Hypervisor Performance Analysis for Real-Time Workloads

Geoffrey Phi C. Tran∗†, Yu-An Chen∗†, Dong-In Kang*, John Paul Walters*, and Stephen P. Crago∗†

∗Information Sciences Institute
University of Southern California, Arlington, VA 22203
Email: {gttran, dkang, jwalters, crago}@isi.edu

†Department of Electrical Engineering
University of Southern California, Los Angeles, CA 90089
Email: {geoffret, chen116}@usc.edu

Abstract—Virtualization has become a key technology used in modern data centers. What began as a tool for server consolidation and energy efficiency has grown into an enabler for cloud computing. Cloud computing has become an accepted best practice for data centers. Virtualization is also becoming a key component of embedded and real-time systems in automotive systems, game consoles, and industrial settings. Many applications that run on embedded systems are bound to hard real-time requirements, meaning that deadlines must be met. While hypervisor performance for non-real-time workloads has been well-documented, comparison of performance across hypervisors for real-time workloads has not been systematically studied.

In this paper, we fill that gap by characterizing Xen (Credit and Real-Time-Deferrable-Server schedulers) and Wind River’s low-latency KVM for hard real-time workloads in one and two virtual machine (VM) cases. We compare each hypervisor to a non-virtualized base system and evaluate the relative merits of each hypervisor for a number of synthetic workloads with a number of varying characteristics.

When using a single VM, and therefore no resource contention, we find that all configurations are capable of completing over 99.8% of their jobs within their deadline. This demonstrates that a virtualized environment can support some real-time applications, such as those with relaxed constraints. We also find that Wind River’s low-latency KVM and Xen-RTDS scheduler are capable of supporting hard real-time constraints over a variety of task set characteristics. However, a use case with multiple VMs creates a need for resource sharing. The sharing of resources results in contention between guests for usage of the said resources. When two VMs are present, the Xen-RTDS and tuned Xen-credit scheduler are superior to the other configurations. Furthermore, we find that the default Xen-credit scheduler provides poor support for hard real-time constraints, regardless of whether resource contention is present. However, by tuning the credit scheduler’s timeslice parameter to benefit latency-sensitive tasks, we observe that the performance of the credit scheduler is indeed able to support hard real-time constraints for the given workloads in both single VM and two VM cases. These developments in real-time virtualization prove to be an exciting step towards real-time enabled clouds.

Index Terms—Virtualization; KVM; Xen; Real-Time; Schedulability; Cloud; Dynamic

I. INTRODUCTION

Many computing scenarios have real-time requirements. Today, everything from game consoles to high-frequency trading platforms to vehicle infotainment systems use virtualization to provide security and isolation while maintaining performance requirements [1][2][3]. Virtualization has been used on a full range of processors, from server and desktop class to embedded class processors. While embedded platforms have limited performance due to constraints such as power, networking limitations, and reliability, the capabilities are improving. Indeed, embedded processors are becoming faster and more capable, while full processors are also becoming more energy efficient [4][5]. For instance, ARMv7, ARMv8, and some Intel Atom processors support virtualization [6].

In systems of embedded processors, virtualization can enable resource sharing and flexibility while still maintaining isolation and security. Virtualization is enabled through hypervisors running either directly on the hardware, or in a host OS. While the performance of traditional hypervisors has been well-studied for general workloads, performance when processing real-time workloads has not been systematically studied. We address that gap by characterizing a number of hypervisor configurations, including Xen and Wind River’s low-latency KVM, and comparing to a base bare-metal system.

This paper makes the following contributions:

1) We compare multiple hypervisor configurations and evaluate each in its ability to meet hard real-time requirements, a key requirement for a real-time cloud. We find that the Xen RTDS outperforms the other hypervisor default configurations, especially when resource contention is present.

2) We find that without resource contention, all three hypervisor configurations are able to schedule a high fraction of the periodic jobs in the task sets. The ability to schedule jobs such that few deadlines are missed is well-suited to soft real-time workloads.

3) We demonstrate that, without careful configuration, Xen-credit is not suitable for hard real-time workloads, despite having similar overheads compared to Xen-RTDS.

Section II presents related and background work. Section III and Section IV present the setup and methodology for this work. Section V discusses the results. Finally, Section VI concludes the paper.

II. BACKGROUND AND RELATED WORK

A. Hypervisors

Wind River KVM is a low-latency hypervisor implementation provided by Wind River. WR-KVM claims to achieve near-native performance compared to the mainstream KVM and other leading hypervisors [7]. It is based on the open-source KVM hypervisor. KVM runs inside a host operating
system, versus the Xen hypervisor, which runs directly on the hardware.

Xen is an open-source hypervisor [8]. A privileged domain is used to manage other guest domains (VMs). The default scheduler in Xen is the credit scheduler. The credit scheduler provides a proportional fair-share of CPU time to each guest based on a weight and cap set by the user.

RT-Xen enables real-time support in the Xen hypervisor by providing a Real-Time-Deferrable-Server (RTDS) scheduler for allocating CPU resources [9]. The RTDS scheduler is a global EDF (Earliest-Deadline-First) scheduler. The VCPU allocations are parameterized as budgets and periods. Each VCPU is guaranteed to receive the assigned budget every period. A version of the RTDS scheduler has been upstreamed into the mainline Xen branch.

B. Real-Time Linux

LITMUS\textsuperscript{RT} is an extension of the Linux kernel that provides a real-time scheduler that runs on single or multi-core systems [10]. It provides a pluggable scheduling interface to implement real-time scheduling policies, including a number of provided default plugins. The global EDF variant, GSN-EDF (Global EDF with synchronization support) is based on the EDF scheduling algorithm. GSN-EDF differs from EDF by providing support for non-preemptible critical sections.

The traditional approach to support real-time workloads, PREEMPT\textsuperscript{RT} [11], focuses primarily on improving scheduling latency by reducing the time spent in the kernel’s non-preemptible sections. Many of the improvements from the PREEMPT\textsuperscript{RT} patch have been merged into the mainline kernel.

While both approaches aim to improve support of real-time processing in the Linux kernel, they each take different approaches. PREEMPT\textsuperscript{RT} improves scheduling latency while LITMUS\textsuperscript{RT} supports real-time processing through task scheduling [12]. Furthermore, the interface to the patches differ in many ways. PREEMPT\textsuperscript{RT} uses the Linux scheduling API. LITMUS\textsuperscript{RT} provides its own API to declare real-time tasks. While both patches provide a way to declare and implement periodic tasks, in our work we use LITMUS\textsuperscript{RT} due to it’s tracing mechanisms.

IV. Methodology

In order to conduct a comprehensive survey of the real-time performance on the various configurations, we utilize a number of workloads with different runtime characteristics. First, we present the definition of tasks and describe our synthetic benchmark. Then, we present the task sets that were executed. We then detail the parameters given to the various hypervisors, if any.

A. Task and Benchmark Definition

We utilize the same definition of tasks presented in [14]: “...real-time applications can be modeled as a collection of real-time tasks, and each real-time task is a sequence of jobs that are released periodically... All jobs are periodic, where each job $T_i$ is defined by a period (and deadline) $p_i$ and a worst-case execution time $e_i$, with $p_i \geq e_i \geq 0$ and $p_i, e_i \in$ integers.”

The benchmark used is a computationally intensive application, as proposed in [15]. This benchmark is comprised of a number of iterations of floating-point operations during each job. This is based on the base_task.c provided with liblitmus, the LITMUS\textsuperscript{RT} userspace library.

The average execution time of a job is determined by the amount of computation that is executed. For a requested execution time, this amount will vary depending on the hypervisor configuration. Therefore, we calibrate the amount of computation to achieve the required execution times. Because the actual execution time varies due to a number of factors such as interrupts, system overhead, or other interferences, we have a range of execution times for a given computation amount. Instead of using computation that provides the worst-case execution time, we use a statistically bounded amount of computation. Our reasoning is that our workloads are a function of utilization. Total utilization is more closely related to average execution time, not worst case. Therefore, we choose the size of the computation by running each job 800 times. The execution times are sorted, and we discard the outliers, defined as those greater than 75th quartile $+ 1.5 \times IQR$. We utilize three different hypervisor configurations in addition to the non-virtualized (bare-metal) OS: low-latency KVM from Wind River (WR-KVM), Xen 4.5 using the credit scheduler (Xen-credit), and Xen 4.5 using the RTDS-based scheduler (Xen-RTDS). As described in Section II-A, the RTDS scheduler is an upstreamed version of the work done by the RT-Xen project. The host OS is Wind River Linux 5.0.1.10 for WR-KVM and Ubuntu 12.04.4 for Xen-Credit and Xen-RTDS. The guest OS for each configuration is Ubuntu 12.04 running the GNU/Linux kernel 3.10.41 x86_64. Each guest OS is allocated 16 GB of memory. In order to eliminate NUMA effects, we also restrict our experiments to a single NUMA node. Furthermore, each guest kernel is patched with LITMUS\textsuperscript{RT} 2014.2. The bare-metal baseline configuration uses Ubuntu 12.04.4 patched with LITMUS\textsuperscript{RT} 2014.2. We used LITMUS\textsuperscript{RT}’s global earliest deadline first with synchronization support (GSN-EDF) scheduler.

The average execution time of a job is determined by the amount of computation that is executed. For a requested execution time, this amount will vary depending on the hypervisor configuration. Therefore, we calibrate the amount of computation to achieve the required execution times. Because the actual execution time varies due to a number of factors such as interrupts, system overhead, or other interferences, we have a range of execution times for a given computation amount. Instead of using computation that provides the worst-case execution time, we use a statistically bounded amount of computation. Our reasoning is that our workloads are a function of utilization. Total utilization is more closely related to average execution time, not worst case. Therefore, we choose the size of the computation by running each job 800 times. The execution times are sorted, and we discard the outliers, defined as those greater than 75th quartile $+ 1.5 \times IQR$. We utilize three different hypervisor configurations in addition to the non-virtualized (bare-metal) OS: low-latency KVM from Wind River (WR-KVM), Xen 4.5 using the credit scheduler (Xen-credit), and Xen 4.5 using the RTDS-based scheduler (Xen-RTDS). As described in Section II-A, the RTDS scheduler is an upstreamed version of the work done by the RT-Xen project. The host OS is Wind River Linux 5.0.1.10 for WR-KVM and Ubuntu 12.04.4 for Xen-Credit and Xen-RTDS. The guest OS for each configuration is Ubuntu 12.04 running the GNU/Linux kernel 3.10.41 x86_64. Each guest OS is allocated 16 GB of memory. In order to eliminate NUMA effects, we also restrict our experiments to a single NUMA node. Furthermore, each guest kernel is patched with LITMUS\textsuperscript{RT} 2014.2. The bare-metal baseline configuration uses Ubuntu 12.04.4 patched with LITMUS\textsuperscript{RT} 2014.2. We used LITMUS\textsuperscript{RT}’s global earliest deadline first with synchronization support (GSN-EDF) scheduler.

IV. Methodology

In order to conduct a comprehensive survey of the real-time performance on the various configurations, we utilize a number of workloads with different runtime characteristics. First, we present the definition of tasks and describe our synthetic benchmark. Then, we present the task sets that were executed. We then detail the parameters given to the various hypervisors, if any.

A. Task and Benchmark Definition

We utilize the same definition of tasks presented in [14]: “…real-time applications can be modeled as a collection of real-time tasks, and each real-time task is a sequence of jobs that are released periodically... All jobs are periodic, where each job $T_i$ is defined by a period (and deadline) $p_i$ and a worst-case execution time $e_i$, with $p_i \geq e_i \geq 0$ and $p_i, e_i \in$ integers.”

The benchmark used is a computationally intensive application, as proposed in [15]. This benchmark is comprised of a number of iterations of floating-point operations during each job. This is based on the base_task.c provided with liblitmus, the LITMUS\textsuperscript{RT} userspace library.

The average execution time of a job is determined by the amount of computation that is executed. For a requested execution time, this amount will vary depending on the hypervisor configuration. Therefore, we calibrate the amount of computation to achieve the required execution times. Because the actual execution time varies due to a number of factors such as interrupts, system overhead, or other interferences, we have a range of execution times for a given computation amount. Instead of using computation that provides the worst-case execution time, we use a statistically bounded amount of computation. Our reasoning is that our workloads are a function of utilization. Total utilization is more closely related to average execution time, not worst case. Therefore, we choose the size of the computation by running each job 800 times. The execution times are sorted, and we discard the outliers, defined as those greater than 75th quartile $+ 1.5 \times IQR$. We utilize three different hypervisor configurations in addition to the non-virtualized (bare-metal) OS: low-latency KVM from Wind River (WR-KVM), Xen 4.5 using the credit scheduler (Xen-credit), and Xen 4.5 using the RTDS-based scheduler (Xen-RTDS). As described in Section II-A, the RTDS scheduler is an upstreamed version of the work done by the RT-Xen project. The host OS is Wind River Linux 5.0.1.10 for WR-KVM and Ubuntu 12.04.4 for Xen-Credit and Xen-RTDS. The guest OS for each configuration is Ubuntu 12.04 running the GNU/Linux kernel 3.10.41 x86_64. Each guest OS is allocated 16 GB of memory. In order to eliminate NUMA effects, we also restrict our experiments to a single NUMA node. Furthermore, each guest kernel is patched with LITMUS\textsuperscript{RT} 2014.2. The bare-metal baseline configuration uses Ubuntu 12.04.4 patched with LITMUS\textsuperscript{RT} 2014.2. We used LITMUS\textsuperscript{RT}’s global earliest deadline first with synchronization support (GSN-EDF) scheduler.
Fig. 1: Fraction of Schedulable Task Sets Across Permutations of Task Set Characteristics for One Virtual Machine
(Moderate period [10 ms, 100 ms])

IQR is the interquartile range, defined as the 75th quartile - 25th quartile. The computation amount is increased until the maximum execution time, after discarding outliers, exceeds the required time. Discarding the outliers allows an average actual utilization nearer to the requested utilization.

B. Workloads

In our experiments, we use synthetic task sets, each composed of a number of tasks with varying characteristics. For instance, video encoding applications may have various periods depending on the frame rate, such as 120, 60, or 30 frames per second. Each task set has a given total utilization, task period distribution, and task utilization distribution. Total utilization is defined as the sum of the task utilization of each task in the task set. The utilization of a single task $i$ is defined as the execution time divided by the period: $e_i/p_i$. This is given in units of processor utilization, with a full core being one. For example, the system described in Section III has a maximum utilization of eight. However, for our experiments we vary the total utilization for each task set from 0.2 to 8.4 in steps of 0.2. The reason that the total utilization exceeds eight is because of the calibration procedure. The calibration procedure attempts to make the benchmark meet the execution time as closely as possible, while not exceeding the required execution times. Because of the variations in execution speed, the actual utilization at a given point may be less than the task definition and therefore less than the total utilization defined. However, as this is constant across platforms, we believe this to be a valid comparison.

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>light</td>
<td>uniform (0.001, 0.1)</td>
</tr>
<tr>
<td>medium</td>
<td>uniform (0.1, 0.4)</td>
</tr>
<tr>
<td>heavy</td>
<td>uniform (0.5, 0.9)</td>
</tr>
</tbody>
</table>
| bimodal      | 0.66: uniform (0.001, 0.5)  
                | 0.33: uniform (0.5, 0.9) |

We randomly generate the task sets that were used on each configuration. For example, a task set could have a utilization of 4.0, with the task periods uniformly distributed between (10ms, 100ms), and a task utilization uniformly distributed between (0.5, 0.9). The task sets are generated as follows: first, a period is randomly selected from the period distribution. A task utilization is then randomly selected and the execution time is determined. This task is then added to the set. If the last task added to the set exceeds the total utilization, the task is “squeezed” to make the given task set utilization exact [16]. “Squeezing” means the execution time, and therefore the utilization, of the last task to be added to the task set is reduced.

Table I summarizes the task utilization distributions. For each distribution, 25 different sets are generated. Each task set uses a moderate period distribution that is uniformly
distributed between (10ms, 100ms). The actual task that is executed is composed of computationally heavy jobs, as described in Section IV-A.

C. Hypervisor Configuration

For each hypervisor configuration, we allocate the full physical CPUs to the guest VCPUs. For the Xen-RTDS scheduler, we configure the guest VCPUs to allocate the full VCPU. This means the period and budget are both configured to 10 ms. If a VCPU runs out of budget, then it is not able to run until the next period [17]. For the Xen-credit scheduler, the weight and cap are also set such that the full physical CPUs are allowed to be used. In this case, they are set to 2048 and 800. Two cases of the timeslice parameter are explored: the default 30 ms and 1 ms. The weight represents a proportional share that a VM will receive, relative to the other guests present. The cap represents the maximum CPU resource that a guest may use, with 100 representing one CPU. The timeslice parameter is the unit size of CPU resource that is allocated to each VM. Smaller timeslices result in the scheduler potentially switching the running guests more often. For both Xen configurations, Dom0 is restricted to the second NUMA node. This eliminates the contention that may be caused by Dom0 competing with the VM for resources. For the WR-KVM configuration, we again allow the full VCPUs to be used. This is done by utilizing control groups to restrict the execution of the VM. The VM is only allowed to run on a single NUMA node, and allowed to use the full capacity of the CPUs it is executing on.

Fig. 2: Fraction of Schedulable Jobs Across Permutations of Task Set Characteristics (Medium/Light)

V. RESULTS AND DISCUSSION

For each task set, as defined in Section IV-B, each task set runs for 10 seconds. During this execution, each task set’s execution is monitored by using the ft_tools package. These are a set of user-space applications for interfacing with the feather-trace mechanism provided by LITMUSRT [18][19][20]. The traced data is then processed to provide information on each individual job according to the metric that is of interest, such as whether all deadlines were met.

We utilize two metrics to compare between the configurations. The first is the Fraction of Schedulable Task Sets, which is the number of task sets where none of the tasks have jobs that miss their deadline divided by the total number of task sets executed. This is useful for evaluating the hard real-time performance of the hypervisor configurations, because a single deadline miss is not acceptable for these workloads. The second is the Fraction of Schedulable Jobs, which is the number of jobs that do not miss their deadline divided by the total number of jobs. This can be used to measure the performance for soft real-time tasks, where deadlines may be missed.

In Figure 1, we illustrate the fraction of schedulable task sets. A fraction of 1.0 shows that all task sets are able to be scheduled successfully. The four subplots each illustrate a different task utilization distribution. The x-axis is the total task set utilization, while the y-axis is the fraction of schedulable task sets.

Overall, we find that the bare-metal configuration outperforms the virtualized configurations in all cases because the schedulability fraction for the bare-metal machine is greater
than or equal to the fraction for the virtualized cases at all utilization points. This applies to all four task utilization distributions. This is expected, as bare-metal does not have a virtualization overhead. Xen-RTDS and WR-KVM exhibit near-native schedulability across all distributions, with slight variations. Xen-credit(ts=1ms) also shows similar behavior to Xen-RTDS and WR-KVM for the bimodal, medium, and light workloads. This is in contrast to Xen-credit(ts=30ms), where the fraction is widely varying. This variability suggests that Xen-credit(ts=30ms) is not suitable for real-time workloads. Xen-credit(ts=1ms), when running heavy workloads, performs similarly to Xen-RTDS and WR-KVM. In Figure 1b, it can be seen that at utilizations 2.8 and 4.2, the fraction of schedulable tasks drops below 1. This suggests that Xen-credit(ts=1ms) is not suitable for some hard real-time workloads. This could be expected, as using a smaller timeslice benefits latency sensitive workloads while increasing the context switching overhead, which reduces performance of high throughput tasks. However, the drops in the ratio may also be due to scheduling jitter.

When comparing between the four subplots, it can be seen that the task sets with heavier task utilization distributions perform poorly compared to those task sets with lighter task utilization distributions. This can be seen when comparing Figure 1b to Figure 1c or Figure 1d. We observe that the heavy task utilization distribution’s fraction starts falling off at a utilization of around 5.4, while the medium and light task utilization distributions both fall off near a total utilization of eight. The bimodal distribution falls off near a utilization of 5.8, which is expected as the distribution is in between the heavy and medium distributions.

![Figure 3: Task Set Schedulability for Two Period Distributions (Xen-credit)](image)

Fig. 3: Task Set Schedulability for Two Period Distributions (Xen-credit)

Furthermore, it is interesting to note the way in which the schedulability curves drop off. While the heavier distributions fall off slowly, the medium and light distributions drop off with a very steep slope in the curve. This can be explained as follows: large task utilizations have fewer jobs which are harder to pack together. Conversely, small utilization jobs are more numerous, but easier to pack. This continues until the total utilization is too high to be scheduled on the resources.

The metric of “fraction of schedulable task sets” is quite strict and does not give a complete picture. Therefore, we investigate the observed miss rates in order to determine by how much a task set is unschedulable. The results for the medium and light distributions are illustrated in Figure 2. The y-axis is the fraction of schedulable jobs. Here, we see that although Xen-credit(ts=30ms) demonstrates high variability in its task set schedulability, its job schedulability is actually very high, with more than 99.9% of jobs meeting deadlines. By comparing Figure 2 to Figure 1, it can be inferred that many task sets are not schedulable by small margins, often only a single job. The bimodal and heavy task sets show similar trends, but are omitted for brevity.

While the job schedulability is very high for all cases, the actual task set schedulability for Xen-credit varies drastically, depending on the timeslice parameter. Therefore, we investigate the effects of the timeslice parameter on the scheduling performance. There are three characteristics that may affect the schedulability: scheduling overhead (time to make scheduling decisions), context switching overhead (time to switch between VMs), and scheduler granularity (unit size of resources allocated). We note that the overall Xen-credit scheduling overhead is small enough to meet real-time requirements. This can be seen when the timeslice is 1 ms. Though there is more context switching than in the 30 ms timeslice case, the real-time performance is still met. The difference is in the scheduler’s granularity. A smaller timeslice leads to less waiting for VMs to receive CPU time, as the resources are reallocated more often. This results in a more fair and responsive credit scheduler as the timeslice becomes more granular. Our results in Figure 1 show that a smaller timeslice leads to better performance. This implies that a balance needs to be met between a more responsive scheduler and the consequent increased context switch overhead.

We also took this one step further and explored what could be done to improve performance without tuning Xen-credit’s timeslice parameter. If a task does not finish executing before the current timeslice is exhausted, it will have to wait for the next timeslice. A task with longer period but identical utilization will have more time per job to finish. And as shown in Figure 2, even one job missing its deadline results in the failure of the whole task set. Therefore, the overheads and granularity of the scheduler have less of an effect on tasks with longer periods. We expect that the real-time performance will be better as the periods of the tasks become larger. Figure 3 illustrates the schedulability for Xen-credit(ts=30ms) across two period distributions: moderate (10ms, 100ms) and long (350ms, 850ms). In this figure, we indeed see that longer period workloads have better performance than shorter period workloads. This supports our assertion that the overheads and granularity have more of an effect on tasks with a shorter period.

The scheduling decisions when only a single VM is present are relatively simple compared to when there are a number of guests contending for a single resource. This results in low scheduler overhead. Only the overhead and the latency of the virtualization affect the performance of real-time tasks in the VM. This is why WR-KVM performs so well with one VM. The real challenge is sharing the resources among
multiple guests. When multiple VMs are scheduled by the hypervisor, it is not enough to ensure that each VM is allotted its fair share; real-time constraints must also be met. This is more representative of resource sharing using virtualization. Therefore, in Figure 4, we evaluate the performance when running two guests on the same host. For each hypervisor configuration, each guest is allocated half of the physical CPU resources available. In Xen-RTDS, each VM is given a budget of half the period, in this case 5000. In both Xen-credit experiments, each VM is assigned a cap of 400. In WR-KVM, cpu.shares is set to half for each VM. The total utilization varies from 0.2-4.2 in 0.2 increments. A consequence of this sharing is that contention for the resources becomes a factor. Here, we see that while WR-KVM demonstrated a high level of performance for the 1 VM cases, it does not perform well here, where resource contention is present. We also observe that Xen-RTDS and Xen-credit(ts=1ms) perform much better compared to the other configurations. Only the Xen-RTDS and Xen-credit(ts=1ms) are able to schedule all task sets for utilizations up until about 2.5. With task sets that have a light task utilization distribution, Xen-credit(ts=1ms) performs well all the way to about 3.8. This suggests that a real-time hypervisor such as Xen-RTDS, or a carefully tuned Xen-credit scheduler, is required to support RT workloads.

VI. CONCLUSION AND FUTURE WORK

In this paper, we characterize the performance of Xen (credit and RTDS schedulers) and Wind River’s low-latency KVM by comparing them to a non-virtualized base system. Each configuration is evaluated based on the ability to successfully schedule task sets of real-time applications. In our results, we observe that the configurations shown to perform sufficiently to support real-time workloads are more prone to fail in scheduling task sets that have heavier task utilization distributions. This is expected because tasks with higher utilization are composed of bigger but fewer jobs, thus having fewer permutations for all the jobs to be scheduled to meet their deadlines. We also observe that all four configurations meet deadlines for more than 99.8% of their jobs over a variety of workload characteristics, and therefore may be suitable for real-time workloads with relaxed constraints. However, when evaluating these configurations for strict hard real-time constraints, where no deadlines may be missed, only WR-KVM, Xen-credit(ts=1ms), and the Xen-RTDS are able to provide reasonable support when virtualizing a single guest. However, the single guest case is not representative of true resource sharing, as no resource contention is present. We then expand the work to include multiple guests. This causes resource contention between the guests, which is a more realistic model of virtualization-enabled sharing. In this case, only Xen-RTDS and Xen-credit(ts=1ms) are able to demonstrate performance suitable for supporting strict real-time constraints, for the types of workloads explored here. Therefore, we conclude that Xen-RTDS and Xen-credit(ts=1ms) have superior performance for supporting hard real-time workloads. The Xen-credit(ts=1ms) scheduler shows much better results in scheduling real-time tasks although it adds more scheduling overhead than Xen-credit scheduler(ts=30ms).

Future work includes exploring various measurements between the hypervisors and virtual machines for a more detailed analysis on scheduler’s performance and behavior. For example, profiling the execution of each scheduler could provide insight on areas for improvement, such as quantifying scheduling and context switching overhead. Other future work is evaluating how best to allocate the resources to guests with real-time requirements. An example of this is a feedback-based scheme. However, care must be made to prevent causing deadline misses by allocating insufficient resources.

Through our analysis of the performance of the three hypervisor solutions, we believe that virtualization support for real-time constraints is reaching a level where real-time constraints can be supported over a variety of task characteristics. This support, in addition to performance profiling and dynamic resource allocation, will enable more advanced systems, such as real-time clouds and frameworks, to be supported.

VII. ACKNOWLEDGEMENTS

This work was developed with support from the Office of Naval Research under grants #N00014-14-1-0035 and #N00014-16-1-2887. The authors would also like to acknowledge Mikyung Kang for her assistance with experimental setup and debugging.
REFERENCES