CPS: A Cooperative Para-virtualized Scheduling Framework for Manycore Machines

Yuxuan Liu
Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University
Shanghai, China

Tianqiang Xu
Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University
Shanghai, China

Zeyu Mi*
Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University
Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China
Shanghai, China

Zhichao Hua
Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University
Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China
Shanghai, China

Binyu Zang
Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University
Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China
Shanghai, China

Haibo Chen
Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University
Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China
Shanghai, China

Abstract

Today's cloud platforms offer large virtual machine (VM) instances with multiple virtual CPUs (vCPU) on manycore machines. These machines typically have a deep memory hierarchy to enhance communication between cores. Although previous researches have primarily focused on addressing the performance scalability issues caused by the double scheduling problem in virtualized environments, they mainly concentrated on solving the preemption problem of synchronization primitives and the traditional NUMA architecture. This paper specifically targets a new aspect of scalability issues caused by the absence of *runtime hypervisor-internal states (RHS)*. We demonstrate two typical RHS problems, namely the invisible pCPU (physical CPU) load and dynamic cache group mapping. These RHS problems result in a collapse in VM performance and low CPU utilization because the guest VM lacks visibility into the latest runtime internal states maintained by the hypervisor, such as pCPU load and vCPU-pCPU mappings. Consequently, the guest VM makes inefficient scheduling decisions.

To address the RHS issue, we argue that the solution lies in exposing the latest scheduling decisions made by both the guest and host schedulers to each other. Hence, we present a cooperative para-virtualized scheduling framework called CPS, which facilitates the proactive exchange of timely scheduling information between the hypervisor and guest VMs. To ensure effective scheduling decisions for VMs, a series of techniques are proposed based on the exchanged information. We have implemented CPS in Linux KVM and have designed corresponding solutions to tackle the two RHS problems. Evaluation results demonstrate that CPS significantly improves the performance of PARSEC by 81.1% and FxMark by 1.01x on average for the two identified problems.

CCS Concepts

• **Computer systems organization → Multicore architectures**; • **Software and its engineering → Virtual machines**; **Operating systems**.

Keywords

Para-virtualized Scheduling, Cache Group, Manycore Machine, Performance Scalability

ACM Reference Format:
Yuxuan Liu, Tianqiang Xu, Zeyu Mi, Zhichao Hua, Binyu Zang, and Haibo Chen. 2023. CPS: A Cooperative Para-virtualized Scheduling Framework for Manycore Machines. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3623278.3624762

*Corresponding author: Zeyu Mi (yzmizeyu@sjtu.edu.cn).

1 Introduction

With the widespread adoption of manycore machines, modern public clouds offer the ability to provision large virtual machine (VM) instances with numerous virtual CPUs (vCPU). For instance, Amazon AWS and Alibaba Cloud provide VM instances with 96 vCPUs [2] and 128 vCPUs [1], respectively. In virtualized environments, the notorious double scheduling problem results in poor scalability of VM performance [45], presenting itself in different variations. One common issue is lock-holder preemption (LHP) [30], where the hypervisor preempts the vCPU holding a lock, causing wasted time slices for other vCPUs waiting for the lock. Other examples include lock-waiter preemption (LWP) [48], blocked-waiter wakeup (BWW) [25, 44], and RCU reader preemption (RRP) [42]. These problems arise due to the semantic gap between the hypervisor and guest VMs, as the hypervisor remains unaware of the internal activities within the VMs. Consequently, a VM may experience preemption during its critical section. Therefore, previous research efforts have mainly focused on proposing various techniques to overcome the semantic gap of virtualized locks from the hypervisor's perspective. These approaches either enhance the hypervisor's capabilities to infer guest behaviors using hardware features (like Pause Loop Exiting in Intel processors) [28], or provide a para-virtualized (PV) interface allowing a guest VM to explicitly communicate critical section information to the underlying hypervisor [32, 48].

This paper distinguishes itself from previous researches on synchronization primitives by focusing on the scalability issues stemming from the absence of *runtime hypervisor-internal states (RHS)*. The *RHS* problem exhibit two distinct characteristics that set it apart from existing problems. First, the problem is caused by the guest's inability to access hypervisor-internal states that are not specific to any particular guest VM, such as runtime physical CPU (pCPU) loads. Second, even when the guest VM is capable of obtaining hypervisor-internal states, the *RHS* problem may persist if the states become outdated.

For example, a guest OS lacks knowledge about the current load of a physical CPU (pCPU), and it may unintentionally schedule a task onto a preempted vCPU on a busy pCPU. Consequently, the task experiences an unpredictable scheduling latency before the vCPU is rescheduled to run. To address this issue, the hypervisor has already introduced PV APIs to inform the guest OS whether a vCPU has been preempted, allowing the guest to migrate tasks to online vCPUs and avoid scheduling latency. However, the PV API-based guest scheduler lacks awareness of the latest pCPU loads and continues to migrate tasks to non-preempted vCPUs, overlooking the potential benefits of utilizing the rapidly waking preempted vCPUs on low-load CPUs. Hence, the mechanism fails to efficiently utilize multiple low-load pCPU in scenarios with under-committed resources, let alone in an over-committed case where there exist some pCPUs that dynamically enter the low-load state. Therefore, the mechanism results in low pCPU utilization and has a negative impact on VM scalability.

The *RHS* problem becomes even more pronounced with the introduction of new micro-architectural features in manycore machines. Some commercial manycore machines employ deep memory hierarchy levels to enhance inter-core communications. Within this memory hierarchy, a single L3 cache partition/slice (referred to as a cache group or CG in this paper) is shared by 2 to 6 physical cores within a NUMA (Non-Uniform Memory Access) node. Threads running on cores within the CG exhibit higher communication throughput compared to those outside the CG [23, 35, 57]. This feature, known as NUCA (Non-Uniform Cache Access), is present in mature x86 servers as well as emerging ARM servers. For instance, in an x86-based server with AMD EPYC processors [3], a CG consists of 3 cores/6 hyperthreads, while in an ARMv8-based server with Kunpeng 920 processors [6], a CG is shared by 4 cores. However, it is challenging to fully leverage the high throughput within a CG for virtual machines (VMs). Although the hypervisor can schedule vCPUs to a CG to create a virtual CG (vCG), the guest scheduler is unable to take advantage of the vCG due to its lack of knowledge regarding the runtime mappings between vCPUs and pCPUs. Moreover, making a VM aware of this information is difficult since the hypervisor constantly migrates vCPUs across pCPUs to mitigate CPU load imbalances, causing the information to quickly become outdated.

In this paper, we argue that **it is necessary to expose the runtime hypervisor-internal information to guest VMs to address the RHS problem**. Therefore, we present CPS, a Cooperative Para-virtualized Scheduling framework that enables a dynamic and bidirectional way for the hypervisor and its VMs to exchange timely internal information. In CPS, a shared structure (Refer-Table) is carefully designed to expose the hypervisor's information to each VM. A backend module in the hypervisor dynamically updates the shared structure upon changes in related information. A frontend module in each guest VM guides the guest's scheduler to make effective decisions based on the Refer-Table. CPS also includes a set of PV APIs for the frontend to deliver hints to the hypervisor when the guest OS needs cooperation from the hypervisor to make guest scheduling decisions.

User-level thread mechanisms for the non-virtualized environment have previously explored various cooperative strategies [14, 16, 37]. These mechanisms enable communication between user-level threads and the OS kernel, with the aim of improving performance and mitigating system integration issues. Although CPS also adopts a cooperative scheduling framework, it distinguishes itself from existing systems through two fundamental aspects. First, in terms of shared information, CPS offers a novel insight by proactively exposing hypervisor-internal states (such as runtime pCPU loads and CG topology) to guest VMs. In contrast, previous user-level threading mechanisms focus solely on userspace-related information. Second, in terms of hardware considerations, existing systems have not explored the impact of the deep memory hierarchy, while CPS provides an efficient frontend that assists guest VMs in effectively utilizing dynamic hardware information, such as CGs.

We have implemented CPS within the Linux KVM [22, 34]. By employing the CPS framework, we can effectively address the RHS problems mentioned earlier. To tackle the issue of preempted vCPU, CPS records the runtime load of each pCPU and communicates this load degree to the corresponding VM. Using this information, the frontend can schedule tasks to preempted vCPUs that are running on low-load pCPUs to harvest the low scheduling latency. This strategy enhances application performance on a single VM by an average of 59%. In an over-committed scenario involving multiple concurrently executing VMs, CPS improves application performance by 1.03x on average. This improvement is attributed to CPS enabling VMs to fully utilize dynamic low-load pCPUs. To leverage CG and enhance application performance, CPS maintains real-time mappings from vCPUs to physical CGs. Additionally, we propose CG-aware scheduling, which guides the guest scheduler in grouping interactive threads within the same CG to maximize their communication throughput. For applications that can fit their interactive threads into a single CG, CPS offers an average performance boost of 1.40x. For applications with a larger number of threads, CPS achieves an average performance improvement of 62.4% (up to 1.21x) in both under-committed and over-committed scenarios.

In this paper, we make the following contributions:

- We introduce the *RHS* problem and analyze its two cases in large VM instances running on manycore servers. (§ 2.2)
- We present CPS, a para-virtualized scheduling framework to mitigate the *RHS* problem in virtualized environments. (§ 3 and § 4)
- We address two types of *RHS* problems with CPS by exposing the pCPU load degree and CG-aware scheduling. (§ 5)
- We evaluate CPS through real-world applications and show that CPS can improve the performance of PARSEC by 81.1% and FxMark by 1.01x on average for the two problems. (§ 6)

2 Motivation and Background

2.1 Double Scheduling and Prior Efforts

Double scheduling is a classic problem that arises from the semantic gap between guest VMs and the hypervisor [24, 25, 30, 44, 48]. Due to the inability to fully grasp the VM's behavior semantics, the hypervisor may preempt a vCPU that is executing critical tasks. Consequently, the inopportune preemption of the vCPU holding a spinlock forces other vCPUs, which are awaiting the lock, to spin for an extended period. This then leads to the Lock Holder Preemption problem (LHP) [30]. Furthermore, if the preempted vCPU is the lock waiter assumed to acquire the lock in a strict order, other vCPUs still have to wait for its wakeup, resulting in Lock Waiter Preemption (LWP) [48]. Previous efforts, mainly from the hypervisor's perspective, offer various solutions to address the double scheduling problem. One approach involves leveraging hardware features (such as Pause Loop Exit) [4, 8] to assist the hypervisor in detecting excessive spinning by a vCPU and reschedule the vCPU that is the root cause of the problem to mitigate the excessive spinning [28]. Another direction focuses on developing a para-virtualized (PV) interface to bridge the semantic gap between the VM and the hypervisor [32, 41, 48, 49]. The third option revolves around optimizing the hypervisor's scheduler by employing gang scheduling and simultaneously scheduling all vCPUs of a VM [21, 33, 38, 45, 46, 50, 53, 55, 58, 60]. But this approach can lead to significant CPU fragmentation. Although the three categories of efforts partially address lock-based double scheduling issues, none of them looks into the discrepancy in scheduler behavior between the guest kernels and the hypervisor, especially the scalability issues caused by the RHS problem.

2.2 Runtime Hypervisor-internal States (RHS)

This paper addresses a distinct scheduling problem that is gaining prominence in manycore machine scenarios. In the existing

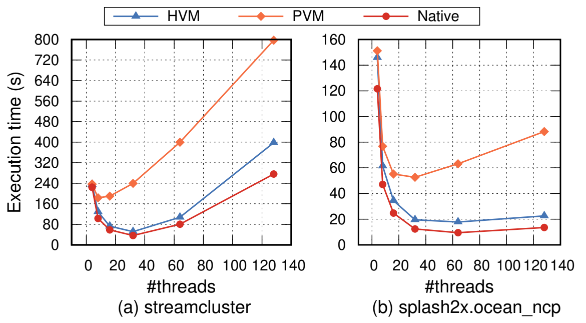

Fig. 1. Negative performance effects of the pvsched optimization.

guest kernels, scheduling decisions are primarily based on their own internal states, such as identifying the vCPU with the lowest load. However, guest kernels lack visibility into the crucial *Runtime Hypervisor-internal States (RHS)* factors, such as the runtime pCPU load managed by the hypervisor. This absence of information can result in suboptimal scheduling decisions made by guest kernels.

In addition to the two properties discussed in § 1, there are three more fundamental differences between the RHS problem and lock-based problems. First, the root cause of lock-based problems is typically attributed to untimely vCPU preemptions by the hypervisor, whereas the RHS problem originates from suboptimal scheduling decisions made by the guest OS. Second, lock-based problems are typically limited to the preemption of critical sections of the lock holder or waiter. In contrast, the RHS problem is more general in nature and encompasses the scheduling of tasks within a vCPU, irrespective of whether or not the task is holding a lock. Third, the solution to a lock-based problem generally involves allocating an extra time slot to a specific vCPU temporarily. Conversely, the solution to the RHS problem involves selecting the optimal vCPU from multiple candidates for a given task. This selection process necessitates considering the conditions of all candidate vCPUs based on runtime hypervisor states, such as vCG and pCPU load.

In this section, we will examine two common scenarios of the RHS problem in a manycore machine. The detailed settings of this machine are outlined in § 6.2.

Invisible pCPU Load. In an over-committed scenario, where multiple VMs' vCPUs can potentially be scheduled on a single pCPU, only one vCPU is able to run at a time on that pCPU. As a result, the other vCPUs remain in a preempted state, waiting in the pCPU's run queue. If the guest scheduler assigns a task to a preempted vCPU, the task has to wait until the hypervisor reschedules that vCPU. Therefore, the scheduling latency for the task on a preempted vCPU becomes unpredictable.

The widely-deployed Linux KVM hypervisor offers a PV mechanism called **pvsched** [7], which allows the guest scheduler to determine if a vCPU has been preempted. With this API, the guest scheduler can choose to schedule tasks on online vCPUs rather than preempted ones, avoiding the uncertainty and prolonged latency associated with task scheduling on preempted vCPUs. However, in scenarios where some pCPUs in the machine are under low stress, our

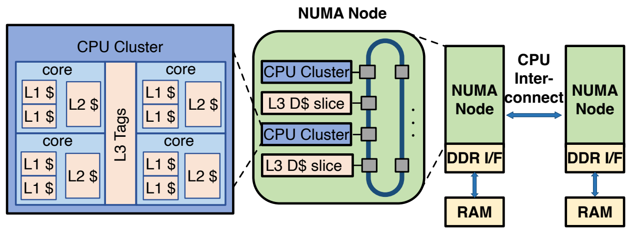

Fig. 2. A deep memory hierarchy consists of NUMA nodes and cache slices (groups).

investigation reveals that enabling the PV API can result in performance degradation and exhibit even worse performance than VMs without using the API. As depicted in Fig. 1, when running a VM with 128 vCPUs on a physical machine with 128 pCPUs, the benchmarks with pvsched-enabled (referred to as PVM, Para-virtualized VM) exhibit significantly poorer performance compared to benchmarks without pvsched (referred to as HVM, Hardware-assisted VM).

In the PVM case, whenever a new task starts or wakes up from a wait queue, the guest scheduler, with the assistance of the pvsched API, avoids assigning the task to any preempted vCPUs. However, in the under-committed case, where each pCPU only runs one vCPU, the preempted vCPUs can quickly acquire a time slice for execution without further waiting. Hence, the PVM fails to leverage the low-load pCPUs, which include the preempted vCPUs, resulting in low CPU utilization and poor scalability of the VM.

Table 1: Throughput of increment operation of an atomic variable among 4 threads located at different memory hierarchy. DS: two cores in different sockets. SS: two cores in different NUMA nodes in the same socket. SN: two cores in the same NUMA node. SCG: two cores in the same CG. Mops/s: Million operations per second.

Config	DS	SS	SN	SCG
Mops/s	2.97	4.45	4.76	7.08

Dynamic Cache Group Mapping. To facilitate efficient intercore communication, modern manycore systems incorporate deep memory hierarchies within each NUMA node [23, 35, 57]. Fig. 2 illustrates an example of such a deep memory hierarchy topology. In this configuration, a manycore server comprises two NUMA nodes, each of which can be further divided into multiple cache groups (CG) [1], which form a unified last-level cache. The access latency of physical cores to their local cache group is lower than their access latency to remote cache groups. Thus, interactive threads within the same CG experience lower communication latency compared to those in separate CGs. To demonstrate this, we conducted an experiment whose results are described in Table 1, where 4 threads increment a single atomic variable using different memory hierarchy settings. The results show that threads in the same CG exhibit higher throughput compared to those in separate CGs. It is important to note that the mapping from a physical core to a CG differs from the mapping of each cache line to a specific CG. The latter mapping is typically determined using an undocumented hashing function [35]. In this paper, our focus is on the pCPU-to-CG mapping, which can be easily detected by clustering physical cores based on the time it takes for each core to access individual CGs.

[1]Cache group is also referred to as a cache slice.

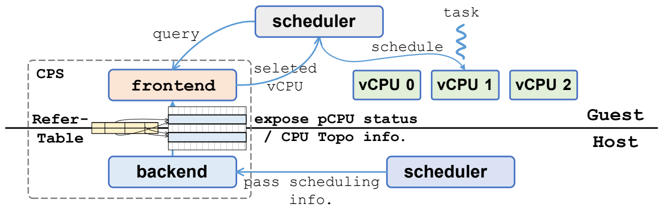

Fig. 3. The architecture of CPS.

The current OS schedulers lack interactive thread identification and do not possess knowledge of fine-grained cache groups (CGs). It should be noted that existing NUMA-related mechanisms cannot directly address the CG problem due to two main reasons. First, while the NUMA mechanism can easily identify interacting threads, CGs-related approaches struggle with this. The OS kernel dynamically disables the access privilege for page mappings and determines whether many threads are sharing a NUMA node by managing page faults. However, as CG behaviors are transparent to software, there are no system-level events, like page faults, designed to identify interactive threads for CGs [29, 47]. Second, virtualization further complicates this problem. Even if the guest scheduler is CG-aware, the hypervisor may frequently migrate vCPUs across CGs to mitigate CPU load imbalance [13]. This dynamic migration of vCPUs alters the mapping of vCPUs to physical CGs, which undermines the scheduling decisions made by the guest scheduler. While various existing works have explored solutions for efficient virtual NUMA structures [15, 18, 19, 27, 36, 43, 51, 54], it is challenging to applying them to the fine-grained CGs consisting of only 2-6 physical cores. This is because hypervisors tend to migrate vCPUs across CGs more frequently compared to NUMA nodes, given that CGs have significantly fewer cores than NUMA nodes.

3 Overview

To address RHS problem, this paper introduces CPS, a Cooperative Para-virtualized Scheduling framework that enables runtime collaboration between a guest VM and the hypervisor to make optimal scheduling decisions. The core concept behind CPS is to establish a cooperative and interactive relationship between the guest VM and the hypervisor, allowing them to proactively exchange their recent scheduling decisions and internal states in a dynamic and bidirectional manner. This includes sharing information such as pCPU load status and mappings from vCPUs to physical cache groups.

The architecture overview of CPS is presented in Fig. 3, consisting of three major components. First, each VM possesses an isolated data structure called the Refer-Table, which is mapped into the VM address space by the hypervisor. The Refer-Table is updated by the hypervisor to provide runtime information to the guest. Second, the hypervisor is extended with a backend module

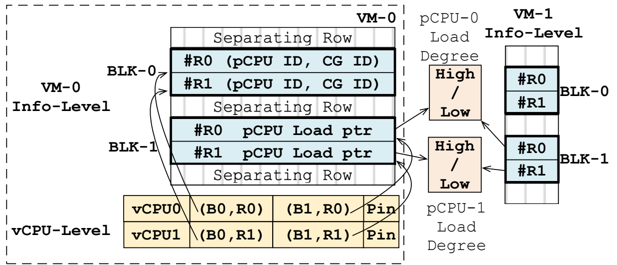

Fig. 4. Refer-Table consists of two levels of blocks, vCPU-level and Info-level blocks, shared by the guest and the hypervisor to mitigate the RHS problem efficiently.

responsible for gathering information from the host scheduler. It dynamically measures the runtime load of each physical core and maintains mappings between vCPUs and physical cache groups (CGs). Whenever the host scheduler makes a scheduling decision, the backend module reevaluates the pCPU load in the Refer-Table following changes in load and updates the exposed vCPU-pCPU mappings if vCPU migration occurs. Third, a frontend module needs to be installed on each guest VM that intends to leverage CPS for improved scalability. The guest scheduler queries the frontend module to obtain guidance on scheduling decisions based on the latest information provided by the backend module. For example, whenever the guest OS schedules a task, it invokes the frontend module to select a suitable vCPU for the task. The frontend module checks whether the target pCPU is in a low-load state or if other interacting threads share the same CG. Furthermore, the frontend module updates the Refer-Table to offer scheduling hints to the backend module, thus guiding the hypervisor in making effective scheduling decisions.

We have devised a series of techniques to ensure the security and general applicability of CPS. Regarding security considerations, CPS follows the principle of exposing only essential information to the guest while avoiding the disclosure of concrete and specific hypervisor details. For instance, the backend module refrains from providing the frontend module with precise details such as the exact number of co-running vCPUs or the specific pCPU ID associated with each vCPU. Instead, the backend module presents a more vague pCPU status, indicating whether it is under low or high load and which vCPUs are sharing the same CG. To enhance the general flexibility of CPS, we have designed a two-level hierarchy for organizing the shared Refer-Table, offering support and accommodation for future extensions. This design enables the incorporation of future extensions by adding a new second level that can be indexed by the existing first level.

4 Design

This section describes the detailed designs of CPS, including its PV interface (§ 4.1), the frontend module in each guest (§ 4.2), and the backend module in the hypervisor (§ 4.3).

4.1 Para-virtualized Interface

CPS provides a PV interface for exchanging runtime information between the frontend and backend. The frontend module utilizes this interface to retrieve the hypervisor's internal states, while the backend module actively exposes this information to each guest OS. The interface is implemented using shared memory pages, referred to as the Refer-Table (shown in Fig. 4), which are shared between these two modules. Note that each VM has its own isolated Refer-Table that is separate from those of other VMs.

During the VM startup process, the frontend module allocates a memory page and invokes a hypercall to provide the page address to the hypervisor. The shared page is then initialized with default values by the hypervisor. Following this initialization, the guest can communicate with the host scheduler simply by accessing the shared memory, without the need for any additional hypercalls. As different vCPUs read distinct portions of the shared page, there is no need to set a lock for multiple vCPUs to access the page concurrently.

The Refer-Table consists of two levels of blocks: the vCPU-level and the Info-level, allowing for future extensions to incorporate additional internal information. Each specific vCPU is represented by a row in the vCPU-level block, which is organized based on vCPU IDs in sequential order. The Info-level blocks are indexed by BLK IDs, with each block containing a specific type of information. In the current design, the Refer-Table contains two types of Info-level blocks: pCPU load degrees and pCPU-CG mappings. To efficiently retrieve information related to a particular vCPU, each row in the vCPU-level block contains multiple (BLK ID, Row ID) pairs, which point to different rows within the Info-level blocks. For example, the *(0, 1)* pair indicates Row-1 in BLK-0, which contains the load degree information for the pCPU that the vCPU locates. If the Refer-Table needs to support a new type of hardware-related information, a new Info-level block will be allocated to track this information. Additionally, each vCPU-level row will have an additional (BLK ID, Row ID) pair to indicate the location of this new information.

The Refer-Table supports two methods for storing data in Info-level blocks: the direct and indirect approach. The direct method involves writing information directly into each row, which is suitable for storing **vCPU-specific** information. An example of this is illustrated in Fig. 4, where the BLK-0 block records a 16-byte pCPU topology information for a vCPU (i.e., pCPU ID, CG ID) directly. The CPU topology information contains virtual values rather than real ones. For instance, the virtual CG (vCG) ID indicates which pCPUs share the same CG. To prevent the exploitation of this value, different VMs perceive different vCG IDs. This ensures that each VM operates with its own isolated view of the CPU topology. When the hypervisor needs to modify the CPU topology for a particular vCPU, it simply adjusts the corresponding BLK-0 row.

However, the direct approach does not align well with the **pCPU-specific** information, as it can lead to significant performance costs when updating this type of information. To demonstrate this limitation, let's consider the example of runtime pCPU load. When two vCPUs belonging to different VMs share a single pCPU, the direct approach mandates that the corresponding rows in the Refer-Tables for these VMs both record the same load value. This leads to a challenge in scenarios where numerous VMs are simultaneously running on a single physical server. In such cases, updating the pCPU's load degree for every scheduling decision becomes a bottleneck as the backend module must sequentially traverse all the Refer-Tables of these VMs.

Therefore, the Refer-Table implements an indirect method that improves the updating efficiency of pCPU-specific information. CPS creates a shared page that is accessible to multiple VMs. Instead of writing the page data directly into the block rows, the Refer-Table stores the base address (GVA) of the shared page. Consider an example where two VMs share a single pCPU. In this scenario, the backend module allocates a physical page specifically for the pCPU load degree. This page is then mapped as read-only to the address spaces of both VMs. Fig. 4 provides a concrete example of this page mapping, where the first rows of BLK-1 for the two VMs respectively point to the same pCPU-0 load-degree page. Consequently, the hypervisor can modify only a single page without the need to traverse all the VM's Refer-Tables. As a result, all VMs simultaneously observe the newly updated value.

The Refer-Table also enables the guest OS to communicate its scheduling requirements to the hypervisor. For example, in order to group certain vCPUs with specific pCPUs and CGs, the frontend module sets a 4-byte "Pin" field in the corresponding rows of the vCPU-level block. This information assists the hypervisor in minimizing vCPU migration frequency, thereby avoiding unnecessary changes to vCPU-vCG mappings. However, it's important to note that the hypervisor isn't consistently bound by the reverse information provided by the VMs, as this information only serves as a hint or suggestion.

4.2 Frontend Module

Within the guest kernel, the frontend module incorporates two mechanisms to assist in making optimal scheduling decisions. First, the frontend provides an interface for applications to deliver their scheduling requirements to the guest kernel. This is achieved through the creation of a *cps* file under the */dev* directory. Applications intending to utilize CPS must create this file and send commands to the frontend using the *ioctl* system call. One typical command is (*define_interactive_threads*), which allows the application to identify which thread interacts with each other and should be placed in the same CG.

Furthermore, the frontend module integrates the runtime information exposed by the backend and guides the guest scheduler in utilizing runtime hypervisor-internal information. Suppose a task is created or a blocked task is awakened, the existing guest scheduler selects a vCPU for the task to run. This selection process is enhanced by invoking the **select_vcpu** function within the frontend module. In the **select_vcpu** function, the frontend module checks the Refer-Table for updates on pCPU information. If the target vCPU has been preempted by the hypervisor, but its associated pCPU is currently being underutilized, the frontend module guides the guest scheduler to run the task on that vCPU (will be explained in § 5.1).

Regarding the CG scenario, if the CG ID of a vCPU has been changed to a new value, it means that the task may run in a CG different from the one where the interactive threads reside. In this case, the frontend module informs the scheduler to migrate this task to the CG where the other interactive threads are located (as described in § 5.2).

4.3 Backend Module

The backend module is responsible for fetching scheduling decisions made by the hypervisor and updating the Refer-Table shared with the frontend module. In addition to recording hardware-related information, such as the load status of each pCPU, the backend provides an *update_cps_info* function for the hypervisor scheduler to awaken the backend. This function modifies the information stored by the backend, which subsequently updates the Refer-Table of VMs affected.

Furthermore, the backend module receives scheduling hints from the frontend module. For instance, the frontend delivers hints to indicate which vCPUs contain interactive threads. Each time a VM exit occurs for a vCPU, the hypervisor invokes the *check_hints* function within the backend module. The function checks the **Pin** field in the Refer-Table's vCPU-level. Based on this information, the backend determines if it should modify the corresponding *task* struct to set CPU affinity accordingly.

Nevertheless, the backend module is not obligated to always follow these hints provided by the frontend module. Instead, the backend module monitors the load of CGs and makes decisions accordingly. If there are already multiple vCPUs pinned to a specific CG, the backend module will avoid pinning the vCPU to that CG, adhering to load balancing principles.

5 Case Study

This section explains how we address the two types of RHS problems (described in § 2.2) via CPS.

5.1 Pload Scheduling

As explained in § 2.2, the problem of invisible pCPU load results in low CPU utilization and performance degradation in an undercommitted scenario. One intuitive solution to addressing this problem is to notify the guest scheduler of information about pCPU idleness using a boolean value [32]. However, this approach is not suitable for resolving the RHS problem in CPS since it could lead to a fairness issue when multiple VMs are running simultaneously. When a VM detects an idle pCPU and successfully schedules its tasks on that pCPU, it prevents other VMs from using that pCPU because it is no longer considered idle.

To mitigate this fairness issue, CPS implements pCPU load (referred to as Pload) scheduling based on the semantics of runtime pCPU load degrees instead of boolean idleness. Pload indicates how busy a pCPU is, categorized as either low or high load. CPS dynamically calculates the number of runnable vCPUs on each pCPU and determines its load degree accordingly. Considering that today's hypervisors typically enforce fair scheduling constraints, where multiple vCPUs from one VM are not allowed to co-run on a single pCPU, the load degree is less influenced by the limited number of vCPUs from one VM compared to the pCPU idleness.

CPS utilizes a Refer-Table for each VM to share the current runtime pCPU load degree information. The Refer-Table includes a block in the Info-level (BLK-1), and an indirect approach is employed to store the pCPU load degree values for each pCPU on which the VM is running. To retrieve the pCPU load degree value, the frontend module needs to dereference the GVA pointers stored in BLK-1 rows. In situations where the hypervisor migrates a vCPU

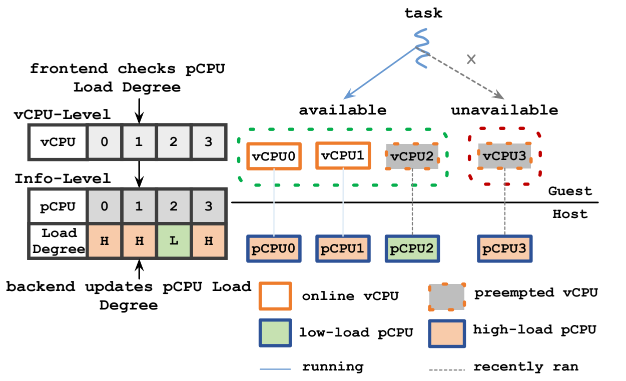

Fig. 5. The workflow of CPS-Pload. The backend updates the pCPU load degree promptly. The frontend checks pCPU loads and enlarges the available vCPU set with preempted but low-scheduling latency vCPUs (vCPU-2).

from one pCPU (pCPU-A) to another (pCPU-B), the backend module is responsible for unmapping the load page associated with pCPU-A from the VM's address space and remapping the GPA to the HPA of the load page corresponding to pCPU-B.

By utilizing the semantic of pCPU load degree, the schedule latency of low-load pCPUs will be predictable to the guest scheduler. In an under-committed scenario, the frontend module includes preempted vCPUs from low-load pCPUs in the available vCPU set. This allows the guest scheduler to distribute tasks among more vCPUs, reducing the length of each vCPU's run queue and minimizing scheduling latency for each task. Even in an over-committed scenario, where there are more active vCPUs than available pCPUs, a preempted vCPU may still have access to a low-load pCPU. This can be identified using CPS-Pload scheduling, leading to improved VM performance and enhanced hardware utilization. A working example of CPS-Pload functionality is illustrated in Fig. 5. When selecting a vCPU for a task, the frontend module first examines the pCPU load state (pCPU-3) of vCPU-3 that is the original vCPU the task ran on. If vCPU-3 has been preempted and the associated pCPU-3 is heavily loaded, the frontend module chooses the preempted vCPU-2 due to its low-load pCPU (pCPU-2).

In our current implementation, we employ a threshold to classify the pCPU load degree. The threshold value is calculated as half of the average vCPU count in the server. For instance, if there are two 128-vCPU VMs running on a 128-pCPU server, the threshold would be set to 1 ($= \frac{2\times128}{128} \times 0.5$). Importantly, the threshold is dynamically adjusted in response to the creation or removal of VMs. This ensures that the threshold remains accurate and aligned with the current system configuration.

5.2 CG-aware Scheduling

Although the CPS framework facilitates cooperation between the hypervisor and its VMs, determining the target vCPU for thread

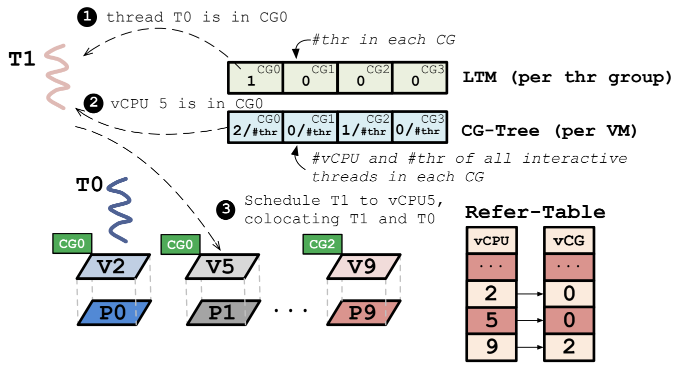

Fig. 6. The algorithm of CPS-CGsched. T0 and T1 are two interactive threads. The frontend uses LTM and CG-Tree to migrate T1 to vCPU-5, co-locating the two interactive threads.

migration to cluster interactive threads can be challenging, especially considering potential vCPU migrations triggered by the hypervisor scheduler. To this end, we propose a CG-aware scheduling mechanism called CPS-CGsched. The frontend module employs this mechanism to dynamically group threads into the same CG and enhance their communication. To accommodate dynamic changes in the CG topology, we extend CPS by enabling it to adjust to the actual CPU topology through the use of a Refer-Table. Additionally, the frontend module can provide hints to the backend module, indicating the need to reduce the frequency of vCPU migrations if necessary.

The concrete CPS-CGsched algorithm and accompanying data structures are shown in Fig. 6. Within the VM frontend, two distinct data structures are maintained. The first is the Local Thread Map (LTM), an array that represents the local process view of interactive thread distribution amongst vCGs during runtime. Each entry in the LTM array indicates the number of interactive threads (*ITN*) within a particular thread group running in a vCG. The second data structure is the CG-Tree, providing a global view of the VM's load status for each vCG. This information is leveraged by the frontend to prevent overloading of vCGs. Each entry in the CG-Tree contains two variables. The first variable is the total ITN of the VM's interactive threads within the corresponding vCG. The second variable is the runtime vCPU count within this vCG, which may be dynamically changed by the hypervisor and is calculated based on the Refer-Table's BLK-0.

Suppose that there is an application aiming to benefit from CPS. During the initialization phase of this application, it issues the *define_interactive_threads* command to inform the frontend module about the collection of threads engaged in inter-communication. Upon receiving this command, the frontend allocates an LTM for the specified thread group. The pointer to this LTM is stored in the *task* structs of these threads. Please note that an application might consist of multiple thread groups, and each group needs to be defined separately using the appropriate *ioctl* command, with each corresponding to their own dedicated LTM.

When scheduling a thread, the objective of the frontend module is to select a vCG that is not overloaded but contains the largest count of threads interacting with the thread being scheduled. To accomplish this, the frontend first calculates the maximum number of threads for each vCG (referred to as *vCG_Quota*). The *vCG_Quota* for a specific vCG is defined using the following formula:

$$vCG\_Quota = \max\{\#vCG\_vCPU, \lceil \frac{\#IT}{\#vCPU} \times \#vCG\_vCPU \rceil\}$$

#IT is the total number of interactive threads in the VM. *#vCPU* and *#vCG_vCPU* stand for the total vCPU count in the VM and this vCG, respectively.

Next, the frontend module retrieves the number of interactive threads (ITN) from the relevant entry in the Local Thread Map (LTM). It then selects the vCG with the largest ITN, provided it is below the calculated *vCG_Quota*. Subsequently, a vCPU within this chosen vCG is selected as the target vCPU. If the original vCPU on which the thread was running is different from the target vCPU, the thread is migrated from the original vCG to the target vCG. Moreover, the frontend module updates the ITN values in the two LTM entries within the process, as well as the corresponding CG-Tree entries within the VM.

Fig. 6 illustrates how the guest OS schedules a new interactive thread (T1) using the frontend module. Consider a scenario with 3 vCPUs and 2 threads within the VM. The frontend calculates the quota of vCG-0 as ($\max\{2, \lceil \frac{2}{3} \times 2 \rceil\} = 2$). Since vCG-0 has the largest ITN that is below this quota, T1 should be scheduled within vCG-0. The frontend then searches for another vCPU that shares vCG-0 in the Refer-Table (i.e., vCPU-5). Consequently, T1 is migrated to vCPU-5, allowing it to co-locate with T0 within vCG-0.

Additionally, CPS-CGsched should be combined with CPS-Pload. In the absence of CPS-Pload, if CPS-CGsched selects a vCPU that has been preempted, the guest scheduler will ignore this vCPU and selects an online vCPU for this thread. By incorporating CPS-Pload, which offers a more informed view of runtime pCPU load, a greater number of available vCPUs can be identified based on their load conditions. Consequently, the frontend module is more likely to identify an available vCPU within the target vCG.

The LTM provides a local view of thread distribution for each process, enabling two thread groups to potentially co-run within the same vCG. In some cases, the vCG with the largest ITN may be the same for both thread groups. As a result, the frontend may schedule both thread groups to the same vCG while leaving other vCGs under-utilized. To overcome this problem, the frontend utilizes the thread count recorded in each vCG within the CG-Tree. By examining the number of interactive threads in each vCG, the frontend can determine if there are other thread groups within the same vCG. If it finds that the number of interactive threads in a vCG is larger than in other vCGs, the frontend will identify this potential conflict and migrate one of the thread groups to another vCG.

The vCPU within each vCG are dynamically migrated due to the operations of the hypervisor scheduler, rendering the guest's CG-aware scheduling decisions less effective. To address this challenge and detect possible updates in the vCG topology, the frontend module incorporates additional checks in the guest's interrupt handling process. This allows it to proactively examine the Refer-Table and quickly adapt to any changes. If updates are detected, the frontend modifies the #vCG_vCPU value within the CG-Tree and repeats the aforementioned procedure for subsequent thread scheduling, ensuring it aligns with the new vCG topology.

However, frequent vCPU migrations can still disrupt cache locality, particularly for correlated vCPUs that hold these interactive threads. To alleviate this issue, cooperation between the frontend and backend modules is necessary to reduce the frequency of vCPU migrations. To achieve this, the frontend pins the vCPUs in a vCG for which its ITN matches its calculated vCG_Quota by setting the *Pin* field in the corresponding rows of the vCPU-level block. Subsequently, the backend checks the *Pin* field to bind the correlated vCPUs to their respective CGs over the lifetime of the interactive threads. Once these threads terminate, the frontend unsets the *Pin* field, and the backend unbinds these vCPUs from their CGs.

6 Evaluation

6.1 Implementation Complexity

CPS has been implemented in Linux KVM 5.10, and the total code size amounts to 424 Lines of Code (LoCs). The frontend module, including the Refer-Table, has a code size of 310 LoCs, while the backend module in the hypervisor totals 114 LoCs. Based on CPS, we introduce another 201 and 558 LoCs to implement CPS-Pload and CPS-CGsched, respectively.

The Refer-Table design in CPS not only addresses the RHS problem but also exhibits extensibility. To evaluate its flexibility, we integrated two other systems, namely eCS [32] and XPV [19], into CPS. In the eCS integration, the guest critical section is transmitted to the hypervisor to resolve lock-based issues. On the other hand, by integrating XPV into CPS, dynamic vCPU-NUMA mappings are exposed from the hypervisor to the guest.

In the eCS integration, the "kvm_steal_time" shared memory struct of the PV mechanism was adjusted to use the Refer-Table, which was straightforward due to the similar interface provided. The porting of existing eCS code, annotation of the guest's critical sections, and modification of the hypervisor scheduler required a total of three person-days. In terms of code size, the implementation consisted of approximately 1000 lines of code (LoCs), which is comparable to the original eCS implementation.

For the integration of XPV, the "shared_info" shared memory struct from Xen was adapted to work with the Ref-Table. Building upon the existing XPV code, six person-days were allocated to modify the guests' memory allocator, scheduler, and automatic NUMA balancing components. The implementation comprised around 900 LoCs, which is on par with the original XPV implementation. It should be noted that the implementation of system runtime libraries was omitted.

6.2 Evaluation Settings

CPS was evaluated on a Huawei Taishan200 manycore server [6] equipped with 128 physical cores (Kunpeng 920-7260 processor, ARMv8.2, 2.6 GHz), 256GB of memory, and 1.9TB of storage capacity. These physical cores are equally distributed across 2 sockets and 4 NUMA nodes. Each core consists of a 64KB L1 instruction cache, a 64KB L1 data cache, and a 512KB L2 cache. Additionally, there are two L3 caches in the server, one per socket, each with a size of 64MB (1MB per core). Furthermore, one L3 cache has 16 cache groups, with 4 physical cores assigned to each group. This configuration results in a total of 32 cache groups for the server.

To demonstrate the performance results for different VM sizes, two types of VMs were used: a small VM with 32 vCPUs, 60GB of memory, and 112GB of storage, and a big VM with 128 vCPUs, 60GB of memory, and 112GB of storage. Both the guest and host kernels are openEuler with Linux 5.10 [11].

Benchmarks. To demonstrate the benefits of CPS on the scalability of varied workloads, we choose four benchmarks that make extensive use of multi-threading and synchronization primitives. 1) **PARSEC 3.0** [17] is a collection of parallel programs that are used for scalability studies of multiprocessor machines. We evaluate the performance of 10 representative programs that frequently trigger scheduling events due to different locks and barriers. 2) **lbzip2** [9] is a multi-threaded compression and decompression utility. We show the time cost of compressing and decompressing the Linux kernel v5.16 source code in parallel via lbzip2. 3) **FxMark** [40] benchmark suite is used to analyze the manycore scalability of file systems. We measure the performance of six typical FxMarks benchmarks that include a large number of parallel operations on a shared file system. 4) **DBx1000** [56] is an in-memory transactional database that is scalable on a manycore machine. We evaluate the performance of running the TPCC workload using two types of two-phase lock(2PL) implementations (WAIT_DIE and NO_WAIT).

6.3 Microbenchmark

Table 2: Average vCPU selection time (unit: ns).

Config	HVM	CPS-Pload	PVM	CPS-CGsched
Time (ns)	103.29	706.60	1835.45	1090.84

In this section, we evaluate the guest scheduling latency of CPS by comparing the average vCPU selection time of CPS-Pload, CPS-CGsched, HVM, and PVM. As shown in Table 2, HVM exhibits the lowest vCPU selection time since it blindly chooses a vCPU without checking if it is preempted by the hypervisor. In the case of CPS-Pload, an additional 603.31 ns is incurred compared to HVM. This is because the guest kernel needs to traverse the vCPU list until it identifies an online vCPU or a preempted one on a lightly loaded pCPU. Similarly, PVM also traverses the vCPU list, but it selects an online vCPU without considering a preempted vCPU on a lightly loaded pCPU. Hence, the guest kernel takes longer to select an idle vCPU, resulting in an additional overhead of 1,128.85 ns compared to CPS-Pload. The additional overhead of 987.55 ns introduced by CPS-CGsched compared to HVM results from operations such as accessing LTM and CG-Tree, as well as calculating the runtime vCG_Quota.

Apart from performance overhead, we measure the additional memory consumption caused by CPS. The memory consumption is classified into three categories, 1) per host's 512KB memory consumption (load-degree pages), 2) per VM's (12KB + 512B) memory consumption (Refer-Table and CG-Tree), and 3) per threads group's 128B memory consumption (LTM).

6.4 Application Performance of CPS-Pload

In this section, we evaluate and analyze the performance of CPS-Pload with PARSEC 3.0 and lbzip2. We begin by evaluating the benchmarks' performance under varying numbers of threads and observe that optimal performance is generally achieved when the working thread count is close to 32. Consequently, we use a small 32-vCPU VM to evaluate CPS-Pload's performance. Our evaluation consists of running one small VM in the under-committed scenario and five small VMs in the over-committed scenario. All these benchmarks are configured with the thread numbers to make them achieve optimal performance.

Under-committed PARSEC 3.0. Fig. 7 shows the under-committed performance of CPS-Pload. The overhead of most PARSEC applications in a PVM is dominated by the data barrier, in which all other threads are put to sleep after reaching the data barrier until the last working thread finishes. These sleeping threads cause their corresponding vCPUs to be preempted. When the last thread wakes all the sleeping threads, the waked threads are scheduled to the non-preempted vCPUs. But the non-preempted vCPU number is limited. Therefore, the waked threads are scheduled to a small number of online vCPUs, even when the pCPU utilization is low, resulting in a negative impact on the performance. For example, in the PVM, the performance overhead caused by data barriers in splash2x.ocean_ncb and fluidanimate is 1.27x and 1.89x, respectively. CPS-Pload provides an effective way to allow these sleeping threads to be scheduled on preempted vCPUs that can be quickly rescheduled to run on low-load pCPUs. Hence, CPS-Pload shows 1.17x improvement on average compared with PVM. Note that there is no performance improvement in the blackscholes benchmark. The reason is that blackscholes benchmark has no data barrier operation. The performance of CPS-Pload in the under-committed scenario is comparable to that of HVM in most benchmarks and slightly outperforms HVM in streamcluster.

Table 3: Breakdown of splash2x.ocean_ncp with PVM and CPS-Pload in the under-committed 32-vCPU VM (unit: s). "Real time" stands for the total elapsed time of the application. "Running" represents how long the application was actually running and "Idle" is the CPU time the application is waiting for available cores. Both "Running" and "Idle" are the averages among 32 vCPU.

Config	PVM	CPS-Pload
Real time (s)	47.62	20.74
Running (s)	11.75	17.56
Idle (s)	35.87	3.18

To figure out how CPS-Pload improves the virtualization performance, we break down the time cost of splash2x.ocean_ncp. Table 3 shows that the idle time with CPS-Pload is 3.18s while the time with PVM is 35.87s. This is because CPS-Pload effectively allows the guest scheduler to run application threads to preempted vCPUs in low-load pCPUs, leading to better CPU utilization and resulting in 1.27x performance improvement.

To compare the thread scalability, we select another representative benchmark from PARSEC 3.0 (splash2x.ocean_cp) and use the

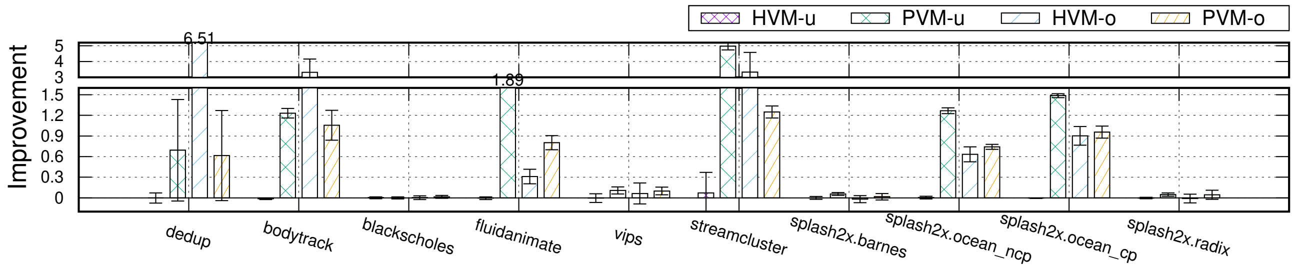

Fig. 7. Performance improvement of CPS-Pload over HVM and PVM in PARSEC 3.0 in the 32-vCPU VM. -u stands for under-committed one VM scenario and -o stands for over-committed five VMs scenario. Higher is better.

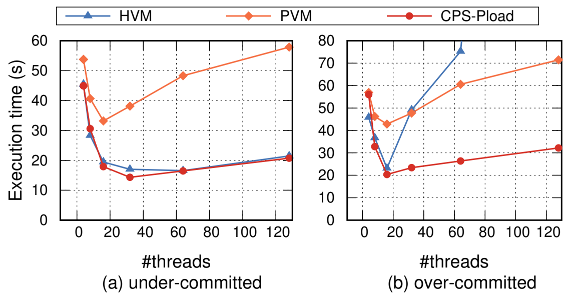

Fig. 8. The thread scalability of CPS-Pload, HVM, and PVM for splash2x.ocean_cp benchmark in the 128-vCPU VM. Lower is better.

big 128-vCPU VM to evaluate its scalability by gradually increasing its thread number up to 128, whose result is shown in Fig. 8(a). Both CPS-Pload and HVM perform better than PVM because both of them can utilize low-load pCPUs. In contrast, PVM puts tasks to a small number of non-preempted vCPUs despite that other preempted vCPUs can be quickly resumed in low-load pCPUs.

Over-committed PARSEC 3.0. In the over-committed scenario, we use 5 small 32-vCPU VMs: one VM runs PARSEC benchmarks while the other four VMs are configured to generate random CPU load with stress-ng [5]. As shown in Fig. 7, HVM is worse than PVM in some cases. In HVM, one thread can be scheduled to a preempted vCPU. Unfortunately, when the pCPU load is high, the preempted vCPU suffers from high latency to be wakened and scheduled to pCPU, causing performance degradation of the thread running on it. PVM, on the other hand, schedules threads to non-preempted vCPUs. The threads can immediately run on vCPUs that are already scheduled on pCPUs. However, the number of non-preempted vCPUs is very limited. Therefore, PVM still cannot fully utilize pCPU resources. CPS-Pload schedules threads to vCPU running on low-load pCPUs. Compared with HVM, CPS-Pload keeps preempted vCPU on high-load pCPU sleeping to reduce contention. Compared with PVM, CPS-Pload utilizes preempted vCPU on low-load pCPU. That explains why CPS-Pload outperforms HVM by 3.31x and PVM by 1.06x in bodytrack.

To analyze this result, we break down *splash2x.ocean_ncp* and compare its idle time and running time of CPS-Pload, HVM, and PVM (Table 4). During the data barrier in HVM, all other threads have to wait for the slowest thread, which may be scheduled to preempted vCPU. Therefore, HVM has the longest Running time. Despite the high CPU load, due to the runtime complexity of scheduling, there still exist low-load pCPUs occasionally in the over-committed case. CPS-Pload can squeeze the performance potential of these low-load pCPUs. Hence, the idle time of CPS-Pload is the shortest, which confirms that the CPS-Pload scheduling strategy harvests the low-load pCPU resources to shorten the scheduling latency of tasks.

Table 4: Breakdown of splash2x.ocean_ncp with PVM, HVM, and CPS-Pload in the over-committed 32-vCPU VM.

Config	PVM	HVM	CPS-Pload
Real time (s)	50.96	49.23	29.92
Running (s)	12.09	17.22	14.57
Idle (s)	38.87	32.01	15.35

Fig. 8(b) shows the thread scalability analysis of a big VM's splash2x.ocean_cp in the over-committed case. When the thread number is small, HVM performs slightly better than CPS-Pload because blindly waking preempted vCPUs in HVM has a smaller penalty. As the thread number further grows, CPS-Pload performs better than HVM and PVM. PVM has a similar trend as in the under-committed scenario, suffering from poor scalability due to the small number of available vCPUs. However, HVM shows severe performance degradation as the thread number increases, because HVM may blindly schedule a task to a vCPU running on a high-load pCPU.

lbzip2. We also evaluate CPS-Pload with a real-world application, lbzip2. In lbzip2, the input file is subdivided into even chunks, and the working threads process their smaller chunks in parallel. Each thread has to select tasks in a task queue protected by a mutex lock, which leads to mutex lock contentions. When the lock holder releases the lock and the guest scheduler selects vCPUs for lock waiters to run, the RHS problem comes into play. We use lbzip2 to compress and decompress the Linux kernel v5.16 source code, and the performance of lbzip2 is shown in Fig. 9. By eliminating the RHS problem, CPS-Pload outperforms HVM by up to 28.7% on average in the over-committed scenario. In the under-committed secnario, CPS-Pload, HVM, and PVM have similar results, because the lock contention of 32 working threads is not severe. To showcase the performance in high contention, we further test lbzip2 with

128 threads in a 128-vCPU VM in the under-committed scenario, in which the CPS-Pload is comparable to HVM and outperforms PVM by 45.2% on average for compression and decompression.

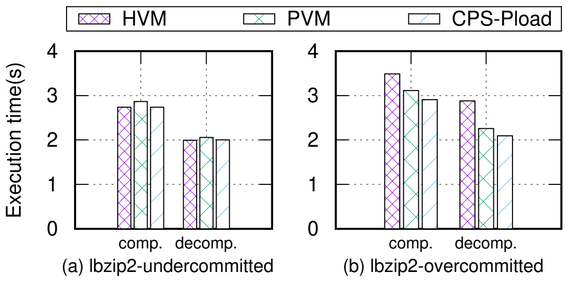

Fig. 9. The CPS-Pload improvement of lbzip2 in the 32-vCPU VM. Lower is better.

6.5 Application Performance of CPS-CGsched

To show how CPS-CGsched helps the guest OS to adapt to dynamic vCPU-vCG mappings in a manycore machine, this section evaluates the performance of two VM applications (FxMarks and DBx1000). Since the two applications can scale to 128 vCPUs, we use the big 128-vCPU VM to evaluate their performance. We run one VM in the under-committed and two VMs in the over-committed scenario.

FxMark. We first leverage FxMark to show the performance improvement of CPS-CGsched compared with **Baseline**, which means the vCPUs are not bound to pCPUs and the threads are not bound to vCGs. In the under-committed scenario as shown in Fig. 10, CPS-CGsched enables guests to enjoy the benefits from the deep memory hierarchy in a manycore server. When the number of interactive threads is small, there always exist idle CGs, in which CPS-CGsched successfully co-locates the interactive threads, providing performance improvement. For example, in the MRDL workload that reads entries of its private directory [2], CPS-CGsched outperforms the baseline by up to 1.61x with four running worker threads. When the number of threads scales to 128, all the CGs have been filled with threads to keep load balanced. It takes more time for CPS-CGsched to find available CGs to co-locate interactive threads. Therefore, CPS-CGsched outperforms the baseline by 76.7% on average with 128 running worker threads.

The improvement of CPS-CGsched for a FxMark benchmark is decided by the benchmark's parallelism degree. For example, MRDM benchmark, which reads entries of its shared directory, can be optimized by 2.35x. The reason for the improvement is that MRDM performs massive parallel reads to the same directory. Meanwhile, DWOM and DWOL benchmarks can only be optimized by 71.4% and 63.5%, because their threads hold a write lock to perform exclusive writes, showing less parallelism. However, they can still benefit from CPS-CGsched because of parallel access to the shared file system metadata and the shared synchronization primitives.

[2]To better showcase the performance benefits using CPS-CGsched, we modify some benchmarks that originally create private files exclusively for each thread, changing them to their counterparts where "private" file can be shared among a small number of threads (4 threads to match the CG core number).

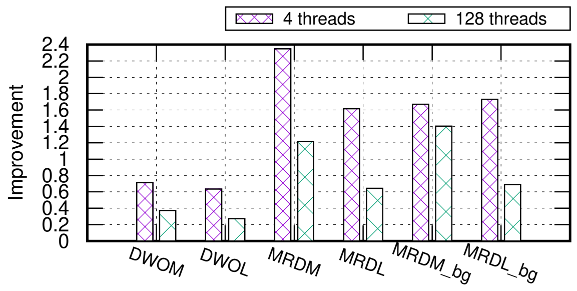

Fig. 10. The improvement of FxMark brought by CPS-CGsched in the under-committed 128-vCPU VM. Higher is better.

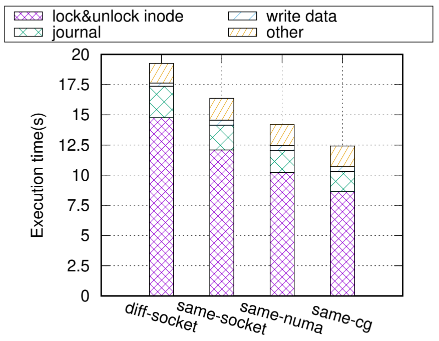

Fig. 11. The impact of memory hierarchy on the performance of DWOM in FxMark with four configurations: 1) diff-socket: 4 vCPUs are distributed in different sockets. 2) same-socket: 4 vCPUs are pinned in the same socket. 3) same-NUMA: 4 vCPUs are pinned in the same NUMA node. 4) same-CG: 4 vCPUs are pinned in the same cache group (a CG contains 4 pCPUs).

We further analyze FxMark DWOM performance to find out how shared file system data structures affect performance by running it under different memory hierarchy configurations. The results are shown in Fig. 11. We can see that the less memory hierarchy the interactive threads share, the worse their performance is. As the performance breakdown further illustrates, the main overhead is that the threads need to excessively access the shared file system metadata, such as inodes and page references, and synchronization primitives. For the diff-socket configuration, the inode lock acquisition operation takes about 15s, while in the same-CG configuration, this operation only takes 8s. When threads reside in different NUMA nodes or CGs, memory consistency, and cache coherency protocols cause data to bounce between NUMA nodes and cache lines, resulting in high performance costs.

The FxMark performance with CPS-CGsched in 2 VMs is shown in Fig. 12. CPS-CGsched brings an average improvement of 1.35x and 48.0% with 4 threads and 128 threads, respectively. CPS-CGsched shows the best improvements of 1.99x in MRDM_bg workload with four threads and 60.0% in MRDM workload with 128 threads. Still, the benchmarks with more parallelism have better improvement as in the under-committed scenario. For example, MRDM improves

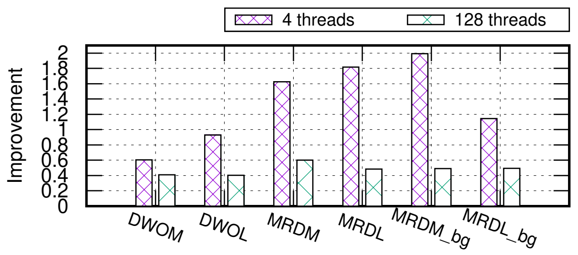

Fig. 12. The improvements of FxMark brought by CPS-CGsched in over-committed 128-vCPU VM. Higher is better.

by 1.63x while DWOM improves by 60.3% with four threads. The improvement of 128 threads also has less performance gain than 4 threads due to more time to find available CGs. For example, MRDL improves by 1.82x with 4 threads but by 48.4% with 128 threads.

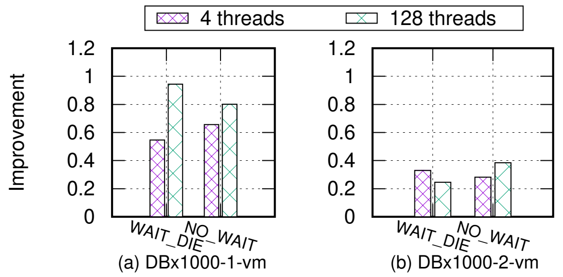

Fig. 13. The improvement of DBx1000 brought by CPS-CGsched in the 128-vCPU VM. Higher is better.

DBx1000. We evaluate CPS-CGsched using a real-world application, DBx1000, which utilizes 2PL for serializable concurrency control. In 2PL, a row lock is acquired for every record access, causing multiple threads to compete for the same lock during transaction conflict, resulting in cache bouncing that can be optimized by CPS-CGsched. To demonstrate CPS-CGsched's efficacy, we amplified the conflict rate of database queries and altered DBx1000 to spawn 128 threads divided into 32 interactive thread groups. As shown in Fig. 13, CPS-CGsched improved DBx1000 by as much as 73.7% in the under-committed scenario and 31.0% in the over-committed scenario. The improvement in the over-committed scenario is lower than that in the under-committed scenario because of cache pollution from the other co-running VM.

7 Limitations and Security Discussion

7.1 Limitations

The current prototype of CPS has three limitations and none of them are fundamental. First, CPS needs to modify the guest OS and only works for PV guest VMs, so it cannot improve the scalability of non-PV VMs that use commercial off-the-shelf OSes, which is still an open research problem. Second, CPS assumes that the application programmers can identify the interactive threads, so that they can slightly modify applications to pass the interaction information to CPS-CGsched. However, CPS cannot support applications that are distributed via pre-built binary format. Third, even though CPS should automatically detect the parameters of hardware CG (e.g., CPU-CG mappings, the number of CG, and CG size) during system startup, these parameters are hardcoded in our current implementation. We plan to extend CPS to support such CG parameters probing in the future.

7.2 Security Implications of Refer-Table

Even though a Refer-Table in CPS is a private data structure visible only to the VM that owns it, it provides hypervisor-internal states to the guest, potentially posing security threats. Therefore, we discuss possible threats and their implications in this section.

Shared Load-Degree Page. The load-degree page, which is mapped in each VM's Refer-Table, is shared by multiple VMs, potentially posing potential security threats. To this end, this page is mapped as read-only, preventing any VM from modifying its contents. Furthermore, the information available to a VM from the page is imprecise, providing only an approximation of the pCPU load without specific knowledge of the vCPUs running on the pCPU. Notably, such information has already been exposed by the existing PV mechanism. For instance, a vCPU can estimate the approximate pCPU load using PV steal time [10].

Side-Channel Threats. Exposing hypervisor-internal states to a VM can potentially facilitate cross-VM side-channel attacks, especially those relying on cache-based attacks [35, 59]. To launch a cache-based attack, an attacker needs to prime the cache set that the victim VM is utilizing. By combining the load degrees of each pCPU and vCG topology it retrieves from the Refer-Table, it may be easier for the attacker to pinpoint the pCPUs that the target VM is executing on, assuming that no other VMs are sharing these pCPUs. Subsequently, the attacker can use their vCPU, which co-resides with the target VM, to quickly prime the cache set. Nonetheless, this assumption is usually not realistic since more VMs (more than 2) typically share the same pCPUs, thus rendering the attack more challenging to execute. Additionally, countermeasures such as allocating sensitive VMs to isolated pCPUs or utilizing existing techniques [20, 35, 39] can be employed to prevent this type of attack, which is outside the scope of this paper.

8 Related Work

8.1 Synchronization in VM

As we have mentioned in § 2.1, synchronization latency caused by the vCPU preemption increases significantly, and this problem has different variants [24, 25, 30, 42, 44, 48]. A long line of methods has been proposed to address these synchronization issues, which can be classified into three categories.

Hardware Approaches. A hypervisor can utilize hardware features [4, 8, 52] to detect vCPUs that keep busy waiting so that the hypervisor has the chance to schedule them out to other vCPUs. For instance, Intel's Pause Loop Exiting (PLE) allows the hypervisor to set a maximum number that a vCPU is able to successively execute the PAUSE instruction. When the upper bound is hit, the hardware considers that this vCPU is contending on a spinlock and triggers a VM exit to wake up the hypervisor, which then schedules a new vCPU or directly boosts the vCPU that is possibly holding the lock. Equipped with such hardware features, the hypervisor effectively reduces the wasted CPU cycles in LHP and LWP. However, it is difficult to precisely pinpoint the lock holder due to the notorious semantic gap in the virtualized environment [28]

Para-virtualized Approaches. To break the semantic gap, the PV approaches modify the guest kernel to pass guest's hints to the underlying hypervisor [41, 49]. Specifically, *paravirtual spinlock* [26] alleviates LHP by modifying the guest kernel's spinlock implementation to proactively give up the pCPU if the guest's waiting time exceeds a threshold. Oticket [30, 31] improves *paravirtual spinlock* through dynamically scaling the waiter's spinning time according to the waiters' distance. Preemptable ticket spinlocks [41] resolve the LWP problem by relaxing the ordering guarantees of ticket spinlocks. It allows an out-of-order waiter to acquire a lock when the earlier waiters are unresponsive. There also exist general solutions to a wider range of synchronization problems [12, 32, 48]. I-Spinlock [48] is a new PV spinlock that only allows lock acquisition when the vCPU's remaining time slice is sufficient, which effectively avoids being preempted when holding a lock. Similarly, eCS [32] proposes PV interface to address all preemption problems, and it lets the guest to annotate critical sections in which the hypervisor will not schedule out the vCPU. These PV approaches only resolve the locking-related issues, not applicable to the RHS problems that are caused by the asymmetric visibility on hypervisor-internal states dynamic (like pCPU load and CG topology).

Scheduler-based Approaches. The third direction is to optimize the hypervisor's scheduler without modifying the guest OS, including the gang-scheduling we have mentioned in § 2.1. Another example is the demand-based coordinate scheduler [33] that co-schedules workloads to reduce communication latency. Song et al. [21, 45] further proposes vCPU ballooning to adjust the number of vCPU according to available pCPU resources. Although these systems resolve the double scheduling related to synchronization to some extent, their methods either lead to serious CPU fragmentation or cannot tackle the RHS problem CPS faces.

8.2 NUMA Virtualization

The double scheduling and semantic gap can further negatively affect VM performance through inefficient NUMA virtualization since the hypervisor may blindly change the NUMA topology of VMs. Therefore, prior efforts aim to designing various solutions to address this issue. Most solutions take a blackbox way [15, 18, 27, 36, 43, 54] in which the actual NUMA topology is hidden to the VM. Different from them, Voron et al. [51] implement four NUMA policies and allow the guest OS to explicitly choose one of them at runtime, but this approach still cannot support topology changes. To this end, XPV [19] exposes a dynamic virtual NUMA topology and provides PV interface to help the guest OS to adapt its NUMA policies. Even though NUMA virtualization looks similar to vCG in CPS, they have fundamental differences (as we have discussed in § 2.2) and CPS needs a mechanism to expose ever-changing vCG topology.

9 Conclusion

To address the lack of *runtime hypervisor-internal states (RHS)* in large VMs, this paper proposes CPS, a cooperative para-virtualized scheduling framework, which proactively allows both guest and host to share dynamic scheduling information made by each other. We have implemented CPS into KVM, and it can improve the performance of PARSEC by 81.1% and FxMark by 1.01x on average for the two RHS problems.

10 Acknowledgments

We express our sincere gratitude to the anonymous ASPLOS reviewers for their insightful suggestions. We are equally grateful to Zhi Guo for performance evaluation, Dingji Li and Tianhong Ding for their help in discussing the initial idea. This work was partially supported by the National Natural Science Foundation of China (No.62132014, 61925206, 62232012, 62002218) and Huawei Innovation Research Plan.

References

[1] Alibaba Cloud: Elastic Compute Service. https://www.alibabacloud.com/product/ecs. Referenced September 2023.
[2] Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/. Referenced September 2023.
[3] AMD. 2021. The 2nd Gen AMD EPYC 7002 Series Processors. www.amd.com/en/processors/epyc-7002-series. Referenced September 2023.
[4] AMD64 Architecture Programmer's Manual, Volume 2: System Programming. https://www.amd.com/system/files/TechDocs/24593.pdf. Referenced September 2023.
[5] Github: stress-ng (stress next generation). https://github.com/ColinIanKing/stress-ng. Referenced September 2023.
[6] Huawei TaiShan Server Data Sheet. https://e.huawei.com/en/material/datacenter/server/7a0b8b0f056f479f909220ac21915999. Referenced September 2023.
[7] implement vcpu preempted check. https://lwn.net/Articles/704904/. Referenced September 2023.
[8] Intel® 64 and IA-32 Architectures Software Developer's Manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3c-part-3-manual.pdf. Referenced September 2023.
[9] lbzip2: parallel bzip2 compression utility. https://github.com/kjn/lbzip2. Referenced September 2023.
[10] LWN.net: Steal time for KVM. https://lwn.net/Articles/449657/. Referenced September 2023.
[11] OpenEuler. https://github.com/openeuler-mirror. Referenced September 2023.
[12] Paravirtualized ticket spinlocks. https://lwn.net/Articles/552696/. Referenced September 2023.
[13] The CPU Scheduler in VMware vSphere® 5.1. https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/vmware-vsphere-cpu-sched-performance-white-paper.pdf. Referenced September 2023.
[14] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *SIGOPS Oper. Syst. Rev.*, 25(5):95–109, sep 1991.
[15] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003.
[16] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 29–44, New York, NY, USA, 2009. Association for Computing Machinery.
[17] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, page 72–81, New York, NY, USA, 2008. Association for Computing Machinery.
[18] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *SIGOPS Oper. Syst. Rev.*, 31(5):143–156, oct 1997.
[19] Bao Bui, Djob Mvondo, Boris Teabe, Kevin Jiokeng, Lavoisier Wapet, Alain Tchana, Gaël Thomas, Daniel Hagimont, Gilles Muller, and Noel DePalma. When EXtended Para - Virtualization (XPV) Meets NUMA. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.
[20] Sanchuan Chen, Fangfei Liu, Zeyu Mi, Yinqian Zhang, Ruby B. Lee, Haibo Chen, and XiaoFeng Wang. Leveraging Hardware Transactional Memory for Cache

Side-Channel Defenses. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS '18, page 601–608, New York, NY, USA, 2018. Association for Computing Machinery.

[21] Luwei Cheng, Jia Rao, and Francis C. M. Lau. VScale: Automatic and Efficient Processor Scaling for SMP Virtual Machines. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[22] Christoffer Dall and Jason Nieh. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, page 333–348, New York, NY, USA, 2014. Association for Computing Machinery.

[23] Rafael Lourenco de Lima Chehab, Antonio Paolillo, Diogo Behrens, Ming Fu, Hermann Härtig, and Haibo Chen. CLoF: A Compositional Lock Framework for Multi-Level NUMA Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 851–865, New York, NY, USA, 2021. Association for Computing Machinery.

[24] Xiaoning Ding, Phillip B. Gibbons, and Michael A. Kozuch. A Hidden Cost of Virtualization When Scaling Multicore Applications. In *5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 13)*, San Jose, CA, June 2013. USENIX Association.

[25] Xiaoning Ding, Phillip B. Gibbons, Michael A. Kozuch, and Jianchen Shan. Gleaner: Mitigating the Blocked-Waiter Wakeup Problem for Virtualized Multicore Applications. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, page 73–84, USA, 2014. USENIX Association.

[26] Thomas Friebel and Sebastian Biemueller. How to Deal with Lock Holder Preemption. 2008.

[27] Jaeung Han, Jeongseob Ahn, Changdae Kim, Youngjin Kwon, Young-Ri Choi, and Jaehyuk Huh. The Effect of Multi-Core on HPC Applications in Virtualized Systems. In *Proceedings of the 2010 Conference on Parallel Processing*, Euro-Par 2010, page 615–623, Berlin, Heidelberg, 2010. Springer-Verlag.

[28] Kenta Ishiguro, Naoki Yasuno, Pierre-Louis Aublin, and Kenji Kono. Mitigating Excessive VCPU Spinning in VM-Agnostic KVM. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2021, page 139–152, New York, NY, USA, 2021. Association for Computing Machinery.

[29] Ali Kamali. *Sharing aware scheduling on multicore systems.* PhD thesis, Applied Science: School of Computing Science, 2010.

[30] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Scalability in the Clouds! A Myth or Reality? In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, APSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[31] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Opportunistic Spinlocks: Achieving Virtual Machine Scalability in the Clouds. *SIGOPS Oper. Syst. Rev.*, 50(1):9–16, mar 2016.

[32] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Scaling Guest OS Critical Sections with eCS. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 159–172, Boston, MA, July 2018. USENIX Association.

[33] Hwanju Kim, Sangwook Kim, Jinkyu Jeong, Joonwon Lee, and Seungryoul Maeng. Demand-Based Coordinated Scheduling for SMP VMs. *SIGARCH Comput. Archit. News*, 41(1):369–380, mar 2013.

[34] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: the Linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230. Dttawa, Dntorio, Canada, 2007.

[35] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622, 2015.

[36] Ming Liu and Tao Li. Optimizing virtual machine consolidation performance on NUMA server architecture for cloud workloads. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 325–336, 2014.

[37] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-Class User-Level Threads. *SIGOPS Oper. Syst. Rev.*, 25(5):110–121, sep 1991.

[38] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Willy Zwaenepoel. Diagnosing Performance Overheads in the Xen Virtual Machine Environment. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, page 13–23, New York, NY, USA, 2005. Association for Computing Machinery.

[39] Zeyu Mi, Haibo Chen, Yinqian Zhang, Shuanghe Peng, Xiaofeng Wang, and Michael K. Reiter. CPU Elasticity to Mitigate Cross-VM Runtime Monitoring. *IEEE Transactions on Dependable and Secure Computing*, 17(5):1094–1108, 2020.

[40] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. Understanding manycore scalability of file systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 71–85, 2016.

[41] Jiannan Ouyang and John R. Lange. Preemptable Ticket Spinlocks: Improving Consolidated Performance in the Cloud. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13, page 191–200, New York, NY, USA, 2013. Association for Computing Machinery.

[42] Aravinda Prasad, K Gopinath, and Paul E. McKenney. The RCU-Reader Preemption Problem in VMs. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 265–270, Santa Clara, CA, July 2017. USENIX Association.

[43] Jia Rao, Kun Wang, Xiaobo Zhou, and Cheng-Zhong Xu. Optimizing virtual machine scheduling in NUMA multicore systems. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 306–317, 2013.

[44] Xiang Song, Haibo Chen, Binyu Zang, X Song, H Chen, and B Zang. Characterizing the performance and scalability of many-core applications on virtualized platforms. *Parallel Processing Institute Technical Report Number: FDUPPITR-2010*, 2, 2010.

[45] Xiang Song, Jicheng Shi, Haibo Chen, and Binyu Zang. Schedule Processes, Not VCPUs. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, APSys '13, New York, NY, USA, 2013. Association for Computing Machinery.

[46] Orathai Sukwong and Hyong S. Kim. Is Co-Scheduling Too Expensive for SMP VMs? In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, page 257–272, New York, NY, USA, 2011. Association for Computing Machinery.

[47] David Tam, Reza Azimi, and Michael Stumm. Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors. *SIGOPS Oper. Syst. Rev.*, 41(3):47–58, mar 2007.

[48] Boris Teabe, Vlad Nitu, Alain Tchana, and Daniel Hagimont. The lock holder and the lock waiter pre-emption problems: Nip them in the bud using informed spinlocks (i-spinlock). In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 286–297, 2017.

[49] Volkmar Uhlig, Joshua LeVasseur, Espen Skoglund, and Uwe Dannowski. Towards Scalable Multiprocessor Virtual Machines. In *Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium - Volume 3*, VM'04, page 4, USA, 2004. USENIX Association.

[50] VMware. The CPU Scheduler in VMware ESX 4.1. *Technical Report*, 2010.

[51] Gauthier Voron, Gaël Thomas, Vivien Quéma, and Pierre Sens. An Interface to Implement NUMA Policies in the Xen Hypervisor. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 453–467, New York, NY, USA, 2017. Association for Computing Machinery.

[52] Philip M. Wells, Koushik Chakraborty, and Gurindar S. Sohi. Hardware support for spin management in overcommitted virtual machines. In *2006 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 124–133, 2006.

[53] Chuliang Weng, Qian Liu, Lei Yu, and Minglu Li. Dynamic Adaptive Scheduling for Virtual Machines. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC '11, page 239–250, New York, NY, USA, 2011. Association for Computing Machinery.

[54] Song Wu, Huahua Sun, Like Zhou, Qingtian Gan, and Hai Jin. vProbe: Scheduling Virtual Machines on NUMA Systems. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 70–79, 2016.

[55] Song Wu, Zhenjiang Xie, Haibao Chen, Sheng Di, Xinyu Zhao, and Hai Jin. Dynamic Acceleration of Parallel Applications in Cloud Platforms by Adaptive Time-Slice Control. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 343–352, 2016.

[56] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endow.*, 8(3):209–220, nov 2014.

[57] Yifan Yuan, Mohammad Alian, Yipeng Wang, Ren Wang, Ilia Kurakin, Charlie Tai, and Nam Sung Kim. Don't Forget the I/O When Allocating Your LLC. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 112–125, 2021.

[58] Lei Zhang, Yu Chen, Yaozu Dong, and Chao Liu. Lock-Visor: An Efficient Transitory Co-scheduling for MP Guest. In *2012 41st International Conference on Parallel Processing*, pages 88–97, 2012.

[59] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 305–316, New York, NY, USA, 2012. Association for Computing Machinery.

[60] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors. *ACM Comput. Surv.*, 45(1), dec 2012.

Received 2022-10-22; accepted 2023-04-28