RT-Xen: Towards Real-time Hierarchical Scheduling in Xen

ABSTRACT
Recent years have seen significant demand for supporting real-time systems in virtualized environments as system integration become an increasingly important challenge for complex real-time systems. This paper presents 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 hierarchical real-time scheduling theory and Xen whose wide-spread adoption makes it an attractive platform to integrate 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 hierarchical real-time scheduling techniques which to date have been studied predominantly via analysis and simulations. Extensive experimental results demonstrates the feasibility, efficiency, and efficacy of fixed-priority hierarchical real-time scheduling in 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 a same virtualization platform. Our empirical evaluation shows that RT-Xen can provide effective real-time scheduling to guest Linux operating systems at 1ms quantum, while incurring moderate overhead with all the fixed server algorithms. While more complex algorithms such as Sporadic Server do incur higher overhead, overhead differences between different server algorithms are insignificant. Deferrable Server generally delivering 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

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

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 become an increasingly important challenge for complex real-time systems. To support real-time and embedded systems a virtual machine monitor (VMM) must deliver desired real-time performance to the virtual machine images to ensure that the integrated subsystems meet their real-time performance 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 [17, 20, 25, 27, 37]. On the other hand, the wide-spread adoption and large user base makes Xen [6] 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 between 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 performed worst in overloaded situations.

While our current implementation of RT-Xen has demonstrated the promise of hierarchical real-time scheduling in Xen, our current implementation supports a relatively simple task model comprising independent and CPU-intensive tasks. We plan to build on the promising initial results presented here, to extend RT-Xen to support more sophisticated task models by leveraging recent advances in real-time hierarchical scheduling theory [8–11, 18, 29, 35, 36].

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 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 [1–3, 5, 13, 17, 19–23, 25, 27, 28, 30, 31, 33, 37, 38, 45]. In this work we focus on fixed-priority scheduling due to its simplicity and efficiency (as demonstrated in our empirical evaluation presented in Section 5). As a starting point for developing a hierarchical real-time scheduling framework in Xen, our current implementation only supports independent, periodic CPU-intensive tasks. Recent advances in hierarchical real-time scheduling theory address more sophisticated task models involving resource sharing [8, 18, 35] and multiprocessors [9–11, 29, 36], which we plan to consider in future work.

Hierarchical real-time scheduling has been implemented primarily within OS kernels [4, 7, 32, 42]. Aswathanarayana et al. [4] 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 [14, 37] which examine real-time virtualization using L4/Fiasco as the hypervisor and L4Linux as the guest OS. Shin et al. [44] use a Periodic Server at the root level, and compare it to a round robin scheduler and a fixed priority scheduler. Crespo et al. [15] 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. [16] 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, Periodic, Deferrable, and Sporadic servers) and provides to our knowledge the first comprehensive empirical comparison of these algorithms in a VMM. Our comprehensive empirical studies lead to important insights on both the 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. [26] boost the response time for both CPU and I/O intensive domains by waking the VCPUs on the RunQ and sorting VCPUs according to priority. Govindan et al. [24] enhance the SEDF scheduler by using a priority inheritance protocol for network I/O. 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. These techniques consider I/O issues which are not addressed by RT-Xen. We plan to build on their insights to develop more rigorous real-time scheduling support for I/O in RT-Xen as future work.

3. BACKGROUND

This section provides background information on the Xen hypervisor, the scheduling framework in Xen, and the two default schedulers provided with Xen.

3.1 Overview of Xen

Xen was developed by Barham et al. in 2003 [6] 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 run at the same time. 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.

3.2 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. Each domain can be configured with one or more VCPUs, up to the number of underlying physical cores, to take advantage of symmetric multiprocessing. 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 block, the scheduler in Xen switches among VCPUs.

Other functions initialize and terminate domains and VCPUs, dump informations, 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. The Credit scheduler, every physical core has one Run Queue (RunQ), which holds all the runnable VCPUs (has task to run). An IDLE VCPU per 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 is 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 they 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, VCPUs are scheduled in a round robin fashion. By default, when picking a VCPU, the scheduler allows it to run for 30ms, and then triggers the do_schedule function again to pick the next one. For a comparison to RT-Xen schedulers in terms of real time performance, this granularity is too coarse. As is discussed in Section 4 and 5, we recomputed this interval to 1ms to improve scheduling precision, and to make a more relevant real-time scheduling comparison.

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.
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 [43].

4. RT-XEN DESIGN AND IMPLEMENTATION

This section presents the design and implementation of RT-Xen. Section 4.1 describes the system model on which RT-Xen is based, and section 4.2 describes the VMM scheduling framework within which different root schedulers can be configured for scheduling guest operating systems.

4.1 System Model

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 is depicted in Figure 3.

![System Model Diagram](image)

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 [17, 25, 37]. We plan to extend RT-Xen to support more sophisticated task models as future work.

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 [17], we use 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 [12] 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.

VMM Scheduler

The VMM scheduler is shaped by both theoretical and practical concerns. In this paper, we consider four servers: Deferrable [41], Sporadic [39], Periodic, and Polling [34]. These scheduling policies have been studied in the recent literature on hierarchical fixed-priority real-time scheduling [17, 25, 37]. 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 [19], 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, given 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 [17]. 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 replenished only after it has been used. We implement the enhanced Sporadic Server algorithm proposed in [40]. 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 are all 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.

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 scheduler 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 pick the next appropriate one to run.

Figure 4 illustrates the three different queues, and how a VCPU migrates between the RunQ and the RdyQ within one physical core.

**do_schedule Function Implementation**

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 to the RdyQ. The Polling Server differs from 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 positive budget on the RdyQ; in line 7, it would compare the VCPUs with 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 [40], 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 only allow each VCPU to run 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 is described in [40]. Our Sporadic Server implementation works as follows: each time the do_schedule function get 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 register a replenishment in the RdyQ. In this way, the replenishment is correctly split and a higher priority VCPU won’t effect the lower priority ones. Interested readers are directed to [40] 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 schedulers and the tools, along with the all the test programs and generated random task sets are available as open-source though due to double blind review, the means for obtaining the code is not given here.

**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

5. **EVALUATION**

In this section, we evaluate the RT-Xen scheduling framework based on the following criteria. First, we measure 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 is performed for each of the four schedulers. Third, we studied the impact of an overloaded domain on both higher and lower priority ones. Finally, an empirical evaluation of soft real-time performance under different system loads is presented.

5.1 **Platform and Methods**
We performed our experiments on a Dell Q9400 quad-core machine without hyper-threading. SpeedStep is disabled by default and each core runs 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, but during the experiments the network service and other inessential applications were shutdown to avoid interference.

Task Implementation

We now describe how we implemented real time tasks in atop the guest operating systems. Currently, the scheduling tick (jiffy) in Linux distributions can be configured at a millisecond level. This quantum is used as a lower bound for our tasks. We first calibrate the amount of work that needs exactly 1ms on one core (using native Linux), and then scale 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 is scheduled using SCHED_FIFO, and the priority is 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 use POSIX real time clocks to generate interrupts to release each job of a task, and record 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 would queue up until their predecessors complete. When each job finishes, its finish time is recorded using the RDTSC instruction which provides 1ns precision with minimal overhead. After all tasks finish, we use the first job’s release time to calculate every job’s release time and deadline, and compare each deadline with the corresponding job finish time. In this way, we can count the deadline miss ratio for each individual task. All the information is stored in locked memory to avoid memory paging overhead. Based on the collected data, we can get the total number of jobs that missed their deadlines within each OS. Dividing by the total number of jobs, we get the deadline miss ratio for each domain.

5.2 Impact of Quantum

In this experiment our goal is to find an appropriately fine-grained scheduling quantum involving acceptable overhead. We define 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 more detailed overhead measurement to Section 5.3.

We ranged 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, is 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 runs a single real time task with a deadline of 100ms, and its cost varies from 1ms to 50ms. For each utilization, we run the task with 600 jobs, and calculate how many deadlines are missed. The low priority domain is configured as domain 2 with a budget of 2 quanta and period of 4 quanta. It runs a busy loop to generate the most possible interference for domain 1. Note that under this setting, whenever domain 1 has 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 only run a single task within domain 1, and the task’s deadline is far larger than the domain’s period, the choice of scheduler does not matter, so we use Deferrable Server as the scheduling scheme.

Figure 5 shows the results for scheduling quanta varying from 1ms to 10 µs. From this figure, we see a deadline miss starting at 48% for 1ms, 44% for 100µs, and 30% for 10µs. When the 1µs was chosen, the overhead is so large that guest OS cannot even be booted. Based on these results, we chose 1ms as our scheduling quantum since it suffers only 4% loss (48%−44%), and would provide 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, and within which we can compare the different server schemes. Therefore we consider the forms of overhead which are most related 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 for the currently running VCPU, and 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 like 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 is set to 70 %, and each domain runs 5 tasks. For completeness, we run the same workload under the Credit and SEDF schedulers and measured their overheads as well. For the Credit scheduler, we keep weight the same for all the domains (because they have the same share\((\text{budget})\), and cap set to 0 by default. Recall that we recompiled it to run at 1ms resolution to give a fair comparison (the original setting was 30ms). For the SEDF scheduler, the same \((\text{slice}, \text{period})\) pair was configured as \((\text{budget, period})\) 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 is set to fire 10 seconds later (to give system some slack time to go back to a normal running state). When it fires, the experiment begins to record the time spent in the do_schedule function, and the time spent in each context switch. After 10 seconds, the recording finishes and the results are dumped via “xm dmesg”.

We have 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 VMM.

- Context switch overhead dominates the scheduling latency overhead, as 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 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. For the Credit and SEDF schedulers, they would return VCPU to run for up to its available credits or slices, and when an IDLE VCPU is selected, it would return it to run forever until interrupted by others. As a result, the time that do_schedule function triggered is significantly less than 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.

The same settings as 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 VCPU are treated equally, so the Credit scheduler picks them in a round-robin fashion, causing domain 1 to miss deadlines, while in the fixed-priority schedulers, it has the highest priority and would always be scheduled first until its budget exhausted.

- Domain 1 has the smallest period, its tasks generated also have the relatively tightest deadlines, which makes the case worse.

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

<table>
<thead>
<tr>
<th></th>
<th>Deferrable</th>
<th>Periodic</th>
<th>Polling</th>
<th>Sporadic</th>
<th>Credit</th>
<th>SEDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>total time in do_schedule</td>
<td>1.435 µs</td>
<td>1.767 µs</td>
<td>1.430 µs</td>
<td>1.701 µs</td>
<td>2.16 µs</td>
<td>519 µs</td>
</tr>
<tr>
<td>percentage of time loss in 10 seconds</td>
<td>0.21 %</td>
<td>0.22 %</td>
<td>0.21 %</td>
<td>0.23 %</td>
<td>0.04 %</td>
<td>0.09 %</td>
</tr>
<tr>
<td>do_schedule overhead (max)</td>
<td>5.642 ns</td>
<td>461 ns</td>
<td>370 ns</td>
<td>469 ns</td>
<td>382 ns</td>
<td>322 ns</td>
</tr>
<tr>
<td>do_schedule overhead (median)</td>
<td>121 ns</td>
<td>159 ns</td>
<td>121 ns</td>
<td>150 ns</td>
<td>108 ns</td>
<td>130 ns</td>
</tr>
<tr>
<td>99 % quantile values in do_schedule</td>
<td>250 ns</td>
<td>328 ns</td>
<td>252 ns</td>
<td>303 ns</td>
<td>328 ns</td>
<td>192 ns</td>
</tr>
<tr>
<td>number of do_schedule called</td>
<td>10,914</td>
<td>10,560</td>
<td>10,807</td>
<td>10,884</td>
<td>1,665</td>
<td>4,126</td>
</tr>
<tr>
<td>context switches overhead (max)</td>
<td>12,456 ns</td>
<td>13,528 ns</td>
<td>8,557 ns</td>
<td>11,239 ns</td>
<td>8,174 ns</td>
<td>8,177 ns</td>
</tr>
<tr>
<td>context switches overhead (median)</td>
<td>1,498 ns</td>
<td>1,555 ns</td>
<td>1,513 ns</td>
<td>1,569 ns</td>
<td>2,896 ns</td>
<td>2,370 ns</td>
</tr>
<tr>
<td>99 % quantile values in context switches</td>
<td>3,087 ns</td>
<td>3,972 ns</td>
<td>3,840 ns</td>
<td>3,881 ns</td>
<td>3,503 ns</td>
<td>3,089 ns</td>
</tr>
<tr>
<td>number of context switches performed</td>
<td>3,254</td>
<td>3,422</td>
<td>2,979</td>
<td>4,286</td>
<td>1,665</td>
<td>3,699</td>
</tr>
</tbody>
</table>

Table 1: Overhead measurement for 10 seconds
### 5.5 Soft Real-Time Performance

<table>
<thead>
<tr>
<th>Domain</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sporadic</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Periodic</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Polling</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Deferrable</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Credit</td>
<td>96%</td>
<td>0.1%</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SEDF</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Sporadic</td>
<td>0</td>
<td>0</td>
<td>49.8%</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Periodic</td>
<td>0</td>
<td>0</td>
<td>49.7%</td>
<td>0.28%</td>
<td>0</td>
</tr>
<tr>
<td>Polling</td>
<td>0.08%</td>
<td>0</td>
<td>48%</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Deferrable</td>
<td>0</td>
<td>0</td>
<td>1.6%</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Credit</td>
<td>100%</td>
<td>0</td>
<td>0</td>
<td>0.08%</td>
<td>0</td>
</tr>
<tr>
<td>SEDF</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3: Budget, Period and Priority Settings for 5 domains

This set of experiments compare the soft real-time performance of different servers. Note that our study differs from and complements previous theoretical comparisons which focus on their capabilities to provide hard real-time guarantees. We also compare the actual real-time performance against an existing response time analysis for fixed-priority servers to assess the pessimism of the analysis.

The experiments are 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 (\( \frac{\text{budget}}{\text{period}} \)). All five domain’s 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 are randomly generated following the steps below. A global variable \( \alpha \) is defined as the total system load. It varies 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 generate 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 * \( \alpha \) (which represents the real domain load) among the five tasks. Using every task’s cost and utilization, we can easily calculate its deadline. Note that all costs and deadlines are integers, so there is 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 scheduler as well. For the Credit scheduler, the scheduling quantum was recomputed at 1ms. The weight was assigned according to the domain’s relative share. For example, if a domain’s share is 20%, its weight takes 20% of the total weight. The cap is 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 (slide, 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 is from 30% to 50%, all deadline miss ratios are 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, 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 “significant” deadline miss? Since we are dealing with soft real-time systems, we consider a 5% deadline miss ratio as significant deadline misses, 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 performed poorly in terms of capacity, even when recomputed 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 performed 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” pre-emption effect in terms of worst-case guarantees, such effects happen rarely in real experiments.
- The Periodic Server scheduler performed worst in the overloaded situation among RT-Xen schedulers. As we discussed in Section 4, to mimic the “as if budget idled away” behavior, when a high priority VCPU
have 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 exhaust 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>
<thead>
<tr>
<th></th>
<th>Deferrable Server</th>
<th>Periodic Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decreasing</td>
<td>[30, 45]</td>
<td>[30, 50], [60, 75]</td>
</tr>
<tr>
<td>Even</td>
<td>[30, 45]</td>
<td>[30, 50], [60, 75]</td>
</tr>
<tr>
<td>Increasing</td>
<td>[30, 45]</td>
<td>[30, 50], [60, 75]</td>
</tr>
</tbody>
</table>

Table 4: Theory Guaranteed Schedulable System Loads

Since we use the same settings as in [17], we also applied that analysis to the task parameters for comparison. Note that all the tasks are considered “unbound” because 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 only guarantees it is schedulable if total system load is under 45 %, while in reality it is schedulable up near 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 hierarchical 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 hierarchical real-time scheduling techniques. Extensive experimental results demonstrate the feasibility, efficiency, and efficacy of fixed-priority hierarchical real-time scheduling in VMM. RT-Xen differ from prior efforts on 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. While the server algorithms are not new, this empirical study represents the first comprehensive experimental comparison of these algorithms in a single virtualization platform. Our empirical studies show that while complex algorithms do incur higher overhead, the overhead between 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 dynamic scheduling, resource sharing, I/O and exploiting multiprocessor and multicore architectures by leveraging recent advances in hierarchical real-time scheduling theory.

7. REFERENCES


