Real-Time Multi-Core Virtual Machine Scheduling in Xen

Sisu Xi† Meng Xu‡ Chenyang Lu† Linh T.X. Phan‡ Christopher Gill† Oleg Sokolsky‡ Insup Lee‡
†*Cyber-Physical Systems Laboratory, Washington University in St. Louis*
‡*University of Pennsylvania*

ABSTRACT

Recent years have witnessed two major trends in the development of complex real-time embedded systems. First, to reduce cost and enhance flexibility, multiple systems are sharing common computing platforms via virtualization technology, instead of being deployed separately on physically isolated hosts. Second, multicore processors are increasingly being used in real-time systems. The integration of real-time systems as virtual machines (VMs) atop common multicore platforms raises significant new research challenges in meeting the real-time performance requirements of multiple systems. This paper advances the state of the art in real-time virtualization by designing and implementing RT-Xen 2.0, a new real-time multicore VM scheduling framework in the popular Xen virtual machine monitor (VMM). RT-Xen 2.0 realizes a suite of real-time VM scheduling policies spanning the design space. We implement both global and partitioned VM schedulers; each scheduler can be configured to support dynamic or static priorities and to run VMs as periodic or deferrable servers. We present a comprehensive experimental evaluation that provides important insights into real-time scheduling on virtualized multicore platforms: (1) both global and partitioned VM scheduling can be implemented in the VMM at moderate overhead; (2) at the VMM level, while compositional scheduling theory shows partitioned EDF (pEDF) is better than global EDF (gEDF) in providing schedulability guarantees, in our experiments their performance is reversed in terms of the fraction of workloads that meet their deadlines on virtualized multicore platforms; (3) at the guest OS level, pEDF requests a smaller total VCPU bandwidth than gEDF based on compositional scheduling analysis, and therefore using pEDF at the guest OS level leads to more schedulable workloads in our experiments; (4) a combination of pEDF in the guest OS and gEDF in the VMM – configured with deferrable server – leads to the highest fraction of schedulable task sets compared to other real-time VM scheduling policies; and (5) on a platform with a shared last-level cache, the benefits of global scheduling outweigh the cache penalty incurred by VM migration.

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. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ESWEEK'14 October 12–17 2014, New Delhi, India
Copyright 2014 ACM 978-1-4503-3052-7/14/10 ...$15.00.
http://dx.doi.org/10.1145/2656045.2656061.

1. INTRODUCTION

Complex real-time systems are moving from physically isolated hosts towards common multicore computing platforms shared by multiple systems. Common platforms bring significant benefits, including reduced cost and weight, as well as increased flexibility via dynamic resource allocation. Virtualization has emerged as a promising technology for integrating systems as virtual machines (VMs) on a common physical embedded computing platform. For example, embedded hypervisors [1, 4] are being developed as automotive computing platforms for integrating both infotainment and safety critical systems. However, the integration of real-time systems as VMs on a common multicore computing platform brings significant challenges in simultaneously meeting the real-time performance requirements of multiple systems, which in turn require fundamental advances in the underlying VM scheduling at the virtual machine monitor (VMM) level.

As a step towards real-time virtualization technology for multicore processors, we have designed and implemented RT-Xen 2.0, a multicore real-time VM scheduling framework in Xen, an open-source VMM that has been widely adopted in embedded systems [9]. While earlier efforts on real-time scheduling in Xen [29,35,45,49] focused on single-core schedulers, RT-Xen 2.0 realizes a suite of multicore real-time VM scheduling policies spanning the design space. We have implemented both global and partitioned VM schedulers (rt-global and rt-partition); each scheduler can be configured to support dynamic or static priorities and to run VMs under periodic or deferrable server schemes. Our scheduling framework therefore can support eight combinations of real-time VM scheduling policies, enables us to perform comprehensive exploration of and experimentation with real-time VM scheduling on multicore processors. Moreover, RT-Xen 2.0 supports a range of resource interfaces used in compositional scheduling theory [27, 28, 43], which enables designers to calculate and specify the resource demands of VMs to the underlying RT-Xen 2.0 scheduler.

We have conducted a series of experiments to evaluate the efficiency and real-time performance of RT-Xen 2.0 with Linux and the LITMUS^{RT} [8] patch as the guest OS. Our empirical results provide insights on the design and implementation of real-time VM scheduling:

- Both global and partitioned real-time VM scheduling can be realized within Xen at moderate overhead.

- At the VMM level, while compositional schedulability analysis shows that partitioned EDF (pEDF) outperforms global EDF (gEDF) in terms of theoretical schedulability guarantees for tasks running in VMs, experimentally gEDF often outperforms pEDF in terms of the fraction of workloads actually schedulable on a virtualized multicore processor.

- At the guest OS level, pEDF requests a smaller total VCPU bandwidth than gEDF based on compositional schedulability analysis, and therefore using pEDF in the guest OS leads to more schedulable workloads. The combination of pEDF in the guest OS and gEDF in the VMM therefore results in the best experimental performance when using a periodic server.

- When pEDF is used as the guest OS scheduler, gEDF in the VMM – configured with deferrable server – leads to the highest fraction of schedulable task sets under system overload compared to other real-time VM scheduling policies.

- On a platform with a shared last-level cache, the benefits of global scheduling outweigh the cache penalty incurred by VM migration.

The rest of this paper is structured as follows: In Section II we first introduce background information on Xen. We then describe the design and implementation of the multicore real-time scheduling framework in RT-Xen 2.0 in Section III. Section IV presents our experimental evaluation. After reviewing other related work in Section V, we conclude the paper in Section VI.

2. BACKGROUND

We first review the scheduling architecture in Xen, and then discuss the compositional schedulability analysis (CSA) that serves as the theoretical basis for the design of our multicore VMM scheduling framework, RT-Xen 2.0.

2.1 Scheduling in Xen

Xen [11] is a popular open-source virtualization platform that has been developed over the past decade. It provides a layer called the virtual machine monitor (VMM) that allows multiple *domains* (VMs) to run different operating systems and to execute concurrently on shared hardware. Xen has a special domain, called *domain 0*, that is responsible for managing all other domains (called guest domains).

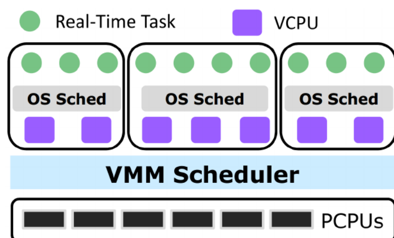


Figure 1: Xen scheduling architecture.

Figure 1 illustrates the scheduling architecture in Xen. As shown in the figure, each domain has one or more virtual CPUs (VCPUs), and tasks within the domain are scheduled on these VCPUs by the domain's OS scheduler. The VCPUs of all domains are scheduled on the physical CPUs (PCPUs), or cores, by the VMM scheduler. At runtime, each VCPU can be in either a *runnable* or a *non-runnable* (blocked by I/O or idle) state. When there are no runnable VCPUs to execute, the VMM scheduler schedules an idle VCPU. By default, Xen boots one idle VCPU per PCPU.

VMM schedulers in Xen. There are three VMM schedulers in Xen: credit, credit2 and simple EDF (SEDF). The credit scheduler is the default one, and it uses a proportional share scheme, where each domain is associated with a weight (which encodes the share of the CPU resource it will receive, relative to other domains) and a cap (which encodes the maximum CPU resource it will receive). The credit scheduler is implemented using a partitioned queue, and it uses a heuristic load balancing scheme: when a PCPU's run queue has only one VCPU with credit left, the PCPU "steals" VCPUs from other run queues. The credit2 scheduler also assigns a weight per domain, but it does not support caps and is not CPU-mask aware. As a result, it cannot limit each domain's CPU resource, nor can it dedicate cores to domains. Unlike the credit and credit2 schedulers, the SEDF scheduler schedules domains based on their interfaces (each being defined as a period and a budget) using an EDF-like scheduling policy. However, SEDF is not in active development and is planned to be "phased out and eventually removed" from Xen [2].

2.2 Compositional Schedulability Analysis

The resource interface for VMs in RT-Xen 2.0 are computed using multicore compositional schedulability analysis (CSA) [28]. This is enabled by a natural mapping between Xen and a two-level compositional scheduling hierarchy in CSA: each domain corresponds to an elementary component, and the system (a composition of the domains under the VMM scheduler) corresponds to the root component in CSA. Using this approach, we can represent the resource requirements of each domain as a multiprocessor periodic resource (MPR) interface, $\mu = \langle \Pi, \Theta, m' \rangle$, which specifies a resource allocation that provides a total of Θ execution time units in each period of Π time units, with a maximum level of parallelism m'. Each interface $\mu = \langle \Pi, \Theta, m' \rangle$ can be mapped to m' VCPUs whose total resource demand is at least equal to the resource supply of the interface. The VCPUs of the domains are then scheduled together on the PCPUs by the VMM scheduler. Based on the VCPUs and the VMM scheduling semantics, we can also derive an interface for the system (i.e., root component), which can be used to check the schedulability of the system or to determine the number of cores needed to schedule the system.

In this paper, we compute the interface of a domain using a state-of-the-art interface computation method corresponding to the scheduling algorithm used by the domain. Specifically, the interfaces of domains under pEDF or partitioned Deadline Monotonic (pDM) are computed using the interface computation method in [43]. The interfaces under gEDF are computed using the multicore interface computation method described in [27]. Finally, since there are currently no methods for gDM, we also propose a new method for computing the interfaces under gDM, which is described in detail in [46].

3. DESIGN AND IMPLEMENTATION

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

3.1 Design Principles

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

- a scheduling interface that is compatible with a range of resource interfaces (e.g., [15, 27, 47]) used in compositional schedulability theory;

- both global and partitioned schedulers, called rt-global and rt-partition, respectively;

- EDF and DM priority schemes for both schedulers;

- for each scheduler, a choice of either a work-conserving deferrable server or a periodic server.

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

Scheduling interface. In RT-Xen 2.0, the scheduling interface of a domain specifies the amount of resource allocated to the domain by the VMM scheduler. In a single-core virtualization setting, each domain has only one VCPU and thus, its scheduling interface can be defined by a *budget* and a *period* [35,45]. In contrast, each domain in a multicore virtualization setting can have multiple VCPUs. As a result, the scheduling interface needs to be sufficiently expressive to enable resource allocation for different VCPUs at the VMM level. At the same time, it should be highly flexible to support a broad range of resource interfaces, as well as different distributions of interface bandwidth to VCPUs (to be scheduled by the VMM scheduler), according to CSA theory.

Towards the above, RT-Xen 2.0 defines the scheduling interface of a domain to be a set of VCPU interfaces, where each VCPU interface is represented by a *budget*, a *period*, and a *cpu_mask* (which gives the subset of PCPUs on which the VCPU is allowed to run), all of which can be set independently of other VCPUs. This scheduling interface has several benefits: (1) it can be used directly by the VMM scheduler to schedule the VCPUs of the domains; (2) it can be configured to support different resource interfaces from the CSA literature, such as MPR interfaces [27], deterministic MPR interfaces [47], and multi-supply function interfaces [15]; (3) it is compatible with different distributions of interface bandwidth to VCPU budgets, such as one that distributes budget equally among the VCPUs [27], or one that provides the maximum bandwidth (equal to 1) to all but one VCPU [47]; and finally (4) it enables the use of CPU-mask-aware scheduling strategies, such as one that dedicates a subsets of PCPUs to some VCPUs and schedules the rest of the VCPUs on the remaining PCPUs [47].

Global vs. partitioned schedulers. Different multicore schedulers require different implementation strategies and provide different performance benefits. A partitioned scheduler only schedules VCPUs in its own core's run queue and hence is simple to implement; in contrast, a global scheduler schedules all VCPUs in the system and thus is more complex but can provide better resource utilization. We support both by implementing two schedulers in RT-Xen 2.0: rt-global and rt-partition.[1] The rt-partition scheduler uses a partitioned queue scheme, whereas the rt-global scheduler uses a global shared run queue that is protected by a spinlock.[2] For each scheduler, users can switch between dynamic priority (EDF) and static priority (DM) schemes on the fly.

Server mechanisms. Each VCPU in RT-Xen 2.0 is associated with a period and a budget, and it is implemented as a server: the VCPU is released periodically and its budget is replenished at the beginning of every period. It consumes its budget when running, and it stops running when its budget is exhausted. Different server mechanisms provide different ways to schedule the VCPUs when the current highest priority VCPU is not runnable (i.e., has no jobs to execute) but still has unused budget. For instance, when implemented as a deferrable server, the current VCPU defers its unused budget to be used at a later time within its current period if it becomes runnable, and the highest-priority VCPU among the runnable VCPUs is scheduled to run. In contrast, when implemented as a periodic server, the current VCPU continues to run and consume its budget (as if it had a background task executing within it). As shown by our experimental results in Section 4.6, the fraction of schedulable task sets can be quite different when different servers are used, even if the scheduler is the same. We implemented the deferrable server scheme in both rt-global and rt-partition, and they can be configured as periodic servers by running a lowest priority CPU-intensive task in a guest VCPU.

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

[2] An alternative approach to approximating a global scheduling policy is to employ partitioned queues that can push/pull threads from each other [12] (as adopted in the Linux Kernel). We opt for a simple global queue design because the locking overhead for the shared global queue is typically small in practice, since each host usually runs only relatively few VMs.

3.2 Implementation

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

Run queue structure. Figure 2 shows the structure of the global run queue (RunQ) of the rt-global scheduler, which is shared by all physical cores. The RunQ holds all the runnable VCPUs and is protected by a global spin-lock. Within this queue, the VCPUs are divided into two sections: the first consists of the VCPUs with a nonzero remaining budget, and the second consists of VCPUs that have no remaining budget. Within each section, the VCPUs are sorted based on their priorities (determined by a chosen priority assignment scheme). We implemented both EDF and DM priority schemes in RT-Xen 2.0. The rt-partition scheduler maintains for each core a separate run queue, which has the same structure as the RunQ, except that it is not protected by a spin-lock.

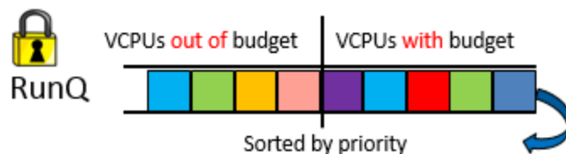


Figure 2: rt-global run queue structure.

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

Algorithm 1 shows the *do_schedule()* function of the rt-global scheduler under the EDF priority scheme (i.e., gEDF). In the first `for` loop (Lines 5–10), it replenishes the budgets of the VCPUs and rearranges the VCPUs in the RunQ appropriately. In the second `for` loop (Lines 11–18), it selects the highest-priority runnable VCPU (`snext`) that can be executed. Finally, it compares the deadline of the selected VCPU (`snext`) with that of the currently running VCPU (`scurr`), and then it returns the VCPU to be executed next (Lines 19–24).

:

Algorithm 1. *do_schedule() function for rt-global under EDF.*

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

end

There are two key differences between RT-Xen 2.0 and RT-Xen [45]: (1) the second `for loop` (Lines 11–18) guarantees that the scheduler is CPU-mask aware; and (2) if the scheduler decides to switch VCPUs (Lines 22–24), the currently running VCPU (scurr) is **not** inserted back into the run queue (otherwise, it could be grabbed by another physical core before its context is saved, since the run queue is shared among all cores, which would then make the VCPU's state inconsistent). For this reason, Xen adds another scheduler-dependent function named *context_saved()*, which is invoked at the end of a *context_switch()* to insert scurr back into the run queue if it is still runnable. Note that both *do_schedule()* and *context_saved()* need to grab the spin-lock before running; since this is done in the Xen scheduling framework, we do not show this in Algorithm 1.

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

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

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

4. EMPIRICAL EVALUATION

In this section, we present our experimental evaluation of RT-Xen 2.0. We have five objectives for our evaluation: (1) to evaluate the scheduling overhead of the rt-global and rt-partition schedulers compared to the default Xen credit scheduler; (2) to experimentally evaluate the schedulability of the system under different combinations of schedulers at the guest OS and VMM levels; (3) to evaluate the real-time system performance under RT-Xen 2.0 schedulers in overload situations; (4) to compare the performance of the deferrable server scheme and the periodic server scheme; and (5) to evaluate the impact of cache on global and partitioned schedulers.

4.1 Experiment Setup

We perform our experiments on an Intel i7 x980 machine, with six cores (PCPUs) running at 3.33 GHz. We disable hyper-threading and SpeedStep to ensure constant CPU speed, and we shut down all other non-essential processes during our experiments to minimize interference. The scheduling quantum for RT-Xen 2.0 is set to 1 ms. Xen 4.3 is patched with RT-Xen 2.0 and installed with a 64-bit Linux 3.9 Kernel as domain 0, and a 64-bit Ubuntu image with a para-virtualized Linux Kernel as the guest domain. For all experiments, we boot domain 0 with one VCPU and pin this VCPU to one core; the remaining five cores are used to run the guest domains. In addition, we patch the guest OS with LITMUS^{RT} [8] to support EDF scheduling.

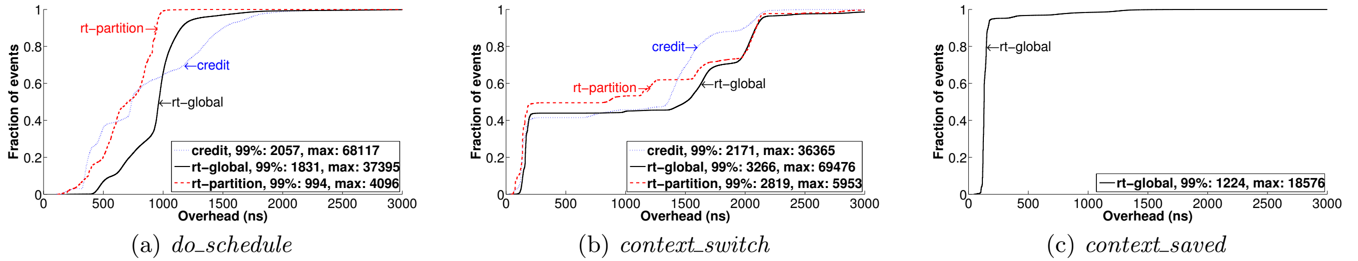
For the partitioned scheduler at the guest OS level and the VMM level, we use a variant of the best-fit bin-packing algorithm for assigning tasks to VCPUs and VCPUs to cores, respectively. Specifically, for each domain, we assign a task to the VCPU with the *largest* current bandwidth[3] among all existing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

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

4.2 Workloads

In our experiments, tasks are created based on the *base_task* provided by LITMUS^{RT}. To emulate a desirable execution time for a task in RT-Xen 2.0, we first calibrate a CPU-intensive job to take 1 ms in the guest OS (when running without any interference), then scale it to the desirable execution time. For each task set, we run each experiment for 60 seconds, and record the deadline miss ratio for each task using the *st_trace* tool provided by LITMUS^{RT}.

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



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

Figure 3: CDF plots for scheduling overhead for different schedulers over 30 seconds.

Following the methodology used in [13] to generate real-time tasks, our evaluation uses synthetic real-time task sets. The tasks' periods are chosen uniformly at random between 350ms and 850ms, and the tasks' deadlines are set equal to their periods. The tasks' utilizations follow the medium bimodal distribution, where the utilizations are distributed uniformly over [0.0001, 0.5) with probability of 2/3, or [0.5, 0.9] with probability of 1/3. Since there are five cores for running the guest domains, we generate task sets with total utilization ranging from 1.1 to 4.9, with a step of 0.2. For a specific utilization, we first generate tasks until we exceed the specified total task utilization, then we discard the last generated task and use a "pad" task to make the task set utilization match exactly the specified utilization. For each of the 20 task set utilizations, we use different random seeds to generate 25 task sets for each bimodal distribution. In total, there are 20 (utilization values) $\times$ 25 (random seeds) = 500 task sets in our experiments.

Each generated task is then distributed into four different domains in a round-robin fashion. We apply compositional scheduling analysis to compute the interface of each domain, and to map the computed interface into a set of VCPUs to be scheduled by the VMM scheduler. In our evaluation, we use harmonic periods for all VCPUs. We first evaluate the real-time schedulers using CPU-intensive tasks in the experiments, followed by a study on the impacts of cache on the different real-time schedulers using cache-intensive tasks with large memory footprints (Section 4.7).

4.3 Scheduling Overhead

In order to measure the overheads for different schedulers, we boot 4 domains, each with 4 VCPUs. We set each VCPU's bandwidth to 20%, and distribute the VCPUs to 5 PCPUs for the rt-partition scheduler in a round-robin fashion; for the rt-global and credit schedulers, we allow all guest VCPUs to run on all 5 PCPUs. We run a CPU-intensive workload with a total utilization of 3.10. We use the EDF scheme in both rt-global and rt-partition schedulers, as the different priority schemes only differ in their placement of a VCPU in the RunQ. In the Xen scheduling framework, there are three key functions related to schedulers as described in Section 3.2. We measure the overheads as the time spent in the *do_schedule* function as *scheduling latency*, the time spent in the *context_switch*, and the time spent in the *context_saved*. Note that *context_saved* is necessary only in rt-global schedulers, as they have shared queues. For rt-partition and credit schedulers, the running VCPU is inserted back to run queue in *do_schedule* function. To record these overheads, we modify *xentrace* [32] and use it to record data for 30 seconds.

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

First, as is shown in Figure 3a, the rt-global scheduler incurred a higher scheduling latency than the rt-partition scheduler. This is because the rt-global scheduler experienced the overhead to grab the spinlock, and it had a run queue that was 5 times longer than that of the rt-partition scheduler. The credit scheduler performed better than the rt-global scheduler in the lower 60%, but performed worse in the higher 40% of our measurements. We attribute this to the load balancing scheme in the credit scheduler, which must check all other PCPUs' RunQs to "steal" VCPUs.

Second, Figure 3b shows that the context switch overheads for all three schedulers were largely divided into two phases: approximately 50% of the overhead was around 200 nanoseconds, and the remaining was more than 1500 nanoseconds. We find that the first 50% (with lower overhead) ran without actually performing context switches, since Xen defers the actual context switch until necessary: when the scheduler switches from a guest VCPU to the IDLE VCPU, or from the IDLE VCPU to a guest VCPU with its context still intact, the time spent in the *context_switch* function is much shorter than a context switch between two different guest VCPUs. We can also observe that *context_switch* in rt-global has a higher overhead. We attribute this to the global scheduling policy, where the VMM scheduler moves VCPUs around all PCPUs, and would cause more preemptions than a partitioned scheduling policy like rt-partition.

Third, Figure 3c shows the time spent in the *context_saved* function for the rt-global scheduler. Recall that this function is NULL in the rt-partition and credit schedulers, since the current VCPU is already inserted back into the run queue by the *do_schedule* function. We observe that, for the rt-global scheduler, around 90% of the overhead was 200 nanoseconds or less, and the 99% value was only 1224 nanoseconds. We attribute this to the extra overhead of grabbing the spinlock to access the shared run queue in the rt-global scheduler.

Overall, in 99% of the cases, the overhead of all three functions (do_schedule, context_switch, and context_saved) for all schedulers was smaller than 4 microseconds. Since we use a 1 ms scheduling quantum for both the rt-global and the rt-partition schedulers, an overhead of 4 microseconds corresponds to a resource loss of only 1.2% per scheduling quantum. Notably, in contrast to an OS scheduler – which is expected to handle a large number of tasks – the VMM scheduler usually runs fewer than 100 VCPUs as each VCPU typically demands much more resources than a single task. As a result, the run queue is typically much shorter, and the overhead for grabbing the lock in a shared run queue is typically smaller than in an OS scheduler.

4.4 RT-Xen 2.0 vs. Credit Scheduler

We conduct experiments to compare the real-time performance of the default credit scheduler in Xen and our RT-Xen 2.0 schedulers. All guest domains run the pEDF scheduler in the guest OS. For the credit scheduler, we configure each domain's credit and cap as the sum of all its VCPU's bandwidths, as described in Section 4.1. For comparison, we also plot the results for the gEDF and gDM schedulers in RT-Xen 2.0 using a periodic server.

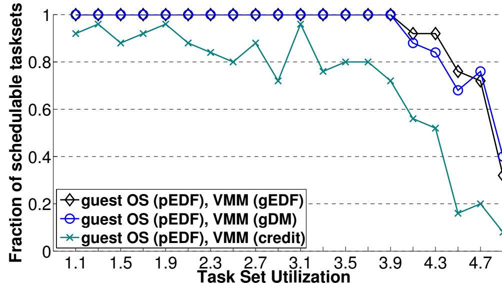


Figure 4: credit vs. RT-Xen 2.0 schedulers

Figure 4 shows the results. With the credit scheduler, even when the total workload utilization is as low as 1.1, 10 % of the task sets experienced deadline misses, which clearly demonstrates that the credit scheduler is not suitable for scheduling VMs that contain real-time applications. In contrast, our real-time VM schedulers based on the gEDF and gDM policies can meet the deadlines of all task sets at utilizations as high as 3.9. This result highlights the importance of incorporating real-time VM schedulers in multicore hypervisors such as Xen. In the following subsections, we compare different real-time VM scheduling policies in RT-Xen 2.0.

4.5 Real-Time Scheduling Policies

We now evaluate different real-time VM scheduling policies supported by RT-Xen 2.0. We first compare their capability to provide theoretical schedulability guarantees based on compositional scheduling analysis. We then experimentally evaluate their capability to meet the deadlines of real-time tasks in VMs on a real multi-core machine. This approach allows us to compare the theoretical guarantees and experimental performance of real-time VM scheduling, as well as the real-time performance of different combinations of real-time scheduling policies at the VMM and guest OS levels. In both theoretical and experimental evaluations, we used the medium-bimodal distribution, and we performed the experiments for all 25 task sets per utilization under the rt-global and rt-partition schedulers.

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

Performance of the four schedulers at the VMM level: Figures 5(a) and 5(b) show the fraction of schedulable task sets for the four schedulers at the VMM level with respect to the task set utilization when fixing pEDF or gEDF as the guest OS scheduler, respectively. The results show that, when we fix the guest OS scheduler, the pEDF and pDM schedulers at the VMM level can provide theoretical schedulability guarantees for more task sets than the gDM scheduler, which in turn outperforms the gEDF scheduler, for all utilizations. Note that the fraction of schedulable task sets of the pEDF scheduler is the same as that of the pDM scheduler. This is because the set of VCPUs to be scheduled by the VMM is the same for both pDM and pEDF schedulers (since we fixed the guest OS scheduler), and these VCPUs have harmonic periods; as a result, the utilization bounds under both schedulers are both equal to 1 [18]. The results also show that the partitioned schedulers usually outperformed the global schedulers in terms of theoretical schedulability.

Combination of EDF schedulers at both levels: Figure 5(c) shows the fraction of schedulable task sets for each task set utilization under four different combinations of the EDF priority assignment at the guest OS and the VMM levels. The results show a consistent order among the four combinations in terms of theoretical schedulability (from best to worst): (pEDF, pEDF), (gEDF, pEDF), (pEDF, gEDF), and (gEDF, gEDF).

Experimental evaluation on RT-Xen 2.0. From the above theoretical results, we observed that pEDF and gEDF have the best and the worst theoretical performance at both levels. Henceforth, we focus on EDF results in the experimental evaluation. We have not observed statistically distinguishable differences between DM and EDF scheduling in their empirical performance, and the DM results follows similar trends to EDF scheduling.

Experimental vs. theoretical results: Figures 6(a) and 6(b) show the fractions of schedulable task sets that were predicted by the CSA theory and that were observed on RT-Xen 2.0 for the two EDF schedulers at the VMM level, when fixing pEDF or gEDF as the guest OS scheduler, respectively. We examine all 25 task sets for each level of system, and we find that whenever a task set used in our evaluation is schedulable according to the theoretical analysis, it is also schedulable under the corresponding scheduler on RT-Xen 2.0 in our experiments. In addition, for both pEDF and gEDF schedulers, the fractions of schedulable task sets observed on RT-Xen 2.0 are always larger than or equal to those predicted by the theoretical analysis. The results also show that, in contrast to the trend predicted in theory, the gEDF scheduler at the VMM level can often schedule more task sets empirically than the pEDF scheduler. We attribute this to the pessimism of the gEDF schedulability analysis when applied to the VMM level, but gEDF is an effective real-time scheduling policy in practice due to its flexibility to migrate VMs among cores.

Combination of EDF schedulers at both levels: Figure 6(c) shows the fraction of empirically schedulable task sets at different levels of system utilization under four different combinations of EDF policies at the guest OS and VMM levels. The results show that, at the guest OS level, the pEDF scheduler always outperform the gEDF scheduler. Further, if we fix pEDF (gEDF) for the guest OS scheduler, the gEDF scheduler at the VMM level can often schedule more task sets than the pEDF scheduler.

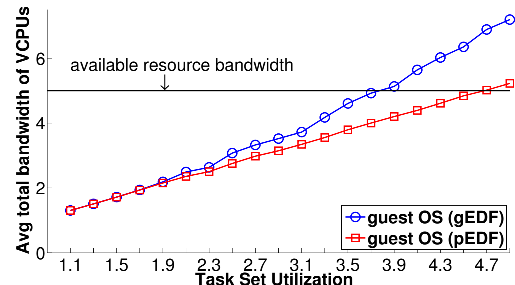
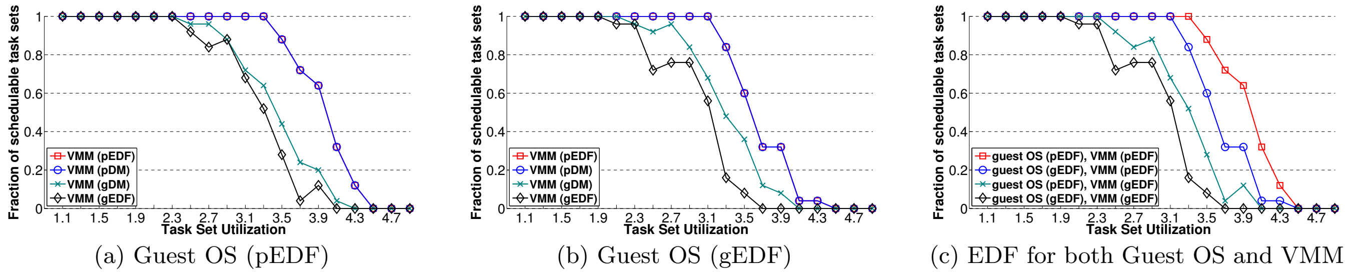


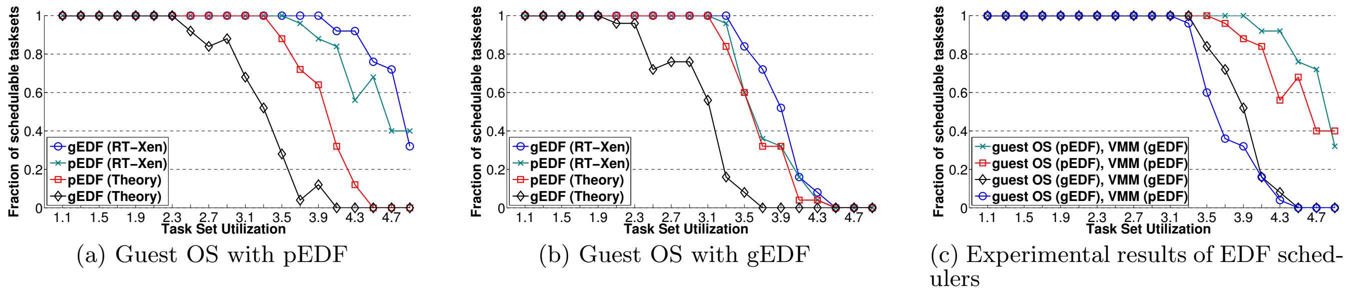
Figure 7: Average total VCPU bandwidth comparison.

To explain the relative performance of pEDF and gEDF in a two-level scheduling hierarchy, we investigate the corresponding set of VCPUs that are scheduled by the VMM when varying the guest OS scheduler. For the same task



(a) Guest OS (pEDF) (b) Guest OS (gEDF) (c) EDF for both Guest OS and VMM

Figure 5: Theoretical results: schedulability of different schedulers at the guest OS and the VMM levels.



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

Figure 6: Experimental vs. theoretical results: schedulability of different schedulers at the guest OS and the VMM levels.

set, the VCPUs of a domain under the pEDF and gEDF schedulers can be different; hence, the set of VCPUs to be scheduled by the VMM can also be different. Figure 7 shows the total bandwidth of all the VCPUs that are scheduled by the VMM – averaged across all 25 task sets – at each level of system utilization for the pEDF and gEDF schedulers at the guest-OS level. The horizontal line represents the total available resource bandwidth (with 5 cores).

The figure shows that gEDF as the guest OS scheduler results in a higher average total VCPU bandwidth compared to pEDF; therefore, the extra resource that the VMM allocates to the VCPUs (compared to that was actually required by their tasks) is much higher under gEDF. Since the resource that is unused by tasks of a higher priority VCPU cannot be used by tasks of lower-priority VCPUs when VCPUs are implemented as periodic servers, more resources were wasted under gEDF. In an overloaded situation, where the underlying platform cannot provide enough resources at the VMM level, the lower-priority VCPUs will likely miss deadlines. Therefore the poor performance of gEDF at the guest OS level results from the combination of pessimistic resource interfaces based on the CSA and the non-work-conserving nature of periodic server. We study deferrable server, a work-conserving mechanism for implementing VCPUs, in Section 4.6

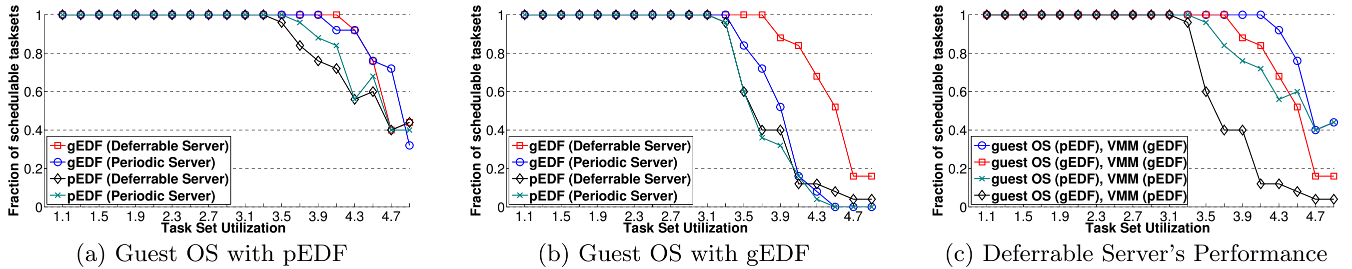
In contrast, when we fix the guest OS scheduler to be either pEDF or gEDF, the set of VCPUs that is scheduled by the VMM is also fixed. As a result, we observe more VCPUs being schedulable on RT-Xen 2.0 under the gEDF scheduler than under the pEDF scheduler at the VMM level (c.f., Figure 6(c)). This is consistent with our earlier observation, that the gEDF scheduler can often schedule more task sets than the pEDF scheduler empirically because of the flexibility to migrate VMs among cores.

4.6 Periodic Server vs. Deferrable Server

As observed in the last set of experiments, realizing VMs as periodic servers suffers from the non-work-conserving nature of the periodic server algorithm. Henceforth, we compare the real-time performance of the periodic server against the deferrable server (which implements VMs in a work-conserving fashion).Theoretically, the deferrable server scheme can suffer from back-to-back effects, in which higher-priority server executes back to back causing lower-priority servers to miss deadlines. While the back-to-back effect affects deferrable server's capability to provide theoretical schedulability guarantees, in practice back-to-back effect happens infrequently and its negative impacts are often dominated by the benefits of the work-conserving property of deferrable server, as shown in our experimental results. For this, we repeat the experiments in Section 4.5 with a deferrable server configuration.

Figure 8a and Figure 8b show the fraction of schedulable task sets with the periodic server and with the deferrable server, respectively. It can be observed that, when pEDF is used in the guest OS (Figure 8a), the two servers are incomparable in terms of the fraction of schedulable task sets. This is because there is little slack time in each VCPU's schedule (recall from Figure 7 that the total VCPU bandwidth for pEDF in the guest OS is close to the actual task set utilization) and thus a deferrable server behaves almost like a periodic server. In contrast, when the guest OS is using gEDF (Figure 8b), using gEDF in the VMM with a deferrable server clearly outperforms the other three combinations. We attribute this to the work-conserving behavior of the deferrable server, which can take advantage of the available slack time at runtime to improve the system schedulability.

Another interesting observation is that, when pEDF is used in RT-Xen 2.0, the difference between the performance of the two servers is not obvious. We attribute this to the VCPU parameters calculated based on CSA: The computed bandwidth of a VCPU is often larger than half of the available bandwidth of a PCPU. As a result, when a partitioned scheduler is used in RT-Xen 2.0, every PCPU is either able to feasibly schedule all tasks (if it only executes one VCPU) or is heavily overloaded (if it executes two or more VCPUs). In the former case, there is no deadline miss on the PCPU under either server; in the latter, using deferrable server can-



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

Figure 8: Fraction of schedulable task sets using deferrable server and periodic server.

not help improve the deadline miss ratio much, since there is often no slack available when the PCPU is heavily overloaded.

Finally, Figure 8c shows four configurations of the gEDF and pEDF scheduling policies with a deferrable server. We can observe that generally global scheduling in VMM outperforms partitioned scheduling empirically. Further, for the same VMM scheduler, using pEDF in the guest OS results in better performance compared to using gEDF.

4.7 Cache Intensive Workloads

Our previous experiments use CPU-intensive workloads with small memory footprints. In comparison, a memory-intensive workload may be more affected by VCPU migrations caused by a global VM scheduler because of cache penalty. To study the impacts of cache effects, we conduct a new empirical comparison between rt-global and rt-partition schedulers using a memory-intensive workload. The Intel i7 processor used in this set of experiments contains 6 cores, each core owns dedicated L1 (32KB data, 32KB instruction) and L2 (256KB unified) cache, while all cores share a unified 12MB L3 cache. The last-level cache is inclusive [5], which means the data that is in a PCPU's L2 cache must also be in the shared L3 cache. Therefore, the cache penalty of a VCPU migration is usually associated with latency difference between core-specific private cache (L1 or L2) and the shared L3 cache. On the i7 processor the latency difference between L2 and L3 cache is 18 cycles [6], about 5 nano-seconds per cache line (64B). The local L2 cache size is 256 KB (4096 cache lines), therefore, a VCPU migration may result in a cache penalty as high as 4096 x 5 ns = 20 μs. However, due to the widely used cache pre-fetch technology, the observed migration penalty is usually much less than the worst case. In comparison to a VMM scheduling quantum of 1 ms, we hypothesize the VCPU migration would not incur a significant performance penalty. [4]

To create significant cache penalty from VCPU migrations, we designed the memory access pattern of our tasks as follows. We allow each task to access a fixed sized array within the L2 cache. The access pattern is one element per cache line, and we store the next element's index in the current element, so that it is data dependent and the improvement from cache pre-fetch can be mitigated. Recent work in compositional scheduling theory also considers cache impact [47], but assumes there is no shared cache. Therefore, we keep the other parameters of the task sets the same as our previously generated workload. The impact of cache has received significant attention in the context of one level scheduling [40, 42, 50]; we defer integrating them into a two-level hierarchal scheduling to future work.

We use pEDF in the guest OS so that the cache penalty

[4]This analysis is valid only for processor with a shared last-level cache. For platforms with multiple last-level caches, global scheduler can have a higher cache-miss penalty, as shown in an earlier study at the OS level [13].

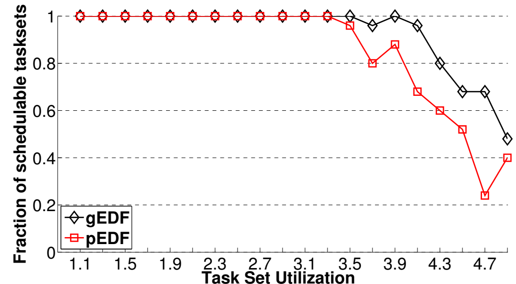


Figure 9: Cache-intensive workloads (guest OS with pEDF)

are attributed only to the VMM-level schedulers. We compare the real-time performance of the gEDF and pEDF VMM schedulers. As shown in Figure 9, gEDF again outperforms pEDF despite the cache penalty. This confirms the benefits of a global scheduling policy outweighs the cache penalty caused by the VCPU migration on a multicore platformed with shared last-level cache.

5. RELATED WORK

Since our RT-Xen 2.0 scheduling framework is shaped by both theoretical and practical considerations, we discuss related work in both the theory and the systems domain.

5.1 Theory perspective

There exists a large body of work on hierarchical scheduling for single-processor platforms (see e.g., [14, 25, 26, 43]), which considers both DM and EDF schemes. In RT-Xen 2.0, we use the method in [43] to derive the interfaces under partitioned DM and partitioned EDF schedulers.

Recently, a number of compositional scheduling techniques for multicore platforms have been developed (e.g., [15, 27, 31, 37, 39, 47]), which provide different resource models for representing the interfaces. For instance, an MPR interface [27] abstracts a component using a period, a budget within each period, and a maximum level of concurrency, whereas a multi-supply function interface [15] uses a set of supply bound functions. Our evaluations on RT-Xen 2.0 were performed using the MPR model for the domains' interfaces [27], but the RT-Xen 2.0 scheduling framework is compatible with most other interfaces, such as [15, 47] and their variations as discussed in Section 3. In the future RT-Xen 2.0 can serve as an open source platform for the community to experiment with different hierarchical schedulability analysis.

5.2 Systems perspective

The implementation of hierarchical scheduling has been investigated for various platforms, including (1) native platforms, (2) Xen, and (3) other virtualization platforms. We describe each category below:

Native platforms: There are several implementations

of hierarchical scheduling within middleware or OS kernels [10, 17, 24, 38, 41, 44]. In these implementations, all levels of the scheduling hierarchy are implemented in one (user or kernel) space. In contrast, RT-Xen 2.0 implements schedulers in the VMM level, and leverages existing real-time schedulers in the guest OS, thereby achieving a clean separation between the two levels of scheduling. Furthermore, leveraging compositional schedulability analysis, RT-Xen 2.0 also enables guest domains to hide their task-level details from the underlying platform, since it only requires a minimal scheduling interface abstraction from the domains.

Other Xen approaches: Several recent efforts have added real-time capabilities to Xen. For example, Lee et al. [36] and Zhang et al. [51] provides a strict-prioritization patch for the credit scheduler so that real-time domains can always get resource before non-real-time domains, and Govindan et al. [29] mitigates priority inversion when guest domains are co-scheduled with domain 0 on a single core. Yoo et al. [49] used similar ideas to improve the credit scheduler on the Xen ARM platform. While these work can help real-time applications, they employ heuristics to enhance the existing schedulers and is not supported by analytical guarantees. In our earlier work RT-Xen 1.0 [35, 45], we focused on *single-core* scheduling with fixed priority. RT-Xen 2.0 provides a new *multicore* real-time scheduling framework.

Other virtualization approaches: There are other virtualization technologies that rely on architectures other than Xen. For instance, KVM [7] integrates the VMM with the host OS, and schedules VCPUs together with other tasks in the host OS. Hence, in principle, any real-time multicore Linux scheduler [13, 16, 19, 22, 30] could be configured to apply two-level hierarchical scheduling in KVM, but with limited server mechanism support. As an example, Checconi et al. [20] implemented a partitioned-queue EDF scheduler for scheduling multicore KVM virtual machines, using hard constant-bandwidth servers for the VCPUs and global fixed-priority scheduling for the guest OS scheduler. The same group also investigated the scheduling of real-time workloads in virtualized cloud infrastructures [23]. Besides KVM, the micro-kernel like L4/Fiasco [3] can also be used to achieve hierarchical scheduling, as demonstrated by Yang et al. [48] using a periodic server implementation. Lackorzynski et al. [34] exported the task information to the host scheduler in order to address the challenge of mixed-criticality systems in a virtualized environment. Härtig et al. [33] studied a multi-resource multi-level problem to optimize energy consumption. Crespo et al. [21] proposed a bare-metal VMM based on a para-virtualization technology similar to Xen for embedded platforms, which uses static (cyclic) scheduling. In contrast, RT-Xen 2.0 provides a scheduling framework spanning the design space in terms of global and partitioned scheduling, dynamic and static priority, periodic and deferrable servers. We also provide an comprehensive experimental study of different combinations of scheduling designs.

6. CONCLUSIONS

We have designed and implemented RT-Xen 2.0, a new real-time multicore VM scheduling framework in the Xen virtual machine monitor (VMM). RT-Xen 2.0 realizes global and partitioned VM schedulers, and each scheduler can be configured to support dynamic or static priorities, and to run VMs as periodic or deferrable servers. Through a comprehensive experimental study, we show that both global and partitioned VM scheduling can be implemented in the VMM at moderate overhead. Moreover, at the VMM scheduler level, in compositional schedulability theory pEDF is better than gEDF in schedulability guarantees, but in our experiments their actual performance is reversed in terms of the fraction of workloads that meet their deadlines on virtualized multicore platforms. At the guest OS level, pEDF requests a smaller total VCPU bandwidth than gEDF based on compositional schedulability analysis, and therefore using pEDF in the guest OS level leads to more schedulable workloads on a virtualized multicore processor. The combination of pEDF in guest OS and gEDF in the VMM therefore resulted the best experimental real-time performance. Finally, on a platform with a shared last-level cache, the benefits of global scheduling outweigh the cache penalty incurred by VM migration.

Acknowledgments

This research was supported in part by ONR (N000141310800 and N000141310802), ARO (W911NF-11-1-0403), NSF (CNS-1117185, CNS-1329984, and CNS-1329861), and Global Research Laboratory Program through NRF of Korea funded by the Ministry of Science, ICT & Future Planning (2013K1A1A2A02078326).

7. REFERENCES

[1] COQOS Operating System. http://www.opensynergy.com/en/Products/COQOS/.
[2] Credit Scheduler. http://wiki.xen.org/wiki/Credit_Scheduler.
[3] Fiasco micro-kernel. http://os.inf.tu-dresden.de/fiasco/.
[4] INTEGRITY Multivisor. http://www.ghs.com/products/rtos/integrity_virtualization.html.
[5] Intel 64 and IA-32 Architectures Optimization Reference Manual. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.
[6] Intel Ivy Bridge 7-Zip LZMA Benchmark Results. http://www.7-cpu.com/cpu/IvyBridge.html.
[7] Kernel Based Virtual Machine. http://www.linux-kvm.org.
[8] LITMUS-RT. http://www.litmus-rt.org/.
[9] The Xen Project's Hypervisor for the ARM architecture. http://www.xenproject.org/developers/teams/arm-hypervisor.html.
[10] T. Aswathanarayana, D. Niehaus, V. Subramonian, and C. Gill. Design and Performance of Configurable Endsystem Scheduling Mechanisms. In *RTAS*, 2005.
[11] 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. *ACM SIGOPS Operating Systems Review*, 2003.
[12] S. Barush and B. Brandenburg. Multiprocessor feasibility analysis of recurrent task systems with specified processor affinities. In *RTSS*, 2013.
[13] A. Bastoni, B. B. Brandenburg, and J. H. Anderson. An Empirical Comparison of Global, and Clustered Multiprocessor EDF Schedulers. In *RTSS*. IEEE, 2010.
[14] 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.
[15] E. Bini, G. Buttazzo, and M. Bertogna. The Multi Supply Function Abstraction for Multiprocessors. In *RTCSA*. IEEE, 2009.

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

[17] 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.

[18] G. C. Buttazzo. Rate monotonic vs. edf: judgment day. *Real-Time Syst.*, 29(1):5–26, Jan. 2005.

[19] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson. LITMUS-RT: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In *RTSS*, 2006.

[20] F. Checconi, T. Cucinotta, D. Faggioli, and G. Lipari. Hierarchical Multiprocessor CPU Reservations for the Linux Kernel. *OSPERT*, 2009.

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

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

[23] T. Cucinotta, F. Checconi, G. Kousiouris, D. Kyriazis, T. Varvarigou, A. Mazzetti, Z. Zlatev, J. Papay, M. Boniface, S. Berger, et al. Virtualised e-learning with real-time guarantees on the irmos platform. In *Service-Oriented Computing and Applications (SOCA)*. IEEE, 2010.

[24] M. Danish, Y. Li, and R. West. Virtual-CPU Scheduling in the Quest Operating System. In *RTAS*, 2011.

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

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

[27] A. Easwaran, I. Shin, and I. Lee. Optimal virtual cluster-based multiprocessor scheduling. *Real-Time Syst.*, 43(1):25–59, Sept. 2009.

[28] A. Easwaran, I. Shin, and I. Lee. Optimal Virtual Cluster-Based Multiprocessor Scheduling. *Real-Time Systems*, 2009.

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

[30] G. Gracioli, A. A. Fröhlich, R. Pellizzoni, and S. Fischmeister. Implementation and evaluation of global and partitioned scheduling in a real-time os. *Real-Time Systems*, 2013.

[31] S. Groesbrink, L. Almeida, M. de Sousa, and S. M. Petters. Towards certifiable adaptive reservations for hypervisor-based virtualization. In *RTAS*, 2014.

[32] D. Gupta, R. Gardner, and L. Cherkasova. Xenmon: Qos Monitoring and Performance Profiling Tool. *Hewlett-Packard Labs, Tech. Rep. HPL-2005-187*, 2005.

[33] H. Härtig, M. Völp, and M. Hähnel. The case for practical multi-resource and multi-level scheduling based on energy/utility. In *RTCSA*, 2013.

[34] A. Lackorzyński, A. Warg, M. Völp, and H. Härtig. Flattening hierarchical scheduling. In *EMSOFT*. ACM, 2012.

[35] J. Lee, S. Xi, S. Chen, L. T. X. Phan, C. Gill, I. Lee, C. Lu, and O. Sokolsky. Realizing Compositional Scheduling Through Virtualization. In *RTAS*. IEEE, 2012.

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

[37] H. Leontyev and J. H. Anderson. A Hierarchical Multiprocessor Bandwidth Reservation Scheme with Timing Guarantees. *Real-Time Systems*, 2009.

[38] B. Lin and P. A. Dinda. Vsched: Mixing batch and interactive virtual machines using periodic real-time scheduling. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, 2005.

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

[40] W. Lunniss, S. Altmeyer, C. Maiza, and R. I. Davis. Integrating Cache Related Pre-emption Delay Analysis into EDF Scheduling. In *RTAS*, 2013.

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

[42] S. Altmeyer, R.I. Davis, and C. Maiza. Improved Cache Related Pre-emption Delay Aware Response Time Analysis for Fixed Priority Pre-emptive Systems. *Real-Time Systems*, 2012.

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

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

[45] S. Xi, J. Wilson, C. Lu, and C. Gill. RT-Xen: Towards Real-Time Hypervisor Scheduling in Xen. In *EMSOFT*. ACM, 2011.

[46] M. Xu, L. T. X. Phan, I. Lee, and O. Sokolsky. Compositional analysis under global deadline monotonic. Technical report, University of Pennsylvania, 2013.

[47] M. Xu, L. T. X. Phan, I. Lee, O. Sokolsky, S. Xi, C. Lu, and C. D. Gill. Cache-Aware Compositional Analysis of Real-Time Multicore Virtualization Platforms. In *RTSS*, 2013.

[48] J. Yang, H. Kim, S. Park, C. Hong, and I. Shin. Implementation of compositional scheduling framework on virtualization. *SIGBED Rev*, 2011.

[49] S. Yoo, K.-H. Kwak, J.-H. Jo, and C. Yoo. Toward under-millisecond I/O latency in Xen-ARM. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, page 14. ACM, 2011.

[50] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. MemGuard: Memory Bandwidth Reservation System for Efficient Performance Isolation in Multi-core Platforms. In *RTAS*, 2013.

[51] W. Zhang, S. Rajasekaran, T. Wood, and M. Zhu. MIMP: Deadline and Interference Aware Scheduling of Hadoop Virtual Machines. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2014.