Optimizing Task Scheduling in Cloud VMs with Accurate vCPU Abstraction

Edward Guo
Hofstra University

Weiwei Jia
The University of Rhode Island

Xiaoning Ding
New Jersey Institute of Technology

Jianchen Shan
Hofstra University

Abstract

The paper shows that task scheduling in Cloud VMs hasn't evolved quickly to handle the dynamic vCPU resources. The existing vCPU abstraction cannot accurately depict the vCPU dynamics in capacity, activity, and topology, and these mismatches can mislead the scheduler, causing performance degradation and system anomalies. The paper proposes a novel solution, *vSched*, which probes accurate vCPU abstraction through a set of lightweight microbenchmarks (*vProbers*) without modifying the hypervisor, and leverages the probed information to optimize task scheduling in cloud VMs with three new techniques: biased vCPU selection, intra-VM harvesting, and relaxed work conservation. Our evaluation of *vSched*'s implementation in x86 Linux Kernel demonstrates that it can effectively improve both system throughput and workload latency across various VM types in the dynamic multi-cloud environment.

CCS Concepts: • **Software and its engineering** → **Operating systems**; **Virtual machines**.

Keywords: Task Scheduling, Virtualization, Cloud Computing, Resource Probing, Operating Systems

ACM Reference Format:
Edward Guo, Weiwei Jia, Xiaoning Ding, and Jianchen Shan. 2025. Optimizing Task Scheduling in Cloud VMs with Accurate vCPU Abstraction. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3689031.3696092

1 Introduction

Cloud virtual machines (VMs) have become the primary means of provisioning computing resources. Even containers are often hosted in VMs for lower cost [1] and better performance isolation [2]. Therefore, operating systems (OSes) are increasingly running in the cloud VMs, managing the virtualized resources. However, as the core component, the task scheduler hasn't evolved quickly to efficiently manage the virtual CPU (vCPU) resource.

The major challenge lies in the dynamic nature of vCPUs, driven by various reasons. The first is double scheduling where cloud applications are scheduled on vCPUs while the vCPUs are scheduled on physical cores by the hypervisor. Unless pinned, vCPUs may be migrated to balance the load, optimize memory access, or increase energy efficiency, thereby altering the vCPU topology. Second, due to cloud multi-tenancy, vCPUs would time-share the physical cores. This can result in vCPU inactive periods [3] and fluctuations in vCPU capacity [4]. Lastly, VM migration is the norm practice, especially in the multi-cloud era [5], to achieve optimal VM deployment. It also happens frequently with spot VM [6] which harvests unused cloud resources. Such VM migration can significantly change vCPU performance features.

Unfortunately, the existing vCPU abstraction cannot accurately depict the aforementioned vCPU dynamics, leading to mismatches in important performance features, including capacity, topology, and activity. For example, hypervisors by default would mistakenly expose vCPUs as static and symmetric CPUs [4] with an UMA (Uniform Memory Access) topology [7]. Being unaware of the dynamic nature of vCPUs, the scheduler cannot make informed decisions.

To mitigate this issue, one approach is to improve task scheduling with accurate vCPU information exposed by the paravirtualized hypervisor. For example, XPV [7] exposes NUMA (Non-Uniform Memory Access) topology to the VM from the hypervisor to make NUMA-aware optimizations effective inside the VM. CPS [8] exposes NUCA (Non-Uniform Cache Access) topology and core load to the VM to further optimize the scalability in big VMs. Another approach is to improve vCPU scheduling [3, 9–15] at the hypervisor layer. For example, the hypervisor can preempt excessively spinning vCPUs to mitigate the lock-holder preemption (LHP) problem [16] caused by vCPU inactivity. Some work shares task information from guest to host to further assist vCPU scheduling optimizations. For instance, eCS [13] provides hints from the VM to allow the hypervisor to boost vCPUs running critical-section tasks. Pillai [12] proposes sharing a task's latency requirements so the hypervisor can prioritize vCPUs running latency-critical tasks.

EuroSys '25, March 30–April 3, 2025, Rotterdam, Netherlands
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1196-1/25/03
https://doi.org/10.1145/3689031.3696092

These state-of-the-art approaches require either communication from the hypervisor to the guest or from the guest to the hypervisor. This necessitates hypervisor modifications, posing several limitations. First, integrating modifications from different solutions is challenging. While some recent solutions involve relatively light code changes (e.g., CPS [8] adds 424 lines to KVM [17]), combining these changes to comprehensively address problems like double scheduling is complex, especially when the solutions have conflicting optimization goals. Second, in the multi-cloud era [5], adopting these modifications is difficult due to the diverse or closed-source nature of hypervisors. Lastly, exposing more hypervisor-internal information to guests can resolve more mismatches in vCPU abstraction but also introduces additional attack surfaces, raising security concerns such as the risk of cross-VM side-channel attacks [18].

Thus, we aim to optimize the task scheduling **with** accurate vCPU abstraction **within** the VM, which has the best knowledge of the workloads, **without** relying on any hypervisor modifications. This approach can avoid the above limitations and allow users to enhance schedulers for specific optimization goals in the multi-cloud environment, making it a versatile and immediate solution that benefits users without waiting for cloud providers to adopt new remedies. To achieve this goal, we made the following contributions.

Our first contribution (§ 2) is to systematically analyze and demonstrate through experiments on a commodity system (x86 Linux VMs on the KVM hypervisor) the three major impacts of inaccurate vCPU abstraction on task scheduling. First, the existing optimizations that depend on accurate CPU abstraction, such as capacity-aware or topology-aware scheduling [19, 20], cannot function as expected. Second, the optimizations for unique vCPU performance features are missing. For instance, there is a lack of consideration for vCPU inactivity, which can cause wasted vCPU time and increased workload latency. Third, existing scheduler design principles may be inappropriately applied. We found that enforcing work conservation invariant [21] (no task waiting on a busy CPU when there are idle CPUs) may hurt the system when a problematic idle vCPU is picked. For instance, an idle vCPU with extremely low capacity can turn a task into a straggler [22] delaying other dependent tasks. Even worse, placing tasks on an idle vCPU stacked with other vCPUs can cause problems like priority inversion.

Our second contribution (§ 3) is to design *vSched*, a novel solution that makes the scheduler aware of vCPU dynamics through resource probing. A suite of lightweight microbenchmarks (named *vProbers*) is designed to expose accurate vCPU abstractions within the VM. Periodic and coordinated sampling is employed to probe dynamic vCPU capacity. To provide smooth estimation and prevent frequent task migrations, we utilize the exponential moving average (EMA), which considers the past while prioritizing the present. For activity, we measure the average vCPU inactive period, referred to as *vCPU latency*, to indicate how quickly a vCPU can schedule tasks. A heartbeat mechanism is designed to probe vCPU states (inactive/active) without requiring paravirtualization [23]. Additionally, the time a vCPU has been active (or inactive) after the previous state change is tracked for fine-granular scheduling decisions. Regarding topology, all levels of hierarchy (stacking, SMT, socket) are dynamically probed by measuring vCPU distance through cache line transfer latency [24].

Existing optimizations can benefit from the accurate vCPU abstraction exposed by *vProbers*. To fully leverage the unique vCPU characteristics, *vSched* further introduces three new optimizing techniques: intra-VM harvesting (IVH), biased vCPU selection (BVS), and relaxed work conservation (RWC). The first two techniques are proposed to implement activity-aware scheduling. Particularly, IVH would proactively migrate a running task from a soon-to-be-inactive vCPU to another unused vCPU where it can continue making progress by harvesting the otherwise wasted vCPU time, leading to improved vCPU utilization. Whereas, BVS aims to reduce workload latency by placing small latency-sensitive tasks on vCPUs with low runqueue latency. RWC is an effort to adapt existing scheduler design principles to the virtualized environment. Its key idea is to hide the problematic idle vCPUs (e.g., straggler and stacking vCPUs) from task placement to avoid system anomalies and improve performance.

Our third contribution (§ 4) is to implement *vSched* in the x86 Linux VM. Our primary goal is to show how easily an existing scheduler can be ported to be vCPU-aware, thereby enhancing performance. Thus, rather than introducing a new scheduling class, we extend the Completely Fair Scheduler (CFS [25]) to implement *vSched*. Three microbenchmarks (VCAP, VTOP, and VACT) are implemented to probe capacity, topology, and activity, respectively. One kernel module is created to expose the probed results to CFS by dynamically rebuilding schedule domains [26] and updating per-vCPU data (e.g., EMA capacity and vCPU latency). A new kernel function is added to query vCPU states. BPF hooks [27] are inserted to bypass the original code paths, such as those in load balancing and CPU selection functions, to realize IVH and BVS. For RWC, we utilize cgroup [28] to hide the problematic vCPUs. Our implementation requires no changes to hardware, hypervisor, or applications, making it a practical solution. The implementation is open-sourced[1] to aid future research on scheduling in virtualized systems.

Our final contribution (§ 5) is to evaluate *vSched* with diverse workloads across various VM types in the multi-tenant environment. We first show that *vProbers* can accurately probe the targeted vCPU performance features under subsecond. Subsequently, upon exposing the probed results to the kernel, we illustrate that existing optimizations, such as

[1]https://github.com/vSched

capacity-aware and topology-aware heuristics, become more effective. Additionally, we individually test IVH, BVS, and RWC to showcase their respective contributions to *vSched*'s effectiveness in increasing throughput, reducing latency, and preventing system anomalies. Through further comprehensive testing, we demonstrate that *vSched* can benefit a wide range of real-world workloads on both resource-constrained and high-performance VMs. Moreover, we show that *vSched* can quickly respond to vCPU changes and maintain high Quality-of-Service (QoS) in dynamic multi-tenant hosts. Finally, we show that *vSched* offers large performance gains at a small cost and it causes a minimal overhead when workloads cannot benefit from the accurate vCPU abstraction.

2 Background and Motivation

2.1 vCPU abstraction

Cloud VMs can offer up to hundreds of vCPUs. As software entities, vCPUs are managed and scheduled by the hypervisor, exhibiting unique performance features that cannot be accurately depicted by current vCPU abstraction.

vCPU is not always active. Due to cloud multi-tenancy, vCPUs in one VM may time-share cores with vCPUs from co-located VMs. When preempted, a vCPU becomes inactive until rescheduled by the hypervisor. Unlike always-active cores, vCPUs exhibit several unique characteristics. First, a running task on a preempted vCPU is technically not running to make any progress. Second, a vCPU doesn't have an intact private cache as a core does. A vCPU cannot allow its tasks to effectively build up data in the cache if the co-running vCPUs constantly pollute the cache during its inactive periods. Third, the latency to start serving a new or wakeup task by a vCPU could be much higher than by a core since it includes both runqueue delay and inactive periods. Therefore, falsely assuming that vCPU is always active with an intact private cache and can start serving tasks with low latency may adversely impact workload performance.

vCPU capacity is not static. CPU capacity is the normalized measurement of CPU performance (instructions per second). In cloud VMs, vCPU capacity can be highly dynamic since it's determined by the hosting core's capacity and the percentage of the CPU time allocated to the vCPU from the hosting core. The former fluctuates with changes in core frequency (cycles per second) and contention from the SMT sibling if hardware threads are used. The latter is influenced by the contention on the core. Therefore, without accurate vCPU capacity, any capacity-aware optimizations become inefficient.

By combining the inactivity and dynamic capacity, vCPUs manifest distinct performance features. Without this knowledge, the scheduler misses opportunities to match vCPUs with tasks having diverse requirements, as illustrated in Figure 1 (left), where the states of four vCPUs are depicted over a repeated period. We assume that the vCPUs are hosted on four cores with identical capacities. vCPU0 stands out as

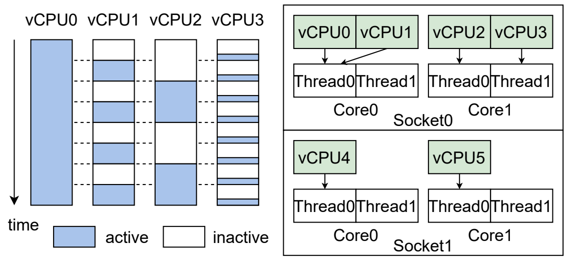


Figure 1. Dynamic vCPU capacity, activity and topology.

an ideal match for any task due to its highest capacity and lowest latency, achieved by exclusively utilizing the core. Between vCPU1 and vCPU2 which share the same capacity, the former can better serve latency-sensitive tasks with shorter inactive periods. Lastly, despite vCPU3 exhibiting the lowest capacity and yielding lower throughput compared to vCPU1 and vCPU2, it excels in handling sporadic and small tasks.

vCPU can be moved around. The vCPU topology is determined by the mappings between vCPUs and cores. Figure 1 (right) illustrates different levels of topology based on the shared resources. vCPU0 and vCPU1 are stacked together, sharing per-hardware-thread resources such as registers. Meanwhile, vCPU2 and vCPU3 are mapped to hardware threads that are SMT siblings, sharing per-core resources like L1/L2 cache. Lastly, vCPU0-vCPU3 and vCPU4-vCPU5 are mapped to different sockets, thereby sharing per-socket resources such as the last-level cache (LLC). The vCPU topology can dynamically change due to vCPU or VM migration. However, it often remains opaque or outdated within cloud VMs [7], impeding any topology-aware optimizations.

2.2 Linux Scheduler

This paper focuses on x86 Linux VMs on the KVM hypervisor, an open-source platform widely used in public clouds. Linux implements various scheduling classes with distinct priorities to manage specific types of tasks [29]. The CFS class, our main focus, handles most tasks. To be scalable, CFS creates one runqueue per CPU where time is shared based on the task's priority. CPU selection and load balancing are the two critical components in achieving scheduling goals through task placement and migration. Their heuristics depend on accurate CPU abstraction.

In particular, the CPU capacity available to CFS tasks is precisely measured as *CFS capacity* to accurately match the CFS task load [30] on that CPU. Runnable tasks are migrated to balance the load-to-capacity ratio on each runqueue. Even in the underloaded system (i.e., there are idle CPUs and no runnable tasks), the active balance is triggered to move a misfit running task to a CPU with a higher capacity that can satisfy its CPU utilization. To leverage the CPU topology, schedule domains [26] are hierarchically constructed, grouping cores based on shared resources. At the lowest level (SMT), each domain contains SMT sibling cores, allowing

SMT-aware algorithms like core scheduling [20] to disfavor an idle hardware thread that has the busy SMT sibling. At the upper level (socket), each domain includes cores sharing resources like LLC, which allows placing dependent tasks within the same socket domain for enhanced communication. Lastly, CPU state is also considered, with strategies like small-task packing [21] implemented to prioritize short-idled CPUs in a shallow sleep state for quicker wake-up and higher-frequency operation.

However, with the inaccurate vCPU abstraction in Linux VMs, the aforementioned optimizations are inefficient. Additionally, workload performance can suffer due to the lack of consideration for unique vCPU performance features, such as capacity asymmetry, inactivity, and stacking topology, as demonstrated by experiments in the following sections. It is important to note that while these issues are demonstrated with CFS, they can also manifest in other schedulers that share similar goals and designs. In addition, although these issues stem from the same root cause (double scheduling) as previously identified problems like lock-holder preemption (LHP) [16], lock-waiter preemption (LWP) [31], and blocked-waiter wakeup (BWW) [32], the issues discussed in the following sections are uniquely identified from a scheduler development perspective, crucial for scheduler improvement within virtualized systems.

2.3 Motivating Experiments

We conduct experiments to demonstrate the performance issues that can be mitigated if new optimizing techniques can be designed to consider the unique performance features of vCPUs. Detailed benchmark descriptions and experimental settings can be found in the evaluation section (§ 5).

Extended Runqueue Latency. Runqueue latency describes how long a runnable task needs to wait on runqueue before execution. In an overcommitted VM, it can be extended by *vCPU latency*, which is the time a runnable vCPU must wait on the host runqueue before being rescheduled. Thus, small latency-sensitive tasks, when scheduled on a vCPU with high *vCPU latency*, can suffer from the extended runqueue latency that dominates the end-to-end execution time.

To illustrate the impact of *vCPU latency* on the runqueue latency, we conducted experiments with two overcommitted 32-vCPU VMs. Each VM's vCPUs are pinned individually on the same set of 32 cores. One VM executed latency-sensitive workloads selected from Tailbench [33], while the other VM stressed its vCPUs using Sysbench [34] to induce *vCPU latency* in the former. We explored two scenarios: one without best-effort tasks, where vCPUs of the primary VM remained idle when the benchmark was inactive, and another with best-effort tasks, where all vCPUs were kept busy by a background workload of the lowest priority (sched_idle [29]). The latter is commonly used to harvest free vCPU cycles [35] without impeding latency-sensitive tasks. In both scenarios, CPU bandwidth control [36] is used on the host with other tunables (e.g., minimum and wakeup granularities [37]) to adjust *vCPU latency* without changing capacity.

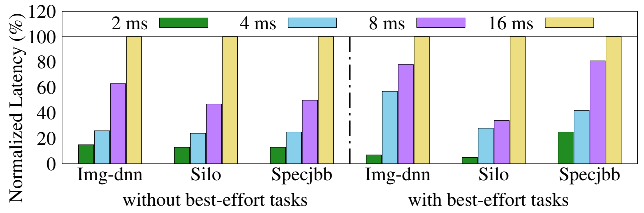


Figure 2. The impact of *vCPU latency* on latency-sensitive workloads. The p95 tail latencies are normalized to when the *vCPU latency* is 16 ms. Lower is better.

The results, as depicted in Figure 2, revealed a significant increase in the 95th tail latency of each benchmark, scaling up to 20x as *vCPU latency* increased from 2 ms to 16 ms. This increase is primarily attributed to *vCPU latency*, as we reduced the arrival rate of requests in each benchmark to minimize the delay on the runqueue while waiting for other requests to be completed. This finding highlights the critical importance of considering *vCPU latency* in vCPU selection for tasks with different latency requirements (§ 3.2).

Stalled Running Task. A running task, during a vCPU's inactive periods, technically ceases to make any progress, leading to what we term the *stalled running task* problem. With CFS, a running task is only migrated if there is an idle vCPU with the capacity that better fits the task's CPU utilization requirement. However, no existing migration aims to prevent a stalled running task. This results in wasted vCPU time in an underloaded VM, as there exist unused idle vCPUs where a stalled running task could immediately progress. Consequently, allocated CPU time is left underutilized. Moreover, the stalled running task may obstruct the critical path, thereby delaying other dependent tasks.

To illustrate the impact of the *stalled running task*, we conducted experiments with two overcommitted 4-vCPU VMs. Each VM's vCPUs were pinned individually on the same set of 4 cores. One VM executed a synthetic single-threaded program that is CPU-intensive, while the other VM stressed its vCPUs using Sysbench. We minimized the wakeup latency and adjusted both the minimum granularity and CPU bandwidth control on the host so that each vCPU would be inactive for 5 ms after every 5 ms active period. The program is executed in two modes: *default mode* and *migration mode*. In the former, the scheduler determined task placement, while in the latter, the thread circularly migrated itself among idle vCPUs every 4 ms.

We generate Figure 3 with KernelShark [38] to provide detailed insights into task execution under the two modes. In the *default mode*, the task becomes stalled 50% of the time, while in the *migration mode*, the proactive task migration can always move the thread before it becomes stalled. As a result, the vCPU utilization is doubled. This observation demonstrates the importance of developing a new running task migration heuristic by considering vCPU inactivity to optimize workload throughput (§ 3.3).

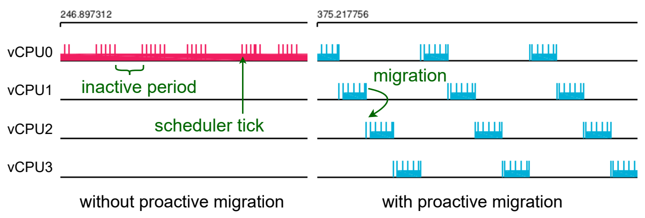


Figure 3. Proactive migration can prevent the stalled running task by harvesting unused vCPU cycles.

Deficient Work Conservation. Most schedulers like CFS, adhere to the work conservation invariant, considering violations as bugs [39]. However, some idle vCPUs might be bad choices for task placement. An idle vCPU with extremely low capacity compared to others can turn a task into a *straggler* delaying other dependent tasks. Scheduling tasks to an idle vCPU stacked with other busy vCPU leads to expensive vCPU switches. It also shifts the control of task scheduling from the VM to the host, resulting in double scheduling problems, such as LHP and *priority inversion* (i.e., a low-priority task may interfere with a high-priority task when they are scheduled on two stacked vCPUs as the host is unaware of the task priority inside vCPUs). To demonstrate departing from strict work conservation could improve performance, we conducted the following experiments.

Idle vCPU with extremely low capacity: We created a VM with 16 vCPUs pinned on 16 cores. By stressing one core with a high-priority task on the host, one vCPU has significantly lower capacity than the others. Throughput-oriented benchmarks, particularly synchronization-intensive ones from Parsec [40], were executed under two scenarios: *work-conserving* and *non-work-conserving*. In the latter, the low-capacity vCPU was excluded from task placement. The results in Figure 4 (left) revealed that leaving the straggler vCPU idle achieves up to 43% higher workload throughput.

Idle vCPU stacked with busy vCPUs: In this experiment, the 16-vCPU VM was configured with vCPUs stacked in pairs on 8 cores. Benchmarks were executed under *work-conserving* and *non-work-conserving* scenarios. In the latter, one vCPU from each stacking group was excluded from task placement. As depicted in Figure 4 (right, first six bars), the non-work-conserving scenario yielded up to 30% higher throughputs without double scheduling issues and expensive vCPU switches. To demonstrate the *priority inversion* problem, we launched a best-effort workload with low priority on one vCPU of each stack group. We repeated the tests, launching each benchmark with 8 threads instead. Additionally, the vCPUs that did not run the low-priority workload were excluded from task placement in the *non-work-conserving* scenario. The results, as shown in Figure 4 (right, last six bars), illustrate that the benchmark suffered from interference from the low-priority workload under the *work-conserving* scenario. In the non-work-conserving scenario, throughput can experience up to a 6.7x improvement.

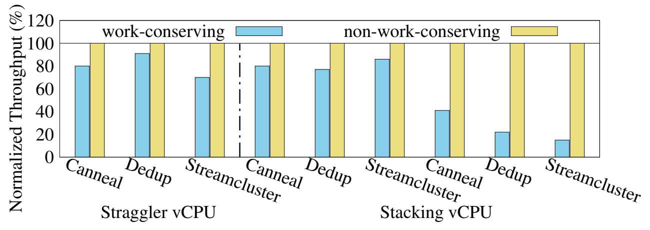


Figure 4. Non-work-conserving policy yields better performance. Throughput is normalized to that with non-work-conserving policy. Higher is better.

These observations demonstrate the critical importance of considering both workload and vCPU characteristics when making scheduling decisions, rather than strictly adhering to work conservation principles (§ 3.4).

3 The vSched Approach

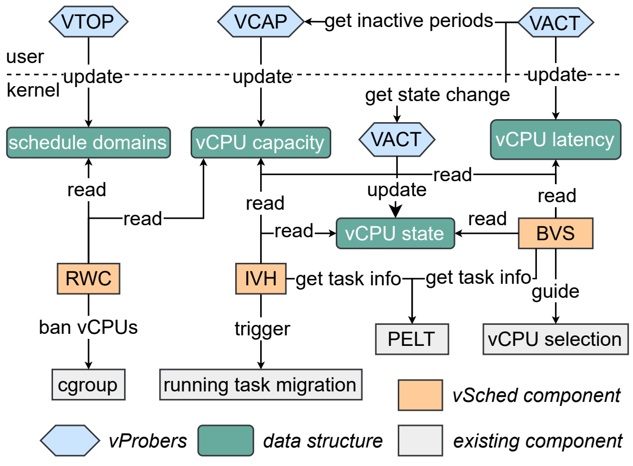


Figure 5. vSched Overview.

vSched aims to optimize task scheduling in Cloud VMs via accurate vCPU abstraction. Figure 5 presents an overview of *vSched*, highlighting its user-level and kernel-level components and their interactions with other system elements within the VM. To enhance the effectiveness of existing capacity-aware and topology-aware heuristics, VCAP and VTOP are designed to periodically probe the capacity and topology of vCPUs, updating the corresponding kernel data structures. To further capture vCPU dynamics, VACT is designed to probe vCPU activity (latency and state) and exposes these metrics to the kernel as new vCPU characteristics.

Leveraging the *vProbers* (VCAP, VTOP, VACT), *vSched* introduces a suite of optimization techniques to address the issues identified in our motivating experiments. To reduce extended runqueue latency, biased vCPU selection (BVS) is employed to match latency-sensitive tasks with vCPUs where they can experience minimal delay. This is achieved by guiding the vCPU selection process. To address the *stalled running task* problem, intra-VM harvesting (IVH) is designed to proactively trigger running task migration to move high-CPU-utilization tasks from a soon-to-be-inactive vCPU to another where they can continue making progress. Both BVS and IVH utilize per-entity load tracking (PELT) [30] to classify tasks and choose target vCPUs based on the probed vCPU capacity and activity. Finally, relaxed work conservation (RWC) is proposed to intentionally leave problematic vCPUs idle by excluding them from task placement using cgroups. The *vSched* approach serves as a practical scheduling framework to make schedulers vCPU-aware. The following section will discuss each *vSched* component in detail.

3.1 vCPU Probing

vCPU Capacity. Probing dynamic vCPU capacity poses several challenges. Firstly, to measure the percentage of CPU time a vCPU can utilize on a core, a prober must continually stress the vCPU. This not only consumes vCPU cycles but also interferes with the user's workload. Secondly, the capacities of hosting cores are unknown and need measurement, especially considering their dynamic nature due to frequency changes or vCPU migration. Thirdly, vCPU capacity cannot be probed individually as capacities of different vCPUs measured at different times can be misleading.

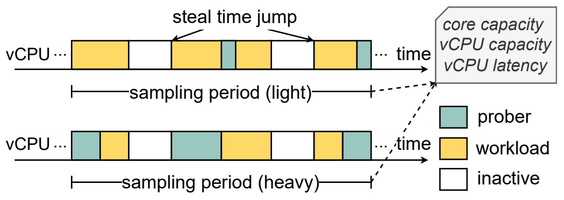


Figure 6. One sampling period for VCAP and VACT.

To overcome these challenges, VCAP adopts a *cooperative and multi-phase sampling* approach, as illustrated in Figure 6. VCAP launches one prober thread per vCPU for simultaneous sampling, which is conducted periodically. This allows VCAP to expose the relative capacities of different vCPUs at the same time with low overhead. There are two types of sampling periods: light and heavy. During the light phase, conducted regularly, each prober thread runs at the lowest priority, ensuring vCPUs are kept busy with a best-effort task when idle. This allows VCAP to profile the percentage of CPU time a vCPU can share on a core by collecting steal time [41]—the time a vCPU, with a running task, waits on the host runqueue. Interference with user workload is minimized by utilizing otherwise unused vCPU cycles.

In the heavy phase, VCAP additionally profiles core capacity by setting prober threads to high priority to guarantee execution, delicately measuring the amount of work a vCPU can complete (i.e., vCPU capacity). Alongside steal time, core capacity is measured. For instance, if prober threads report a vCPU capacity of 50 while using 50% vCPU time (the other 50% is used by workload and system), the actual vCPU capacity is 100. If the stolen time amounts to 80% of the sampling period, then the hosting core capacity would be 500. Knowing core capacity, vCPU capacity can be easily calculated during the light sampling phase after probing the percentage of CPU time the vCPU receives from the core.

All capacities are normalized to show relative performance. To prevent excessive scheduling events due to capacity fluctuation, history is incorporated using exponential moving average (EMA), which gives more weight to recent capacity and decays history for a smooth estimate. To ensure prompt updates of capacity with high accuracy and low overhead, the sampling period is set to be long enough (e.g., 100 ms) for a vCPU to execute at least once on the core. Additionally, the heavy sampling frequency is kept low to restrict probing costs. In the light phase, VCAP consumes minimal CPU time because vCPUs typically run user workloads or get preempted during the sampling period.

vCPU Activity. Designing VACT presents considerable challenges, particularly in providing real-time vCPU state without hypervisor support and estimating *vCPU latency* with fine granularity. To tackle the first challenge, VACT employs a heart-beat mechanism. It instruments the kernel to record a system-wide timestamp (e.g., sched_clock [42]) per scheduler tick on each vCPU. Upon a vCPU state query, if the examiner finds that the most recent timestamp reported by the examinee has been outdated for several ticks, the target vCPU is considered inactive; otherwise, it is still active. This approach provides the near real-time state without requiring hypervisor support and incurs minimal overhead.

To address the second challenge, VACT focuses on measuring the vCPU inactive periods to estimate its latency as these periods can indicate how quickly a vCPU can become active to respond to events such as rescheduling interrupts. To achieve this, VACT leverages the sample periods of VCAP to gather activity-related information. In particular, VACT instruments the kernel to check for increases in steal time since the previous tick on a vCPU. A notable increase in steal time indicates that the vCPU was preempted and has just been rescheduled as illustrated in Figure 6. Small jumps are filtered out to eliminate noise caused by instantaneous system tasks on the host. For each vCPU, VACT maintains a preemption counter which is updated upon each qualified steal time jump. The user-space component of VACT collects this metric before resetting it at the end of each VCAP sampling period. Using this data, the average vCPU inactive period can be calculated. For instance, if during a sampling period, the stolen time is 60 ms and 5 preemptions are detected, the average inactive period would be 12 ms. The average inactive period is then exposed as the new vCPU abstraction (*vCPU latency*). This method ensures high accuracy with minimal overhead.

vCPU Topology. vTop aims to construct the vCPU topology by probing the distance between vCPUs, which can be estimated by measuring the *cache line transfer latency* [43]. To achieve this, vTop launches two prober threads on a pair of target vCPUs, each conducting the following operations: 1) Atomically read and write a 64-byte memory block (the cache line size) and record the latency. 2) Spin until the other thread atomically reads and writes the same memory block. 3) Repeat the first step until the block has been successfully updated a certain number of times or a timeout is reached. The lowest latency is recorded into a matrix where latencies between each vCPU pair are stored.

When two vCPUs run on sibling hardware threads, the latency is *low* since the block can be fetched directly from the shared private cache. In the case where two vCPUs are scheduled to cores in the same socket, the latency is *medium* as one vCPU's copy of the block in its private cache is invalidated for cache coherence after the other updates it, leading to a transfer from the other vCPU's private cache or the shared LLC. For vCPUs mapped to cores in different sockets, cache line transfer occurs through the inter-socket bus [44], resulting in *high* latency. When two vCPUs are stacked, the prober would mostly spin since the stacked vCPUs can never run simultaneously, resulting in very few cache transfers, as shown in Figure 7 (vCPU0 and vCPU1). If the number of successful updates is extremely low after the timeout, we return *infinitely high* latency. These distinct latencies enable vTop to differentiate between various vCPU topologies.

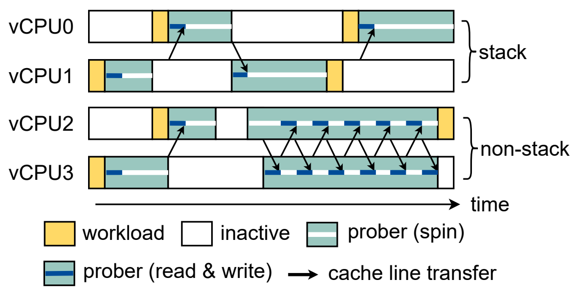


Figure 7. Probing vCPU distance with cache line transfer latency under stacking and non-stacking topologies.

One challenge of this method is to avoid misidentifying non-stacking vCPUs as stacking ones. Even when two vCPUs are not stacked, they might not be active simultaneously throughout the probing process due to inactive periods or interference from user workloads. As a result, they could be incorrectly labeled as stacking vCPUs. For example, consider vCPU2 and vCPU3 in Figure 7: they are not stacked, but initially, due to the limited overlap between their active periods, the prober threads mostly spin, resulting in only one transfer. However, with extended effort, more transfers can then be observed. Therefore, in vTop, the probing threads are assigned high priority to minimize interference within the VM, and the timeout is extended if few transfers occur to increase the likelihood that both vCPUs are active during probing, thus helping to avoid misjudgment.

Another challenge is ensuring that measurements are fast enough to enable periodic probing for prompt detection of topology changes, especially in large VMs where measurement complexity is exponentially increased. vTop proposes three optimizations. First, it skips probing between vCPUs whose distances can be inferred from existing probed results. For example, if we know that vCPU0 and vCPU1 are stacked, and vCPU0 and vCPU2 are on different sockets, then vCPU1 and vCPU2 can be skipped since they must also be on different sockets. Second, it probes the socket topology first, then other topologies within each socket can be probed in parallel. Lastly, it introduces the validation period. A full topology probing is only conducted if periodic validation fails. The validation period is much lighter since fewer pairs need to be probed to validate the current topology. Additionally, the validation can be done with higher parallelism without causing interference between different probing pairs. For example, each pair of SMT vCPUs can be validated in parallel, and once confirmed, one vCPU of each SMT group can participate in the socket topology validation. These optimizations significantly reduce the probing time to just a few hundred milliseconds, as demonstrated later in the evaluation section.

3.2 Biased vCPU Selection (BVS)

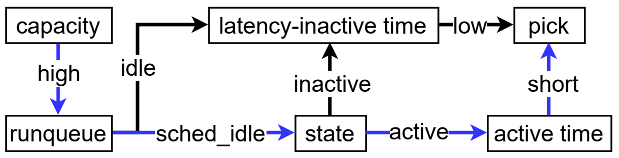


Figure 8. BVS heuristic for vCPU search.

BVS is proposed to match small latency-sensitive tasks with vCPUs where they can minimize the extended runqueue latency. BVS is an activity-aware optimization where vCPU selection considers vCPU latency and state as illustrated in Figure 8. Initially, PELT and user-space tools [45, 46] are used to identify small tasks with low latency requirements. When placing such tasks, BVS prioritizes high-capacity vCPU to prevent runqueue saturation. BVS then checks the runqueue: if empty, it examines vCPU latency and idle duration. A vCPU with low latency and prolonged idleness (i.e., long inactive time) is selected as it tends to wake up quickly. If low-priority tasks (sched_idle ones) are in the queue, BVS assesses vCPU state: if inactive for a long time with low latency, it's likely to be active soon and is a good choice. Otherwise, if a vCPU becomes active recently with sched_idle tasks, it's an ideal destination as the task can start there immediately and finish within the remaining active period, as highlighted with the blue path. vCAP and vACT expose the median capacity and latency to allow BVS to quickly identify high-capacity and low-latency vCPUs. vACT tracks average active and inactive periods, allowing BVS to determine if a vCPU has been long-inactive or recently active. To reduce the vCPU selection latency, a first-fit policy is used to quickly pick any acceptable vCPU as shown in Figure 8. Only when none is found, the CFS heuristic is used. This allows BVS to perform aggressive searches without being limited by the CFS heuristic, which searches vCPUs within the preferred LLC domains.

3.3 Intra-VM Harvesting (IVH)

Proactive task migration is shown to be effective in mitigating the *stalled running task* issue by harvesting the unused vCPU resource. However, a significant obstacle is the *migration delay*-the time taken to reschedule a task on the target vCPU after initiating the migration. This delay, often due to extended runqueue latency, can greatly diminish the effectiveness of this approach. To overcome this challenge, IVH proposes an *activity-aware running task migration* technique that pre-wakes the target vCPU, initiating migration only when both the source and target vCPUs are active to minimize migration delay. Additionally, IVH employs a heuristic similar to BVS to search for a target vCPU capable of quickly engaging in the migration process. In each scheduler tick, IVH identifies CPU-intensive tasks (using PELT) that have been running for a minimum duration (e.g., 2 ms) on a vCPU with inactive periods. It then aggressively searches for migration targets. Once a suitable target is found, the migration proceeds in three steps, illustrated in Figure 9 (left).

Firstly, the source vCPU sends an interrupt to prompt the target vCPU to initiate the migration and continue executing the task (❶). If the target is idle, it wakes up, sends a task pull request, and begins spinning until the migration completes or the source vCPU is preempted (❷). Upon receiving the pull request, a stopper thread [47] is activated to enqueue the running task, which can then be detached and attached to the target vCPU's runqueue (❸). Migration to an active vCPU with sched_idle tasks can be accomplished with minimal delay, even if the target is identified towards the end of the source's active period (Figure 9-middle). IVH would abandon migration if the target issues a late pull request after the task has already stalled due to extended delay, as there would be no benefit (Figure 9-right). Note that frequent task migrations are generally discouraged due to the inability of tasks to effectively utilize cache [48]. However, tasks targeted by IVH are less vulnerable to this issue, as the cache is polluted during vCPU inactive periods even without migration.

3.4 Relaxed Work Conservation (RWC)

RWC proposes intentionally leaving some problematic vCPUs idle by hiding them from task placement to prevent relevant

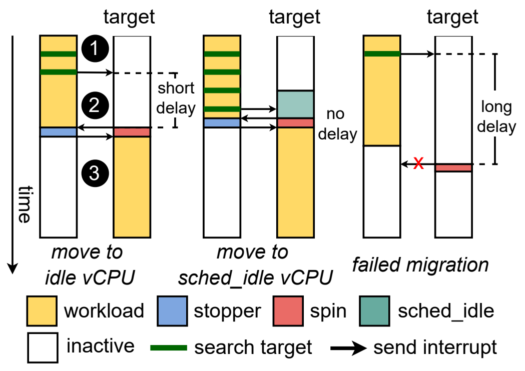


Figure 9. IVH's activity-aware running task migration.

issues. To address the *straggler* problem, vCPUs with capacities significantly lower (e.g., 10x lower) than the average capacity are considered straggler vCPUs, intended only for running best effort (sched_idle) tasks. Therefore, vCAP, during the light sampling, can still run on a straggler vCPU to probe capacity in case it increases. When other workloads, such as synchronization-intensive ones, are present, the straggler vCPU remains idle. To prevent double scheduling problems such as priority inversion and LHP, and to avoid expensive vCPU switches, only one vCPU from each stacking group is considered for task placement, while the remaining ones are banned from running any tasks, including vCAP, which could lead to priority inversion issues. The only exception is vTOP, which is still allowed to conduct probing on all vCPUs to detect any changes in stacking topology, allowing RWC to adjust hidden vCPUs.

4 Implementation Details

vSched has been implemented in Linux kernel v6.1.36 on top of CFS. Although it can be easily ported to the latest kernel that uses the Earliest Eligible Virtual Deadline First (EEVDF) scheduler [48], we found this new scheduler hasn't been fully tested and presents issues when interacting with cgroup [49]. Therefore, we returned to CFS since our main goal is to demonstrate how accurate vCPU abstraction can help improve task schedulers that share similar goals. For the same reason, instead of implementing a new scheduling class using the existing framework (e.g., sched_ext [50]), we added BPF hooks [27] to CFS core selection paths and scheduler tick handler to implement the activity-aware optimizations (BVS and IVH). Since *vProbers* are implemented in user space mostly, the probed results are exposed to the scheduler through a new kernel module. For example, the probed topology is stored as three lists for each vCPU to store the siblings for each topology level. The kernel module uses *rebuild_sched_domains* to update the schedule domains based on those lists. vACT instruments the scheduler tick handler to monitor vCPU state changes and provides a kernel function to query the vCPU state. To implement RWC, vTOP collaborates with vcap to halt the sampling on the banned stacked vCPUs and guides cgroup [51] to hide problematic vCPUs from user workloads. Overall, *vSched* has a code size of 1612 Lines of Code (LoCs). The kernel portion, including the kernel module, kernel functions, and BPF hooks, amounts to 636 LoCs, and the BPF program involves 417 LoCs. The user-level portion of *vProbers* involves 559 LoCs.

5 Evaluation

Our evaluation aims to demonstrate the following:

- The ability of *vProbers* to expose accurate vCPU abstraction with high performance (§ 5.2).
- The effectiveness of the probed accurate vCPU abstraction in enhancing existing scheduling optimizations (§ 5.3).
- The impact of activity-aware optimizing techniques, namely bvs and ivh, on improving workload latency (§5.4) and throughput (§5.5), respectively.
- The versatility of *vSched* in optimizing diverse workloads across various VM types (§5.6).
- The adaptability of *vSched* in maintaining Quality-of-Service in the face of dynamic vCPU resource changes (§5.7).
- The *vSched* improvement in multi-tenant hosts under realistic interference from multiple co-located VMs (§5.8).
- The small cost incurred by *vSched* when offering significant performance gains, and the minimal overhead incurred by *vSched* when workloads cannot benefit from the accurate vCPU abstraction (§5.9).

5.1 Evaluation Settings

An HPE ProLiant DL580 Gen10 server (4 Intel Xeon Gold 6138 20-core CPUs, 256 GB memory, and 2 TB SSD) is used to host Linux VMs on the KVM hypervisor. Hyper-threading and Dynamic voltage and frequency scaling (DVFS) are enabled. Both the host OS and guest OS are kernel v6.1.36. The CPU bandwidth control is employed on the host along with *sched_min_granularity_ns* and *sched_wakeup_granularity_ns* to adjust the inactive and active periods of a vCPU to accurately control its capacity and activity (latency and state) when competing with a CPU-bound vCPU. The vCPU topology is controlled by pinning vCPUs on cores using virsh [52]. The values of the tunables in *vSched* are listed in Table 1.

Table 1. Chosen values of *vSched* tunables.

Tunables Description	Value
vcap sampling period	100 milliseconds
vcap light sampling frequency	Every 1 second
vcap heavy sampling frequency	Every 5 light samplings
vcap EMA decay factor	50% per 2 periods
vtop sampling frequency	Every 2 seconds
vtop targeted cache transfers	500 times
vtop cache transfer timeout	15000 transfer attempts
ivh migration threshold	After 2 milliseconds

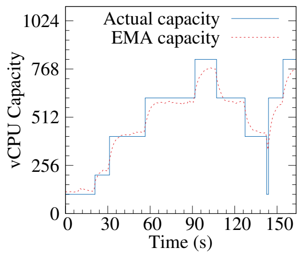
To showcase *vSched*'s capacity to benefit diverse workloads, we carefully selected 34 benchmarks exhibiting different characteristics across various domains. Specifically, we chose 8 Tailbench benchmarks [33] to highlight *vSched*'s efficacy in enhancing small latency-sensitive tasks. Additionally, we employed 10 Parsec benchmarks and 11 Splash-2x benchmarks [40] to demonstrate *vSched*'s effectiveness in improving throughput-oriented workloads, encompassing synchronization-intensive and scientific computing workloads, respectively. Nginx [53] and Pbzip2 [54] were selected to illustrate *vSched*'s impact on memory-intensive web server operations and parallel file compression tasks. Further, microbenchmarks like Hackbench [55], Fio [56], and Sysbench [34] were utilized to generate synthetic workloads, stressing the scheduler. We conducted warm-up runs for each benchmark, followed by five runs to obtain average values.

To demonstrate the applicability of *vSched* across various VM types with diverse vCPU performance features, we employ two representative cloud VMs: the resource-constrained VM (rcvm) and the high-performance VM (hpvm). rcvm, typically utilized for cloud resource harvesting or in resource-limited edge environments [6, 9, 57], is characterized by its small size and frequent hosting on highly-contended hosts. It comprises 12 vCPUs, with the first 10 vCPUs mapped to 5 pairs of SMT siblings and the last two stacked together. Among the first 10 vCPUs, two are designated as straggler vCPUs. The remaining 8 vCPUs are categorized into four types: high-capacity-high-latency (hchl), high-capacity-low-latency (hcll), low-capacity-high-latency (lchl), and low-capacity-low-latency (lcll), with two vCPUs in each category. For example, the hcll vCPU possesses double the capacity and one-third the latency of the lchl vCPU. In contrast, hpvm is larger, spanning multiple sockets with less contention on the host, often serving performance-critical workloads. It features 32 vCPUs divided into 4 groups, each mapped to 4 pairs of SMT siblings in a separate socket. Similarly categorized into 4 types, the vCPUs in three groups mirror those of rcvm, with the last group dedicatedly utilizing the hosting cores. Unlike rcvm, hpvm does not have stacked or straggler vCPUs.

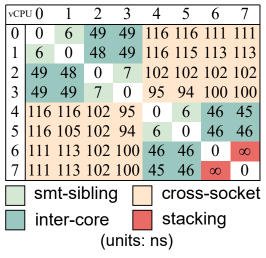
5.2 Accuracy and Performance of vProbers

To show the advantage of the EMA capacity probed by vcap, we manually adjust a vCPU's capacity to observe the probed result. Figure 10 (a) illustrates the comparison between actual capacity (solid line) and probed EMA capacity (dotted line). The EMA capacity successfully provides a trend that matches the capacity changes while smoothing out spikes to avoid unnecessary task migrations.

To assess the accuracy of vtop, we created a 8-vCPU VM with all topology hierarchies. In particular, vCPU0-vCPU3 are pinned to two pairs of SMT siblings in one socket, while in another socket, vCPU6 and vCPU7 are stacked, and vCPU4 and vCPU5 are mapped to a pair of SMT siblings. Figure 10 (b) presents the probed cache line transfer latency matrix, showing distinct latency for different topologies, enabling vtop to accurately update schedule domains. Additionally,



(a) EMA capacity

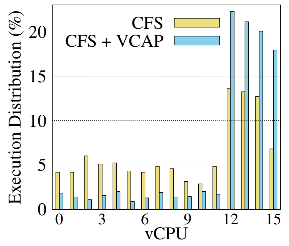


vCPU	0	1	2	3	4	5	6	7
0	0	6	49	49	116	116	111	111
1	6	0	48	49	116	115	113	113
2	49	48	0	7	102	102	102	102
3	49	49	7	0	95	94	100	100
4	116	116	102	95	0	6	46	45
5	116	105	102	94	6	0	46	46
6	111	113	102	100	46	46	0	∞
7	111	113	102	100	45	46	∞	0

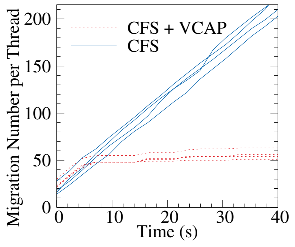
smt-sibling
cross-socket
inter-core
stacking
(units: ns)

(b) Cache line transfer latency

Figure 10. Accuracy measurements of vcap and vtop.



(a) Asymmetric Capacity



(b) Symmetric Capacity

Figure 11. Impact of the accurate vCPU capacity.

we measured how quickly vtop can probe the topologies in rcvm and hpvm, respectively. As shown in Table 2, the probing time is sub-second, especially, validation tasks up to 4x shorter time than full probing, allowing quick detection of topology changes. Note that validation takes longer in rcvm compared to the larger hpvm due to the additional effort made to confirm stacking topology.

Table 2. vtop probing time (unit: ms).

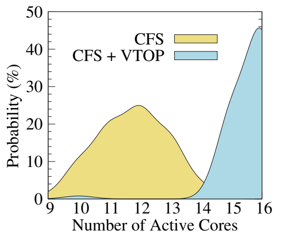
Config	rcvm-full	rcvm-validate	hpvm-full	hpvm-validate
Time (ms)	547	388	665	160

5.3 Enhancing Existing Scheduling Optimizations

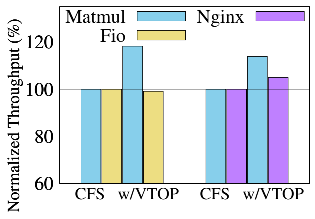
This section demonstrates the impacts of *vProbers* on existing capacity-aware and topology-aware optimizations. For the former, Linux measures *CFS capacity* to represent the capacity that can be used for CFS tasks. It excludes the steal time, enabling better measurement in an overcommitted VM. However, steal time can only be observed when a vCPU is busy. A low-capacity idle vCPU without steal time can appear to be a high-capacity vCPU, misleading the scheduler. To show how vcap can address such issues, we conducted tests under two scenarios: *asymmetric capacity* and *symmetric capacity*.

In the first scenario, we created a 16-vCPU VM where the last four vCPUs have 2x higher capacity than others. Sysbench is launched with 4 CPU-bound threads. As shown in Figure 11 (a), under CFS, Sysbench spent only 44% of the time on high-capacity vCPUs, whereas these vCPUs are selected 81% of the time when vcap is enabled. Consequently, Sysbench achieves a 32% higher throughput with accurate vCPU capacity. In the second scenario, all vCPUs are set to have the same capacity and we observe 4% higher throughput with vcap enabled. This improvement is attributed to the prevention of adverse task migrations to idle vCPUs that appear stronger due to the absence of steal time. Profiling migrations during Sysbench execution with and without vcap revealed a 74% reduction in migrations with vcap enabled, leading to reduced overhead, as shown in Figure 11 (b).

For topology, we conducted two sets of experiments to illustrate *vProbers*'s impact on SMT-aware and LLC-aware optimizations, respectively. In the SMT-aware optimization experiments, we configured a VM with 32 vCPUs pinned to 16 pairs of SMT siblings on 16 cores and tested under two scenarios: *underloaded system* and *mixed workloads*. In the underloaded system scenario, Sysbench was initiated with 16 CPU-bound threads. Monitoring the number of cores utilized over time revealed that CFS typically employed 11-12 out of 16 cores due to the absence of SMT topology awareness. Conversely, with vtop enabled, 15-16 cores could be utilized as idle vCPUs with a busy SMT sibling were appropriately avoided, as demonstrated in Figure 12 (a). In the mixed workload scenario, we mixed CPU-intensive Matmul (matrix multiplication) with memory-intensive Nginx or I/O-intensive Fio, each launching 16 threads. Figure 12 (b) illustrates that, by resolving resource conflicts with accurate SMT topology, CFS combined with vtop allows for up to an 18% enhancement in Matmul performance, with a minor (5%) improvement in Nginx and no degradation in Fio.



(a) Underloaded System



(b) Mixed Workloads

Figure 12. Effective SMT-aware scheduling with vtop.

To illustrate the impact of vtop on LLC-aware optimizations, we pinned the 32 vCPUs to two sets of 16 cores across two sockets. Three benchmarks—Hackbench, Dedup, and Nginx—each showcasing unique intra-thread communication patterns, were selected. Two instances of each benchmark were launched for testing. Figure 13 demonstrates that vtop's correct socket topology exposure effectively segregates each instance's threads into separate LLC domains, resulting in higher communication efficiency through LLC with up to a 99% reduction in inter-process interrupts (IPIs) and a 14.5% increase in instructions per cycle (IPC) on average. Consequently, workload throughput (averaged across two instances) increased by 26% on average compared to CFS. We utilized IPC to indirectly demonstrate cache efficiency, as cache miss profiling tends to be less reliable in VMs [58].

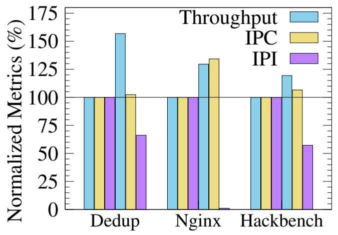


Figure 13. Effective LLC-aware optimizations with vTOP. For each benchmark, the first three bars represent performance metrics under CFS, while the last three bars depict performance with vTOP enabled. Metrics are normalized to that with vTOP disabled.

5.4 Latency Reduction with BVS

To demonstrate the effectiveness of BVS in reducing workload latency, we conducted experiments using workloads from Tailbench. We compared their performance with and without BVS. *vProbers* are enabled in both configurations. In addition, the benchmarks were executed with and without best-effort tasks, respectively. The best-effort tasks were assigned with the lowest priority (sched_idle) to harvest vCPU cycles when the benchmark was inactive. For the BVS tests, we created an overcommitted 16-vCPU VM on 16 cores in one socket, configured with asymmetric vCPU latency and symmetric capacity. Half of vCPUs have 2x lower latency than the others. This setup minimizes the noise from capacity and topology. The results in Figure 14 demonstrate that BVS can effectively reduce 95th tail latency by 42% on average.

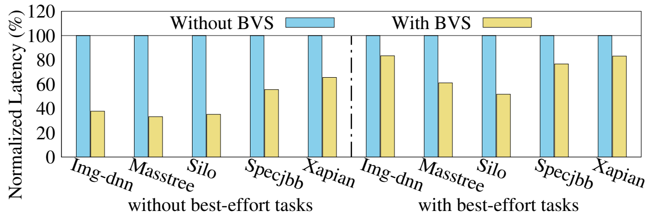


Figure 14. The effectiveness of BVS. The p95 tail latencies are normalized to when BVS is disabled. Lower is better.

To show that this improvement stems from reduced runqueue latency, we provide queue time (runqueue latency), service time (running time), and end-to-end time reported by *Masstree* in Table 3. BVS reduces queue time by 44% and 70% with and without best-effort tasks, respectively. Due to the small service times, the runqueue latency reduction directly translates to lower end-to-end tail latency. Furthermore, we introduce a new configuration: BVS that doesn't consider the vCPU state, to understand how considering the vCPU state in BVS aids in reducing latency on sched_idle vCPUs running best-effort tasks. As shown in the table (second-to-last column), the latencies increase in this new configuration compared to BVS, indicating that prioritizing active sched_idle vCPUs helps to further reduce workload latency.

Table 3. Masstree p95 latency breakdown (unit: ms).

Setting	No best-effort tasks		With best-effort tasks		
	No BVS	BVS	No BVS	BVS (no state check)	BVS
Queue	32.73	9.92	20.66	15.47	11.48
Service	0.36	0.39	0.32	0.36	0.37
End-2-end	33.44	10.25	21.28	16.29	11.95

5.5 Increased Throughput with IVH

IVH proactively migrates running tasks to mitigate the *stalled running task* problem. It can effectively improve throughput in an underloaded system by harvesting unused vCPUs. To show this, we run various throughput-oriented workloads with different thread counts in a 16-vCPU VM, overcommitted with another VM on 16 cores in one socket, where each vCPU shares 50% of core time. The results in Figure 15 illustrate that IVH achieves significant throughput improvements, up to 82%, particularly when there are fewer threads and more unused vCPUs to harvest. Note that even with 16 threads, there's a 17% improvement on average. This suggests that in certain phases, the system is underloaded with only a few active threads, which can be accelerated by IVH.

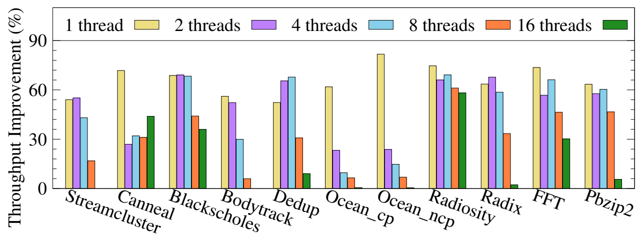


Figure 15. Throughput improvement with IVH compared to when IVH is disabled. Higher is better.

IVH employs activity-aware migration, which involves pre-waking the target vCPU and executing the migration only when both the source and target vCPUs are active. To demonstrate its advantage, we compare IVH with its activity-unaware counterpart, which migrates tasks directly without pre-waking the target. We use canneal as an example and present its performance with and without activity-aware migration in Table 4. The results indicate that migration delays resulting from activity unawareness can impede the effective utilization of unused vCPU cycles.

5.6 Overall Improvement with vSched

We conducted comprehensive experiments with a wide range of workloads in both RCVM and HPVM to evaluate the overall performance of *vSched*. Each workload was executed under three configurations: CFS, enhanced CFS, and *vSched*, with

Table 4. Canneal execution time (unit: s).

#Threads	1	2	4	8	16
IVH (activity-unaware)	408.0	298.2	210.7	149.8	124.9
IVH (activity-aware)	347.9	274.2	172.7	129.2	100.7

the number of threads equal to or greater than the number of vCPUs. In enhanced CFS, *vProbers* and RWC were enabled to provide accurate vCPU abstraction and hide problematic vCPUs. The results are presented in Figure 18 and Figure 19. In RCVM, enhanced CFS can reduce latency by 1.4x and increase throughput by 59%, whereas *vSched* can reduce latency by 1.6x and increase throughput by 69% on average compared to CFS. In HPVM, compared to CFS, enhanced CFS can reduce latency by 1.5x and increase throughput by 13%, while *vSched* can reduce latency by 2.3x and increase throughput by 18% on average. It is clear that BVS and IVH provide additional improvements by considering vCPU activity. Moreover, the overall throughput improvement with *vSched* is higher in RCVM where RWC can effectively hide stragglers and stacking vCPUs, and IVH yields more benefit when *stalled running tasks* problems are severe due to high host contention. On the other hand, *vSched* achieves more latency reduction in HPVM where BVS can maximize minimal latency from dedicated vCPUs. Streamcluster and volrend in HPVM are two exceptions, experiencing slight performance degradation when the correct socket topology is exposed. They employ user-level spin-based synchronization, which leads to the unaddressed LHP-like problem [59] that becomes more severe when more imbalance is tolerated among sockets exposed by VTOP.

5.7 Adaptability of vSched

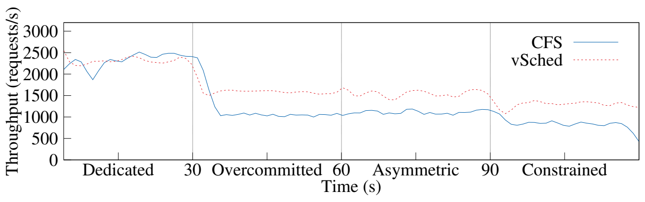


Figure 16. *vSched* can respond quickly to vCPU changes.

To demonstrate that *vSched* can maintain high QoS in the cloud by quickly adapting to vCPU dynamics, we created a 16-vCPU VM running Nginx to report live throughput as vCPUs underwent multiple phases of changes with CFS and *vSched*. The results are shown in Figure 16. Initially, vCPUs were dedicatedly hosted on 16 cores in one socket, and *vSched* yielded similar performance to CFS since the existing vCPU abstraction was already accurate. Next, the VM was overcommitted with a competing VM, with each vCPU sharing half of a core. Under CFS, the throughput dropped to half due to contention, whereas *vSched* maintained higher throughput by effectively preventing stalled running tasks with IVH. We then configured the host to give half of the vCPUs 2x higher capacity than the others without changing the overall capacity. CFS struggled to leverage the high-capacity vCPUs for higher throughput, while *vSched* maintained the same throughput since the vCPU utilization was already maximized with IVH. Finally, we stacked two vCPUs and significantly reduced the capacity of another two to simulate a resource-constrained host. *vSched* can quickly recover the throughput by using RWC to hide problematic vCPUs, whereas Nginx suffered lower throughput with CFS.

5.8 vSched Improvement in Multi-tenant Hosts

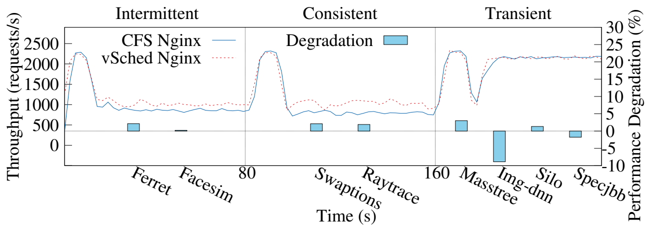


Figure 17. *vSched* improves QoS under varying interference in a multi-tenant environment. Performance degradation refers to throughput reduction or latency increase experienced by co-located VMs under *vSched* compared to CFS.

In this section, we demonstrate that *vSched* can maintain high QoS in a multi-tenant environment where vCPUs are highly dynamic and unpredictable. We co-located multiple 16-vCPU VMs, allowing their vCPUs to be freely scheduled on 16 cores. We deployed Nginx in one VM and compared its live throughput with *vSched* and CFS while other VMs ran diverse workloads over time to generate realistic interference. As shown in Figure 17, we first launched two synchronization-intensive workloads (facesim and ferret) in two co-located VMs to create *intermittent interference*. *vSched* outperformed CFS, consistently achieving 15% higher throughput, while the average slowdown of facesim and ferret due to *vSched* is only 1.2%.

We then terminated facesim and ferret and launched two computation-intensive workloads (swaptions and raytrace) to generate *consistent interference*. The large spikes in Nginx reflect periods of no interference before the new workloads were launched in co-located VMs. Under heavier interference, Nginx throughput with CFS dropped, while *vSched* maintained high QoS, outperforming CFS by 24% on average. Although swaption and raytrace performance degraded by 2.1% and 1.9%, respectively, due to increased vCPU utilization from Nginx with *vSched*, we don't consider this a penalty, as *vSched* simply ensures its VM claims a fair share.

Following the completion of swaption and raytrace, four latency-sensitive workloads with small tasks were launched in four co-located VMs to create *transient interference*. In this phase, under lighter interference, Nginx throughput under CFS increased significantly. *vSched* yielded a similar increase, as the vCPUs behaved similarly to dedicated vCPUs, leaving

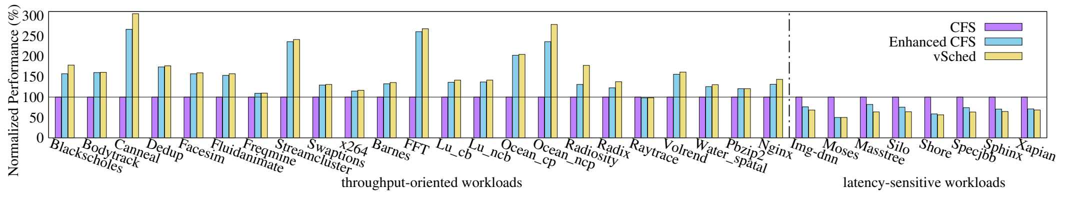


Figure 18. RCVM results. Both throughput and p95 tail latency are normalized to CFS. Higher throughput and lower latency are better.

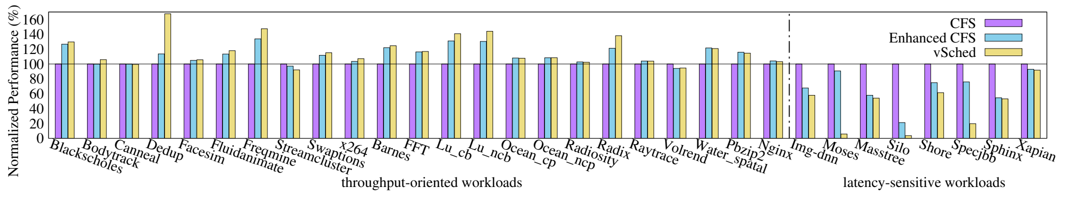


Figure 19. HPVM results. Both throughput and p95 tail latency are normalized to CFS. Higher throughput and lower latency are better.

little mismatch in vCPU abstraction for *vSched* to address. Surprisingly, the p95 tail latency for latency-sensitive workloads was reduced by 3.1% under *vSched*'s impact. This is attributed to the prevention of adverse task migrations by VCAP, as shown in Figure 11, which reduces interference with latency-sensitive workloads compared to CFS.

5.9 Cost and Overhead of vSched

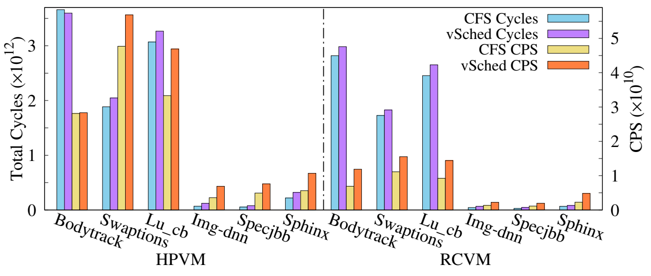


Figure 20. *vSched* cost for throughput-oriented and latency-sensitive workloads on RCVM and HPVM. Lower total cycles are better. Higher cycles per second indicate higher vCPU utilization.

To analyze the cost of *vSched* for various workloads on diverse VMs, we repeated some tests from Section 5.6 and collected the total cycles and cycles per second (CPS) consumed by the VM during workload execution under *vSched* and CFS. As shown in Figure 20, when running throughput-oriented workloads, the VMs consumed only 5.5% more cycles but achieved a 38% higher CPS on average under *vSched* compared to CFS. This demonstrates that the small *vSched* cost can effectively improve vCPU utilization and workload performance. Notably, bodytrack even consumed 1.7% fewer cycles, indicating a reduction in workload cycles due to more coordinated thread execution. For latency-sensitive workloads, VMs consumed 50.5% more cycles and achieved 81.4% higher CPS on average with *vSched* compared to CFS. Despite the higher increase in cycle consumption, the *vSched* cost remains small since these workloads are much lighter with small tasks, evidenced by an average 8.4x lower CPS compared to the throughput-oriented workloads under CFS. Moreover, the improved CPS with *vSched* translated to a substantial reduction in tail latency, as shown in Section 5.6.

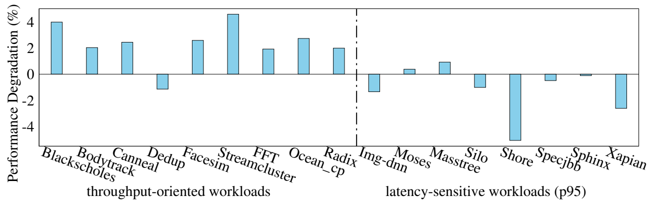


Figure 21. Performance degradation refers to throughput reduction or latency increase with *vSched* compared to CFS.

To measure *vSched*'s overhead, we created a 16-vCPU VM dedicatedly hosted on 16 cores in one socket. This ensures that the vCPUs are always active with symmetric capacity and UMA topology, matching the default vCPU abstraction exposed to the kernel. In this VM, a workload does not benefit from *vSched*, which improves performance through accurate vCPU abstraction. Therefore, comparing workload performance with CFS and *vSched* in this setup reveals the extra overhead caused by *vSched*. The results in Figure 21 indicate that *vSched* incurs low overhead, with only a 0.7% performance degradation on average. Throughput-oriented workloads with high CPU utilization are slightly affected by probing costs, whereas latency-sensitive workloads with small tasks even benefit from the probing periods, which keep vCPUs active and increase core frequency.

6 Discussion

Limitations of vSched. *vSched* requires changes to the guest OS and is not compatible with VMs running commercial OSes. Additionally, since *vSched* is designed to operate without modifying the hypervisor, it lacks control over vCPU scheduling, limiting its ability to engage in cooperative scheduling optimizations that involve information exchange between the guest and the host to improve resource allocation to a VM. However, *vSched* is the best choice for optimizing resource utilization within a VM when hypervisor modifications are not feasible, and it can be seamlessly integrated with any host-side vCPU scheduling enhancements. Lastly, *vSched* uses sampling to minimize overhead. While it can respond to typical vCPU changes [4, 7, 24] within seconds, solutions like XPV [7] and CPS [8] that expose real-time vCPU information from the hypervisor are necessary for sub-second vCPU changes.

vSched Tunables Configuration. The *vSched* tunables are experimentally determined in our evaluation. Specifically, the VCAP sampling period is adjusted to be long enough to include at least one active period for each vCPU. The probing frequencies are set to ensure that *vSched* can respond to vCPU changes within seconds. The VCAP EMA decay factor is calibrated to prevent excessive migrations due to fluctuating capacity. The VTOP targeted cache transfer and cache transfer timeout are tuned to minimize failed probing attempts that result from misidentifying non-stacking vCPUs. The IVH migration threshold is aligned with the scheduler tick, set to trigger migration proactively within 2 ticks after vCPU rescheduling. These tunables can therefore be easily auto-configured across different platforms.

Security Implication of vSched. Probing vCPUs within a VM may expose it to attacks from a co-located adversarial VM. The attacker might run concurrently during the victim's probing phase, deceiving the victim about its actual capacity and topology, and thereby manipulating its workload execution. However, performing such attacks is challenging, if not impractical. The attacker lacks control over the placement of its vCPUs and has limited knowledge of the victim's vCPU topology. Additionally, since probing is integrated with the victim's workload execution, and given the presence of other co-running VMs, identifying distinct probing patterns becomes difficult. It's also unrealistic for *vSched* to exploit the probed results for attacks without precise host information.

7 Related Work

Task scheduling optimizations mostly stem from new hardware features, such as multicore [39], DVFS [21], hierarchical memory [7], and special instruction sets [60], to name a few. *vSched* aims to improve scheduling on virtualized resources. Existing solutions achieve this goal by exposing accurate vCPU information to the VM. XPV [7] exposes NUMA topology changes to the VM, enabling NUMA-aware optimization. CPS [8] reveals NUCA topology and core load changes to improve scalability. UFO [61] provides VMs with the number of cores allocated, allowing the VM to optimize the number of online vCPUs for latency-critical tasks. I-Spinlock [31] addresses LHP and LWP problems by ensuring that a thread only acquires a lock when it can complete the critical section within the vCPU quantum exposed by the hypervisor. However, these solutions require hypervisor modifications, which increase security risks and are hard to deploy in multi-cloud scenarios. However, the new scheduling heuristics in *vSched* can be integrated into these paravirtualized solutions, and we plan to explore this integration in future work.

vCPU scheduling optimizations are developed at the hypervisor layer to provision high-quality vCPUs that can best serve VM workloads. UFO [61] isolates vCPUs from different VMs, as well as the vCPU threads and emulator threads within each VM, to reduce vCPU latency for latency-critical tasks. DASEC [9] implements dynamic asymmetric vCPU scheduling to shift inter-vCPU interference away from tasks on the critical path, redirecting it to non-critical tasks. Some solutions enhance vCPU scheduling by receiving hints from the VM. Pillai [12] proposes sharing task information between the guest and host, allowing a vCPU to be boosted when running latency-sensitive tasks. eCS [13] enables the hypervisor to boost vCPUs running critical-section tasks based on VM-provided hints. CPS [8] notifies the hypervisor when vCPUs within the same cache group are executing interactive threads, helping to avoid migrations of such vCPUs. UFO [61] shares scheduling frequency information within the VM to the hypervisor, offering a hint for determining the optimal number of cores to allocate to the VM for lower tail latency. However, combining these solutions is challenging, particularly when their optimization goals conflict. Despite this, *vSched*, as an orthogonal solution, can help better leverage the high-quality vCPUs provided by these optimizations.

Resource probing tools are implemented to increase user visibility in the cloud. CacheInspector [58] probes CPU cache resources available to the VM, improving QoS for cache-sensitive workloads. vMitosis [24] probes NUMA topology to optimize page table access within the VM. VTOP extends it by probing additional topologies and reducing the probing time from tens of seconds to hundreds of milliseconds.

8 Conclusion

This paper employs innovative probing techniques to unveil the accurate vCPU abstraction, enabling the scheduler to make well-informed decisions. Coupled with virtualization-aware heuristics leveraging the probed results, significant throughput and latency enhancements are achieved in the dynamic multi-cloud environment. We plan to extend our probing efforts to other resources for further optimizations.

9 Acknowledgments

We thank the anonymous reviewers for their constructive comments, and Alain Tchana for his helpful suggestions as the shepherd for this paper. Edward Guo and Jianchen Shan are supported in part by NSF CNS-2324923. Weiwei Jia is supported in part by NSF grant CRII-SHF-2348066.

References

[1] Zhang, Yanqi and Goiri, Íñigo and Chaudhry, Gohar Irfan and Fonseca, Rodrigo and Elnikety, Sameh and Delimitrou, Christina and Bianchini, Ricardo. Faster and Cheaper Serverless Computing on Harvested Resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 724–739, 2021.

[2] Davood Ghatrehsamani, Chavit Denninnart, Josef Bacik, and Mohsen Amini Salehi. The Art of CPU-pinning: Evaluating and Improving the Performance of Virtualization and Containerization Platforms. In *Proceedings of the 49th International Conference on Parallel Processing*, pages 1–11, 2020.

[3] Weiwei Jia, Cheng Wang, Xusheng Chen, Jianchen Shan, Xiaowei Shang, Heming Cui, Xiaoning Ding, Luwei Cheng, Francis CM Lau, Yuexuan Wang, et al. Effectively Mitigating I/O Inactivity in vCPU Scheduling. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 267–280, 2018.

[4] Matthew Elbing and Jianchen Shan. The Linux Load Balance: Wasted vCPUs in Clouds. In *2020 IEEE Cloud Summit*, pages 174–175. IEEE, 2020.

[5] Zongheng Yang, Zhanghao Wu, Michael Luo, Wei-Lin Chiang, Romil Bhardwaj, Woosuk Kwon, Siyuan Zhuang, Frank Sifei Luan, Gautam Mittal, Scott Shenker, et al. SkyPilot: An Intercloud Broker for Sky Computing. In *20th USENIX Symposium on Networked Systems Design and Implementation*, pages 437–455, 2023.

[6] Amazon. EC2 Spot Instances. https://aws.amazon.com/ec2/spot/.

[7] 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*, pages 1–15, 2019.

[8] Yuxuan Liu, Tianqiang Xu, Zeyu Mi, Zhichao Hua, Binyu Zang, and Haibo Chen. CPS: A Cooperative Para-virtualized Scheduling Framework for Manycore Machines. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, pages 43–56, 2023.

[9] Weiwei Jia, Jiyuan Zhang, Jianchen Shan, Jing Li, and Xiaoning Ding. Achieving Low Latency in Public Edges by Hiding Workloads Mutual Interference. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 477–492, 2022.

[10] Jianchen Shan, Xiaoning Ding, and Narain Gehani. APPLES: Efficiently Handling Spin-Lock Synchronization on Virtualized Platforms. *IEEE Transactions on Parallel and Distributed Systems*, 28(7):1811–1824, 2016.

[11] Weiwei Jia, Jianchen Shan, Tsz On Li, Xiaowei Shang, Heming Cui, and Xiaoning Ding. vSMT-IO: Improving I/O Performance and Efficiency on SMT Processors in Virtualized Clouds. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 449–463, 2020.

[12] Vineeth Pillai. Dynamic vCPU Priority Management in KVM. https://lwn.net/Articles/955145/.

[13] 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, 2018.

[14] Xiang Song, Jicheng Shi, Haibo Chen, and Binyu Zang. Schedule Processes, not vCPUs. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, pages 1–7, 2013.

[15] Kenta Ishiguro, Naoki Yasuno, Pierre-Louis Aublin, and Kenji Kono. Revisiting VM-Agnostic KVM vCPU Scheduler for Mitigating Excessive vCPU Spinning. *IEEE Transactions on Parallel and Distributed Systems*, 2023.

[16] Thomas Friebel and Sebastian Biemueller. How to Deal with Lock Holder Preemption. *Xen Summit North America*, 164, 2008.

[17] 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.

[18] 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. IEEE, 2015.

[19] Linux Kernel Documentation. Capacity Aware Scheduling. https://docs.kernel.org/scheduler/sched-capacity.html.

[20] Linux Kernel Documentation. Core Scheduling. https://docs.kernel.org/admin-guide/hw-vuln/core-scheduling.html.

[21] Julia Lawall, Himadri Chhaya-Shailesh, Jean-Pierre Lozi, Baptiste Lepers, Willy Zwaenepoel, and Gilles Muller. OS Scheduling with NEST: Keeping Tasks Close Together on Warm Cores. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 368–383, 2022.

[22] James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory R Ganger, Garth Gibson, Kimberly Keeton, and Eric Xing. Solving the Straggler Problem with Bounded Staleness. In *14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*, 2013.

[23] Pan Xinhui. Implement vCPU Preempted Check. https://lwn.net/Articles/704904/.

[24] Ashish Panwar, Reto Achermann, Arkaprava Basu, Abhishek Bhattacharjee, K Gopinath, and Jayneel Gandhi. Fast Local Page-tables for Virtualized NUMA Servers with vMitosis. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 194–210, 2021.

[25] Linux Kernel Documentation. CFS Scheduler. https://docs.kernel.org/scheduler/sched-design-CFS.html.

[26] Linux Kernel Document. Scheduler Domains. https://docs.kernel.org/scheduler/sched-domains.html.

[27] Roman Gushchin. [PATCH rfc 0/6] Scheduler BPF. https://lwn.net/ml/linux-kernel/20210916162451.709260-1-guro@fb.com/.

[28] Paul Menage at Linux Kernel Document. Control Groups. https://docs.kernel.org/admin-guide/cgroup-v1/cgroups.html.

[29] Viresh K. Fixing SCHED_IDLE. https://lwn.net/Articles/805317/.

[30] Jonathan C. Per-entity Load Tracking. https://lwn.net/Articles/531853/.

[31] 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.

[32] Xiaoning Ding, Phillip B Gibbons, Michael A Kozuch, and Jianchen Shan. Gleaner: Mitigating the Blocked-Waiter Wakeup Problem for Virtualized Multicore Applications. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 73–84, 2014.

[33] Harshad Kasture and Daniel Sanchez. Tailbench: a Benchmark Suite and Evaluation Methodology for Latency-critical Applications. In *2016 IEEE International Symposium on Workload Characterization*, pages 1–10.

[34] Alexey Kopytov. Sysbench. https://github.com/akopytov/sysbench.

[35] Song L. Introduce cpu.headroom knob to CPU controller. https://lore.kernel.org/lkml/20190408214539.2705660-1-songliubraving@fb.com/.

[36] Linux Kernel Documentation. CFS Bandwidth Control. https://docs.kernel.org/scheduler/sched-bwc.html.

[37] SUSE. Tuning the task scheduler. https://documentation.suse.com/sles/15-SP4/html/SLES-all/cha-tuning-taskscheduler.html.

[38] Steven Rostedt. Using KernelShark to analyze the real-time scheduler. https://lwn.net/Articles/425583/.

[39] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The Linux Scheduler: a Decade of Wasted Cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.

[40] 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, pages 72–81, 2008.

[41] Anthony O Ayodele, Jia Rao, and Terrance E Boult. Performance Measurement and Interference Profiling in Multi-tenant Clouds. In *2015 IEEE International Conference on Cloud Computing*, pages 941–949. IEEE, 2015.

[42] Linux Kernel Documentation. Clock sources, Clock events, sched_clock() and delay timers. https://docs.kernel.org/timers/timekeeping.html.

[43] Georgios Chatzopoulos, Rachid Guerraoui, Tim Harris, and Vasileios Trigonakis. Abstracting multi-core topologies with MCTOP. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 544–559, 2017.

[44] Markus Velten, Robert Schöne, Thomas Ilsche, and Daniel Hackenberg. Memory Performance of AMD EPYC Rome and Intel Cascade Lake SP Server Processors. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, pages 165–175, 2022.

[45] Linux Kernel Documentation. Utilization Clamping. https://docs.kernel.org/scheduler/sched-util-clamp.html.

[46] Jonathan Corbet. Improved response times with latency nice. https://lwn.net/Articles/887842/.

[47] Linux Kernel v6.1.36. Stopper thread. https://elixir.bootlin.com/linux/v6.1.36/source/kernel/stop_machine.c#L384.

[48] Jonathan Corbet at LWN. An EEVDF CPU scheduler for Linux. https://lwn.net/Articles/925371/.

[49] Peter Zijlstra. [RFC][PATCH 00/10] sched/fair: Complete EEVDF. https://lore.kernel.org/lkml/20240405102754.435410987@infradead.org/.

[50] Jonathan Corbet. The extensible scheduler class. https://lwn.net/Articles/922405/.

[51] Red Hat Resource Management Guide. Default Cgroup Hierarchies. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/resource_management_guide/sec-default_cgroup_hierarchies.

[52] Libvirt Virtualization API. virsh. https://www.libvirt.org/index.html.

[53] Will Reese. Nginx: the high-performance web server and reverse proxy. *Linux Journal*, 2008(173):2, 2008.

[54] Jeff Gilchrist. Parallel BZIP2 (PBZIP2). http://compression.great-site.net/pbzip2/?i=1.

[55] Ubuntu Manpage. hackbench - scheduler benchmark/stress test. https://manpages.ubuntu.com/manpages/xenial/man8/hackbench.8.html.

[56] Jens Axboe. fio - Flexible I/O tester. https://fio.readthedocs.io/en/latest/index.html#.

[57] Pradeep Ambati, Íñigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, et al. Providing SLOs for Resource-Harvesting VMs in Cloud Platforms. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 735–751, 2020.

[58] Weijia Song, Christina Delimitrou, Zhiming Shen, Robbert Van Renesse, Hakim Weatherspoon, Lotfi Benmohamed, Frederic De Vaulx, and Charif Mahmoudi. CacheInspector: Reverse Engineering Cache Resources in Public Clouds. *ACM Transactions on Architecture and Code Optimization*, 18(3):1–25, 2021.

[59] Stijn Schildermans, Jianchen Shan, Kris Aerts, Jason Jackrel, and Xiaoning Ding. Virtualization Overhead of Multithreading in X86 State-of-the-Art & Remaining Challenges. *IEEE Transactions on Parallel and Distributed Systems*, 32(10):2557–2570, 2021.

[60] Mathias Gottschlag, Philipp Machauer, Yussuf Khalil, and Frank Bellosa. Fair Scheduling for AVX2 and AVX-512 Workloads. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 745–758, 2021.

[61] Yajuan Peng, Shuang Chen, Yi Zhao, and Zhibin Yu. UFO: The Ultimate QoS-Aware Core Management for Virtualized and Oversubscribed Public Clouds. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1511–1530, 2024.