Multi-Mode Virtualization for Soft Real-Time Systems

Haoran Li*, Meng Xu†, Chong Li*, Chenyang Lu*, Christopher Gill*, Linh Phan†, Insup Lee†, Oleg Sokolsky†
*Washington University in St. Louis, †University of Pennsylvania

***Abstract*—Real-time virtualization is an emerging technology for embedded systems integration and latency-sensitive cloud applications. Earlier real-time virtualization platforms require offline configuration of the scheduling parameters of virtual machines (VMs) based on their worst-case workloads, but this static approach results in pessimistic resource allocation when the workloads in the VMs change dynamically. Here, we present *Multi-Mode-Xen (M2-Xen)*, a real-time virtualization platform for dynamic real-time systems where VMs can operate in modes with different CPU resource requirements at run-time. M2-Xen has three salient capabilities: (1) dynamic allocation of CPU resources among VMs in response to their mode changes, (2) overload avoidance at both the VM and host levels during mode transitions, and (3) fast mode transitions between different modes. M2-Xen has been implemented within Xen 4.8 using the real-time deferrable server (RTDS) scheduler. Experimental results show that M2-Xen maintains real-time performance in different modes, avoids overload during mode changes, and performs fast mode transitions.**

I. Introduction

Consolidating multiple systems into a multi-core host can reduce the cost of real-time systems while enhancing their flexibility. Virtualization is gaining traction as a platform for integrating real-time systems [1]. Virtualization maintains the modularity of systems as virtual machines (VMs), while allowing multiple subsystems to share the hardware resources of a single physical machine. Each VM contains both the guest OS and the application tasks of the subsystem. Compositional analysis then allows a subsystem vendor to abstract the application timing requirements as a resource interface of the VM [2–4]. Real-time schedulers [5–7] have been developed and adopted in hypervisors to satisfy the resource interfaces of VMs, thereby allowing application tasks to meet their timing constraints. This real-time virtualization approach benefits both subsystem vendors and system integrators. It does not require the system vendors to expose the details of their subsystem implementations, while still allowing the system integrator to meet the real-time performance of each VM based on its resource interface.

For example, an *in-vehicle infotainment (IVI)* system may include traffic information, cellphone interfaces, and vehicle self-diagnostics, and auxiliary hands-free functions for drivers. These systems may be provided by different vendors and then integrated by the vehicle manufacturer. Virtualization technology offers a promising solution for IVI systems: multiple systems can be virtualized and consolidated in a multicore host. Each system can be implemented as a VM, which reduces hardware components and simplifies the design. The Nautilus Platform [8], Renesas R-Car [9], and INTEGRITY Multivisor [10] are examples of automotive virtualization solutions. Specifically, the Nautilus Platform, which collaborates with the Xen Automotive Project, leverages the Xen hypervisor and RTDS scheduler to host VMs running infotainment software on different operating systems [11].

Traditionally, computational resources for each VM are statically allocated through the hypervisor, in a configuration that is based on the worst-case CPU requirment of the VM. However, this static configuration approach may underutilize hardware resources when VMs may operate in different modes each involving a different task set. Supporting multi-mode VMs becomes increasingly important as real-time systems operate in dynamic settings. Representative examples include IVI systems that need to adapt to driver behavior, and edge clouds that need to be reconfigured for agile manufacturing.

To support dynamic soft real-time systems, we develope *Multi-Mode-Xen (M2-Xen)*, a real-time virtualization platform that adapts to mode changes of VMs. A VM may operate in different modes, each comprising a task set and its corresponding resource interface. To maintain real-time performance of dynamic systems, M2-Xen provides three novel capabilities: (1) *dynamic reallocation* of CPU resources among VMs in response to their mode changes, (2) *overload avoidance* at both the VM and host levels during a mode transition, and (3) *fast mode transitions*. M2-Xen has been implemented in Xen 4.8 using the real-time deferrable server (RTDS) scheduler. [1] Experimental results show that M2-Xen: (1) maintains real-time performance in different modes, (2) avoids overload despite mode changes, and (3) significantly reduces mode switching latency when compared to standard approaches.

This paper is structured as follows. Section II introduces background information. Section III identifies system requirements. Section IV describes the design and implementation of M2-Xen. Section V focuses on design choices to achieve fast mode transition. Section VI presents the evaluation of M2-Xen. Section VII discusses other relevant research, and Section VIII summarizes the contributions of this work.

II. Background

M2-Xen is built on top of RT-Xen [5, 13], a real-time scheduling framework for the Xen hypervisor [14]. This section introduces Xen and RT-Xen as background. We provide a broader review of related work in Section VII.

A. Xen

Xen [14] is a widely used open-source hypervisor. A Xen-based virtualization system includes a privileged administration VM (i.e., Domain 0) and multiple unprivileged VMs

[1] Although other hypervisors (with certification capability, e.g. [10, 12]) are more frequently adopted in industrial applications, they are often proprietary and inaccessible to us. Xen is an open-source hypervisor that allows us to experiment with new designs that may be extended to industrial hypervisors.

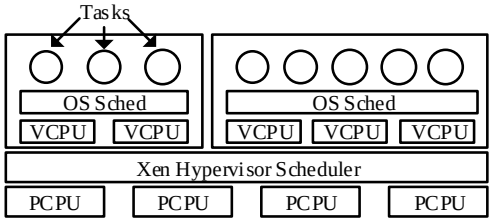


Fig. 1: Scheduling Architecture of Xen

(i.e., guest domains). The privileged VM is used by system operators to manage the unprivileged VMs. Each VM has its own operating system that schedules its tasks. We use the terms domain and VM interchangeably in the rest of this paper.

A two-level hierarchical scheduling framework is used in a virtualized host based on Xen, as in most virtualization systems. A VM runs on multiple *virtual CPUs (VCPUs)*, which are scheduled on multiple *physical CPUs (PCPUs)* by the hypervisor. Tasks in a VM are scheduled on VCPUs by the guest OS of the VM.

B. RT-Xen and Its Limitations

The RT-Xen project [5, 13] developed a suite of real-time schedulers for the Xen hypervisor, including the RTDS scheduler [13] that was adopted in Xen in 2015. Industry has shown increasing interest in RTDS. For example, the Xen Automotive project [11] is using the RTDS scheduler to achieve real-time performance for automotive applications.

When the RTDS scheduler is used in the Xen hypervisor, the user must specify the *resource interface* of each VM. A resource interface $\omega = (N, B, P)$ specifies that the VM has N VCPUs, and that each VCPU should be executed no more than $B\mu s$ (budget) every $P\mu s$ (period).

To guarantee the budget and period assigned to each VCPU, RTDS treats each VCPU as a deferrable server, where the budget of the VCPU is consumed only when a task is running on the VCPU. RTDS schedules all VCPUs based on the global Earliest Deadline First (gEDF) policy.

To meet the desired real-time performance of the tasks in a VM, we may calculate its resource interface ω based on *Compositional Scheduling Analysis (CSA)* [15, 16]. Given (1) the periods and the *worst-case execution times (WCET)* of a set of periodic tasks and (2) the real-time scheduling policy of the guest OS, we can compute the resource interface ω of the VM using CSA (e.g., the Multi-processor Periodic Model [17]). If the underlying hypervisor scheduler (RTDS) satisfies the resource interface of the VM at run time, the tasks in this VM are expected to meet their deadlines [18]. If a host does not have enough CPU capacity to accommodate the resource interfaces of all VMs, it cannot guarantee the schedulability of the tasks based on CSA.

It is important to note that RT-Xen supports a *static* resource interface per VM, which must be configured when the VM is initialized. If the VM may change its workloads or real-time requirements at run time, the designer has to calculate the resource interfaces based on the *worst-case* scenarios. Such a static configuration can result in significant resource underutilization at run time except when all VMs experience their worst-case scenarios at the same time. In contrast, M2-Xen supports dynamic VMs through on-line reallocation of CPU resources based on *multi-mode* resource interfaces.

III. Problem Statement

A *multi-mode* virtualization system may operate in different *modes* at run-time. Each VM can have different sets of soft real-time tasks in different modes. The task set of V_k in mode m is $S_{k,m} = \{\tau_{k,m,i}\}$. A dynamic real-time system may switch its mode (e.g., triggered by a user command or external events) at run time. When the system changes from mode m to mode m', each VM will switch from its task set in mode m to that in mode m'. Each periodic task $\tau_{k,m,i}$ is characterized by $(C_{k,m,i}, T_{k,m,i})$, where C and T are WCET and period, respectively. The *utilization* of a task set $S_{k,m}$ is defined as $U_{k,m} = \sum_i \frac{C_{k,m,i}}{T_{k,m,i}}$.

For each VM, M2-Xen allows a user to specify multiple resource interfaces corresponding to its different modes. Specifically, V_k runs on N_k VCPUs; each VCPU of V_k has a budget $B_{k,m}$ and a period $P_{k,m}$ in mode m. The resource interfaces of V_k in mode m is specified as $\omega_{k,m} = (N_k, B_{k,m}, P_{k,m})$. The *bandwidth* of resource interface $\omega_{k,m}$ is defined as $W_{k,m} = N_k \times \frac{B_{k,m}}{P_{k,m}}$. Given a task set $S_{k,m}$ in mode m, VM V_k, and the real-time scheduling policy of the guest OS, we can use CSA to calculate the resource interface $\omega_{k,m}$.

A multi-mode virtualization system must meet the following requirements in response to a mode change.

- **Dynamic resource reallocation:** The system should dynamically change the resource interface and task set of each VM. When the system's mode changes from m to m', for each VM V_k, the system should change the resource interface and task set, from $\omega_{k,m}$ and $S_{k,m}$, to $\omega_{k,m'}$ and $S_{k,m'}$, respectively. The change in the resource interface consequently causes the underlying RTDS scheduler to reallocate CPU resources among the VMs.
- **Overload avoidance in VMs:** The system must avoid overloading a VM, i.e., the utilization of the task set and the mode management system in a VM cannot exceed the bandwidth of its resource interface. An overload in a VM can lead to deadline misses and even a kernel panic in the guest OS, when it runs out of CPU cycles allocated by the hypervisor.
- **Overload avoidance in the physical host:** The total bandwidth of all the VMs should not exceed the total number of PCPUs. If the total bandwidth exceeds the CPU capacity of a host, some VMs will not receive their required CPU bandwidth, which may lead to deadline misses and even a kernel panic.
- **Fast mode switching:** The system needs to complete the mode switch quickly so that applications can start

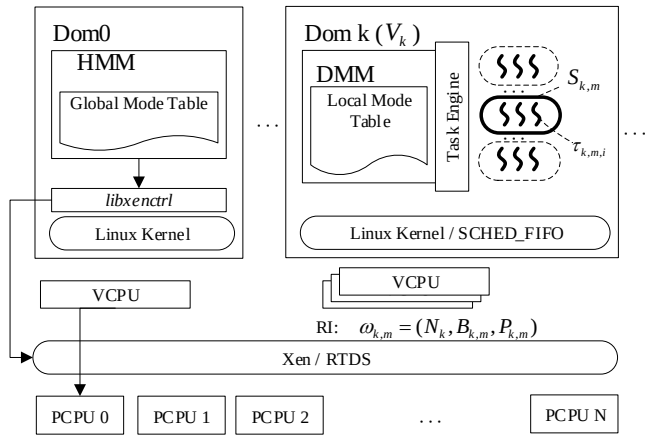


Fig. 2: Architecture of M2-XEN

operating in the new mode without significant delays [2].

IV. M2-Xen Architecture

In this section we first describe the main components of M2-Xen. We then explain the mode switching process, followed by our design and implementation choices.

A. System Architecture

Fig. 2 illustrates the system architecture of M2-Xen. M2-Xen comprises multiple VMs running on the Xen hypervisor in a multi-core host. The administrative VM, Domain 0, is pinned on a dedicated PCPU, PCPU 0. Each VM comprises soft real-time tasks running on a guest OS and a set of VCPUs.

The system administrator specifies a multi-mode system in a *Global Mode Table*. A *global mode* specifies a *local mode* for each VM, where each local mode is associated with a task set and the corresponding resource interface for the VM. Given a global mode m, we can find the task set $S_{k,m}$ and the resource interface $\omega_{k,m}$ of the VM k.

M2-Xen manages mode switching through a centralized *Host Mode Manager (HMM)* in Domain 0 and a *Domain Mode Manager (DMM)* in each VM. The HMM schedules and coordinates the local mode changes of the VMs based on the Global Mode Table. During mode switch, the HMM sends commands to the DMMs of VMs, requesting them to change the task sets to those of the new mode. The HMM is also responsible for changing the resource interfaces of VMs (via the *libxenctrl* library provided by the Xen hypervisor).

The DMM in each VM has a local mode table, which maps each local mode to a set of tasks. The DMM uses a task engine to change the task set in response to a mode change.

The HMM and DMMs communicate through TCP sockets to coordinate the mode switching process. The HMM contains a TCP server to which DMMs connect. When initialized, each DMM establishes a long-lived TCP connection to the HMM.

[2]While M2-Xen can meet the real-time performance requirements in different modes in steady state, it does not guarantee system schedulability during a mode change. M2-Xen is suitable for applications that can tolerate potential deadline misses during the transient mode switching process. However, for hard real-time systems, we plan to explore using mode switching protocols [19–25] in the future to avoid deadline misses during mode switching.

As DMM consumes CPU cycles to change task sets and to communicate with HMM, we need to account for the overhead when we compute the resource interface of a VM. Specifically, we can model the DMM as a periodic task whose period is the minimum inter-arrival time of mode switch requests. The WCET of a local mode switch is 100 μs in our current implementation. In practice, the mode switch frequency is usually not high and a VM can account for its DMM overhead in its resource interface, which allows M2-Xen to avoid CPU overload at both the VM and host levels despite the DMM overhead.

M2-Xen currently provide the mechanisms to change the modes of individual VMs at run-time. To support dynamic mode changes of individual VMs, M2-Xen may employ an online admission controller to perform compositional schedulability analysis and schedule VM mode changes on demand. A potential challenge in online admission control is that its schedulability analysis needs to be efficient in comparison to the required mode switching latency. The current M2-Xen avoided this issue by supporting a known set of global modes that are schedulable based on offline compositional analysis. This approach is applicable if the set of (global) system modes are well defined at design time or when the number of VMs and modes are limited.

B. Mode Switching Procedure

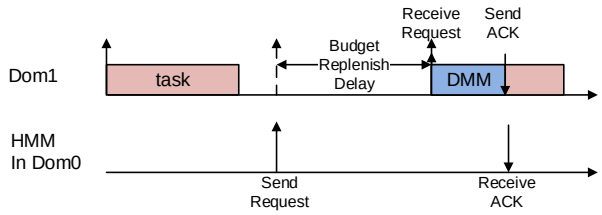
The mode switching procedure works as follows:

Step 1. Mode changes may be triggered by a predefined set of events from an unprivileged VM. The DMM of the VM then sends a predefined request to the HMM to trigger a mode change. This request contains a new global mode identifier. Based on the Global Mode Table, the HMM identifies the VMs whose local modes need to be changed.

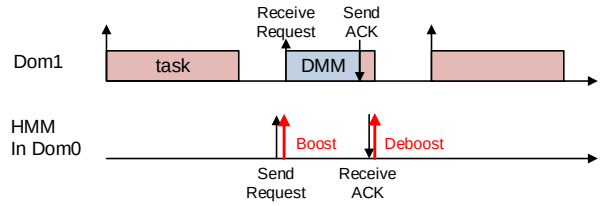
Step 2. The HMM schedules the mode changes of the VMs . The mode changes of different VMs may be scheduled to occur sequentially or in parallel. Importantly, to prevent *PCPU overload* the HMM must enforce the PCPU capacity constraint when scheduling the mode changes. That is, the total bandwidth of all the VCPUs should always remain below the total PCPU capacity throughout the mode switching process.

Step 3. To change the mode of a VM, if the bandwidth of the VM increases in the new mode, the HMM first changes the VM's resource interface before sending the mode change command to the VM. Otherwise, the HMM sends the mode change command to the DMM, waits for the completion acknowledgement from the VM, and changes the resource interface of the VM. Note that this policy prevents *VM overload* by ensuring that the bandwidth of the VM remains larger than the total utilization of its tasks during the mode change process.

Step 4. When a DMM receives a mode change command, the DMM uses the task engine to switch the old task set to that in the new mode. The DMM sends a completion acknowledgement to the HMM once it finishes changing the task set.



(a) Budget Replenishment



(b) Elimiate Budget Replenishment by Boosting VCPU

Fig. 3: Typical Timelines of Sending Mode Switch Request

Step 5. The HMM continues to schedule mode changes of the remaining VMs until all the VMs have completed their mode changes.

In the current implementation, M2-Xen accepts a new mode change request only after finishing the current mode change. In the future, we can explore other policies, e.g., (1) queuing mode change requests and serve them in order, or (2) allowing a new mode change request to preempt the ongoing mode change.

C. Implementation Issues

Configuration of Domain 0. In current implementation, M2-Xen allocates a full-capacity VCPU to Domain 0 and pins the VCPU onto a dedicated PCPU0. Domain 0 hosts both the HMM, which manages the mode changes for all VMs, and the inter-VM virtual network. Dedicating a PCPU to Domain 0 mitigates interference between Domain 0 and the applications running in the guest VMs.

Changing the task set. Each DMM uses a task engine to change its local task sets. Our current implementation employs the signal mechanism in Linux. Given a task set identifier, the task engine sends SIGSTOP signals and SIGCONT signals to the tasks to be stopped and started, respectively. However, M2-Xen is not limited to a specific implementation of the task engine, and different VMs may adopt different approaches to change their task sets.

Inter-VM communication. We choose to use TCP connections for message exchanges between the HMM and the DMMs. Using TCP makes our implementation portable across different operating systems and extensible to support mode switching in distributed systems in the future. As discussed in the next section, a challenge of TCP-based inter-VM communication is that its latency depends on VCPU scheduling. We present solutions to reduce inter-VM communication latency in the next section.

V. Reducing Mode Switching Latency

A key challenge to M2-Xen is to reduce the mode switch latency. Fast mode switch not only allows the system to enter the new node quickly, but also shortens the transient states in which tasks may potentially miss deadlines: the mode switching latency is largely attributed to communication latency between Domain 0 and the VMs, which can be heavily influenced by VCPU scheduling. When the VCPU of a VM runs out of budget and when a mode switch request arrives, the DMM received the request will not run until the VCPU's budget is replenished, a delay referred to as the *budget replenishment delay*, as shown in Fig. 3(a).

In this section, we frist analyse the delay experienced in a mode switching process, and then introduce approaches to reduce the delay through (1) user-level mode switch schedulings or (2) VM scheduling in the hypervisor.

The latency of changing a VM's mode comprises four parts: (1) the time required to send a local mode change request from the HMM to a DMM in a VM, (2) the overhead of changing the resource interface of a VM, (3) the latency of changing the task set of a VM, and (4) the latency of receiving a completion acknowledgement from the DMM. Note the above steps are ordered for the case when the bandwidth of a VM needs to be increased in order to avoid overload a VM. When the bandwidth of a VM needs to be reduced, the task set needs to be changed before the resource interface is reduced, as discussed in Section IV.

(1) The HMM sending a mode change request to a DMM. When a mode change starts, the HMM sends a mode change request message to each affected VM. The message includes a new resource interface and the local mode identifier. The latency of an inter-VM socket communication can be as small as several microseconds if the VCPU hosting the DMM, is running on a PCPU when the request message arrives. However, as shown in Fig. 3(a), if the DMM-VCPU has exhausted its budget and hence is not eligible to run, the DMM will not be able to receive a message from the HMM immediately. As a result, the message has to wait until the the DMM-VCPU's budget is replenished, as shown in Fig. 3(a). The budget replenishment delay significantly increase the latency of a mode change [3].

(2) Changing resource interface of the VM. The HMM uses *xl*, an administration tool, to change resource interfaces. Internally, the xl invokes a hypercall *Hypercall_DOMCTL*. The hypercall is wrapped into the *IOCTL* system call in Domain 0's kernel. Invoking such a hypercall will cause a context switch in the Linux Kernel and a VM-exit in Xen, which introduces overheads at the order of tens of microseconds.

(3) Changing task sets in VMs. Changing the task set to

[3]The communication delay induced by budget replenishment was also observed in Quest-V [26].

the one corresponding to the new mode can be implemented a domain-specific way. Our current implementation based on Linux *SIGNAL* takes no more than 100 μs to change a task set, as shown in Section VI.

(4) Receiving an acknowledgement from the DMM. As a PCPU is dedicated to Domain 0, there is no budget replenishment delay before the HMM receives the acknowledgement from the DMM.

As analyzed above, the mode switch latency can be dominated by the budget replenish delay of the VM. In the rest of this section, we first introduce three user-level mode switch scheduling methods to mitigate the mode switching delay and then present a technique to significantly reduce the budget replenish delay by modifying the hypervisor scheduler.

A. User-level Mode Switch Scheduling

To mitigate the impact of inter-VM communication delays, we propose three policies to schedule the mode change requests to different VMs.

Sequential. To avoid overloading the host during a mode transition, the HMM first sorts the mode change requests in increasing order of the difference between the new bandwidth and the old bandwidth of the VM. Then the HMM will change the mode of each VM one by one, starting from the VM that will incur the largest bandwidth decrease. The HMM does not send the next local mode change request until it has received the acknowledgement message from the previous VM.

Batch. The batch approach shortens the mode transition by allowing the HMM to issue mode switch requests to multiple VMs in parallel through non-blocking send over the TCP sockets. After collecting all the completion acknowledgements from those VMs, we then issue another batch of messages to VMs on the "increase" sublist. To avoid overloading in the physical host, the VMs in first batch that the HMM sends mode change requests are the ones whose bandwidthes are to be reduced.

Greedy. Like the batch approach, the Greedy approach also exploits concurrent requests to speed up the mode transition, but in a more aggresive manner. Rather than wait for *all* the acknowledgements from the "decrease" sublist, we can issue mode change requests to some of VMs on the "increase" sublist as soon as it can be performed without overloading the host. In order to avoid overloading the host, the HMM performs a bandwidth check every time we receive an completion acknowledgement from any VM whose bandwidth is decreased, and issue an increase message as soon as enough bandwidth becomes available.

B. Reduce Mode Switching Delay through VCPU Boost

While the user-level approaches described above can mitigate the impact of inter-VM communication delay through concurrent requests, they do not fundamentally reduce the inter-VM communication delay caused by the underlying VM scheduler. To this end, we introduce a *VCPU-boost* mechanism in the hpyervisor scheduler. The key idea of achieving fast message delivery is to temporarily boost the priority of the VCPU hosting the DMM of the receiving VM, when the HMM sends a mode switch request; and then return the DMM to its normal priority after the HMM receives the acknowledgement from the DMM. Following this insight, we propose and implement VCPU Boost feature based on the RTDS scheduler in Xen hypervisor.

RTDS architecture. The RTDS scheduler in Xen 4.8 is event-driven. The scheduler is triggered by the following events: (1) the VCPU budget replenishment event, when a VCPU replenishes its budget; (2) the VCPU budget exhaustion event, when a VCPU runs out of its current budget; (3) the VCPU sleep event, when a VCPU has no task running; and (4) the VCPU wake-up event, when a task starts to run on a VCPU.

Whenever the RTDS scheduler is triggered, it applies the global EDF algorithm to schedule VCPUs: A VCPU with an earlier deadline has higher priority; at any scheduling point, the scheduler picks the highest priority VCPU to run on a feasible PCPU. A PCPU is feasible for a VCPU if the PCPU is idle or has a lower-priority VCPU running on it.

The RTDS scheduler uses two global queues to keep track of all VCPUs' runtime information. The first is a *run queue*, which has all runnable VCPUs sorted in increasing order of their priorities. A VCPU is runnable if the VCPU still has budget in the current period. The second is a *depleted queue*, which keeps all VCPUs that run out of budget in their current periods. The VCPUs in the *depleted queue* are sorted based on their release time, i.e., the next budget replenish time.

Design of the VCPU boost feature. We introduce the *boost* priority level to the RTDS scheduler. A VCPU with boost priority always has a higher priority than non-boosted VCPUs. All boosted VCPUs have the same priority and are scheduled based on the First Come First Served (FCFS) policy. The number of boosted VCPUs is no larger than the number of PCPUs, so that the boosted VCPUs can always be scheduled immediately.

We introduce two hypercalls for the administrative VM (Domain 0), to change a VCPU's boost status: (1) a *boost hypercall*, which promotes a VCPU's priority to boost priority, and (2) a *de-boost hypercall*, which degrades a boosted VCPU's priority back to non-boost priority. We allow only Domain 0 to issue these two commands by installing rules in the Xen Security Model (XSM). Note that it is important to restrict the boost hypercall to Domain 0 to prevent VMs from abusing the boost feature. As the boost mechanism effectively grants the boosted VCPU additional CPU time beyond its resource interface, it is important for Domain 0 to minimize the boost duration. As shown in our experimental results, the time needed for a mode change of a single VM is limited to within 200 μs. The impact of boost on the resource allocation is therefore negligible for many applications. Nevertheless, the boost feature limits M2-Xen to soft real-time applications because it affects the accuracy of CPU allocation to VCPUs.

The RTDS scheduler with the VCPU boost feature has two new scheduling events: (1) a *VCPU boost event*, when a VCPU's priority is promoted to the *boost* priority by Domain

0; (2) a *VCPU de-boost event*, when a boosted VCPU's priority is degraded to that of a non-boosted VCPU.

The scheduler is triggered by these two boost-related events: (1) At the *VCPU boost event*, the scheduler will find the PCPU the lowest priority, remove the boosted VCPU from the global queue (which can be the run queue or the depleted queue), and schedule the boosted VCPU on the PCPU. A PCPU's priority is equal to the current running VCPU's priority on the PCPU. A PCPU without any VCPU running on it always has lower priority than a PCPU with a VCPU running on it; (2) At the *VCPU de-boost event*, the scheduler will update the de-boosted VCPU's deadline and its budget. The scheduler will insert the de-boosted VCPU back into the run queue if the de-boosted VCPU still has budget in the new period, or into the depleted queue otherwise.

Boost. We leverage the proposed VCPU boost feature to further reduce mode change latency. The HMM first sorts the mode change requests in ascending order of the difference between the new bandwidth and the old bandwidth of the VMs. Then the HMM changes the mode of each VM one by one, based on the sorted list of the mode change requests. After sending the mode switch request to a VM's DMM, the HMM immediately boosts the DMM-VCPU. Once the HMM receives the acknowledgement from the DMM, the HMM de-boosts the DMM-VCPU as illustrated in Fig. 3(b).

As a boosted VCPU receives extra CPU cycles, it is therefore necessary to reserve CPU bandwidth to accommondate the worst-case boost time in order to avoid host overload, i.e., the total bandwidth of all VCPUs should be no larger than the total number of PCPUs (as defined in Section III). We can account for the additional bandwidth consumed by a boosted VCPU as a hypothetical VCPU. Based on the minimum inter-arrival time of mode switch request (the period) and the worst-case mode switch times of the VMs (the budget). To avoid host overload, we reserve the total bandwidth of the hypothetical VCPU of all the VMs. Furthermore, the HMM can use a timeout mechanism, to avoid a malfunctioning DMM from occupying a PCPU for an excessive amount of time. If the DMM fails to send an acknowledgement before the timeout, and hence enforcies an upper bound of the boost time of each VM.

C. Comparison of Different Approaches

As an example, Fig. 4 presents the mode switch timelines under different policies in real experiment. We used 6 VMs (exclude Dom 0) to share a 15-PCPU host. Each VM got 3 VCPUs. The timelines are plotted based on real experiment trace files. In this mode switch, Dom 1, 3, 4, and 5 decrease their bandwidth, while Dom 2 and 6 increase theirs. When decreasing a VM's bandwidth, the HMM sends a mode switch request to each DMM, receives the acknowledgement, then changes the resource interface. When increasing bandwidth, the resource interface should be changed before sending the request to the DMM. The Sequential approach changes VMs modes one by one, which introduces extra delay and is also vulnerable to budget replenishment delays. The Batch approach exploits parallelism to mitigate the budget replenish issue: "decreasing" requests are sent in a batch, followed by the "increasing" sublist. The Greedy approach opportunistically increases resource interfaces of VMs as soon as possible, and thus yields a better result than Sequential or Batch. The Boost approach avoids budget replenishment delays by boosting VCPUs on demand. Boost consistently outperforms the others by providing a lower overall mode switch latency. Boost also reduces the variability of mode switch delays.

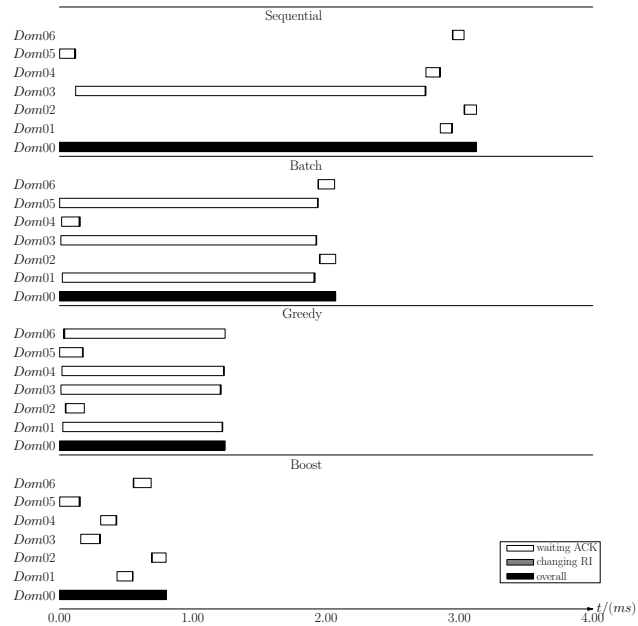


Fig. 4: Example Mode Change Timelines: 6 3-VCPU VMs

Compared to the user-level approaches, VCPU boost can drastically reduce mode switch latency, but it requires patching the hypervisor. The choice between the user-level approaches and the Boost approach therefore involve a trade-off between performance and hypervisor modifications.

VI. Evaluation

To evaluate M2-Xen, we conducted an extensive set of experiments, using randomly generated real-time workloads and micro-benchmarks. We had three main objectives: (1) evaluate the real-time performance improvement (in terms of missed deadlines) of M2-Xen over vanilla Xen, which is Xen with static configuration, for multi-mode systems; (2) evaluate the latency overhead incurred by mode-switch operations with micro-benchmarks; and (3) compare the overall mode-switch latencies of four fast mode switch policies.

A. Experimental Setup

Hardware. We conducted the experiment on a machine with an Intel E5-2683v4 16-core chip and 64GB memory. We disabled hyper-threading and power saving features and fixed the CPU frequency at 2.1 GHz to reduce the system's unpredictability as in [13, 27, 28].

Hypervisor and VMs. We used Xen 4.8.0 with the RTDS scheduler as the baseline hypervisor. We used our modified

Xen with the VCPU boost feature for the overall latency evaluation. We used Linux 4.4.19 for all VMs. We configured Domain 0 with one full-capacity VCPU pinned to one dedicated core, i.e., PCPU 0. For each VM, We set the irq-affinity of the network interface (*eth0-q0*) to the VM's VCPU 0 and disabled *irqbalance*. In the deadline miss experiment, we used 12 VMs, each of which had 3 VCPUs. In other experiments, we used 6 - 12 VMs, each of which had multiple VCPUs. In each experiment, we randomly generated test cases, each of which had different numbers of VCPUs and different resource interfaces for the VMs.

Mode change manager. The HMM and the mode request generator were deployed in Domain 0. The HMM and the DMM processes were scheduled by the Linux SCHED_FIFO scheduler with a priority of 98. The mode request generator randomly generated mode switch requests that were sent to the HMM through a socket. The DMM in each VM was pinned onto the VCPU 0 of the VM.

Real-time workload. Each real-time workload was an independent process. We set real-time workloads' priorities to 1 - 95 under the Linux *SCHED_FIFO* scheduler so that the scheduler becomes the Rate Monotonic scheduler to schedule the real-time workloads. The WCET and the period of each task were determined by test cases.

Test case generation. We randomly generated test cases each comprising a set of VMs hosted by a multicore host. Each test case has two system modes. Each VM has different task sets and resource interfaces in different modes. The task sets in both modes are schedulable based on CSA performed using the CARTS tool [29].

The test case generator guaranteed that each generated test case had the following properties: (1) the total bandwidth of both modes are the same; (2) the number of tasks generated for a VM V_k with N_k VCPUs was no smaller than the number of VCPUs N_k; (3) the system was claimed schedulable in each mode by CARTS [29]; (4) the resource interfaces and task sets of a VM are different in different modes. In case a randomly generated test case violated these properties, the test case generator discarded the test case and regenerated one.

B. Real-Time Performance in Multiple Modes

We compared M2-Xen against Xen in term of real-time performance of multi-mode systems. As baselines for comparison, we considered two representative approaches to configure resource interfaces of VMs on Xen: (1) based on the *initial mode*, or (2) based on the *worst-case mode*. When the resource interfaces were configured based on the initial mode, the system may suffer deadline misses after a mode switch if a VM requires larger bandwidth in the new mode. On the other hand, Xen can avoid no deadline miss in any mode if the resource interface of each VM is configured based on the largest bandwidth in any mode. However, the system would be forced to over-provision resources unless all VMs experience their respective worst-case workloads in the same mode.

We first compared the empirical performance of M2-Xen against Xen when the latter was configured based on its initial mode. In this set of experiments, we randomly generated 40 test cases each has two modes. In each mode, each test case has 12 3-VCPU VMs running on 15 PCPUs, and each VM contained a set of randomly generated tasks whose periods was uniformly distributed in $[100, 1000]ms$. The total VCPU bandwidth of all VMs was 8.0 in each mode, but the required bandwidth of each individual varied between modes. The period of each VMs resource interface is $5ms$. Under M2-Xen each VM had a resource interface for each mode. In contrast, under Xen each VM had a fixed resource interface computed based on the initial system mode (Mode 0).

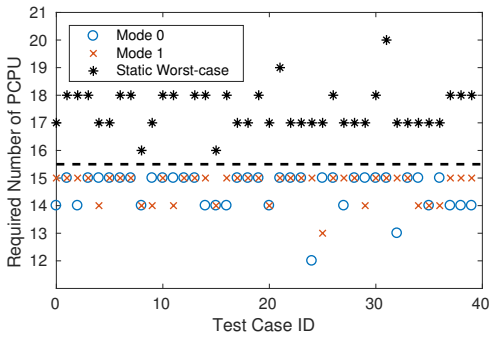


Fig. 5: Number of PCPU required for different test cases

We ran each test case for $200s$. The system started in Mode 0 and switched to Mode 1 at $100s$. The system recorded deadline misses during the experiments. As expected M2-Xen consistently met deadlines in both system modes, while Xen suffered deadline misses after mode changes. Specifically, the system experienced deadline misses in 6 out of 40 test cases under Xen. In contrast, no test case witnessed any deadline miss under M2-Xen because it dynamically changed the resource interfaces in response to the mode switches.

We then assessed the cost of over-provisioning resources in Xen based on each VM's worst-case bandwidth need in comparison to the multi-mode interfaces supported by M2-Xen. In this analysis, each VM was assigned the resource interface with the largest bandwidth in both system modes. As a result, tasks in each VM are schedulable in all modes, but this approach led to resource over-provisioning assuming all VMs experience their respective worst-case workloads at the same time, a rare situation in practice. To guarantee the schedulability of the entire system, we used CARTS [29] to compute the required number of PCPUs for these resource interfaces. If the required number of PCPUs was no larger than the available number of PCPUs (i.e., 15 PCPUs in our setting), the system was schedulable. In contrast, M2-Xen provisions different resource interfaces for each VM in different modes. The system is schedulable as long as the required number of PCPUs are smaller than the number of available PCPUs in each mode.

For the same 40 test cases used in the last set of experiments, we computed the required number of PCPUs for each test case (denoted *Static Worst-case* in Fig. 5). Notably, none of the 40 test cases was schedulable on the 15 PCPUs available on

TABLE I: Global Mode Setting for Micro Benchmark

Global Mode	**Dom 0** (B, P)	**Dom 1** (B, P)
0	(10ms, 10ms)	(10ms, 10ms)
1	(10ms, 10ms)	(4ms, 10ms)

our experimental platform. In contrast, all 40 test cases were schedulable in both Mode 0 and 1, which means that they were schedulable under M2-Xen that can dynamically switch to the resource interfaces corresponding to the current mode.

In summary, this set of experiments and analyses have shown that M2-Xen can significantly maintain real-time performance by adapting to mode changes while avoiding resource over-provisioning.

C. Latency Breakdown among Mode-Switch Operations

We measured the latency introduced by the mode-switch procedures with micro benchmarks. We created a VM with one VCPU and two modes. Each mode had 4 randomly generated tasks. We had a Domain 0 to run the HMM and the mode switch generator. The resource interface for the VM and the Domain 0 in both modes are shown as Table I. We toggled the system mode between Mode 0 and Mode 1 500 times with the mode-switch interval randomly picked between 200 ms and 400 ms. We measured the delay introduced by each mode-switch procedure at each mode switch event. We classified the delay into two cases: (1) case 1 when the system changed from Mode 0 to Mode 1; and (2) case 2 when the system changed from Mode 1 to Mode 0.

Fig. 6 shows the latency of the four procedures that contribute to the overall mode-switch latency: (1) Send, which sends a mode change request from the HMM to a DMM; (2) Signal, which uses the task engine to change the task set; (3) Receive, which receives an acknowledgement from the VM; and (4) Change, which uses the HMM to change the VM's resource interface.

We observed in Fig. 6 that the latencies of the Signal, Receive, and Change procedures are relatively short and stable. In both cases, the latencies stay within $100\mu s$ most of the time. We also observed that the Send procedure did not suffer a significant delay when system changed from Mode 0 to 1. This is because Domain 1 had a full-capacity VCPU in Mode 0 at the mode switch, which eliminated the budget replenishment overhead in the Send procedure and allowed the DMM to immediately process an inter-VM message from the HMM.

However, when the system changed from Mode 1 to 0, Domain 1 had a partial-capacity VCPU at the mode switch. The VCPU running the DMM may exhaust its budget and be de-scheduled, resulting in budget replenishment delay of up to 6 ms, which was consistent with the upper bound of $P - B = 10ms - 4ms = 6ms$. This latency dominated the overall mode change latency. The long tail of the Send latency increased not only the worst-case latency but also the average latency, as shown in Fig. 7.

D. Overall Mode Change Latency

This experiment aims to evaluate the budget replenishment delay of the Send procedure for four different mode switch policies: Sequential, Batch, Greedy, and Boost.

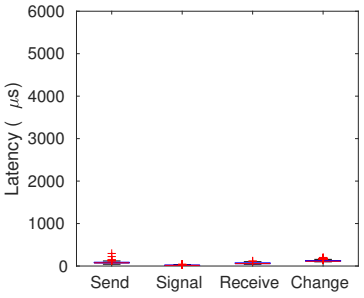
We had 15 PCPUs for guest VMs. Each VM had 3 VCPUs. We generated test cases by varying the total bandwidth W_{total} and number of VMs. We had 10 test cases for each combination of total bandwidth and number of VMs. We had 10 test cases for each $\{W_{total}, \#VMs\}$ setting. For each test case, we switched the system mode between Mode 0 and 1 50 times. The interval between two mode switch requests was randomly choosed [2000, 3000] ms. We ran each test case with different mode switch policies and measured 100 overall mode change latencies for each test case. We had 1000 latency results in total for each test case.

Latency distribution. Fig. 8 showed the overall latency in boxplot for the four mode switch policies in different bandwidth settings. We observed that the median latency increased monotonically with the number of VMs. This is reasonable because the HMM was implemented as a single thread server, and all communication and resource interface management were handled on PCPU 0 serially. The more VMs to manage, the longer median overall latency was expected. We observed that the Greedy approach outperformed the other two user-level approaches. In general, the Greedy approach had smaller median latency than the other two approaches, because it leveraged parallelism in an opportunistic manner. However, all three user-level approaches witnessed large variance of latencies.

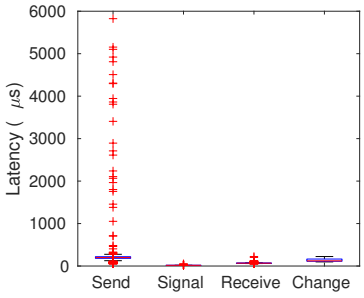
We also observed that the Boost approach significantly outperformed all three user-level approaches across all system bandwidths. To be specific, the Boost approach's overall latencies were consistently smaller than 1.5 ms, while the best user-level approach, i..e, the Greedy approach, had its largest overall latency larger than 10 ms (see the results for 11 VMs in Fig. 8 (a)). This is reasonable because the Boost approach avoided the extra delay caused by the budget replenishment.

Fig. 9(a) showed the 99th percentile latencies of 1000 samples across different number of VMs. We observed that the 99th percentile latency under all three user-level approaches was always larger than 5 ms, while it was less than 1.5 ms under the Boost approach. This demonstrated that user-level approaches suffered from long budget replenishment delays which were significantly reduced by the Boost approach.

In order to show the distribution of the latencies of the four mode switch policies, we picked the results for the setting that had 12 VMs and 9.0 total bandwidth. We plotted the CDF of the 1000 latency samples for all four policies in Fig. 9(b). We observed that all the latency samples of the Boost approach were distributed in a narrow range $[1.09, 1.49]$ ms. This demonstrated that the Boost approach could provide very stable and small mode switch latencies. In comparison, the Greedy approach, the best user-level approach, had the minimal latency 2.25 ms, and had more than 10% latency



(a) Change from Mode 0 to Mode 1



(b) Change from Mode 1 to Mode 0

Fig. 6: Boxplot of Mode Change Latencies

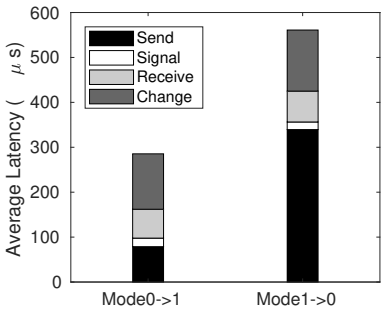
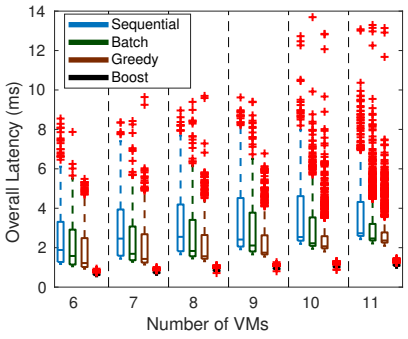
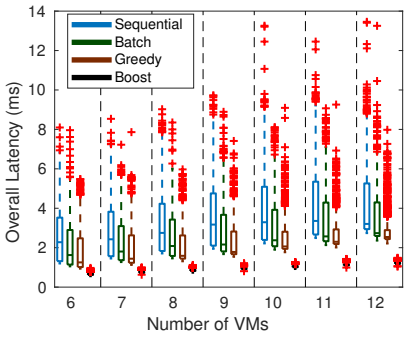


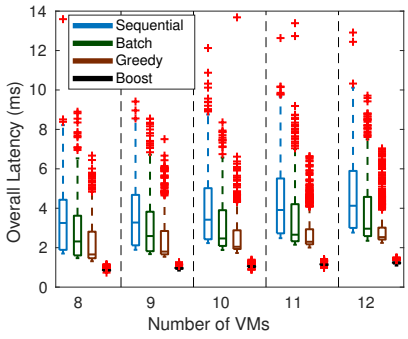
Fig. 7: Overall Mode Change Latency



(a) $W_{total} = 7.0$

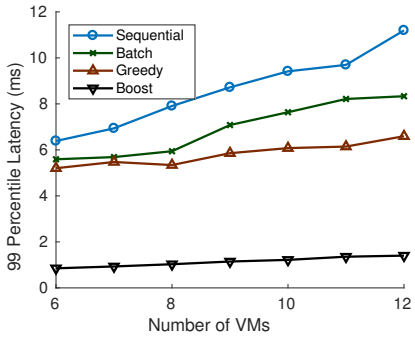


(b) $W_{total} = 8.0$

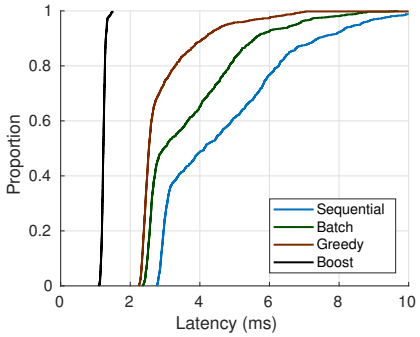


(c) $W_total = 9.0$

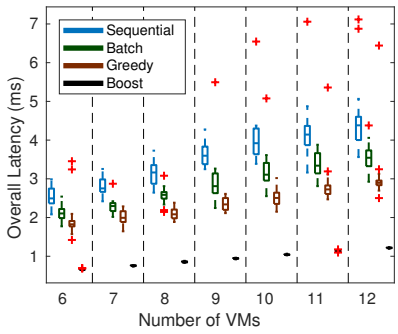
Fig. 8: Boxplot of Mode Change Latencies



(a) 99th Percentile Latency, $W_{total} = 8.0$



(b) Empirical CDF, $W_{total} = 9.0$, 12 VMs



(c) Boxplot of Average Latency Among Testcases

Fig. 9: Latency Distributions for different settings

samples greater than 4 ms, which is $4/1.49 = 2.68\times$ larger than that of the Boost approach. This again demonstrated the effectiveness of the Boost approach in eliminating the budget replenishment delay.

Performance among different mode settings. Since the mode setting affects the distribution of budget replenishment delay during mode switch, a VCPU with higher bandwidth suffers from less budget replenishment penalty. In this experiment, we evaluated the stability of our mode change policies: the mode switch latency of a stable mode switch policy should not be seriously affected by different mode settings. We randomly generated test cases that had different resource interface settings for this experiment.

We created a boxplot to evaluate the impact of VCPU bandwidth on the average latency among different policies. Each data point was the average latency of 50 delay samples at mode switches of a test case. Fig. 9(c) showed the boxplot of four mode switch policies across different number of VMs.

We observed that the Batch and the Greedy approaches were less sensitive to mode settings when compared to the Sequential. This is because these two approaches leveraged the parallelism mitigate the impact of the variable budget replenishment delay.

We also observed that the Boost approach was the most robust one: Fig. 9(c), showed that the latencies of the Boost approach were aggregated in a very small range that is less

than 100 μs. This is because the Boost approach boosted the VCPU hosting the DMM and scheduled it immediately whenever the HMM sent a mode switch request, making the overall mode switch latency insensitive to resource interfaces.

When the budget replenishment delay was removed, the mode switch latency was dominated by the cost of invoking hypercall to change resource interfaces in Domain 0. Since changing a VM's resource interface usually took constant time and the HMM changed resource interfaces sequentially, the overall latency was expected to have linear relation with the number of VMs, which was confirmed in our empirical results of the Boost approach.

VII. Related Work

A. Real-Time Virtualzation Approaches

Real-time virtualization has received significant attention as a promising approach for embedded system integration and latency-sensitive cloud computing [1]. KVM [30], as type-II hypervisor, incorporates itself within the host OS. VCPUs are treated as processes and are scheduled together with other tasks in the host. In KVM, it is possible to apply a real-time scheduler [31, 32] for VCPUs, thus achieving hierarchical real-time scheduling. The deadline-based real-time scheduler [6] in the IRMOS project [33] is an instance of hierarchical real-time scheduling based on KVM. Another KVM-based scheduler, ExVM [34], adopts flattened scheduling design [35]. It exposes scheduling information within VMs to the host scheduler. While Flattening may potentially improve the performance and flexibility of multi-mode scheduling, an advantage of hierarchical scheduling approach as in M2-Xen is to avoid exposing task-specific information within a VM, which can be appealing to a subsystem vendor who may not want to disclose their proprietary design and intellectual property.

Quest-V [36, 37], a separation kernel, divides physical resources, such as cores and I/O devices, into separated sandboxes. Each sandbox runs its own kernel and schedules tasks directly on the cores in this sandbox. With this architecture, Quest-V establishes a predictable model and addresses the communication delay due to scheduling [26].

The MARACAS scheduling framework [7] addresses memory-aware scheduling, shared cache, and memory bus contention, for multicore scheduling. MARACAS throttles the execution of threads running on specific cores when memory contention exceeds a certain threshold.

While the aforementioned work focused on developing novel architectures and scheduling policies for real-time virtualization, they largely assumed static configuration for real-time VMs. In contrast, M2-Xen incorporates dynamic mode switching and run-time resource allocation within a conventional two-level hierarchical scheduling framework provided by RT-Xen [5, 13].

B. Real-Time Mode Change Protocols

Burns [38] gave an overview of mode change in real-time systems. Switching between two modes requires eliminating some tasks in the old mode and establishing other tasks in the new mode. For hard real-time systems, the greatest challenge is to guarantee hard real-time requirements during the transition, when the tasks of the old and the new modes may be scheduled simultaneously. For uniprocessor, the *Idle Time Protocol* [39], the *Maximum-Period-Offset Protocol* [40], and the *Minimum Single Offset Protocol* [25] are existing synchronization protocols. Research has also focused on multi-core real-time mode change [41]. Although multi-core mode switch protocols with task-level mode switch have been proposed [19–25], no relevant study has focused on solution that leverage mode change protocols for virtualized systems, which can change the resource interface and task set. In M2-Xen, we do not guarantee deadline misses are avoided entirely during the transition, although reducing the mode switching latency mitigates the potential for deadline misses during the mode switch. Our current system is therefore designed for soft real-time systems.

Timeliness guarantees for M2-Xen during mode transitions is an interesting yet challenging future direction. To ensure the schedulability of tasks during mode transitions, (1) the resource interface that each VM exposes to the hypervisor must be sufficient to account for the effects of task-level mode changes on the tasks resource demands; and (2) the analysis at the hypervisor level must consider the impact of VM-level mode changes on the timing behaviors of the VCPUs of the VMs; neither of these is supported by current interface models and analysis. Prior work on multi-mode interfaces and compositional analysis such as [3] can serve as a starting point towards this direction; however, this line of work targets uniprocessors only and assumes the same mode change semantics at both VM and task levels, and thus it must be substantially extended to work with M2-Xen. We intend to build on this experimental work to develop more rigorous mode switching protocols for Xen in the future.

VIII. Conclusions

We have designed and implemented M2-Xen, a virtualization platform for dynamic real-time systems. In contrast to existing real-time virtualization platforms supporting static resource allocations, M2-Xen can adapt resource allocations among real-time virtual machines in response to system mode changes. As a result, M2-Xen can maintain real-time performance for multi-mode real-time systems without over-provisioning resources. Furthermore, M2-Xen avoids transient system overloads during mode switches and employs user-space and hypervisor scheduling techniques to reduce mode switching latency. M2-Xen has been implemented and evaluated in Xen 4.8 with the RTDS scheduler. Experiments on a 16-core host demonstrated the efficacy, efficiency, and scalability of M2-Xen in comparison to existing static real-time scheduling approaches in Xen.

Acknowledgment

This research was supported in part by ONR grant N000141612108 and the Fullgraf Foundation.

References

[1] M. García-Valls, T. Cucinotta, and C. Lu, "Challenges in real-time virtualization and predictable cloud computing," *Journal of Systems Architecture*, vol. 60, no. 9, pp. 726–740, 2014.

[2] I. Shin and I. Lee, "Compositional real-time scheduling framework," in *Real-Time Systems Symposium (RTSS)*, 2004.

[3] L. T. Phan, I. Lee, and O. Sokolsky, "Compositional analysis of multi-mode systems," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2010.

[4] A. Boudjadar, A. David, J. H. Kim, K. G. Larsen, M. Mikučionis, U. Nyman, and A. Skou, "Hierarchical scheduling framework based on compositional analysis using uppaal," in *International Workshop on Formal Aspects of Component Software*, 2013.

[5] S. Xi, J. Wilson, C. Lu, and C. Gill, "RT-Xen: towards real-time hypervisor scheduling in xen," in *Embedded Software (EMSOFT)*, 2011.

[6] F. Checconi, T. Cucinotta, and M. Stein, "Real-time issues in live migration of virtual machines," in *Euro-Par 2009 – Parallel Processing Workshops*, 2009.

[7] Y. Ye, R. West, J. Zhang, and Z. Cheng, "Maracas: A real-time multicore vcpu scheduling framework," in *Real-Time Systems Symposium (RTSS)*, 2016.

[8] "Solution accelerators and services from globallogic," https://cdn10.globallogic.com/wp-content/uploads/2016/12/GlobalLogic-Nautilus-Platform.pdf, accessed: 2017-03-08.

[9] "Infotainment and telematics solutions with renesas r-car," https://goo.gl/sAMdpi, accessed: 2017-10-06.

[10] "Integrity multivisor for IVI," http://www.ghs.com/products/multivisor_infotainment.html, accessed: 2016-10-12.

[11] "Xen hypervisor targets automotive virtualization role," https://goo.gl/Uwj8U3/, accessed: 2017-10-06.

[12] "Mentor embedded hypervisor," ihttps://www.mentor.com/embedded-software/hypervisor, accessed: 2018-01-21.

[13] 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 *Embedded Software (EMSOFT)*, 2014.

[14] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *ACM SIGOPS operating systems review*, 2003.

[15] A. Easwaran, I. Shin, and I. Lee, "Optimal virtual cluster-based multiprocessor scheduling," *Real-Time Systems*, vol. 43, no. 1, pp. 25–59, 2009.

[16] S. K. Baruah and N. Fisher, "Component-based design in multiprocessor real-time systems," in *International Conference on Embedded Software and Systems (ICESS)*, 2009.

[17] I. Shin, A. Easwaran, and I. Lee, "Hierarchical scheduling framework for virtual clustering of multiprocessors," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2008.

[18] M. Xu, L. T. X. Phan, O. Sokolsky, S. Xi, C. Lu, C. Gill, and I. Lee, "Cache-aware compositional analysis of real-time multicore virtualization platforms," *Real-Time Systems*, vol. 51, no. 6, pp. 675–723, 2015.

[19] J. Lee, H. S. Chwa, L. T. Phan, I. Shin, and I. Lee, "MC-ADAPT: Adaptive task dropping with task-level mode switch in mixed-criticality scheduling," in *Embedded Software (EMSOFT)*, 2017.

[20] D. de Niz and L. T. Phan, "Partitioned scheduling of multi-modal mixed-criticality real-time systems on multiprocessor platforms," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[21] A. Burns and S. Baruah, "Towards a more practical model for mixed criticality systems," in *Workshop on Mixed-Criticality Systems (colocated with RTSS)*, 2013.

[22] X. Gu, A. Easwaran, K.-M. Phan, and I. Shin, "Resource efficient isolation mechanisms in mixed-criticality scheduling," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2015.

[23] J. Ren and L. T. X. Phan, "Mixed-criticality scheduling on multiprocessors using task grouping," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2015.

[24] P. Huang, P. Kumar, N. Stoimenov, and L. Thiele, "Interference constraint grapha new specification for mixed-criticality systems," in *Emerging Technologies & Factory Automation (ETFA)*, 2013.

[25] J. Real and A. Crespo, "Mode change protocols for real-time systems: A survey and a new proposal," *Real-time systems*, vol. 26, no. 2, pp. 161–197, 2004.

[26] Y. Li, R. West, Z. Cheng, and E. Missimer, "Predictable communication and migration in the quest-v separation kernel," in *Real-Time Systems Symposium (RTSS)*, 2014.

[27] H. Kim and R. Rajkumar, "Real-time cache management for multi-core virtualization," in *Embedded Software (EMSOFT)*, Oct 2016, pp. 1–10.

[28] M. Xu, L. T. X. Phan, H. Y. Choi, and I. Lee, "vcat: Dynamic cache management using cat virtualization," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017.

[29] L. T. X. Phan, J. Lee, A. Easwaran, V. Ramaswamy, S. Chen, I. Lee, and O. Sokolsky, "Carts: A tool for compositional analysis of real-time systems," *SIGBED Rev.*, vol. 8, no. 1, pp. 62–63, Mar. 2011.

[30] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the Linux symposium*, 2007.

[31] A. Bastoni, B. B. Brandenburg, and J. H. Anderson, "An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers," in *Real-Time Systems Symposium (RTSS)*, 2010.

[32] G. Gracioli, A. A. Fröhlich, R. Pellizzoni, and S. Fischmeister, "Implementation and evaluation of global and partitioned scheduling in a real-time os," *Real-Time Systems*, vol. 49, no. 6, pp. 669–714, 2013.

[33] T. Cucinotta, F. Checconi, G. Kousiouris, K. Konstanteli, S. Gogouvitis, D. Kyriazis, T. Varvarigou, A. Mazzetti, Z. Zlatev, J. Papay *et al.*, "Virtualised e-learning on the irmos real-time cloud," *Service Oriented Computing and Applications*, vol. 6, no. 2, pp. 151–166, 2012.

[34] M. Drescher, V. Legout, A. Barbalace, and B. Ravindran, "A flattened hierarchical scheduler for real-time virtualization," *Embedded Software (EMSOFT)*, 2016.

[35] A. Lackorzyński, A. Warg, M. Völp, and H. Härtig, "Flattening hierarchical scheduling," in *Proceedings of the tenth ACM international conference on Embedded software*. ACM, 2012, pp. 93–102.

[36] Y. Li, R. West, and E. Missimer, "A virtualized separation kernel for mixed criticality systems," in *ACM SIGPLAN Notices*, vol. 49, no. 7, 2014, pp. 201–212.

[37] E. Missimer, R. West, and Y. Li, "Distributed real-time fault tolerance on a virtualized multi-core system," *Operating Systems Platforms for Embedded Real-Time Application*, 2014.

[38] A. Burns, "System mode changes-general and criticality-based," in *Proc. of 2nd Workshop on Mixed Criticality Systems (WMC)*, 2014, pp. 3–8.

[39] K. Tindell and A. Alonso, "A very simple protocol for mode changes in priority preemptive systems," *Universidad Politécnica de Madrid, Tech. Rep*, 1996.

[40] C. Bailey, "Hard real-time operating system kernel. investigation of mode change," *British Aerospace Systems Ltd*, 1993.

[41] M. S. Mollison, J. P. Erickson, J. H. Anderson, S. K. Baruah, and J. A. Scoredos, "Mixed-criticality real-time scheduling for multicore systems," in *Computer and Information Technology (CIT)*, 2010.