# A Flattened Hierarchical Scheduler for Real-Time Virtualization

Michael Drescher, Vincent Legout, Antonio Barbalace, Binoy Ravindran
Dept. of Electrical and Computer Engineering
Virginia Tech, Virginia, USA
{mdresch, vlegout, antoniob, binoy}@vt.edu

## ABSTRACT

Migrating legacy real-time software stacks to newer hardware platforms can be achieved with virtualization which allows several software stacks to run on a single machine. Existing solutions guarantee that deadlines of virtualized real-time systems are met but can only accommodate a reduced number of systems. Therefore, this paper introduces ExVM, a new scheduling framework to maximize the number of legacy uniprocessor real-time systems able to run on a single machine. Contrary to most existing solutions, ExVM uses a flattening approach where the host schedules the virtual machine which contains the task with the earliest deadline. The real-time characteristics of tasks are obtained through introspection during the execution. We implemented this framework using Linux's SCHED_DEADLINE real-time scheduling policy in the host. Simulations using an exact schedulability test show that ExVM is able to schedule 96% of randomly generated tasksets with a utilization of at least 0.8, while state-of-the-art solutions are only able to schedule 40% of the same tasksets. Experimental evaluations performed using synthetic benchmarks and production real-time applications show that ExVM always outperforms the existing solutions, always meeting more than 80% of deadlines while these solutions fall below 50% when the utilization increases.

## 1. INTRODUCTION

Many real-time systems are designed to run on the same hardware platform during their entire lifecycles. However, organizations periodically go through hardware refreshes for multiple reasons – e.g., hardware becomes obsolete, increasing maintenance costs; advanced features become available in newer platforms; security concerns, etc. Migrating existent, enterprise-class real-time software to new hardware is expensive, as it may involve costly re-writing of large legacy codebases and/or re-validation of timing requirements.

Virtualization is therefore an appealing solution for this problem. Full virtualization, which allows a guest OS to run unmodified atop a hypervisor is particularly compelling, as it enables a software stack (i.e., OS, applications, and associated libraries) to be easily migrated onto new hardware into a virtual machine (VM), without any modifications. With the commodity-scale availability of virtualization extensions, e.g., Intel VT-x, and AMD-V on the x86 architecture, full virtualization has become a game changer, as it reduces the need to emulate guest software, thereby reducing overhead and yielding good performance. In contrast, paravirtualization is a less appealing solution: it involves modifying the guest OS so that hypercalls can be made to the hypervisor. Though this also yields near native performance, the need to modify guest operating systems can be expensive, especially when large legacy codebases are migrated. Moreover, hardware virtualization extensions are likely to further improve with newer processor generations, further reducing the overheads of full virtualization. For the same motivations, operating-system-level virtualization technologies such as LXC [26] or jails [15], that further requires to eventually port the application to another operating system, are definitely not a viable solution either.

Using full virtualization, an existent real-time software stack can be migrated on to new hardware with relative ease. However, guest applications' time constraints must (continue to) be satisfied on the new hardware. This will require hypervisor-level scheduling policies that ensure temporal isolation of the guest VMs, especially when multiple VMs are consolidated on the same platform, which is often done in enterprise settings to reduce hardware costs, increase resource utilization, reduce maintenance costs, etc. One approach to schedule VMs is hierarchical scheduling [32], wherein VMs are scheduled as independent schedulable entities, and within each VM's scheduled time duration, its guest OS scheduler schedules its guest tasks.

Previous efforts using hierarchical scheduling to virtualize real-time systems use a periodic server-based approach, where guests are seen as periodic servers which take turns executing on the processor. This approach requires an offline analysis of the real-time tasks on each guest (e.g., execution time, period) to compute the proportion of CPU time to allocate to each server to satisfy all the deadlines. Then, during the execution, the hypervisor only schedules VMs and is not aware of the guest tasks inside each VM. This approach thus suffers from wasted CPU cycles and improvements can be obtained if guests are allowed to communicate with the host scheduler. Lackorzynski et al. [17] were the first to go beyond servers and introduced flattening scheduling to improve the schedulability of the system. Using a flattening approach, the host is aware of the tasks inside VMs and chooses the VM to schedule according to the characteristics of every task. However, their solution nevertheless still relies on servers (see discussion in Section 2).

Flattening scheduling alters the temporal isolation between guests. However, this paper targets legacy real-time systems, thus trusted systems in which the whole software stack is assumed not to be hostile. For example, we assume that no denial of service attack where guests launch tasks with short deadlines is going to occur during the execution. Thus, security issues are out of the scope of this paper.

## 1.1 Contribution

The contributions presented in this paper are fourfold: the ExVM framework for flattening hierarchical scheduling, its exact schedulability test, its implementation, and its evaluation using different scenarios and competitors. ExVM improves the schedulability of the system by directly scheduling guest real-time tasks via their associated VM using a flattened scheduler, while most of the existing solutions rely on temporally-isolated servers (Section 2 contrasts past work with ours). ExVM uses a periodic task model detailed in Section 3, supports guests which use either EDF or RM, and schedules VMs on multiprocessor systems using worst-fit partitioning. An exact schedulability test in Section 4 demonstrates that ExVM can schedule a higher proportion of tasksets than state-of-the-art server-based approaches.

An implementation of the algorithm based upon Linux and KVM is provided (Section 5). This implementation is the first open-source implementation of a real-time hierarchical scheduler for virtual machines in Linux which does not use servers but a flattened scheduler. To support legacy real-time guests, this implementation uses introspection [5] rather than paravirtualization to expose the real-time characteristics of the guests to the host. Contrary to other introspection engines, it only introspects real-time events and thus comes with a low overhead. The implementation of ExVM is open-source and is available online.

Using this implementation, Section 6 provides evaluations of the real-time performance of ExVM in comparison to vanilla Linux and a state-of-the-art server-based approach. Evaluations demonstrate that ExVM satisfies a higher number of deadlines than the state-of-the-art alternative when utilization is increased to near-overload and overload levels for multiple real-world real-time applications.

# 2. RELATED WORK

Scheduling virtual machines inherently implies a hierarchy of schedulers [10]; thus a plethora of hierarchical scheduling literature is relevant to the work presented in this paper. Servers provide a method of representing an application or virtual machine with its own real-time tasks and scheduler as a single entity to be scheduled by the host OS or VM monitor. Lipari et al. extend and improve upon scheduling with server by integrating the Constant Bandwidth Server framework in [21], examining how to partition resources among servers in [22], and how to extend hierarchical scheduling to multiprocessor systems [23]. Other aspects of hierarchical scheduling have also been studied, for example resource sharing [8] or cache related preemption delays [25]. There also exists analytical work for the hierarchical scheduling of tasks without real-time servers, such as an analysis of EDF within priorities presented by Harbour et al. in [14] or hybrid-priority real-time scheduling by Baruah et al. [2]. However, these works do not focus on virtualization.

In [28], Shin and Lee introduced the Compositional Scheduling Framework (CSF). CSF provides a method to compute the budget of a periodic server given the set of guest tasks, the guest scheduler, and the period of the server on the host. A smaller period in the host to schedule servers will result in a better schedulability but it will also increase the number of preemptions. CSF is fully composable, meaning that a server can schedule servers which schedules its own servers. In [19], Lee et al. introduce a method to support soft real-time tasks in the Xen hypervisor by reducing scheduling latency and managing shared caches among VMs. Real-Time Xen [32] was introduced by Xi et al. to schedule real-time virtual machines in Xen. RT-Xen uses the concept of servers to schedule virtual machines. Each virtual machine is limited to a single virtual CPU that is pinned to a single physical CPU core; a similar model is used in ExVM. In 2012, Lee et al. added CSF to Real-Time Xen [18]. With CSF, RT-Xen is able to guarantee that deadlines are met when the taskset is schedulable. Multicore scheduling was introduced in RT-Xen in [33]. This work evaluates the use of both global and partitioned EDF schedulers as the top-level scheduler for the servers, as well as for the guest schedulers.

Some approaches (e.g., [13, 4]) supporting real-time virtualization use a micro-kernel, and for example NOVA [30] could also be used as a microhypervisor. However, as in ExVM, many solutions make use of the widely available and well-known Linux kernel. Furthemore, Kiszka [16] showed that Linux using KVM and the PREEMPT_RT patch set has promise as a real-time hypervisor. In [7], Cucinotta et al. examine the use of KVM as a hypervisor when the guest is running IRMOS, an operating system designed to run real-time multimedia applications. IRMOS partitions its real-time tasks using *cgroups* in Linux, while Cucinotta et al. modified KVM to partition host resources among real-time guest tasks in an effort to reduce network latencies. The authors evaluate their implementation using measurements of network response times from the guest. Such work, differently from ExVM, is limited to a specific use case, which is reducing network latencies in the guest rather than modifying the host scheduler to provide real-time guarantees.

The idea of "flattening scheduling" was first introduced in [17], which observed that removing the concept of servers can improve the schedulability of a system. However, no hierarchical algorithms were presented in this work which allow servers to be discarded. Guests are still scheduled using servers; however the authors note that servers present a problem when trying to maintain hard real-time guarantees, and thus introduce their "flattening scheduling" idea. This work suggests using hypercalls to flatten guest and host schedulers, but does not go into detail about how this could be implemented and how the host could use this information to schedule guests. Additionally, the authors did not evaluate the feasibility of a flattening solution in the context of real-time guarantees. This paper expands on the ideas presented in [17] by providing an algorithm, theoretical analysis, and implementation of flattening scheduling.

# 3. MODELS AND ASSUMPTIONS

## 3.1 Task Model

A real-time task $\tau_i$ is characterized by its period $T_i$, its deadline $D_i$, and its worst-case execution time $C_i$. We consider that every task's deadline $D_i$ is equal to its period $T_i$. Each task releases a single job once per period and all tasks are released simultaneously. A job is denoted as $\tau_i^j$, where $j$ represents the $j^{th}$ release of task $\tau_i$. The absolute deadline of a task $\tau_i$ job is $d_i$. It is assumed that no job will ever exceed its worst case execution time $C_i$. However, the implementation provides safeguards against this possibility. $P_i$ is the priority of task $\tau_i$.

A set of tasks is denoted by $\Gamma$ and referred to as a taskset. In this model, the taskset to schedule contains $n$ tasks and is

divided into non-overlapping subsets $V_k$, where $V_k$ contains the tasks which belong to guest operating system $k$. The hyperperiod of a taskset $\Gamma$ is equal to the least common multiple (LCM) of $T_i$ for all $\tau_i \in \Gamma$. A task's utilization $U_i$ is defined as $C_i/T_i$. The utilization of a guest $V_k$ is defined as $\sum_{\forall \tau_i \in V_k} U_i$. The overall utilization $U$ of a taskset $\Gamma$ is defined as $\sum_{\forall \tau_i \in \Gamma} U_i$. $V_{\tau_i}$ is the guest which owns task $\tau_i$.

## 3.2 Guest Model

This work aims to schedule the taskset $\Gamma$ split in a collection of real-time guests, each guest $k$ being in charge of its own taskset $V_k$. Each guest is a VM running on top of a host operating system known as a hypervisor. The guests in this model are fully virtualized; this means they are executing under the assumption that they are running on real hardware, and thus make no attempts to communicate with the hypervisor. Furthermore, the guests operate without privileged hardware access, just as any user-space Linux process.

Each guest is assumed to be using either EDF or RM as its scheduler. $V_{EDF}$ and $V_{RM}$ are respectively the set of VMs scheduled with EDF and RM. In the remainder of this paper, a EDF VM will be a VM scheduled using EDF (respectively a RM VM for a VM scheduled using RM). All of the solutions presented in this work also support a mixture of EDF and RM guests. The guest schedulers are uninfluenced by both the hypervisor and other guests. Therefore, an EDF guest will always schedule the active task $\tau_i^j \in V_k$ which has the earliest absolute deadline. Similarly, an RM guest $k$ will always schedule the highest-priority active task which belongs to $V_k$. It is assumed that each guest runs on a single virtual CPU (vCPU).

## 3.3 Hardware Model

All theoretical and experimental results presented in this work assume a homogeneous x86-based multicore platform. The implementation presented in this work relies on the Intel-VT or the AMD-V virtualization hardware extensions. It is assumed that the host system consists of $m$ identical, independent cores. This work does not consider the effects of cache on performance. The model used by this work and the work of the competitors does not assume any preemption overhead or scheduling overhead. However, and contrary to most studies in this field, this paper includes experimental evaluations which incorporate caches as well as the overheads of preemption and scheduling.

In uniprocessor evaluations, the host is only able to use a single physical CPU to schedule multiple guests, each having a single vCPU. In multiprocessor evaluations, partitioned scheduling is used, such that vCPUs are unable to migrate to other physical CPUs after they begin executing.

This work examines a hierarchical scheduling model in which a single host schedules multiple guests. The role of the hypervisor scheduler is to choose which guest to schedule at any given point in time. Because the guest operating systems are unaware of the hypervisor and other guests, at any given point in time each guest operating system will have a preferred task $\tau_i^j$. The hypervisor cannot alter this preference. Essentially the hypervisor must determine which preferred task $\tau_i^j$ among the guests to schedule at each time.

No priorities are assumed between different guest operating systems. Supporting systems with priority requirements between different virtual machines are left to future work.

The host operating system for both the implementation presented in this paper as well as the competitors presented uses a constant-bandwidth-server (CBS) implementation. This means that tasks are preempted (i.e., they are not given any more execution time) if they attempt to overrun their specified worst case execution times $C_i$.

# 4. ALGORITHM

This section presents the uniprocessor flattened scheduling algorithm implemented by ExVM. This algorithm is work-conserving: it never leaves the processor idle when any guest has real-time jobs ready to execute. It removes the concept of servers and *flattens* the hierarchy of schedulers such that the host can schedule guest tasks directly. The term "flattening" was introduced in [17] but the algorithm presented here goes further by completely removing servers.

The scheduling algorithm uses EDF as the top-level scheduler in the hierarchy. This is because EDF has been shown to be an optimal algorithm for scheduling tasks on a uniprocessor system. Additionally, using EDF as the host scheduler eliminates the need to assign priorities to each guest, but rather schedule them with no assumption of priority, as specified in the scheduling model in Section 3.

Removing the concept of servers and bridging the communication gap between the host and guest schedulers comes with many advantages. First, it allows a reduction of idle CPU time, which directly leads to increased schedulability and performance. Additionally, ExVM has the potential to support any guest scheduler, not just EDF and RM. It can also better adapt to less-predictable guest tasksets, through the ability to gather task information on-the-fly. For example, aperiodic tasks, which only appear once and at any time. ExVM can also accommodate tasks which do not specify a worst-case execution time; the only parameter each task is required to report is its absolute deadline since the top-level scheduler uses EDF. Finally, ExVM does not require any offline calculations to compute a server budget, allowing guests and tasks to be added and removed on-the-fly at runtime without the recalculation of budgets.

## 4.1 Overview

The algorithm assumes no priority relationship between the virtual machines, is online, and executed in the host at each scheduling event. No offline processing is required, which makes it easier to add and remove tasks or VMs during the execution. The algorithm is detailed in Figure 1.

```
1 begin
2 |  Select active job τ^j with earliest absolute deadline;
3 |  Let g be the guest which contains τ^j;
4 |  if g is scheduled using EDF then
5 |  |  Schedule τ^j
6 |  else
7 |  |  Schedule the highest priority active job on g
8 |  end
9 end
```

Figure 1: ExVM Scheduling Algorithm

Note that a system with multiple guests all using fixed-priority schedulers such as RM does not necessarily produce the same schedule as running RM on all of the tasks without a host scheduler would; this is because the top-level sched-

uler is still EDF. In fact, a system in which every guest runs the RM scheduler with one task per guest will actually result in a pure EDF schedule under this algorithm.

## 4.2 Schedulability Test

The best-case schedule for the flattening algorithm occurs when the system only contains EDF VMs or RM VMs with only one task. In this case, all tasks are scheduled using EDF and the utilization bound is 1. When RM VMs with more than one task are added to the system, the utilization bound decreases. If the system only contains one RM VM, the algorithm actually reduces to RM, and has a utilization bound of $U_{bound} = n(2^{1/n} - 1)$ [24]. While this represents the utilization bound for the ExVM flattening algorithm, it is a poor schedulability test for the algorithm. In cases where one guest is scheduled with EDF, or even in cases where there are multiple guests all scheduled with RM, the actual utilization bound is higher than the bound for RM on the taskset.

In an effort to demonstrate ExVM algorithm's superiority over both RM and server-based approaches to hierarchical scheduling, this section proposes a schedulability test for the algorithm based on response-time analysis. The theory presented below is built upon the response-time analysis techniques developed by Spuri [29] for EDF scheduling, and the notation used below is based upon the notation used in [14], a response-time analysis of EDF within fixed-priorities.

This section computes the worst-case response-time $R_a$ of task $\tau_a$ which can be in a VM scheduled with EDF or RM. If this worst-case response time is larger than the period of $\tau_a$, $\tau_a$ will miss a deadline. The worst-case response time happens when $\tau_a$ is released at the critical instant. However the critical instant cannot be known in advance as in e.g. EDF, thus all the possible instants in the "busy period" must be examinated. The "busy period" is the period in which the processor is never idle. The busy period is assumed to have length $t$, and maximum length $L$. The busy period begins at time 0, and continues until a processor idle period is reached.

The worst-case response-time of $\tau_a$ depends on the interference of all the other tasks of the systems, tasks which can be in the same or another VM. The next section details the contribution of tasks scheduled using EDF. Then, the contributions of tasks scheduled in the same RM VM with higher priority and in a different RM VM are examined.

### 4.2.1 Interference from tasks scheduled under EDF

Spuri [29] provides the conditions necessary to calculate the worst-case contribution of an interfering task $\tau_i$ to the response time of the analyzed task $\tau_a$. Only the activations of $\tau_i$ that fall in the interval $[0, t)$ contribute to the worst-case response time of $\tau_a$. This follows from the work-conserving property of the scheduling algorithm. Additionally, only activations with a deadline earlier than $d_a$, the relative deadline of $\tau_a$, should be considered, since these tasks are scheduled using EDF.

Let each activation be identified by the sequence number $p$. Then the sequence number of interest identifies the last activation which contributes to the worst-case response time of $\tau_a$. The number of activations of task $\tau_i$ in the busy period is given by:

$$p_t = \left\lceil \frac{t}{T_i} \right\rceil \tag{1}$$

Similarly, since we assume that deadlines are equal to periods, the number of activations of task $\tau_i$ with deadlines before $d_a$ is given by:

$$p_{d_a} = \left\lfloor \frac{d_a}{T_i} \right\rfloor \tag{2}$$

By combining Equations 1 and 2, the worst-case contribution of an EDF task $\tau_i$ to the busy period is:

$$W_i^{EDF}(t, d_a) = min(\left\lceil \frac{t}{T_i} \right\rceil, \left\lfloor \frac{d_a}{T_i} \right\rfloor) \cdot C_i \tag{3}$$

Equations 1, 2, and 3 are adapted from [14]. Next, the worst-case contribution of a fixed-priority task from the same virtual machine is examined.

### 4.2.2 Interference from higher priority tasks in the same VM scheduled with RM

Naturally, if the task under analysis, $\tau_a$, is in a EDF VM (i.e., $V_{\tau_a} \in V_{EDF}$), the contribution of other tasks $\tau_i \in V_{\tau_a}$ can be calculated using Equation 3 for $W_i^{EDF}(t, d_a)$. However, if the task $\tau_a$ is in an RM VM (i.e., $V_{\tau_a} \in V_{RM}$), the worst case contribution of other tasks has nothing to do with deadlines and everything to do with priority.

THEOREM 1. *The worst-case contribution of a task $\tau_i \neq \tau_a$, $\tau_i \in V_{\tau_a} \in V_{RM}$ to the busy period is equal to the number of releases of $\tau_i$ in the busy period multiplied by $C_i$ if $P_i \geq P_a$, and equal to 0 if $P_i < P_a$.*

PROOF. Because guest schedulers are honored, it follows that every release of $\tau_i$ will preempt $\tau_a$ for its full execution time $C_i$ if $P_i \geq P_a$. Similarly, no task with a lower priority than $P_a$ will preempt $\tau_a$'s execution, thus lower priority tasks do not contribute to the busy period. □

Given the results of Theorem 1, the contribution of tasks in $V_{\tau_a} \in V_{RM}$ with a higher priority than $P_a$ can be calculated using the number of releases during the busy period, which is given by Equation 1. This leads to the following equation for the worst-case contribution of a task in the same VM with a higher priority, also adapted from [14]:

$$W_i^{HI}(t) = \left\lceil \frac{t}{T_i} \right\rceil \cdot C_i \tag{4}$$

### 4.2.3 Interference from tasks in other VMs scheduled with RM

The final set of tasks for which worst-case contribution must be computed is the set of tasks which reside in a fixed-priority virtual machine which is not the same machine in which $\tau_a$ resides.

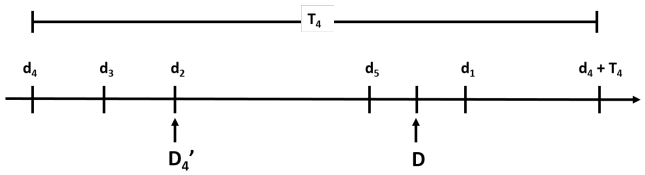
Unlike tasks in the same fixed-priority VM as $\tau_a$, there is no priority relationship between these tasks and $\tau_a$. However, it is also not as simple as the EDF case, because $\tau_i$ has the capability to contribute to the busy period even if its relative deadline $d_i$ falls after $d_a$ when another task in $V_{\tau_i}$ has a deadline earlier than $d_a$.

To obtain the number of releases of $\tau_i$ which contribute to $\tau_a$, the term $D_i'(d_a)$ is defined as follows:

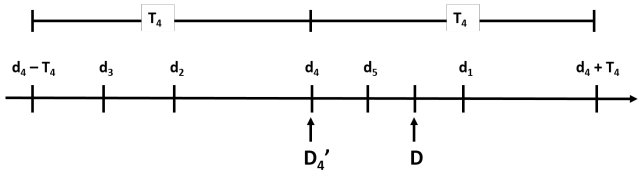
$$D_i'(d_a) = max(\forall \tau_j \in V_{\tau_i} | P_j \leq P_i, \left\lfloor \frac{d_a}{T_j} \right\rfloor T_j) \tag{5}$$

$D_i'(d_a)$ is the closest deadline in $V_{\tau_i}$ to $d_a$ such that $D_i'(d_a) \leq d_a$ and the deadline belongs to a task with a priority less than or equal to $P_i$. Figure 2 represents what the value of

$D'_i$ means graphically for $i = 4$. Let $d_j$ represent the relative deadline of $\tau_j \in V_{\tau_i}$, where a lower value of $j$ represents a lower priority.




(a) $D'_4 \neq d_4$




(b) $D'_4 = d_4$

Figure 2: Graphical Representation of $D'_4$

$D'_i$ represents the latest lower-priority deadline in $V_{\tau_i}$ that still falls before $d_a$. In Figure 2a, note that the last release of $\tau_i$, which in this case occurs at the first $d_4$, has its deadline well beyond the deadline $d_a$. Thus, if scheduled using EDF, this release would not contribute to the busy period. However, since $V_{\tau_i}$ is scheduled using a fixed-priority algorithm, this last release can still contribute to the busy period if a lower priority task in $V_{\tau_i}$ is active and has an earlier deadline than $d_a$. In this case, $\tau_i$ would preempt the lower priority task and execute, even though its deadline falls after $d_a$. This is the case in Figure 2a, where the lower-priority task $\tau_2$ has its deadline before $d_a$ but after the last release of $\tau_i$. Even though $d_5$ falls closer to $d_a$ than $d_2$, since $\tau_5$ has a higher priority than $\tau_i$, $\tau_i$ will not preempt $\tau_5$ and thus $d_5$ is not considered in the calculation of $D'_i$.

Figure 2b shows the case where the closest deadline in $V_{\tau_i}$ to $d_a$ belongs to $\tau_i$ itself (the period of $\tau_i$ has been divided by 2). In this case, the interference from $\tau_i$ on $\tau_a$ is the same as if $\tau_i$ was scheduled using EDF, since it gets no deadline advantage from lower priority tasks on its VM.

THEOREM 2. *Every release of a task $\tau_i \in V_{RM}, V_{\tau_i} \neq V_{\tau_a}$ which occurs strictly before $D'_i$ contributes to the busy period for $\tau_a$.*

PROOF. According to the flattening scheduling algorithm, $\tau_i \in V_{\tau_i} \in V_{RM}$ is scheduled when any task on $V_i$ has the earliest overall deadline, and $\tau_i$ is the highest priority active task in $V_{\tau_i}$. Therefore, for an activation of $\tau_i$ to preempt $\tau_a$, it is required that $V_{\tau_i}$ contains a task $\tau_j$ with priority $P_j \leq P_i$ and deadline $d_j \leq d_a$. Note that it is possible that $\tau_j \equiv \tau_i$. Since $D'_i$ is equal to the latest absolute deadline that falls no later than $d_a \, \forall \tau_j$ , releases of $\tau_i$ that fall before $D'_i$ will contribute to the busy period, while releases that fall no earlier than $D'_i$ will not preempt because there will be no lower priority task with a deadline between $D'_i$ and $d_a$. Therefore, it follows that every release of $\tau_i$ that falls strictly before $D'_i$ contributes to the busy period. □

With the result of Theorem 2, it is possible to define the worst-case contribution of a task $\tau_i$ which belongs to different fixed-priority guest from $\tau_a$:

$$W_i^{RM}(t, d_a) = min(\left\lceil \frac{t}{T_i} \right\rceil, \left\lceil \frac{D'_i(d_a)}{T_i} \right\rceil) \cdot C_i \tag{6}$$

Note that when $D'_i(d_a)$ coincides with $D_i$, Equation 6 reduces to Equation 3. This is because tasks in other VMs provide the same interference as EDF tasks except in special cases such as the one shown in Figure 2a.

#### 4.2.4 Calculating the Worst-Case Response Time of a Task

Table 1: Set Notation

| Name | Definition | Description |
|---|---|---|
| $S_{EDF}$ | $\cup V_i, \forall i \| V_i \in V_{EDF}$ | The set of all tasks which belong to EDF guests |
| $S_{HI}$ | $\cup\{\tau_i\}, \forall i \| \tau_i \in V_{\tau_a}, P_i \geq P_a$ | The set of all tasks in the same guest as $\tau_a$ with a higher priority than $\tau_a$ |
| $S_{RM}$ | $\cup V_i, \forall i \| V_i \in V_{RM}, V_i \neq V_{\tau_a}$ | The set of all tasks which belong to a different fixed-priority guest than $\tau_a$ |

To lighten the notation, this paper defines the sets in Table 1. With the worst-case contribution of fixed-priority tasks in other VMs accounted for, the following analysis is adapted directly from Harbour [14]. As demonstrated by Spuri [29], the release time of the task under analysis $\tau_a$ is not necessarily the beginning of the busy period. It is possible that the first release of $\tau_a$ which leads to the worst-case response time occurs at an offset from the start of the busy period, such that a future release shares its deadline with another task in the system. Thus, response-time analysis for $\tau_a$ must examine release times which coincide with other task deadlines, $pT_i$. The set of all of these release times which must be examined, $\Psi$, is given below:

$$\begin{gathered} \Psi = \cup\{p \cdot T_i\} \\ \forall p = 1 \ldots \left\lceil \frac{L}{T_i} \right\rceil, \forall i \in S_{RM} \cup S_{EDF} \cup S_{HI} \end{gathered} \tag{7}$$

In Equation 7, $L$ refers to the longest busy period, which is calculated recursively using the following equation:

$$L = \sum_{\forall i \in S_{RM} \cup S_{EDF} \cup S_{HI}} \left\lceil \frac{L}{T_i} \right\rceil \cdot C_i \tag{8}$$

The critical release for $\tau_a$ is found by subtracting $T_a$ from each value in the set $\Psi$. Since $\tau_a$ may have multiple activations within the busy period, every activation must be analyzed for worst-case response time. If the first activation of $\tau_a$ within the busy period occurs at time $A$, and the busy period begins at time 0, the completion time $w_a^A(p)$ of acti-

vation $p$ of $\tau_a$ can be calculated by the following equation:

$$w_a^A(p) = p \cdot C_a + \sum_{\forall \tau_i \in S_{HI}} W_i^{HI}(w_a^A(p)) + \sum_{\forall \tau_i \in S_{RM}} W_i^{RM}(w_a^A(p), D_a^A(p)) + \sum_{\forall \tau_i \in S_{EDF}} W_i^{EDF}(w_a^A(p), D_a^A(p)) \quad (9)$$

Equation 9 combines the worst-case contributions from Equations 4, 6, and 3, as well as the contribution from $\tau_a$ itself. The term $D_a^A(p)$ refers to the deadline of activation $p$ when the first activation occurs at $A$:

$$D_a^A(p) = A + p \cdot T_a \quad (10)$$

Finally, the worst-case response time $R_a^A(p)$ for activation $p$ of task $\tau_a$ is calculated by subtracting the activation time of iteration release $p$ from the completion time:

$$R_a^A(p) = w_a^A(p) - A - (p-1)T_a \quad (11)$$

It is only necessary to check values of $A$ which fall between 0 and $T_a$, since the first release must fall within this range. Thus, it is necessary to check the values of $\Psi$ in the subset:

$$\Psi^* = \{\Psi_x \in \Psi | p \cdot T_a \leq \Psi_x < (p+1)T_a\} \quad (12)$$

Thus, we define the set of $A$ values to check as:

$$A(\Psi_x) = \Psi_x - (p \cdot T_a) \quad (13)$$

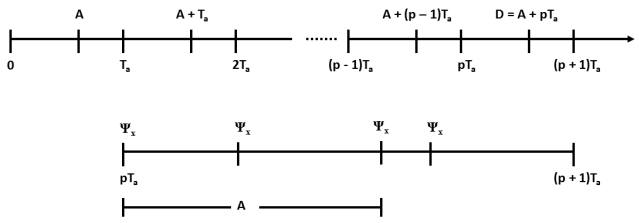



Figure 3: Graphical Representation of $A$ and $\Psi_x$

Figure 3 demonstrates the graphical meaning of the different deadlines to analyze for task $\tau_a$, which this paper denotes $\Psi_x$. The top timeline begins at the start of the busy period (time 0) and ends with the $p^{th}$ release of $\tau_a$. The value of $A$ represents an offset from the start of the busy period. The first release of task $\tau_a$ occurs at $A$, and subsequent releases occur at $A + pT_a$ for integer values of $p$. Clearly, the value of $A$ must fall between 0 and $T_a$, since $A$ is defined as the first release of $\tau_a$ in the busy period and if $A > T_a$, there would be another release of $\tau_a$ which falls between 0 and $T_a$ and thus is the first release.

The second timeline in Figure 3 demonstrates how the values of $A$ to iterate over are calculated. Each $\Psi_x$ on this timeline represents an absolute deadline of an interfering task in the system (or $\tau_a$ itself) that falls between $pT_a$ and $(p+1)T_a$. Each of these values of $\Psi_x$ is offset from $pT_a$ by the same offset of the first release, $A$. Only values of $A$ which correspond with $\Psi_x$ need to be checked for the $p^{th}$ release of $\tau_a$ because having $\tau_a$'s deadline coincide with the deadline of an interfering task provides a (local) maximum of interference on $\tau_a$.

Finally, the absolute worst-case response time for the task $\tau_a$, denoted $R_a$, can be calculated:

$$R_a = max(R_a^A(p)) \quad \forall p = 1 \ldots \left\lceil \frac{L}{T_a} \right\rceil, \forall A(\Psi_x) | \Psi_x \in \Psi^* \quad (14)$$

Thus, a taskset is schedulable by the ExVM scheduling algorithm if the worst-case response time for every task is less than the relative deadline for the task. In other terms, the taskset is schedulable if:

$$R_i \leq D_i, \forall i \quad (15)$$

# 5. IMPLEMENTATION

We implemented the algorithm presented in Section 4 in an existing real-time operating system, here the Linux kernel. The implementation is based on SCHED_DEADLINE [12], an Earliest Deadline First scheduling policy which has been part of the Linux kernel since version 3.14.0. Guest real-time parameters are gathered by a low overhead introspection engine [5]. It captures the real-time requirements of the guest VM by instrumenting its kernel scheduler by adding undefined instructions, which force the VM to exit thus the host scheduler to gain control. Because instrumentation happens at initialization time there is no need to modify the guest software. The introspection engine comes with a plugin for each supported operating system and requires the kernel symbol table to be available (e.g., DWARF infos, Linux's `/proc/kallsysm`, or `System.map`).

## 5.1 Deadline Scheduling in Linux

The EDF scheduling class in Linux is implemented based on the assumption that each Linux thread represents a single real-time task. As a result of this assumption, each `task_struct`, the kernel data structure representing a single Linux thread, only contains a single, statically-allocated deadline scheduling entity. The deadline scheduling entity, represented by the `sched_dl_entity` structure, contains fields for the task period, deadline, and worst-case execution time.

In the Linux kernel, each physical CPU has an associated runqueue, which contains the threads which are active on that CPU. The thread selected by the `pick_next_task` function is the thread which will gain control of the CPU next. The `pick_next_task` function gives priority to threads scheduled using the SCHED_DEADLINE scheduling class. To decide between two deadline-scheduled threads, each CPU also has a red-black tree data structure, sorted by deadline. The left-most node in the tree contains the `sched_dl_entity` with the earliest absolute deadline. When the red-black tree is not empty, the `pick_next_task` function will return the `task_struct` associated with such left-most node.

A deadline scheduling entity will remain in the red-black tree until it depletes its available runtime. When the runtime is fully depleted, the entity is marked as *throttled*, its associated thread is put to sleep, and a timer is started which will wake the thread up at the beginning of the next period. When the thread is awoken by the timer, the entity's budget is *replenished*, i.e., it is set to the task's worst-case execution time and the entity is added back into the red-black tree.

## 5.2 Flattened Deadline Scheduling

In KVM, the hypervisor used by ExVM, a guest's virtual CPU, or vCPU, is implemented as a Linux thread itself. As

a result, the vCPU is represented in the kernel by a single `task_struct` with a single `sched_dl_entity`. However, when the guest contains multiple, concurrently-executing real-time tasks, a single `sched_dl_entity` is not sufficient to store the real-time information of every guest task. Note that tasks can be released simultaneously and have different parameters hence an aggregate of real-time characteristics is not possible. Because of this, the `task_struct` structure was modified to accommodate multiple scheduling entities, dynamically-allocated for each guest real-time task.

Scheduling entities are stored in a linked-list that is pointed by the owner `task_struct`, each scheduling entity contains a pointer to its thread's `task_struct`. Scheduling entities are added to the red-black tree in the same way as vanilla Linux does; when an entity has runtime it is in the tree, and the `pick_next_task` function chooses the `task_struct` associated with the left-most entity in the tree.

Rather than rely on timers for budget replenishment, this implementation is able to leverage introspection. Introspection provides the host scheduler with the exact time of every guest task release and termination; as a result the scheduler implementation is able to replenish scheduling entities at the exact moment of task release without using a timer. This ensures that each scheduling entity remains synchronized with the actual task period on the guest; timers have the possibility of drifting away from the actual task period if either the period parameter or the timer object is not perfectly accurate. For feasible schedules, the guest receives the execution time necessary to release each task via the non-real-time task support, presented in Section 5.3. Additionally, introspection provides the capability to deplete the budget at the exact moment of task termination, which frees the processor to execute other real-time tasks when any given task completes its execution prior to its WCET.

SCHED_DEADLINE does not natively support RM; the support for it was added using data structures similar to the deadline-sorted red-black tree. `task_struct` was augmented with `is_dm` flag, which is set if the guest is scheduled using the deadline-monotonic algorithm. When the `is_dm` flag is set, scheduling entities from the guest are added to a second per-guest linked-list (`dm_list`) which is sorted by relative deadline. (Entities are still present in the red-black tree.)

The head of the `dm_list` is the scheduling entity with the smallest relative deadline. When the `pick_next_task` function executes, it chooses the left-most node of the red-black tree (the earliest deadline), and then checks if the associated `task_struct` has its `is_dm` flag set. If so, it depletes the budget of the entity at the top of the `dm_list` when the guest is executed. Otherwise, it depletes the entity with the earliest deadline. Entities are removed from both the `dm_list` and the red-black tree when their budgets are depleted. Similarly, entities are added to both the list and the tree when their budgets are replenished.

## 5.3 Non-Real-Time Guest Processes

In addition to real-time task budgets, most guests require execution time for general processes such as a shell or `ssh` server. To provide this execution time without compromising the real-time schedule, an extra deadline scheduling entity is added to each `task_struct`. This entity is guaranteed to never deplete its runtime and to always have the latest possible absolute deadline. Because of these properties, it will only execute when there are no active releases of real-time tasks in any guest. When the red-black tree contains more than one of these entities, they are scheduled in a round-robin fashion in order to provide time to each guest.

# 6. EVALUATION

This section evaluates ExVM through simulations and experimental evaluations. Simulation is first used to compare ExVM with CSF (Section 6.1), a server-based approach used by RT-Xen, based on the number of tasksets that both solutions can schedule. Then, overheads are evaluated in Section 6.2 using the implementation discussed in section 5. Finally, experiments are performed using the implementation on uniprocessor and multiprocessor systems, first using a synthetic benchmark (Section 6.3), then using real-time applications (Section 6.4).

Simulations and experiments are performed using different periods for CSF and for multiple sets of VMs. However, the implementation of ExVM is not compared to RT-Xen because the evaluation would be more focused on the differences between Xen and KVM than on the differences between the two real-time scheduling solutions. Regarding [17], it does not describe an actual algorithm, we were thus unable to compare ExVM against this solution.

Evaluations were performed on 2 different servers: an Intel server with a 8-core Intel Xeon E5520 CPU running at 2.27 GHz with 8MB of cache and 12 GB of RAM; and an AMD server with an AMD Opteron 6168 processor running at 1.9 GHz with 12MB of cache and 12 GB of RAM. The Intel server was used to measure the virtualization and scheduling overheads and to run the synthetic benchmarks. Both server machines were used to evaluate the real-time production applications. For all but the multicore evaluations, execution was restricted to a single core using Linux's *cpusets*.

The host runs a patched Linux 3.16 and QEMU 1.6.50. Both the host and the guests run Ubuntu 14.04. All guests are similar; they run a ChronOS 3.0 real-time kernel based on Linux kernel 3.0.24. ChronOS [9] is a real-time Linux-based operating system which implements a number of traditional real-time scheduling algorithms. The choice of the guest operating system is however not critical and other real-time operating systems like Linux/RTAI, LITMUS$^{RT}$ [6], or RTEMS, could have also been chosen.

## 6.1 Comparison with CSF

Figure 4 shows the percentage of schedulable tasksets for both CSF and ExVM for different configurations and tasksets. For these simulations, each taskset contains 6 individual tasks, divided randomly among 3 VMs. The utilization of each task is computed randomly to be between 0.01 and 0.99 using a uniform distribution and the UUniFast algorithm [3]. The period of each task is chosen randomly as a multiple of 100 between 100 and 1000. The global utilization of the taskset is set between 0.1 and 0.95. 500 tasksets were generated for each utilization. For CSF [28], 3 different server periods were used and all servers in the same simulation share the same period. CSF-20 denotes CSF with a server period of 20, while CSF-100 denotes a server period of 100, and so on. In Figure 4a two VMs are scheduled using EDF and one with RM, while in Figure 4b 1 VM is scheduled using EDF and 2 using RM. In Figures 4c and 4d, the global utilization is fixed respectively at 60% and 80% and only the scheduling algorithms used inside VMs evolves: the x axis represents the percentage of VMs scheduled with EDF.

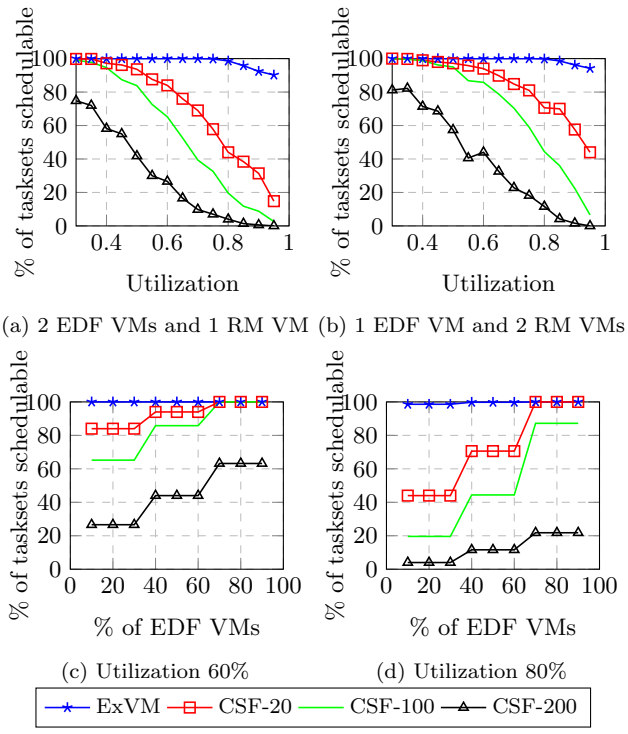



Figure 4: % of tasksets schedulable by ExVM and CSF

ExVM is able to schedule the largest proportion of tasksets, regardless of the utilization. As expected, CSF performs better with a smaller server period; however, this smaller period comes with the trade-off of a higher number of virtual machine preemptions. Even though preemption overhead is not considered in this simulation, it is a concern at runtime that we address in the next subsections where evaluations are performed using an actual implementation. When the proportion of EDF VMs increases, the number of tasksets that are schedulable increases as expected, but ExVM always outperforms CSF. We also performed the same simulations with a varying number of tasks and VMs with similar results: ExVM always outperforms CSF and the difference grows when the average number of tasks per VM increases. More comprehensive results can be found in [11].

## 6.2 Virtualization and Scheduling Overheads

This section first evaluates the cost of the introspection mechanism [5] using microbenchmarks in the guest and by instrumenting the host kernel source code. The results are calculated over 100 runs. Figure 5 shows that total cost for introspection is around 13.8k CPU cycles ( 6us) with a standard deviation of less than 5%. Thus, introspection can be approximated as a constant overhead and assimilated into the WCET of each task. The introspection cost includes switching the execution from guest code to the host kernel code (exit_VM), saving guest's information (save_VM), handling the exit (vmx_handle_exit), passing the control back to the main KVM loop (__vcpu_run), restoring guest's information (restore_VM), and re-entering the VM (enter_VM). The cost of all scheduling related mechanisms that we developed are integrated in vmx_handle_exit cost. ExVM is not responsible for the 20% deviation in the run-time; this

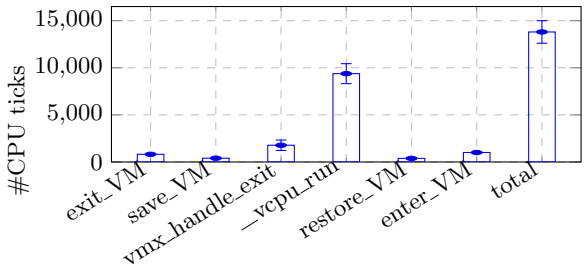



Figure 5: VM introspection cost breakdown

is due to the Linux kernel. The same applies to __vcpu_run.

Then, to measure the overheads of both virtualization and the scheduling algorithms on the guest and host, the deadline miss load metric, first introduced in [20], is used. Deadline miss load presents a useful method of characterizing scheduler overheads by running a schedulable taskset and reducing the average period and execution time for the tasks until a deadline is missed, while keeping the utilization of the taskset constant. The end result is a minimum average period for tasksets that run on the system.

Three sources of overhead are of interest: overhead due to the guest scheduler (EDF) and kernel (ChronOS Linux), overhead due to the virtualization software (KVM/QEMU), and overhead due to the host scheduler (ExVM scheduling algorithm) and kernel (Linux). In order to isolate each of these overheads, the deadline miss load was calculated for 3 different experimental setups: 1) the taskset is run on a single processor core (Hardware), 2) the taskset is run on a single virtualized processor core (Virtualization) and 3) the taskset is run on multiple virtualized processor cores, scheduled onto a single physical processor core (ExVM). The first setup characterizes the overhead of the EDF algorithm. The second setup characterizes the overhead added by the virtualization layer. The third setup is used to characterize the additional overhead added by ExVM.

This evaluation was run using a taskset with a utilization of 0.95, 12 tasks, and an average period which was varied between $4.4ms$ and $155ms$. The deadline miss load value for Hardware is $4.5ms$ while for both Virtualization and ExVM is $154.4ms$. The experiment was repeated for 16, 20, 24, 32, and 64 tasks with similar results. This shows that the overhead added by ExVM is negligible compared to virtualization, but also that ExVM is able to schedule VMs appropriately when there are no active real-time tasks, including the timely release of real-time tasks.

## 6.3 Evaluations using synthetic benchmarks

This section evaluates ExVM using `sched_test_app` [9], a synthetic benchmark developed for ChronOS. ExVM is compared with vanilla KVM, which is oblivious of the real-time tasks in the guest, and with a periodic server-based VM scheduling solution, which is instead informed about the workload in the guests. KVM schedules each VM with a weighted round-robin policy (Linux's Completely Fair Scheduler), while the server-based configuration schedules VMs as periodic servers (Linux's SCHED_DEADLINE) with budget and period calculated offline with CSF (similarly to [34]).

Figure 6 shows the percentage of deadlines satisfied for ExVM and Vanilla KVM for different utilizations between 0.7 and 1.3. Since CSF is designed without a mechanism to

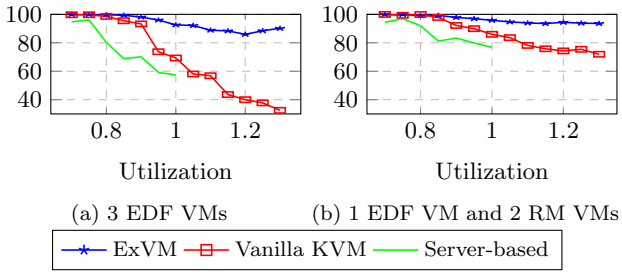



Figure 6: Average % of deadlines satisfied

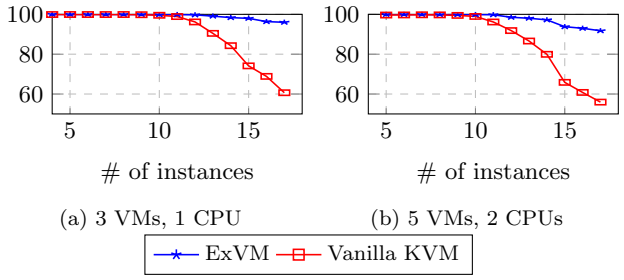



Figure 7: Average % of deadlines satisfied among 3 VMs

handle overload conditions, CSF's budgets and periods can only be evaluated with tasksets which are schedulable, thus the Server-based configuration was examined only between an utilizations of 0.7 and 1. For each utilization, 10 random tasksets have been generated using the Baker model [1] and have been scheduled using `sched_test_app`. Each point has been computed using the average percentage of deadlines satisfied among all the tasks of all VMs for 10 tasksets.

As expected, more and more deadlines are missed once the utilization increases. But ExVM still achieves more than 85% of deadlines satisfied for a utilization of 1.2 in Figure 6a. On the other hand, the performance of Vanilla KVM drops quickly to reach only 40%. The disparity between VMs and the standard deviation for Vanilla KVM is higher than for ExVM which makes ExVM more deterministic than Vanilla KVM. As seen in Figure 6b, deadline misses appear earlier when the ratio of EDF VMs decreases but the percentage of deadlines satisfied drops slower than with more EDF VMs. This is due to the fact that RM performs better than EDF in overloaded situations. The Server-based solution perform the worst, with up to 30% more deadline misses than Vanilla KVM in Figure 4a. This is because a portion of the budget is consumed by non-real-time tasks during every server period, delaying the earlier release of the server itself. Note that this does not happen when using Vanilla KVM nor ExVM, which both perform better in this scenario. In RT-Xen, paravirtualization and idle-budgets are used, which should also avoid this budget-consuming problem.

We performed the same evaluations for all different combinations of scheduling algorithms inside VMs, i.e., with 3 VMs RM scheduled, and 2 VMs EDF scheduled and 1 with RM. Evaluations show that for all configurations, the utilization for which the percentage of deadline satisfied falls below 99% is always higher for ExVM than for vanilla KVM.

## 6.4 Evaluations using real-time applications

This section presents evaluations performed using production real-time applications from Table 2. Each application has been modified to run as a single-threaded real-time task in ChronOS Linux. Due to space constraints, this paper only present the results with x264, but the evaluations performed with the two other benchmarks yielded similar conclusions [11]. The average execution time of x264 on the Intel server is $40ms$ for a period of $120ms$. Evaluations are performed by spawning a number of identical instances of each application with identical input. These instances are spread as evenly as possible among the virtual machines in the system. As the number of instances increases, so does the overall taskset utilization. Once the number of instances increases to the point where the system is overloaded, the percentage of deadline satisfied begins to drop from 100%.

Table 2: Benchmarks

| | |
|---|---|
| x264 [27] | Video encoding application. |
| disparity [31] | Motion-tracking application which uses stereo-vision. Part of the San Diego Vision Benchmark Suite (SD-VBS) |
| multi n-cut [31] | An image segmentation application. Part of SD-VBS. |

We also conducted experiments on multiple processors using partitioned scheduling. We choose the commonly used worst-fit heuristic to partition virtual machines among different processors in order to share the load between all available processors. Given a virtual machine, the algorithm works by assigning that virtual machine to the processor with the lowest utilization, even if the utilization becomes larger than 1. The multiprocessor setup consists of 5 VMS and 2 CPUs. Each virtual CPU is assigned to a specific physical CPU. This pinning of vCPUs is done before the release of any real-time jobs on any of the virtual machines.

Figure 7 shows the percentage of deadlines satisfied for the two configurations with one and two physical CPUs. ExVM performs better than Vanilla KVM, for which the number of deadline misses increases quickly when more than 10 task instances are executed. Similarly to the evaluations with `sched_test_app`, the standard deviation of deadlines satisfied between VMs in a single trial, and between multiple trials, is lower with ExVM than with Vanilla KVM.

# 7. CONCLUSION

This paper introduced ExVM, a real-time scheduling framework for virtualized systems with a hierarchy of schedulers. It schedules virtual machines using flattening scheduling instead of servers, therefore improving the schedulability of the system. The host scheduler is aware of the real-time characteristics of the guests (scheduling policy, tasks) and can therefore schedule guests accordingly, allowing greater flexibility. An exact schedulability test was introduced showing that ExVM decreases the CPU bandwidth required to feasibly schedule a taskset. The scheduler makes all its decisions online, and therefore does not require any offline computations based on the WCET of tasks. An implementation is provided based on Linux and KVM. Compared to state-of-the-art solutions, we showed through simulations of random tasksets that for a utilization larger than 0.8, ExVM can schedule 96% of the tasksets while this percentage drops to 40% for state-of-the-art solutions. Experimental evaluations

using real-time applications show that ExVM allows more deadlines to be satisfied at higher utilizations. In future work, we would like to allow migrations of guests between physical CPUs to improve the schedulability of the system. We also plan to support multiprocessor guests, i.e., guests with two or more vCPUs, first using partitioned scheduling and then global scheduling. Finally, we would like to expand the work to other task models (e.g., aperiodic).

## Acknowledgments

This work was supported by US NSWC under grant N00178-13-D-1031/0005 and by US NAVSEA/NEEC under grant 3003279297.

# 8. REFERENCES

[1] T. P. Baker. Comparison of empirical success rates of global vs. partitioned fixed-priority and EDF scheduling for hard real time. Technical report, 2005.

[2] S. Baruah and N. Fisher. Hybrid-priority real-time scheduling. In *ISPDP*, April 2008.

[3] E. Bini and G. C. Buttazzo. Biasing effects in schedulability measures. In *16th ECRTS*, 2004.

[4] F. Bruns, D. Kuschnerus, and A. Bilgic. Virtualization for safety-critical, deeply-embedded devices. In *28th SAC*, 2013.

[5] K. Burns, A. Barbalace, V. Legout, and B. Ravindran. KairosVM: Deterministic introspection for real-time virtual machine hierarchical scheduling. In *2nd VtRES*, September 2014.

[6] 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 *27th RTSS*, 2006.

[7] T. Cucinotta, F. Checconi, and D. Giani. Improving responsiveness for virtualized networking under intensive computing workloads. In *13th RTLWS*, 2011.

[8] R. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *27th RTSS*, 2006.

[9] M. Dellinger, P. Garyali, and B. Ravindran. ChronOS Linux: a best-effort real-time multiprocessor Linux kernel. In *48th DAC*, 2011.

[10] Z. Deng and J. W.-S. Liu. Scheduling real-time applications in an open environment. In *18th RTSS*, 1997.

[11] M. Drescher. *A Flattened Hierarchical Scheduler for Real-Time Virtual Machines*. MS thesis, Virginia Tech, 2015. *http://scholar.lib.vt.edu/theses/available/etd-05182015-121808/*.

[12] D. Faggioli, F. Checconi, M. Trimarchi, and C. Scordino. An EDF scheduling class for the linux kernel. In *11th RTLWS*, 2009.

[13] S. Groesbrink, L. Almeida, M. de Sousa, and S. M. Petters. Towards certifiable adaptive reservations for hypervisor-based virtualization. In *20th RTAS*, 2014.

[14] M. Harbour and J. Palencia. Response time analysis for tasks scheduled under edf within fixed priorities. In *24th RTSS*, Dec 2003.

[15] P.-H. Kamp and R. N. Watson. BSD Jails. http://phk.freebsd.dk/pubs/sane2000-jail.pdf.

[16] J. Kiszka. Towards Linux as a real-time hypervisor. In *11th RTLWS*, 2009.

[17] A. Lackorzyński, A. Warg, M. Völp, and H. Härtig. Flattening hierarchical scheduling. In *10th EMSOFT*, 2012.

[18] J. Lee, S. Xi, S. Chen, L. T. Phan, C. Gill, I. Lee, C. Lu, and O. Sokolsky. Realizing compositional scheduling through virtualization. In *18th RTAS*, 2012.

[19] M. Lee, A. Krishnakumar, P. Krishnan, N. Singh, and S. Yajnik. Supporting soft real-time tasks in the Xen hypervisor. In *ACM Sigplan Notices*, volume 45, pages 97–108, 2010.

[20] P. Li, B. Ravindran, S. Suhaib, and S. Feizabadi. A formally verified application-level framework for real-time scheduling on POSIX real-time operating systems. *IEEE TSE*, 30(9):613–629, 2004.

[21] G. Lipari and S. Baruah. A hierarchical extension to the constant bandwidth server framework. In *19th RTAS*, 2001.

[22] G. Lipari and E. Bini. Resource partitioning among real-time applications. In *15th ECRTS*, 2003.

[23] G. Lipari and E. Bini. A framework for hierarchical scheduling on multiprocessors: from application requirements to run-time allocation. In *31st RTSS*, 2010.

[24] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, Jan. 1973.

[25] W. Lunniss, S. Altmeyer, G. Lipari, and R. Davis. Cache related pre-emption delays in hierarchical scheduling. *Real-Time Systems*, pages 1–38, 2015.

[26] LXC. Linux Containers. http://linuxcontainers.org/.

[27] L. Merritt and R. Vanam. x264: A high performance H. 264/AVC encoder. 2006.

[28] I. Shin and I. Lee. Compositional real-time scheduling framework with periodic model. *ACM TECS*, 7(3):30:1–30:39, May 2008.

[29] M. Spuri. Analysis of deadline scheduled real-time systems. *Rapports de recherche- INRIA*, 1996.

[30] U. Steinberg and B. Kauer. Nova: A microhypervisor-based secure virtualization architecture. In *5th EuroSys*, pages 209–222.

[31] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor. SD-VBS: The San Diego vision benchmark suite. In *IISWC '09*.

[32] S. Xi, J. Wilson, C. Lu, and C. Gill. RT-Xen: towards real-time hypervisor scheduling in Xen. In *10th EMSOFT*, 2011.

[33] S. Xi, M. Xu, C. Lu, L. T. Phan, C. Gill, O. Sokolsky, and I. Lee. Real-time multi-core virtual machine scheduling in Xen. In *12th EMSOFT*, 2014.

[34] J. Yang, H. Kim, S. Park, C. Hong, and I. Shin. Implementation of compositional scheduling framework on virtualization. *SIGBED Rev.*, Mar. '11.