi, and L. Abeni. Respecting Temporal Constraints in Virtualised Services. In *COMPSAC*, 2009.

[15] R. Davis and A. Burns. Hierarchical Fixed Priority Pre-Emptive Scheduling. In *RTSS*, 2005.

[16] R. Davis and A. Burns. Resource Sharing in Hierarchical Fixed Priority Pre-Emptive Systems. In *RTSS*, 2006.

[17] R. Davis and A. Burns. An Investigation into Server Parameter Selection for Hierarchical Fixed Priority Pre-emptive Systems. In *RTNS*, 2008.

[18] Z. Deng and J. Liu. Scheduling Real-Time Applications in an Open Environment. In *RTSS*, 1997.

[19] A. Easwaran, M. Anand, and I. Lee. Compositional Analysis Framework Using EDP Resource Models. In *RTSS*, 2007.

[20] A. Easwaran, I. Shin, O. Sokolsky, and I. Lee. Incremental Schedulability Analysis of Hierarchical Real-Time Components. In *EMSOFT*, 2006.

[21] X. Feng and A. Mok. A Model of Hierarchical Real-Time Virtual Resources. In *RTSS*, 2002.

[22] S. Govindan, A. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam. Xen and Co.: Communication-aware CPU Scheduling for Consolidated Xen-based Hosting Platforms. In *VEE*, 2007.

[23] T. Kuo and C. Li. A Fixed Priority Driven Open Environment for Real-Time Applications. In *RTSS*, 1999.

[24] M. Lee, A. Krishnakumar, P. Krishnan, N. Singh, and S. Yajnik. Supporting Soft Real-Time Tasks in the Xen Hypervisor. In *VEE*, 2010.

[25] G. Lipari and E. Bini. Resource Partitioning among Real-Time Applications. In *ECRTS*, 2003.

[26] G. Lipari and E. Bini. A Methodology for Designing Hierarchical Scheduling Systems. *Journal of Embedded Computing*, 2005.

[27] G. Lipari and E. Bini. A Framework for Hierarchical Scheduling on Multiprocessors: From Application Requirements to Run-time Allocation. In *RTSS*, 2010.

[28] A. Mok and X. Feng. Towards Compositionality in Real-Time Resource Partitioning Based on Regularity Bounds. In *RTSS*, 2001.

[29] A. Mok, X. Feng, and D. Chen. Resource Partition for Real-Time Systems. In *RTAS*, 2001.

[30] J. Regehr and J. Stankovic. HLS: A Framework for Composing Soft Real-Time Schedulers. In *RTSS*, 2001.

[31] S. Saewong, R. Rajkumar, J. Lehoczky, and M. Klein. Analysis of Hierarchical Fixed-Priority Scheduling. In *ECRTS*, 2002.

[32] L. Sha, J. Lehoczky, and R. Rajkumar. Solutions for Some Practical Problems in Prioritized Preemptive Scheduling. In *RTSS*, 1986.

[33] I. Shin, M. Behnam, T. Nolte, and M. Nolin. Synthesis of Optimal Interfaces for Hierarchical Scheduling with Resources. In *RTSS*, 2008.

[34] I. Shin, A. Easwaran, and I. Lee. Hierarchical Scheduling Framework for Virtual Clustering of Multiprocessors. In *ECRTS*, 2008.

[35] I. Shin and I. Lee. Periodic Resource Model for Compositional Real-Time Guarantees. In *RTSS*, 2003.

[36] I. Shin and I. Lee. Compositional Real-Time Scheduling Framework. In *RTSS*, 2004.

[37] B. Sprunt. *Aperiodic Task Scheduling for Real-Time Systems*. PhD thesis, 1990.

[38] M. Stanovich, T. Baker, A. Wang, and M. Harbour. Defects of the POSIX Sporadic Server and How to Correct Them. In *RTAS*, 2010.

[39] J. Strosnider, J. Lehoczky, and L. Sha. The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. *IEEE Transactions on Computers*, 1995.

[40] Y. Wang and K. Lin. The Implementation of Hierarchical Schedulers in the RED-Linux Scheduling Framework. In *ECRTS*, 2000.

[41] Xen Wiki. Credit-based cpu scheduler. `http://wiki.xensource.com/xenwiki/CreditScheduler`.

[42] J. Yang, H. Kim, S. Park, C. Hong, and I. Shin. Implementation of Compositional Scheduling Framework on Virtualization. In *CRTS*, 2010.

[43] F. Zhang and A. Burns. Analysis of Hierarchical EDF Pre-emptive Scheduling. In *RTSS*, 2007.