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 and multi-core systems. Recently, some researchers have applied the real-time hierarchical scheduling theory to virtualization environments, such as RT-Xen.

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 and MSRP, can lead to excessive blocking times due to the preemption and budget depletion of VCPUs. Available solutions in the uni-core hierarchical scheduling context 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 parallelization 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 and deferrable server 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 replenishment period of VCPU $v_i$

\(^1\)The superscript $v$ denotes that the parameter is a VCPU parameter.

## 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 $\tau_i$ has lower priority than task $\tau_j$. Each task has an alternating sequence of normal execution segments and critical section segments. Task $\tau_i$ is thus represented as follows:

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

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

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

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

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 PCPs. 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 \( \tau_i \). The function \( \text{type}(\tau_i, j) \) returns gcs or lcs, which is the type of the \( j \)-th critical section of \( \tau_i \). In addition, we use \( S_i^{gcs} \) and \( S_i^{lcs} \) to denote the number of global and local critical section segments of \( \tau_i \), 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 \( \tau_i \)
- \( P(v_i) \): 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 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 \( \tau_i \) tries to access a global resource held by another task on a different core, task \( \tau_i \) suspends by itself until the resource-holding task finishes its corresponding critical section. Multiprocessor real-time synchronization protocols such as MPCP 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.

- **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, 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.

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 \( \tau_i \) 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 \( \tau_i \) 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) 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...

---

More information on this issue can be found in [18].
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 \(\tau_i\) 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 \(\tau_i\) requests an access to a global resource \(R_k\), the resource \(R_k\) can be granted to the task \(\tau_i\), if it is not held by another task.

3) While a task \(\tau_i\) in a VCPU \(v_j\) is holding a resource for its global critical section (gcs), the priority of \(\tau_i\) is raised to \(\pi_B, v_j + \pi_i\), where \(\pi_B, v_j\) is a base task-priority level greater than that of any task in the VCPU \(v_j\), and \(\pi_i\) is the normal priority of \(\tau_i\). We refer to \(\pi_B, v_j + \pi_i\) as the task-level priority ceiling of the gcs of \(\tau_i\).

4) While a task \(\tau_i\) executes a gcs, the priority of its VCPU \(v_j\) is raised to \(\pi_B + \pi_j\), where \(\pi_B\) is a base VCPU-priority level greater than that of any other VCPUs in the system, and \(\pi_j\) is the normal priority of the VCPU \(v_j\). We refer to \(\pi_B + \pi_j\) as the VCPU-level priority ceiling of the gcs of \(\tau_i\).

5) When a task \(\tau_i\) requests access to a resource \(R_k\), the resource \(R_k\) cannot be granted to \(\tau_i\), if it is already held by another task. In this case, the task \(\tau_i\) 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) \(4\) for accessing local resources. 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 \(\tau_i\) can access a local resource \(R_k\), if the priority of \(\tau_i\) is higher than the priority ceilings of any other mutexes currently locked by other tasks in that VCPU.

3) If a task \(\tau_i\) is blocked on a local resource by another task that has a lower priority than \(\tau_i\), the lower-priority task inherits the priority of \(\tau_i\).

**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 \(\tau_i\) waiting for a global resource \(R\) that is held by another task on a different physical core. The VCPU of \(\tau_i\) is currently suspended due to its budget depletion. If the resource \(R\) is released while the VCPU of \(\tau_i\) is suspended, the task \(\tau_i\) 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 \(\tau_i\) waiting for a global resource \(R\) that is held by another task on different physical core. The VCPU of \(\tau_i\) has exhausted its regular budget. If the resource \(R\) is released, the VCPU of \(\tau_i\) is allowed to overrun its budget and the task \(\tau_i\) can execute its gcs corresponding to \(R\). Once the task \(\tau_i\) finishes its gcs, the VCPU of \(\tau_i\) 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 \([12]\).

**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 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 \(\pi_B\) 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

---

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

4Para-virtualization is a technique involving small modifications to guest operating systems or device drivers to achieve high performance and efficiency.
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 block the higher-priority VCPU \( v_h \) multiple times during \( v_l \)'s period. The maximum accumulated global resource holding time of the tasks of \( v_l \) during a time interval \( t \) is given by:

\[
\text{sum}_\text{ght}(v_l, t) = \sum_{\tau_j \in v_l} \left( \left\lfloor \frac{t}{T_j} \right\rfloor + 1 \right) \cdot \sum_{1 \leq k \leq S_j, \text{type}(\tau_j, k) = \text{gcs}} E_{j,k}
\]

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:

\[
B^i_v(t) = \sum_{v_i \in P(v_i) \setminus v_h \leq i} \text{sum}_\text{ght}(v_i, t)
\]

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:

\[
B^i_v(t) = \sum_{v_i \in P(v_i) \setminus v_h \leq i} \text{sum}_\text{ght}(v_i, t)
\]

**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 \( v_i \) (\( O^i_t \)) is equal to \( \text{ght}(v_i) \) 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:

\[
W^{n+1}_i = C^i_n + O^i_n + B^i_v(W^{n+1}_i) + \sum_{v_h \in P(v_i) \setminus v_h \leq i} \left[ \frac{W^{n+1}_i + J^i_h}{T^i_h} \right] \cdot (C^i_h + O^i_h)
\]

where \( W^{n+1}_i \) is the worst-case response time of \( v_i \) at the \( n \)th iteration \((W^{0+1}_i = C^i_0 + O^i_0)\) and \( J^i_h \) is a VCPU release jitter \((J^i_h = 0 \text{ under the periodic server policy and } J^i_h = T^i_h - C^i_h \text{ under the deferrable server policy})\). Eq. (5) is based on the iterative response time test [14]. It terminates when \( W^{n+1}_i = W^{n+1}_i \), and the VCPU \( v_i \) is schedulable if its response time does not exceed its period: \( W^{n+1}_i < T^i \).

In this equation, \( O^i_t \) and \( O^i_h \) are used to represent the budget overrun of \( v_i \) and its higher-priority VCPUs, respectively. 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 \( \tau_i \) 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 remote 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 \( \tau_i \) in a VCPU \( v_k \) under vMPCP:

\[
W^{n+1}_i = C^i_n + B^i_t + \sum_{\tau_h \in V(\tau_i) \setminus v_h \leq i} \left[ \frac{W^n_i + J^i_h + (W^i_h - C^i_h)}{T^i_h} \right] C^i_h + \left[ \frac{W^n_i + C^n_k}{T^i_k} \right] (T^i_k - C^i_k)
\]

5vMPCP 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.
where \( B_i^l \) is the local blocking time for \( \tau_i \), \( B_i^r \) is the remote blocking time for \( \tau_i \), and \( J_h \) is the release jitter of each higher-priority task \( \tau_h \). It terminates when \( W_i^{n+1} = W_i^n \) and the task \( \tau_i \) 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 \( \tau_h \) is captured by adding \( W_h - C_h \) in the summing term.

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 \( \tau_i \). Similarly, \( vc_{i,j} \) is used to represent the VCPU-level priority ceiling of the \( j \)-th critical section segment of task \( \tau_i \).

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

\[
B_i^{l,\text{cs}} = \max_{\tau_j \in V(\tau_i) \land n_i} \left( 1 \leq u \leq S_i \land \text{type}(\tau_i, u) = \text{cs} \right) E_{l,u} \tag{7}
\]

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

\[
B_i^{gcs} = \sum_{\tau_j \in V(\tau_i) \land n_i} \left( 1 \leq u \leq S_i \land \text{type}(\tau_i, u) = gcs \right) E_{l,u} \tag{8}
\]

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

\[
B_i^l = (B_i^{l,\text{cs}} + B_i^{gcs}) \cdot (S_i^{gcs} + 1) \tag{9}
\]

Here, the reason for multiplying by \( S_i^{gcs} + 1 \) is that, before a task \( \tau_i \) executes or whenever \( \tau_i \) 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 \( \tau_i \) is given by:

\[
B_i^r = \sum_{1 \leq j \leq S_i \land \text{type}(\tau_i, j) = gcs} B_{i,j}^r \tag{10}
\]

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 \( [3] \).

**Remote blocking time:** The remote blocking time \( B_i^r \) of \( \tau_i \) in acquiring the global resource associated with the \( j \)-th critical section of \( \tau_i \). Note that \( B_{i,j}^r = 0 \) if the \( j \)-th critical section of \( \tau_i \) is a lcs.

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

\[
B_{i,j}^{r,n+1} = \max_{V(\tau_i) \land \text{type}(\tau_i, u) = \text{gcs}} W_{l,u}^{gcs} + \sum_{V(\tau_j) \land \text{type}(\tau_j, u) = \text{lcs}} \left( \left[ B_{i,j}^{r,n} \right] + 1 \right) W_{h,u}^{gcs} \tag{11}
\]

where \( B_{0,j}^{r} = \max_{V(\tau_j) \land \text{type}(\tau_j, u) = \text{lcs}} W_{l,u}^{gcs} \) (the first term of the equation), \( \text{hpcpus}(V(\tau_j)) \) is the set of lower-priority VCPUs than the VCPU of \( \tau_j \) in the system, \( \text{hpcpus}(V(\tau_i)) \) is the set of higher-priority VCPUs than the VCPU of \( \tau_i \), 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, \( \text{load}_{l,u} \) and \( \text{vcpu}_{\text{prm}}_{l,u} \), as follows:

\[
\text{load}_{l,u} = E_{l,u} + \sum_{\tau_x \in V(\tau_i)} \left( 1 \leq y \leq S_x \land \text{type}(\tau_i, y) = \text{gcs} \right) E_{x,y} \tag{12}
\]

\[
\text{vcpu}_{\text{prm}}_{l,u} = \sum_{\tau_x \in V(\tau_i)} \sum_{\tau_z \in V(\tau_i)} \left( 1 \leq y \leq S_z \land \text{type}(\tau_i, y) = \text{gcs} \right) E_{x,y} \tag{13}
\]

The term \( \text{load}_{l,u} \) 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 \( \text{vcpu}_{\text{prm}}_{l,u} \) 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 \( \text{vcpu}_{\text{prm}}_{l,u} \) 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 \( T_{V_i} - C_{V_i}^{\tau_i} \) 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, \( W_{l,u}^{gcs} \)
Other VCPUs

VCPU $V(\tau_i)$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$

$C'(V(\tau_i))$
We have implemented vMPCP on the KVM hypervisor. Implementation: We have implemented vMPCP on the KVM hypervisor.

### 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 hypervisor. (Kernel-based Virtual Machine) hypervisor [17] of the latest version of Linux/RK [24, 27]. 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 synchronizatio-
TABLE II: Implementation cost of vMPCP on KVM

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

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