vMPCP: A Synchronization Framework for Multi-Core Virtual Machines

Hyoseung Kim*, Shige Wang†, Ragunathan (Raj) Rajkumar*
*Carnegie Mellon University
†General Motors R&D
hyoseung@cmu.edu, shige.wang@gm.com, raj@ece.cmu.edu

***Abstract*—The virtualization of real-time systems has received much attention for its many benefits, such as the consolidation of individually developed real-time applications while maintaining their implementations. However, the current state of the art still lacks properties required for resource sharing among real-time application tasks in a multi-core virtualization environment. In this paper, we propose vMPCP, a synchronization framework for the virtualization of multi-core real-time systems. vMPCP exposes the executions of critical sections of tasks in a guest virtual machine to the hypervisor. Using this approach, vMPCP reduces and bounds blocking time on accessing resources shared within and across virtual CPUs (VCPUs) assigned on different physical CPU cores. vMPCP supports periodic server and deferrable server policies for the VCPU budget replenish policy, with an optional budget overrun to reduce blocking times. We provide the VCPU and task schedulability analyses under vMPCP, with different VCPU budget supply policies, with and without overrun. Experimental results indicate that, under vMPCP, deferrable server outperforms periodic server when overrun is used, with as much as 80% more tasksets being schedulable. The case study using our hypervisor implementation shows that vMPCP yields significant benefits compared to a virtualization-unaware multi-core synchronization protocol, with 29% shorter response time on average.**

I. Introduction

The adoption of multi-core CPUs in real-time embedded systems is increasing dramatically. This trend creates the opportunity to consolidate multiple real-time applications into a single hardware platform. Such consolidation leads to a significant reduction in space and power requirements while also reducing installation, management and production costs by reducing the number of CPUs and wiring harnesses among them.

Virtualization plays a key role in the successful consolidation of real-time applications. Specifically, each application in use could have been developed independently by different vendors, but can maintain its own implementation by using virtualization. Since re-programming or re-structuring of real-time embedded software requires going through a rigorous and expensive re-certification process, virtualization offers multiple benefits. In addition, virtualization can also provide fault isolation, IP (intellectual property) protection and license segregation in consolidated embedded systems.

Modern virtualization solutions generally provide a two-level hierarchical scheduling structure. Each virtual machine (VM) has one or more virtual CPUs (VCPUs) on which tasks of that VM are scheduled. Then, VCPUs are scheduled by the hypervisor on physical CPUs (PCPUs). Many researchers in the real-time systems community have studied such hierarchical scheduling for both uni-core systems [11, 12, 30, 35] and multi-core systems [20, 33]. Recently, some researchers have applied the real-time hierarchical scheduling theory to virtualization environments, such as RT-Xen [38].

While real-time system virtualization can benefit from previous work on hierarchical scheduling, resource sharing and task synchronization issues still remain an open research question. Consolidating multiple tasks into a single hardware platform inevitably introduces the sharing of logical and physical resources, i.e. shared memory for communication, network stacks and I/O devices. The more real-time tasks are consolidated as the number of processing cores increases, the more we need a synchronization mechanism with bounded blocking times for multi-core real-time virtualization. Unfortunately, multi-core synchronization mechanisms designed for non-hierarchical scheduling, such as MPCP [25, 28] and MSRP [13], can lead to excessive blocking times due to the preemption and budget depletion of VCPUs. Available solutions in the uni-core hierarchical scheduling context [4, 6, 12] have not yet been extended to multi-core platforms. More importantly, in current virtualization solutions, the hypervisor is unaware of the executions of critical sections of application tasks within VCPUs and there is no systematic mechanism to do so.

In this paper, we propose a virtualization-aware multiprocessor priority ceiling protocol (vMPCP) and its framework to address the synchronization issue in multi-core virtualization. vMPCP extends the well-known multiprocessor priority ceiling protocol (MPCP) to the multi-core two-level hierarchical scheduling context. vMPCP enables the sharing of resources in a bounded time within and across VCPUs that could be assigned on different PCPUs. To do so, it uses a para-virtualization approach to expose the executions of critical sections in VCPUs to the hypervisor. Each guest VM can maintain its own priority-numbering scheme and task priorities do not need to be compared across VMs. For the VCPU budget supply and replenishment policy, vMPCP supports both periodic server [32] and deferrable server [36] policies. In addition, vMPCP provides an option for VCPUs to overrun their budgets while their tasks are executing critical sections. The effect of the overrun is analyzed and evaluated in detail.

Contributions: The main contributions of this paper are as follows:

- We propose a new synchronization protocol for the virtualization of multi-core real-time systems. We characterize timing penalties caused by shared resources in a virtualization environment and propose a protocol to address such penalties.

- We analyze the impact of different VCPU budget supply policies, namely periodic and deferrable servers, on synchronization in a multi-core virtualization environment. We also analyze each of the policies with and without VCPU budget overrun.
- From our analysis and experimental results, we found that the periodic server policy, which has been considered to dominate the deferrable server policy in the literature, does not dominate the deferrable server policy when overrun is used. We also found that the use of overrun does not always yield better results, especially for tasks with relatively long critical sections.
- We have implemented the prototype of vMPCP on the KVM hypervisor running on a multi-core platform. Using this implementation, we identify the effect of vMPCP on a real system by comparing it against a virtualization-unaware synchronization protocol (MPCP).

Organization: The rest of this paper is organized as follows. Section II describes the system model used in this paper. Section III presents the vMPCP framework. Section IV provides the analysis on VCPU and task schedulability under vMPCP. A detailed evaluation is provided in Section V. Section VI reviews related work, and Section VII concludes the paper.

II. System Model

In this section, we first describe the hypervisor, virtual machine model, the task model and shared resource model used in this work. Then, we characterize scheduling penalties that arise from shared resources in the multi-core virtualization environment.

A. Hypervisor and Virtual Machines

Figure 1 shows an example system considered in this work. We assume a uniform multi-core system where each core runs at a fixed clock frequency. The system runs a hypervisor hosting multiple guest VMs, each of which has one or more VCPUs. The system has a two-level hierarchical scheduling structure: VCPU scheduling at the hypervisor level and task scheduling at the VCPU level. In this work, we consider *partitioned fixed-priority preemptive scheduling* for both the hypervisor and the VMs, because it is widely used in many commercial real-time embedded hypervisors and OSes such as OKL4 [2] and PikeOS [3]. Under partitioned scheduling, each VCPU is statically assigned to a single PCPU and each task is statically assigned to a single VCPU. Any fixed-priority assignment can be used for both VCPUs and tasks, such as Rate-Monotonic [21].

VCPU v_i is represented to the hypervisor as follows:

$$v_i = (C_i^v, T_i^v)$$

- C_i^v: the maximum execution budget of VCPU v_i[1]
- T_i^v: the budget replenish period of VCPU v_i

[1]The superscript v denotes that the parameter is a VCPU parameter.

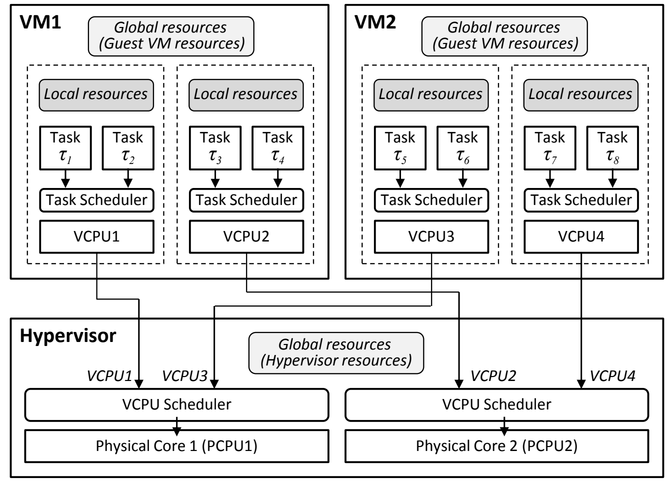


Fig. 1: Multi-core virtualization and shared resources

VCPUs are ordered in increasing order of priorities, i.e. $i < j$ implies that a VCPU v_i has lower priority than a VCPU v_j.

For the VPCU budget supply and replenishment policies, we consider both a *periodic server* [32] and a *deferrable server* [36]. Under the periodic server policy, each VCPU becomes active periodically and executes its tasks that are ready to be executed until the VCPU's budget is exhausted. When a VCPU has no task ready to execute, the VCPU cannot preserve its budget; the budget is idled away. Under the deferrable server policy, a VCPU can preserve its budget until the end of its current period. Hence, the tasks of the VCPU can execute any time while the VCPU's budget remains. The budget-preserving feature of the deferrable server policy causes a jitter equal to $T^v - C^v$ [7].

B. Tasks and Shared Resources

We consider periodic tasks with implicit deadlines. Each task has a unique priority within its VCPU. Note that each task does not need to have a unique priority across VCPUs and there is no need to compare task priorities in one VPCU with those in other VCPUs. In each VCPU, tasks are ordered in increasing order of priorities, i.e. $i < j$ implies that task τ_i has lower priority than task τ_j. Each task has an alternating sequence of normal execution segments and critical section segments. Task τ_i is thus represented as follows:

$$\tau_i = ((C_{i,1}, E_{i,1}, C_{i,2}, E_{i,2}, ..., E_{i,S_i}, C_{i,S_i+1}), T_i)$$

- $C_{i,j}$: the worst-case execution time (WCET) of the j-th normal execution segment of task τ_i
- $E_{i,j}$: the WCET of the j-th critical section segment of τ_i
- T_i: the period of task τ_i
- S_i: the number of critical section segments of τ_i

We use C_i and E_i to denote the sum of the WCETs of all the segments of task τ_i and the sum of the WCETs of the critical section segments of τ_i, respectively. Hence,

$$C_i = \sum_{j=1}^{S_i+1} C_{i,j} + \sum_{j=1}^{S_i} E_{i,j}, \text{ and } E_i = \sum_{j=1}^{S_i} E_{i,j}$$

Shared resources considered in this work are protected by suspension-based mutually-exclusive locks (mutexes). Tasks access shared resources in a non-nested manner, meaning that each task can hold only one resource at a time. There are two types of shared resources: *global* and *local* resources. Global resources are the resources shared among tasks from different VCPUs that may be located on different PCPUs. The critical sections corresponding to the global resources are referred to as global critical sections (gcs's). Conversely, local resources are shared among tasks assigned to the same VCPU. The corresponding critical sections are local critical sections (lcs's). Each resource has a unique index and the function $R(\tau_i, j)$ returns the index of the resource used by the j-th critical section of task τ_i. The function $type(\tau_i, j)$ returns gcs or lcs, which is the type of the j-th critical section of τ_i. In addition, we use S_i^{gcs} and S_i^{lcs} to denote the number of global and local critical section segments of τ_i, respectively. Hence, $S_i = S_i^{gcs} + S_i^{lcs}$.

For brevity, we will also use the following notation in the rest of the paper:

- $V(\tau_i)$: the VCPU assigned to a task τ_i
- $P(v_i)$: the PCPU assigned to a VCPU v_i

C. Penalties from Shared Resources

Scheduling penalties caused by accessing shared resources in a multi-core platform can be categorized into *local blocking* and *remote blocking*. Local blocking time is the duration for which a task needs to wait for the execution of lower-priority tasks assigned on the same core. Uniprocessor real-time synchronization protocols like PCP [31] can bound the local blocking time to at most the duration of one local critical section. Remote blocking time is the duration that a task has to wait for the executions of tasks of any priorities assigned on different cores. If a task τ_i tries to access a global resource held by another task on a different core, task τ_i suspends by itself until the resource-holding task finishes its corresponding critical section. Multiprocessor real-time synchronization protocols such as MPCP [28] are proposed to bound and minimize the duration of remote blocking.

Unlike local blocking, remote blocking causes additional timing penalties even though a multiprocessor synchronization protocol like MPCP is used:[2]

- *Back-to-back execution*: If a task suspends by itself due to remote blocking, its self-suspending behavior can cause a back-to-back execution phenomenon [26], resulting in additional interference to lower-priority tasks.
- *Multiple priority inversions*: Whenever a medium-priority task suspends due to remote blocking, lower-priority tasks get a chance to execute and issue requests for local or global resources. In case of local resources under PCP, every normal execution segment of a medium-priority task can be blocked at most once by one of the lower-priority tasks executing their local critical sections with inherited higher priorities. In case of global resources under MPCP, every normal execution segment of a task can be preempted at most once by each of the lower-priority tasks executing global critical sections. Consequently, multiple priority inversions caused by remote blocking increase the local blocking time.

[2]More information on this issue can be found in [18].

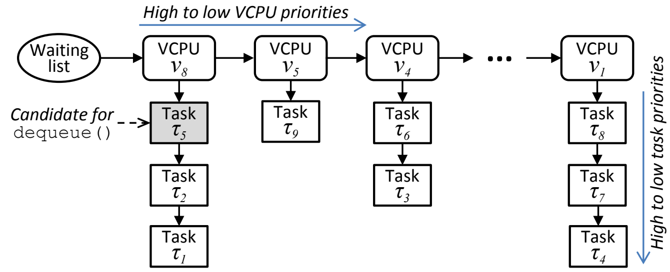


Fig. 2: Two-level priority queue of a global mutex

In the multi-core virtualization environment, the length of remote blocking time may become even significantly longer due to:

- *Preemptions by higher-priority VCPUs*: Consider a task τ_i in a VCPU v_j waiting on a global resource held by another task in a VCPU v_k assigned on a different physical core. If the VCPU v_k is preempted by higher-priority VCPUs on its core, the remote blocking time for τ_i is increased by the execution times of those higher-priority VCPUs.
- *VCPU budget depletion*: Tasks in a VCPU are scheduled by using their VCPU's budget. When the VCPU budget of a resource-holding task is depleted, a task waiting remotely on that resource needs to wait at least until the start of the next replenishment period of the resource-holding task's VCPU.

Goal: In this work, our goal is to minimize the remote blocking in a multi-core virtualization environment. Another goal is to bound the remote blocking time of a task as a function of the duration of global critical sections of other tasks (and the parameters of VCPUs having those tasks when overrun is not used), and *not* as a function of the duration of normal execution segments or local critical sections.

III. vMPCP Framework

In this section, we present the virtualization-aware multiprocessor ceiling protocol (vMPCP). We first define vMPCP and explain the optional VCPU budget overrun mechanism for periodic server and deferrable server replenishment policies under vMPCP. Then, we provide the details on the software design to implement vMPCP in the hypervisor.

A. Protocol Description

vMPCP is specifically designed to reduce and bound remote blocking times for accessing global shared resources in a multi-core virtualization environment. To do so, vMPCP uses hierarchical priority ceilings for global critical sections. This approach suppresses both task-level and VCPU-level preemptions while accessing a global resource, thereby reducing the remote blocking times of other tasks waiting on that resource. Global and local resource access rules under vMPCP are defined as follows.

Global shared resources: vMPCP is based on the multiprocessor priority ceiling protocol (MPCP) [25, 28], and extends it to the hierarchical scheduling context.

1) Under vMPCP, each mutex protecting a global resource uses a two-level priority queue for its waiting list. Figure 2

shows a logical structure of this two-level priority queue, where the first level is ordered by VCPU priorities and the second level is ordered by task priorities. The key for queue insertion is a pair of VCPU priority and task priority, i.e. (j, i) is a key for a task τ_i in a VCPU v_j. The queue has a `dequeue` function, which returns the highest priority task of the highest priority VCPU and removes it from the queue.
2) When a task τ_i requests an access to a global resource R_k, the resource R_k can be granted to the task τ_i, if it is not held by another task.
3) While a task τ_i in a VCPU v_j is holding a resource for its global critical section (gcs), the priority of τ_i is raised to $\pi_{B,v_j} + \pi_i$, where π_{B,v_j} is a base task-priority level greater than that of any task in the VCPU v_j, and π_i is the normal priority of τ_i. We refer to $\pi_{B,v_j} + \pi_i$ as the *task-level priority ceiling* of the gcs of τ_i.
4) While a task τ_i executes a gcs, the priority of its VCPU v_j is raised to $\pi^v_B + \pi^v_j$, where π^v_B is a base VCPU-priority level greater than that of any other VCPUs in the system, and π^v_j is the normal priority of the VCPU v_j. We refer to $\pi^v_B + \pi^v_j$ as the *VCPU-level priority ceiling* of the gcs of τ_i.
5) When a task τ_i requests access to a resource R_k, the resource R_k cannot be granted to τ_i, if it is already held by another task. In this case, the task τ_i is inserted to the waiting list (two-level priority queue) of the mutex for R_k.
6) When a global resource R_k is released and the waiting list of the mutex for R_k is not empty, a task dequeued from the head of the queue is granted the resource R_k.

Local shared resources: vMPCP follows the uniprocessor priority ceiling protocol (PCP) [31] for accessing local resources.[3] Unlike the global resource case, a VCPU priority is not affected while its task is accessing a local resource.

1) Each mutex associated with a local resource R_k is assigned a task-level priority ceiling, which is equal to the highest priority of any task accessing R_k. Note that this is valid only within this VCPU.
2) A task τ_i can access a local resource R_k, if the priority of τ_i is higher than the priority ceilings of any other mutexes currently locked by other tasks in that VCPU.
3) If a task τ_i is blocked on a local resource by another task that has a lower priority than τ_i, the lower-priority task inherits the priority of τ_i.

B. VCPU Budget Overrun

vMPCP provides an option for VCPUs to overrun their budgets when their tasks are in gcs's. This allows tasks to complete their gcs's, even though their VCPU has exhausted its budget. Hence, remote blocking time can be significantly reduced. We present the detailed behavior of the VCPU budget overrun under periodic server and deferrable server policies.

Periodic server with overrun: The VCPU budget overrun with VCPUs under the periodic server policy works similar to the one presented in [12]. Suppose that a VCPU's budget is exhausted while one of its tasks is in a gcs. If overrun is enabled, the task can continue to execute and finish the gcs. Recall that vMPCP immediately increases the priority of any task executing a gcs to be higher than that of any other normally executing tasks or tasks accessing local resources. Therefore, the amount of overrun time is only affected by the lengths of global critical sections in a VCPU.

If a VCPU's budget is exhausted while no task of the VCPU is in a gcs, the VCPU suspends until the start of its next replenishment period. Once the VCPU suspends, overrun has no effect. This is to maintain the good property of the periodic server policy, no potential back-to-back interference to lower-priority VCPUs. For instance, consider a task τ_i waiting for a global resource R that is held by another task on a different physical core. The VCPU of τ_i is currently suspended due to its budget depletion. If the resource R is released while the VCPU of τ_i is suspended, the task τ_i needs to wait until the next replenishment period of its VCPU although overrun is enabled.

Deferrable server with overrun: Unlike the periodic server policy, VCPUs under the deferrable server policy can overrun more flexibly. Consider a task τ_i waiting for a global resource R that is held by another task on different physical core. The VCPU of τ_i has exhausted its regular budget. If the resource R is released, the VCPU of τ_i is allowed to overrun its budget and the task τ_i can execute its gcs corresponding to R. Once the task τ_i finishes its gcs, the VCPU of τ_i suspends again. This difference between periodic server and deferrable server with overrun leads to different values in remote blocking time. We will analyze the details in Section IV-B.

C. vMPCP Para-virtualization Interface

vMPCP increases both the priorities of a task and its VCPU when the task executes a gcs. If a lock corresponding to a global resource is implemented at the hypervisor, e.g., resource sharing among VCPUs from different guest VMs, the hypervisor can manage the priorities of VCPUs appropriately. However, if a lock for a global resource is implemented within a guest VM image, e.g., resource sharing in a multi-core guest VM hosted on the hypervisor, there is no way for the hypervisor to know if any task of a VCPU of the VM executes a gcs associated with the lock.

To address this issue, vMPCP provides a para-virtualization[4] interface for a VCPU to let the hypervisor know the executions of gcs's in the VCPU. The interface consists of the following two functions:

- `vmpcp_start_gcs()`: If any task of a VCPU acquires a lock for a global resource, this function is called to let the hypervisor increase the priority of the VCPU by the base VCPU-priority level π^v_B of the system. If overrun is enabled, the hypervisor allows the VCPU to continue to execute until `vmpcp_finish_gcs()` is called. The hypervisor may implement an enforcement mechanism for the VCPU not to

[3] As an alternative to PCP, the highest locker priority protocol (HLP) can also be used for local resources.

[4] Para-virtualization is a technique involving small modifications to guest operating systems or device drivers to achieve high performance and efficiency.

exceed its pre-determined overrun time that will be given in Sec. IV-A.

- `vmpcp_finish_gcs()`: When there is no global-resource lock held by any task in a VCPU, this function is called to let the hypervisor reduce the priority of the VCPU to its normal priority. Also, if the VCPU's budget is exhausted, the hypervisor suspends the VCPU.

IV. vMPCP Schedulability Analysis

In this section, we present the schedulability analysis under our proposed vMPCP. Our analysis considers each of the periodic server and deferrable server policies with and without VCPU budget overrun. We first analyze the VCPU schedulability on a physical core and the task schedulability on a VCPU.

A. VCPU Schedulability

vMPCP increases the priority of a VCPU while any task of the VCPU is holding a global resource, which enables a lower-priority VCPU to block a higher-priority VCPU.[5] Also, vMPCP results in increased VCPU execution times when overrun is enabled. We now analyze these worst-case effects on VCPU schedulability and derive the VCPU schedulability test under vMPCP.

Blocking from lower-priority VCPUs: We first focus on the case where the periodic server policy is used. Consider a higher-priority VCPU v_h and a lower-priority VCPU v_l, both assigned to the same core. Under the periodic server policy, the higher-priority VCPU v_h never suspends by itself until its budget is exhausted. Hence, the lower-priority VCPU v_l can block v_h only when any global resource that v_l's task has been waiting on is released from another core. The blocking time is equal to the duration of the corresponding gcs (global resource holding time). The worst case happens when all the tasks of v_l have been waiting on global resources and these resources are released from other cores while the higher-priority VCPU v_h is executing. The maximum global resource holding time of v_l is as follows:

$$ght(v_l) = \sum_{\tau_j \in v_l \wedge S_j^{gcs} > 0} \max_{1 \le k \le S_j \wedge type(\tau_j,k)=gcs} E_{j,k} \quad (1)$$

Using Eq. (1), the worst-case blocking time imposed on a VCPU v_i during a time interval t under the periodic server policy is given as follows:

$$B_i^v(t) = \sum_{v_l \in P(v_i) \wedge l < i} ght(v_l) \quad (2)$$

Note that the parameter t is used to be consistent with the deferrable server case which will be shown in Eq. (4).

We now consider the case where the deferrable server policy is used. Under this policy, a higher-priority VCPU v_h may suspend itself several times every period. This means that, unlike the periodic server case, the tasks of a lower-priority VCPU v_l may get a chance to request global resources whenever v_h suspends. Hence, each task of the lower-priority VCPU v_l may block the higher-priority VCPU v_h multiple times during v_h's period. The maximum accumulated global resource holding time of the tasks of v_l during a time interval t is given by:

$$sum\_ght(v_l,t) = \sum_{\tau_j \in v_l} \left\{ \left(\left\lceil \frac{t}{T_j} \right\rceil + 1 \right) \cdot \sum_{\substack{1 \le k \le S_j \wedge \\ type(\tau_j,k)=gcs}} E_{j,k} \right\} \quad (3)$$

Note that the "+1" term is to capture the carry-in job of each task during a given time interval t. By using Eq. (3), the worst-case blocking time imposed on a VCPU v_i during a time interval t under the deferrable server policy is represented as follows:

$$B_i^v(t) = \sum_{v_l \in P(v_i) \wedge l < i} sum\_ght(v_l,t) \quad (4)$$

Budget overrun time: If the VCPU budget overrun option is enabled, a VCPU can overrun its budget only when its tasks are executing gcs's. Hence, the maximum time that a VCPU v_i can overrun is bounded by the maximum global resource holding time of that VCPU, which is given in Eq (1). Therefore, the maximum overrun time of a VCPU v_i (O_i^v) is equal to $ght(v_i)$ if overrun is enabled, and zero if overrun is not enabled.

VCPU schedulability: The schedulability of a VCPU v_i can be determined by the following recurrence equation:

$$\begin{aligned} W_i^{v,n+1} =& C_i^v + O_i^v + B_i^v(W_i^{v,n}) + \\ & \sum_{v_h \in P(v_i) \wedge h > i} \left\lceil \frac{W_i^{v,n} + J_h^v}{T_h^v} \right\rceil \cdot (C_h^v + O_h^v) \end{aligned} \quad (5)$$

where $W_i^{v,n}$ is the worst-case response time of v_i at the n^{th} iteration ($W_i^{v,0} = C_i^v + O_i^v$) and J_h^v is a VCPU release jitter ($J_h^v = 0$ under the periodic server policy and $J_h^v = T_h^v - C_h^v$ under the deferrable server policy). Eq. (5) is based on the iterative response time test [14]. It terminates when $W_i^{v,n+1} = W_i^{v,n}$, and the VCPU v_i is schedulable if its response time does not exceed its period: $W_i^{v,n} <= T_i^v$. In this equation, O_i^v and O_h^v are used to represent the budget overrun of v_i and its higher-priority VCPUs, repectively. The third term represents the blocking time from lower-priority VCPUs during v_i's response time.

B. Task Schedulability

To determine the schedulability of a task τ_i under vMPCP, we need to consider the factors discussed in Section II-C: (i) local blocking time, (ii) remote blocking time, (iii) back-to-back execution due to remote blocking, (iv) multiple priority inversions, (v) preemptions by higher-priority VCPUs, and (vi) VCPU budget depletion. We take into account factor (iv) when analyzing local blocking time, and factors (v) and (vi) when analyzing remote blocking time. By considering factors (i), (ii) and (iii), we use the following recurrence equation that bounds the worst-case response time of a task τ_i in a VCPU v_k under vMPCP:

$$\begin{aligned} W_i^{n+1} =& C_i + B_i^l + B_i^r + \sum_{\tau_h \in V(\tau_i) \wedge h > i} \left\lceil \frac{W_i^n + J_h + (W_h - C_h)}{T_h} \right\rceil C_h \\ & + \left\lceil \frac{W_i^n + C_k^v}{T_k^v} \right\rceil (T_k^v - C_k^v) \end{aligned} \quad (6)$$

[5] vMPCP does not increase the priority of a VCPU when its task is holding a local resource. Hence, local resources do not affect the VCPU schedulability.

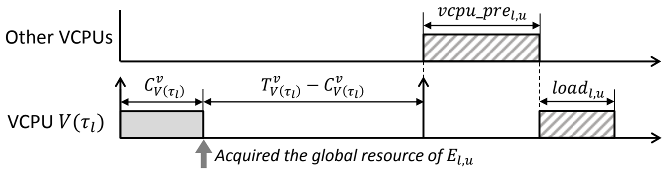


Fig. 3: Periodic server with overrun

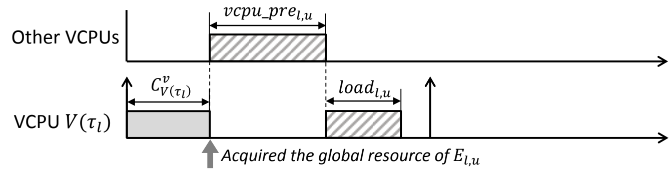


Fig. 4: Deferrable server with overrun

where B_i^l is the local blocking time for τ_i, B_i^r is the remote blocking time for τ_i, and J_h is the release jitter of each higher-priority task τ_h ($J_h = T_k^v - C_k^v$). It terminates when $W_i^{n+1} = W_i^n$, and the task τ_i is schedulable if its response time does not exceed its implicit deadline: $W_i^n <= T_i$. Eq. (6) is based on the response-time test for independent tasks under hierarchical scheduling given in [30]. Specifically, the last term of Eq. (6) is from [30], which captures the execution gap due to the periodic budget supply of the VCPU. The back-to-back execution due to remote blocking from each higher-priority task τ_h is captured by adding $W_h - C_h$ in the summing term.[6]

In the rest of this section, we shall analyze the local and remote blocking times, B_i^l and B_i^r. We use $tc_{i,j}$ as the task-level priority ceiling of the j-th critical section segment of task τ_i. Similarly, $vc_{i,j}$ is used to represent the VCPU-level priority ceiling of the j-th critical section segment of task τ_i.

Local blocking time: The local and global critical sections of lower-priority tasks can block the normal execution segment of a higher-priority task τ_i. With the local resource access rule of vMPCP based on PCP [31], only one lower-priority task with a priority ceiling higher than the normal priority of τ_i can block each normal execution segment of τ_i. Hence, the maximum per-segment blocking time from the local critical sections of lower-priority tasks is given by:

$$B_i^{l\_lcs} = \max_{\substack{\tau_l \in V(\tau_i) \wedge l < i \\ \wedge S_l^{lcs} > 0}} \left(\max_{\substack{1 \le u \le S_l \wedge type(\tau_l,u)=lcs \\ \wedge tc_{l,u} > i}} E_{l,u} \right) \tag{7}$$

Unlike lcs's, the gcs's of each lower-priority task can block the normal execution segment of τ_i. The maximum per-segment blocking time from the gcs's of lower-priority tasks is given by:

$$B_i^{l\_gcs} = \sum_{\substack{\tau_l \in V(\tau_i) \wedge l < i \\ \wedge S_l^{gcs} > 0}} \left(\max_{1 \le u \le S_l \wedge type(\tau_l,u)=gcs} E_{l,u} \right) \tag{8}$$

The total local blocking time from both the local and global critical sections of lower-priority tasks is given by:

$$B_i^l = (B_i^{l\_lcs} + B_i^{l\_gcs}) \cdot (S_i^{gcs} + 1) \tag{9}$$

Here, the reason for multiplying by $S_i^{gcs} + 1$ is that, before a task τ_i executes or whenever τ_i self-suspends due to a global resource, lower-priority tasks may issue requests for local or global resources.

Remote blocking time: The remote blocking time B_i^r of a task τ_i is given by:

$$B_i^r = \sum_{1 \le j \le S_i \wedge type(\tau_i,j)=gcs} B_{i,j}^r \tag{10}$$

[6]This is a correction made from our original manuscript appeared in RTSS'14. More details on this correction and a suspension-based blocking term in a response-time test can be found in [8].

where $B_{i,j}^r$ is the remote blocking time for τ_i in acquiring the global resource associated with the j-th critical section of τ_i. Note that $B_{i,j}^r = 0$ if the j-th critical section of τ_i is a lcs.

The term $B_{i,j}^r$ is bounded by the following recurrence equation:

$$B_{i,j}^{r,n+1} = \max_{\substack{V(\tau_l) \in lpvcpus(V(\tau_i)) \\ \wedge R(\tau_l,u)=R(\tau_i,j)}} W_{l,u}^{gcs} + \sum_{\substack{V(\tau_h) \in hpvcpus(V(\tau_i)) \\ \wedge R(\tau_h,u)=R(\tau_i,j)}} \left(\left\lceil \frac{B_{i,j}^{r,n}}{T_h} \right\rceil + 1 \right) \cdot W_{h,u}^{gcs} \tag{11}$$

where $B_{i,j}^{r,0} = \max_{V(\tau_l) \in lpvcpus(V(\tau_i)) \wedge R(\tau_l,u)=R(\tau_i,j)} W_{l,u}^{gcs}$ (the first term of the equation), $lpvcpus(V(\tau_i))$ is the set of lower-priority VCPUs than the VCPU of τ_i in the system, $hpvcpus(V(\tau_i))$ is the set of higher-priority VCPUs than the VCPU of τ_i, and $W_{l,u}^{gcs}$ represents the worst-case response time of the execution $E_{l,u}$ of a gcs after acquiring the corresponding global resource. The first term of Eq. (11) captures the time for a task in a lower-priority VCPU to finish its gcs. The second term represents the time for tasks in higher-priority VCPUs to execute their gcs's.

We now analyze $W_{l,u}^{gcs}$, the amount of which depends on which VCPU policy is used and whether overrun is used. We first define two terms, $load_{l,u}$ and $vcpu\_prm_{l,u}$, as follows:

$$load_{l,u} = E_{l,u} + \sum_{\tau_x \in V(\tau_l)} \max_{1 \le y \le S_x \wedge tc_{x,y} > tc_{l,u}} E_{x,y} \tag{12}$$

$$vcpu\_prm_{l,u} = \sum_{\substack{v_z \in P(V(\tau_l)) \\ \wedge v_z \neq V(\tau_l)}} \sum_{\tau_x \in v_z} \max_{\substack{1 \le y \le S_x \wedge \\ vc_{x,y} > vc_{l,u}}} E_{x,y} \tag{13}$$

The term $load_{l,u}$ bounds the maximum VCPU budget required to execute the critical section $E_{l,u}$. It captures the execution time of $E_{l,u}$ and the execution times of gcs's with higher task-level priority ceilings in the same VCPU. Since every gcs has a higher priority than any normal execution segment, we only need to consider one global critical section per task. The term $vcpu\_prm_{l,u}$ bounds the VCPU-level preemptions while $E_{l,u}$ executes. The VCPU of $E_{l,u}$ can only be preempted by other VCPUs that have tasks being executing gcs's with higher VCPU-level priority ceilings. Note that $vcpu\_prm_{l,u}$ increases the response time of $E_{l,u}$ ($W_{l,u}^{gcs}$), but does not consume the budget of $E_{l,u}$'s VCPU.

- *Periodic server with overrun*: The worst-case response time of the execution $E_{l,u}$ of a gcs happens when the corresponding resource is acquired right after its VCPU is suspended. In this case, the execution is delayed until the start of its VCPU's next replenishment period, and this waiting time is up to $T_{V(\tau_l)}^v - C_{V(\tau_l)}^v$, as shown in Figure 3. Once the next period of the VCPU starts, the VCPU can execute and finish $E_{l,u}$ within this period due to overrun. Therefore, $W_{l,u}^{gcs}$

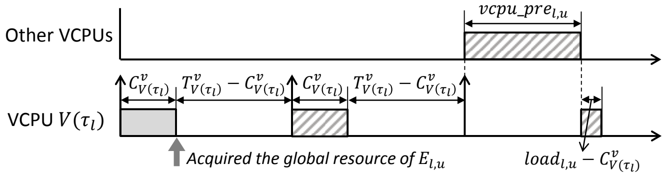


Fig. 5: Periodic/deferrable server without overrun

under the periodic server policy with overrun is given by:

$$W^{gcs}_{l,u} = T^v_{V(\tau_l)} - C^v_{V(\tau_l)} + load_{l,u} + vcpu\_prm_{l,u} \quad (14)$$

- *Deferrable server with overrun*: In this case, $E_{l,u}$ can be executed without the need to wait until the VCPU's next replenishment period (Figure 4). Therefore, $W^{gcs}_{l,u}$ under the deferrable server policy with overrun is given by:

$$W^{gcs}_{l,u} = load_{l,u} + vcpu\_prm_{l,u} \quad (15)$$

- *Periodic/deferrable server without overrun*: When overrun is not used, the execution of $load_{l,u}$ may span over multiple of its VCPU periods (Figure 5). The total execution gap is bounded by $\lceil \frac{load_{l,u}}{C^v_{V(\tau_l)}} \rceil (T^v_{V(\tau_l)} - C^v_{V(\tau_l)})$. Therefore, $W^{gcs}_{l,u}$ under the periodic or deferrable server policy without overrun is given by:

$$W^{gcs}_{l,u} = \left\lceil \frac{load_{l,u}}{C^v_{V(\tau_l)}} \right\rceil (T^v_{V(\tau_l)} - C^v_{V(\tau_l)}) + load_{l,u} + vcpu\_prm_{l,u} \quad (16)$$

Note that, if the amount of $load_{l,u}$ is smaller than the per-period execution budget of the VCPU ($C^v_{V(\tau_l)}$), Eq. (16) becomes equal to Eq. (14).

V. EVALUATION

This section presents our experimental evaluation on vMPCP. To our knowledge, vMPCP is the first virtualization-aware multi-core synchronization protocol and there is no schedulability test for existing protocols in the multi-core virtualization environment. We first empirically investigate the performance characteristics of vMPCP in terms of task schedulability, and then compare vMPCP against a virtualization-unaware protocol (MPCP) in terms of response times on a real hardware platform.

A. Comparison of Different Configurations

The purpose of this experiment is to explore the impact of different uses of vMPCP on task schedulability. To do this, we use randomly-generated tasksets and capture the percentage of schedulable tasksets as the metric.[7]

Experimental Setup: The base parameters we use for experiments are summarized in Table I. As the main interest of our work is in the timing penalties caused by global resources, local resources are not considered. For each experimental setting, we first generate the defined numbers of physical CPU cores in the system, VCPUs for each core, and tasks for each VCPU. Task periods are randomly selected within the defined min/max task period range. On each VCPU, the VCPU task utilization is split into k random-sized pieces,

TABLE I: Base parameters for experiments

Parameters	Values	Parameters	Values
# of physical cores	8	# of VCPUs per core	2
# of tasks per VCPU	3	Period of a VCPU	5 msec
Min. task period	100 msec	Max. task period	500 msec
Per-VCPU task util	15%	# of gcs's per task	1
# of lockers per mutex	2	Size of a gcs	10 μsec

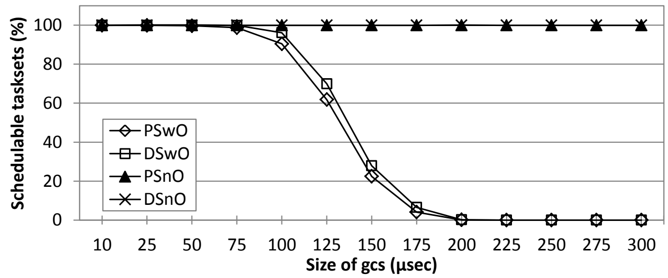


Fig. 6: Taskset schedulability as the size of a gcs increases

where k is the number of tasks in the VCPU. The size of each piece represents the utilization of the corresponding task. Then, the WCET of each task is calculated by dividing its utilization by its period. The priorities of tasks and VCPUs are assigned by the Rate-Monotonic approach [21] (ties are broken arbitrarily). Once the task information is generated, we determine a VCPU budget value that is used for all VCPUs in the system. Starting from a value equal to the VCPU period, we decrease the VCPU budget by 10 μsecs until all VCPUs pass the VCPU schedulability test given in Eq. (5).[8] We generate 10,000 tasksets for each experimental setting, and record the percentage of tasksets where all the tasks pass the task schedulability test given in Eq. (6).

Results: We consider the following four uses of vMPCP: periodic server with overrun (PSwO), deferrable server with overrun (DSwO), periodic server with no overrun (PSnO), and deferrable server with no overrun (DSnO). The main factors affecting task schedulability under vMPCP are: (i) the size of a gcs, (ii) the number of lockers per mutex, (iii) the number of gcs's per task, (iv) the VCPU period, and (v) the utilization of tasks in each VCPU. By exploring these factors, we identify the characteristics of the four schemes of vMPCP.

Figure 6 shows the percentage of schedulable tasksets as the size of a gcs increases. The schemes with no overrun, PSnO and DSnO, are almost unaffected by the size of a gcs. Conversely, the schedulability under the schemes with overrun, PSwO and DSwO, decreases as the size of a gcs increases. This is due to the fact that, without overrun, more VCPU budget can be used for the executions of normal execution segments of tasks. DSwO performs better than PSwO because DSwO results in a shorter response time of the execution of a gcs, as given in Eq. (15).

Figure 7 shows the percentage of schedulable tasksets as the number of lockers per mutex increases. Points on the x-axis represent all possible values for the number of lockers per mutex in our experimental setting. The performance degradation happens only when the number of lockers per muxex is very high (> 12). This is because vMPCP uses a two-level priority queue as the waiting list for a mutex. Hence, higher priority tasks or tasks in higher-priority VCPUs do not need to

[7]The experiments in this subsection have been re-conducted with the corrected equation given in Eq. (6). However, the trends remain the same as those in our original manuscript appeared in RTSS'14.

[8]As the minimum time unit in Table I is 10 μsec, the step size of 10 μsec is fine-grained enough to find the VCPU budget values in this experiment.

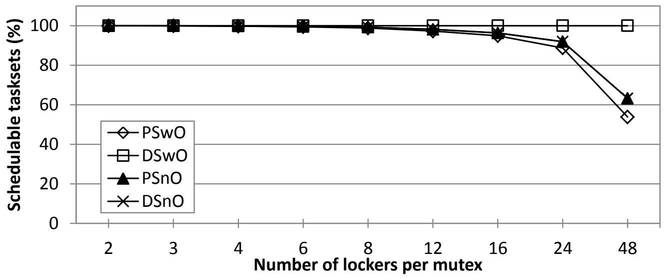


Fig. 7: Taskset schedulability as the number of lockers per mutex increases

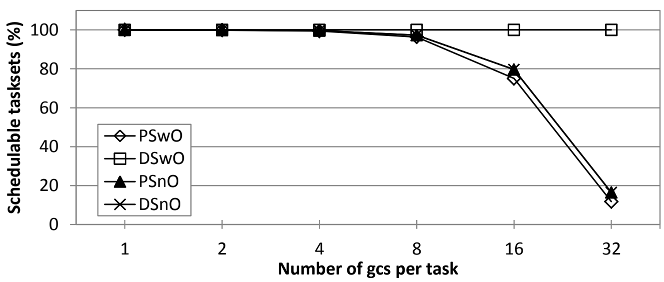


Fig. 8: Taskset schedulability as the number of gcs's per task increases

wait until all the lower-priority tasks or tasks in lower-priority VCPUs finish their gcs's.

Figure 8 shows the percentage of schedulable tasksets as the number of gcs's per task increases. The performance difference between DSwO and the other three schemes becomes larger as the number of gcs's per task increases. Even if the number of gcs's per task reaches 32, DSwO does not show any noticeable performance degradation due to its short gcs response time.

Figure 9 shows the percentage of schedulable tasksets as the VCPU period increases. DSwO performs much better than the other three schemes. Especially, when the VCPU period is 40 msec, the difference in the percentage of schedulable tasksets between DSwO and the other schemes is about 80%. This big difference is due to the fact that PSwO, PSnO and DSnO are sensitive to the VCPU period when accessing global resources, as given by Eq. (14) and Eq. (16).

Lastly, Figure 10 shows the percentage of schedulable tasksets as the utilization of tasks per VCPU increases. For all schemes, the percentage decreases when the per-VCPU utilization is greater than 17.0%. Interestingly, when the utilization is 19.0%, DSwO performs better than PSnO and DSnO, but when the utilization is 20.0%, the result is the opposite.

In summary, we observe from the results that there is no single scheme that can dominate the others. DSwO generally performs better than PSwO, PSnO and DSnO, due to its short gcs response time. In some cases, PSnO and DSnO outperform DSwO by allowing more VCPU budgets for the normal execution segments of tasks. PSwO gives the worst performance in our experiments. This is because PSwO allows less VCPU budget for normal execution segments than PSnO and DSnO, and gives longer gcs response time than DSwO.

B. Case Study: KVM Hypervisor

We now present a case study demonstrating the benefit of vMPCP by using our implementation on the KVM hypervisor.

Implementation: We have implemented vMPCP on the KVM

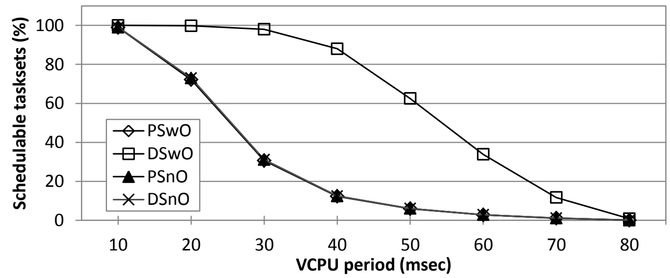


Fig. 9: Taskset schedulability as the VCPU period increases

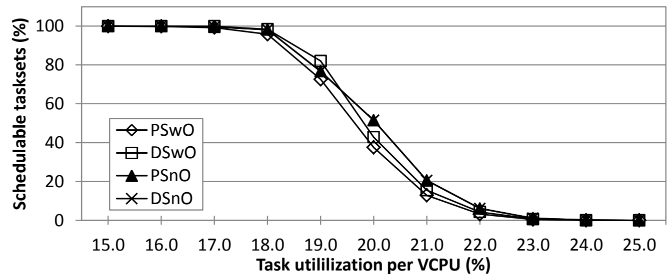


Fig. 10: Taskset schedulability as the task utilization per VCPU increases

(Kernel-based Virtual Machine) hypervisor [17] of the latest version of Linux/RK [24, 27].[9] The host machine runs on Linux/RK, and uses KVM to execute guest VMs that also run on Linux/RK. Our implementation supports the deferrable server policy and an optional overrun mechanism. The vMPCP mutex data structures and APIs (e.g., open, lock, unlock) are implemented as part of the Linux/RK kernel module. Specifically, the vMPCP mutexes are classified into *intra-VM* and *inter-VM* mutexes based on the memory spaces their corresponding global resources belong to. The intra-VM mutexes are for resources shared within a guest VM and use the `vmpcp_start_gcs()` and `vmpcp_finish_gcs()` hypercalls internally. The inter-VM mutexes are for resources shared among guest VMs and the hypervisor. They are implemented by using the per-VCPU `virtqueue` interface of `virtio` [29] for hypervisor-VM communication.

Table II lists the implementation costs of vMPCP APIs on the KVM hypervisor. The target system used is equipped with an Intel Core i7-2600 quad-core processor running at 3.4 GHz and 8GBytes of RAM. To reduce measurement inaccuracies, we have disabled the simultaneous multithreading and dynamic clock frequency scaling of the processor. The `open` and `destroy` APIs take longer times for intra-VM mutexes than for inter-VM mutexes. This is mainly due to the performance difference between a VM and the hypervisor in memory allocation and deallocation for mutex data structures. The costs of `lock`, `trylock` and `unlock` APIs are similar for both intra- and inter-VM mutexes. The major factor contributing to the lock/unlock costs is the "world switch" between a VM and the hypervisor. Since the intra-VM mutexes cause the `vmpcp_start_gcs` and `vmpcp_finish_gcs` hypercalls, the world switch happens for intra-VM mutexes as well.

Case Study: In this case study, we compare the response times of tasks sharing a global resource under vMPCP and those under a virtualization-unaware multi-core synchroniza-

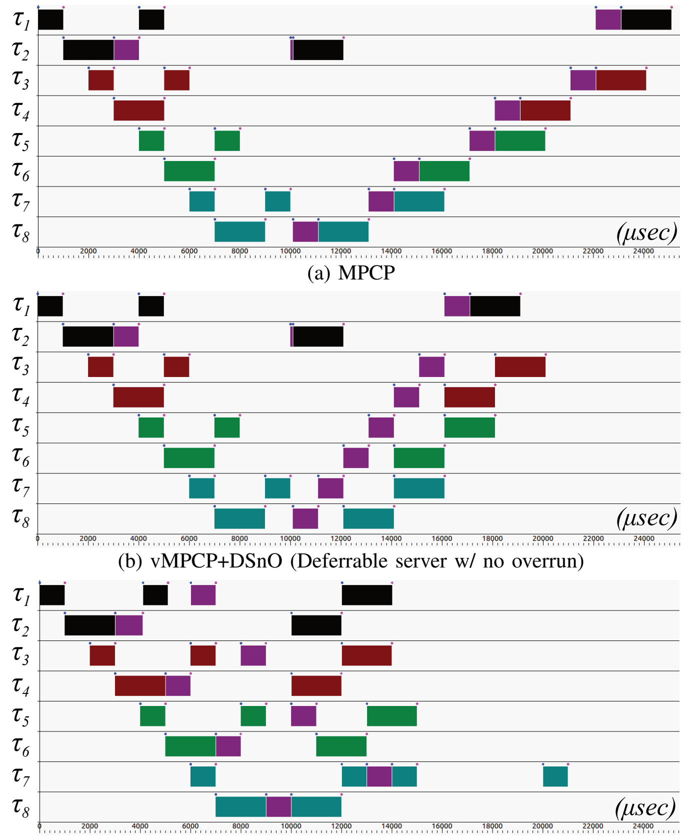
[9] Linux/RK is available at https://rtml.ece.cmu.edu/redmine/projects/rk.

TABLE II: Implementation cost of vMPCP on KVM

Types	Mutex APIs	Avg (μsec)	Max (μsec)
Intra-VM	`open` (create new mutex)	4.16	7.14
	`open` (existing mutex)	1.87	3.64
	`destroy`	1.83	3.50
	`lock`	3.51	5.69
	`trylock`	2.75	5.15
	`unlock`	2.26	2.68
	`*vmpcp_start_gcs`	2.05	2.88
	`*vmpcp_finish_gcs`	1.40	1.60
Inter-VM	`open` (create new mutex)	1.79	3.48
	`open` (existing mutex)	1.76	3.35
	`destroy`	1.49	1.78
	`lock`	3.09	5.31
	`trylock`	2.80	5.29
	`unlock`	1.93	2.57

tion protocol, MPCP. The target system hosts two guest VMs, each of which has four VCPUs (VM1: $\{v_1, v_3, v_5, v_7\}$, VM2: $\{v_2, v_4, v_6, v_8\}$). All VCPUs have the same budget and period: $v_i = (3, 10)$, units in msec. The release offset of each VCPU is zero. The target machine has four processing cores, Core 1, 2, 3 and 4. Each core is assigned two VCPUs: Core 1 $= \{v_1, v_2\}$, Core 2 $= \{v_3, v_4\}$, Core 3 $= \{v_5, v_6\}$, Core 4 $= \{v_7, v_8\}$. For a taskset, we use eight synthetic tasks, each of which has one gcs. There is one global resource shared among all these tasks. Each task is assigned to a VCPU with the same index number: $\tau_5 \in v_5$. All tasks except τ_2 have the same timing parameters: $\tau_i = ((2, 1, 2), 200)$, where $i \neq 2$, units in msec. Task τ_2 has a slightly longer gcs: $\tau_2 = ((2, 1.1, 2), 200)$. Each task τ_i also has a release offset of $i - 1$ msec, e.g., τ_5 is released at $t = 4$ msec. We used Linux/RK to set the periods, release offsets, and real-time priorities of VCPUs and tasks. In accordance with our system model, tasks and VCPUs with higher indices are assigned higher priorities.

Figure 11 shows the execution timelines of tasks captured under MPCP, vMPCP with deferrable server and no overrun (vMPCP+DSnO), and vMPCP with deferrable server and overrun enabled (vMPCP+DSwO). As can be seen, the response times of tasks are much shorter under vMPCP+DSnO and vMPCP+DSwO, compared to those under MPCP (7.5% of resposne time decrease on average under vMPCP+DSnO, and 29.1% under vMPCP+DSwO). The shared resource is first held by τ_2 at $t = 3$, but under MPCP and vMPCP+DSnO, it cannot release the resource due to its VCPU's budget depletion. Hence, the resource is held by τ_2 until the start of its VCPU's next replenish period. Conversely, under vMPCP+DSwO, τ_2 can finish its gcs and release the resource. This allows other tasks to access the resource within the first VCPU period, thereby significantly reducing the response times of tasks. In case of task τ_8, it acquires the resource at $t = 10$ under both MPCP and vMPCP+DSnO. Here, the difference happens when τ_8 finishes its gcs. Under MPCP, τ_8 continues to execute because its VCPU has the highest priority on that core. This causes a delay to task τ_7, which is the highest-priority task among the tasks waiting on the resource, to enter its gcs. However, under vMPCP+DSnO, τ_7 starts its gcs right after the resource is released by τ_8. This slightly lengthens the response time of τ_8, but allows other tasks to access the resource much faster. Under vMPCP+DSwO, the response times of all tasks except τ_7 are shorter than those under the other two schemes. The increase in τ_7's response



(a) MPCP

(b) vMPCP+DSnO (Deferrable server w/ no overrun)

(c) vMPCP+DSwO (Deferrable server w/ overrun)

Fig. 11: Task execution timelines under MPCP, vMPCP+DSnO and vMPCP+DSwO

time is due to the back-to-back execution of the VCPU of τ_8, the amount of which is bounded by our analysis. The case study results show that vMPCP is effective in reducing the response times of tasks accessing shared resources in a multi-core virtualization environment.

VI. Related Work

Resource sharing and task synchronization issues in multi-core and multiprocessor systems have been intensively studied in the non-hierarchical scheduling context. MPCP (Multiprocessor Priority Ceiling Protocol) [25, 28] provides bounded remote blocking time on accessing global shared resources under partitioned fixed-priority scheduling. MPCP uses the uniprocessor PCP [31] for accessing local resources. Recently, a new schedulability analysis for MPCP is proposed in [18]. MSRP (Multiprocessor Stack-based Resource Policy) [13] is an extension of the uniprocessor SRP [5] for resource sharing under partitioned EDF scheduling. A comparison of MPCP and MSRP is also provided in [13]. FMLP (Flexible Multiprocessor Locking Protocol) [9] is the first protocol that supports both partitioned and global EDF scheduling. MSOS (Multiprocessors Synchronization for real-time Open Systems) [23] is designed for resource sharing among independently-developed systems where each processor uses different scheduling algorithms. All these protocols, however, are designed for non-hierarchical scheduling, so they may cause indefinite remote blocking time in multi-core virtual machines.

In the hierarchical scheduling context, much research has been conducted on the schedulability analysis of independent tasks on uniprocessors [11, 30, 34, 35] and multiprocessors [20, 33]. For tasks with shared resources, HSRP (Hierarchical Stack Resource Policy) [12] is the first synchronization protocol proposed in the context of uniprocessor hierarchical scheduling. HSRP uses budget overrun and payback mechanisms to limit priority inversion. SIRAP (Subsystem Integration and Resource Allocation Policy) [6] uses the idea of self-blocking to bound delays on accessing shared resources without knowing the timing parameters of other subsystems. RRP (Rollback Resource Policy) [4] uses a rollback mechanism to avoid a lock-holding task to be blocked while holding a lock. However, none of these protocols has been extended to the multiprocessor hierarchical scheduling context.

In [22], the authors propose to group tasks sharing a resource into a single component and to use the hierarchical scheduling model to schedule the tasks and the component. The purpose of this approach is to avoid global resource sharing in a multi-core system, but it limits the sum of the utilization of tasks sharing a resource to be less than one.

Real-time virtual machines have recently received much attention. RT-Xen [19, 38] is the first hierarchical real-time scheduling framework for the Xen hypervisor. RT-Xen implements a suite of fixed-priority servers for the VCPU budget replenishment policy. The work in [10] investigates the real-time performance of the L4/Fiasco microkernel-based hypervisor [1]. However, these approaches have not considered resource sharing and synchronization issues.

VII. Conclusions

In this paper, we have proposed vMPCP to provide bounded blocking time on accessing shared resources in a multi-core virtualization environment. vMPCP reduces the major inefficiencies caused by shared resources, by exposing the executions of global critical sections to the hypervisor. We have presented the schedulability analysis under vMPCP, with the periodic and deferrable server policies with and without the budget overrun mechanism. From our analysis and experimental results, we made two important findings: (i) the deferrable server outperforms the periodic server when overrun is used, and (ii) the use of overrun does not always yield better schedulability, especially for tasks with long critical sections. We also have implemented vMPCP on the KVM hypervisor and demonstrated the effect of vMPCP in reducing task response times by an average of 29% in our case study.

There are several directions for future work. First, we would like to extend real-time cache [15] and DRAM [16, 37] management schemes to the virtualization environment. Second, more detailed theoretical and empirical evaluations remain to be conducted. Third, extending our schedulability analysis to the compositional framework [34, 35] and developing an efficient algorithm to choose VCPU parameters are also interesting topics. Lastly, we plan to implement vMPCP on other hypervisors, such as L4/Fiasco [1], and port to other architectures, such as ARM.

References

[1] L4/Fiasco Microkernel. https://os.inf.tu-dresden.de/fiasco.

[2] OKL4 Microvisor. http://www.ok-labs.com.

[3] SYSGO PikeOS Embedded Virtualization. http://sysgo.com.

[4] M. Asberg et al. Resource sharing using the rollback mechanism in hierarchically scheduled real-time open systems. In *RTAS*, 2013.

[5] T. P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67–99, 1991.

[6] M. Behnam et al. SIRAP: a synchronization protocol for hierarchical resource sharingin real-time open systems. In *EMSOFT*, 2007.

[7] G. Bernat and A. Burns. New results on fixed priority aperiodic servers. In *RTSS*, 1999.

[8] K. Bletsas et al. Errata for three papers (2004-05) on fixed-priority scheduling with self-suspensions. Technical Report CISTER-TR-150713, CISTER, 2015.

[9] A. Block et al. A flexible real-time locking protocol for multiprocessors. In *RTCSA*, 2007.

[10] F. Bruns et al. An evaluation of microkernel-based virtualization for embedded real-time systems. In *ECRTS*, 2010.

[11] R. I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *RTSS*, 2005.

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

[13] P. Gai et al. A comparison of MPCP and MSRP when sharing resources in the Janus multiple-processor on a chip platform. In *RTAS*, 2003.

[14] M. Joseph and P. K. Pandya. Finding response times in a real-time system. *Comput. J.*, 29(5):390–395, 1986.

[15] H. Kim et al. A coordinated approach for practical OS-level cache management in multi-core real-time systems. In *ECRTS*, 2013.

[16] H. Kim et al. Bounding memory interference delay in cots-based multi-core systems. In *RTAS*, 2014.

[17] A. Kivity et al. KVM: the Linux virtual machine monitor. In *Linux Symposium*, volume 1, pages 225–230, 2007.

[18] K. Lakshmanan et al. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *RTSS*, 2009.

[19] J. Lee et al. Realizing compositional scheduling through virtualization. In *RTAS*, 2012.

[20] H. Leontyev and J. H. Anderson. A hierarchical multiprocessor bandwidth reservation scheme with timing guarantees. *Real-Time Systems*, 43(1):60–92, 2009.

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

[22] F. Nemati, M. Behnam, and T. Nolte. Multiprocessor synchronization and hierarchical scheduling. In *ICPPW*, 2009.

[23] F. Nemati, M. Behnam, and T. Nolte. Independently-developed real-time systems on multi-cores with shared resources. In *ECRTS*, 2011.

[24] S. Oikawa and R. Rajkumar. Linux/RK: A portable resource kernel in Linux. In *RTSS Work-In-Progress*, 1998.

[25] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *ICDCS*, 1990.

[26] R. Rajkumar. Dealing with suspending periodic tasks. *IBM Thomas J. Watson Research Center*, 1991.

[27] R. Rajkumar et al. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *SPIE/ACM Conference on Multimedia Computing and Networking*, 1998.

[28] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *RTSS*, 1988.

[29] R. Russell. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[30] S. Saewong, M. H. Klein, R. R. Rajkumar, and J. P. Lehoczky. Analysis of hierarhical fixed-priority scheduling. In *ECRTS*, 2002.

[31] L. Sha et al. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. on Comp.*, 39(9):1175–1185, 1990.

[32] L. Sha, J. P. Lehoczky, and R. Rajkumar. Solutions for some practical problems in prioritized preemptive scheduling. In *RTSS*, 1986.

[33] I. Shin, A. Easwaran, and I. Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *ECRTS*, 2008.

[34] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *RTSS*, 2003.

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

[36] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers*, 44(1):73–91, 1995.

[37] N. Suzuki et al. Coordinated bank and cache coloring for temporal protection of memory accesses. In *ICESS*, 2013.

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