UFO: The Ultimate QoS-Aware Core Management for Virtualized and Oversubscribed Public Clouds

Yajuan Peng, *Southern University of Science and Technology and Shenzhen Institutes of Advanced Technology, Chinese Academy of Science;* Shuang Chen and Yi Zhao, *Shuhai Lab, Huawei Cloud;* Zhibin Yu, *Shuhai Lab, Huawei Cloud, and Shenzhen Institutes of Advanced Technology, Chinese Academy of Science*

https://www.usenix.org/conference/nsdi24/presentation/peng

This paper is included in the Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation.

April 16–18, 2024 • Santa Clara, CA, USA

978-1-939133-39-7

Open access to the Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation is sponsored by

UFO: The Ultimate QoS-Aware CPU Core Management for Virtualized and Oversubscribed Public Clouds

Yajuan Peng*[1,3], Shuang Chen*[2], Yi Zhao[2], and Zhibin Yu[2,3]

[1]*Southern University of Science and Technology, China*
[2]*Shuhai Lab, Huawei Cloud*
[3]*Shenzhen Institutes of Advanced Technology, Chinese Academy of Science*

Abstract

Public clouds typically adopt (1) multi-tenancy to increase server utilization; (2) virtualization to provide isolation between different tenants; (3) oversubscription of resources to further increase resource efficiency. However, prior work all focuses on optimizing one or two elements, and fails to considerately bring QoS-aware multi-tenancy, virtualization and resource oversubscription together.

We find three challenges when the three elements coexist. First, the double scheduling symptoms are 10× worse with latency-critical (LC) workloads which are comprised of numerous sub-millisecond tasks and are significantly different from conventional batch applications. Second, inner-VM resource contention also exists between threads of the same VM when running LC applications, calling for inner-VM core isolation. Third, no application-level performance metrics can be obtained by the host to guide resource management in realistic public clouds.

To address these challenges, we propose a QoS-aware core manager dubbed UFO to specifically support co-location of multiple LC workloads in virtualized and oversubscribed public cloud environments. UFO solves the three above-mentioned challenges, by (1) coordinating the guest and host CPU cores (vCPU-pCPU coordination), and (2) doing fine-grained inner-VM resource isolation, to push core management in realistic public clouds to the extreme. Compared with the state-of-the-art core manager, it saves up to 50% (average of 22%) of physical cores under the same co-location scenario.

1 Introduction

Motivation. Most cloud data centers operate at very low resource utilization[25, 26, 29, 44]. Other than users over-provisioning their resources [19, 26, 46], resource is especially underutilized in public clouds due to two additional reasons. First, most cloud providers, if not all, provide monthly or yearly subscriptions, which are much cheaper than on-demand pricing[1, 3, 15]. Most users therefore almost never release their resources even when their virtual machines (VMs) are completely idle. Second, there is no explicit label such as online/offline, or high-/low-priority workloads for users' VMs. This forms the black-box nature of public clouds: co-located VMs on the same host machine are equally important, similar to the co-location of multiple LC workloads studied in recent years[22, 48, 51]. This leads to generally lower resource utilization of public clouds than private clouds.

Resource isolation leverages software and hardware mechanisms to partition resources between co-located workloads, and to increase the degree of workload co-location without hurting application's quality of service (QoS)[23, 37, 38]. Resource isolation is especially important for interactive LC workloads such as web search and key-value stores[22, 26, 51]. However, when attempting to apply resource isolation in real public clouds, we find that the state-of-the-art QoS-aware resource managers are (1) *sub-optimal* due to lack of consideration of the virtualization layer and resource over-provisioning, and (2) *inapplicable* to public clouds due to lack of labels, inputs, or feedback for dynamic resource management.

Challenge 1. In virtualized environments, there are two layers of resource management: one in the host OS (e.g., scheduling physical CPU cores, pCPUs, for VMs), and another in the guest OS (e.g., scheduling virtual CPU cores, vCPUs, for users' applications). This is a common problem known as double scheduling, and prior work[36, 41, 55, 58, 62] has proposed co-scheduling or guest-host coordination to improve locks, interrupts, synchronization, preemption, load balancing, etc.

However, we find previous studies are ineffective for LC applications that suffer uniquely from double scheduling. Specifically, if the guest OS is oblivious of the allocated resources on the host, the host OS (rather than guest OS) would experience extensive context switches and high scheduling delay, leading to substantial QoS violations. This is intrinsically related to the sub-millisecond nature of tasks in LC applications. In addition, resource contention also happens between thread groups that belong to the same VM. Such inner-VM contention is especially intense under LC applications.

Challenge 2. Prior work relies on application-level input (such as real-time 99^{th} percentile latency) to guide QoS-aware resource management [22, 23, 43, 44]. This makes two assumptions that do not hold for realistic public clouds. The first assumption is that an application has to monitor its own latency. In reality, many applications do not trace their own

*Equal contribution.

latency, and it is impossible to ask users to provide tracing points to probe their user-level QoS satisfaction degree. The second assumption is that the monitored value can be leveraged by the host OS. This requires interfaces for users to communicate with the cloud provider, to pass application-level information down to components outside the user's VM.

Opportunities. Major cloud providers all provide their own guest OS images [2, 4]. With Cent OS Linux reaching its end of life[5], an increasing number of users are migrating to in-house guest OS images. This leads to a great opportunity for cloud providers to explore guest-host coordination in guest/host kernels, to tackle the two challenges above. By transmitting host's core isolation decision to the guest OS and adjusting core management in the guest accordingly, we observe up to 4.3× throughput improvement under QoS requirements for LC workloads. By leveraging guest's scheduling frequency as a VM's performance indicator, the host OS is able to make QoS-aware core allocation decisions without disturbing users' applications inside the VMs. Furthermore, we observe that core isolation between different thread groups of the same VM leads to an additional 20% of improvement under the same amount of hardware.

Our work. In this paper, we propose UFO, a practical QoS-aware resource manager that targets virtualized public cloud environments, and push core isolation to the next level. Leveraging guest-host coordination, it solves the above-mentioned two challenges that prior work fails or misses to address, and aims to accommodate as many VMs as possible without QoS violations. We make the following contributions:

1. We find that the state-of-the-art QoS-aware core managers neglected the virtualization layer, and made unrealistic assumptions of the interactions between users and the cloud platform. Therefore, they are sub-optimal and inapplicable for virtualized public clouds.
2. We present a comprehensive characterization study to showcase how guest-host coordination and inner-VM isolation can greatly improve resource efficiency in virtualized environments in a black-box manner.
3. We devise UFO, a practical QoS-aware core manager for virtualized and oversubscribed public clouds. UFO does not require user-level QoS input, and instead leverages scheduling frequency from the guest OS as QoS indicators; this makes UFO applicable for public clouds. UFO dynamically adjusts core allocations for each VM on the host and the guest OS concurrently; this makes UFO considerate for virtualization overhead, especially when cores are oversubscribed. Furthermore, UFO dynamically manages an emulator pool to avoid core contention between emulator and vCPU threads.
4. We evaluate UFO under constant and dynamic loads, and show that UFO outperforms prior work by up to 50% (average of 22%) in core saving under the same co-location scenario, and up to 60% (average of 27%) higher input load under the same amount of cores, with negligible overhead.

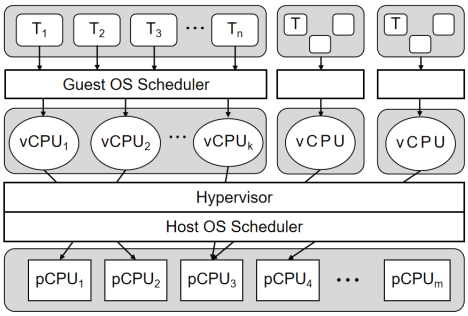


Figure 1: Double scheduling in a typical cloud server. The host OS schedules VMs' vCPU threads to physical cores (pCPUs), while the guest OS of each VM schedules users' application threads to vCPUs.

2 Background and Related Work

2.1 QoS-Aware Resource Management

There are mainly two types of workloads in the public cloud. Latency-critical (LC) workloads have strict quality-of-service (QoS) requirements, usually defined in tail latency. Best-effort (BE) applications are typically throughput-oriented, and are at lower priority than LC applications. QoS-aware resource management refers to techniques that can co-locate various workloads on the same server node [26, 45] while meeting LC workloads' QoS requirement; BE applications can be slowed down or even suspended when needed. Recent work is able to co-locate multiple LC applications on the same node while meeting each LC application's QoS target [22, 43, 48, 51].

However, previous studies have two drawbacks. First, they all rely on application-level performance metrics to guide QoS-aware resource management, and claim this could be achieved by monitoring on the client side, or relying on the application itself to report latency[22, 43]. In public clouds, there is no explicit QoS target defined for LC workloads, and it is impossible to ask users to provide tracing points to probe their user-level QoS satisfaction degree. Instead, cloud providers should find server-side indicators that generally correlate well with QoS, and the cloud providers can legally obtain the indicators without user intervention. A series of work like Arachne [53], Shinjuku [34] and Caladan [28] aim to provide sub-millisecond QoS satisfaction. However, they all rely on deep coordination with users' applications and make *intrusive modifications* to applications or their runtime libraries. Second, the virtualization overhead is significantly overlooked under (most if not) all QoS-aware resource managers.

2.2 Virtualization and Double Scheduling

In public cloud, virtualization is prevalent to ensure user privacy and security. Virtual machines and hosts interact through the hypervisor, and are independent and unaware of each other. Figure 1 shows the current software stack in a typical public cloud. The host OS scheduler schedules the logical CPUs

(vCPUs) of many virtual machines on the physical CPUs (pCPUs) of a server. The users' applications are scheduled by a task-level scheduler (Guest OS scheduler) of VMs, and most operations within the virtual machines are opaque to the hypervisor. Due to the use of multi-core processors and multi-threaded applications, this mutually unaware double-scheduling [57] introduces a semantic gap between hosts and VMs, which leads to problems including lock-holder preemption (LHP) [35, 59], lock-waiter preemption (LWP) [59], blocked-waiter wakeup (BWW) [27], etc. There are two types of solutions to resolve the double scheduling symptom.

2.2.1 Pure-host Solutions

These solutions modify the hypervisor and the host OS to reduce the overhead of virtualization and double scheduling. Adjusting scheduling priority on the host OS is one of the most common approaches, to alleviate LHP and LWP [27, 33], achieve fairness between VMs or load balancing [40, 63], and reduce CPU fragmentation [52, 54]. Advanced hardware supports, such as Intel's Pause Loop Exit (PLE) [41, 56] and AMD's Pause Filter (PF) [47], are also leveraged by the hypervisor and the host OS to detect excessive spinning in the guest OS, to address various preemption problems.

2.2.2 Guest-Host Coordination

These solutions bridge the semantic gap by coordinating the guest and the host OSes, and involves modifications to both the guest and the host OSes.

vCPU ballooning [24, 57] is one of the pioneer works in this line. It leverages CPU hot-(un)plugging to make #vCPUs match #pCPUs, so it completely avoids the double scheduling symptoms. The core allocation is static based on VM priority, making vCPU ballooning fail to adapt to varying load, and fail to handle co-location of LC applications. UFO inherits the same philosophy of vCPU ballooning, to make #vCPUs match #pCPUs. However, UFO targets LC applications, and features in finding the right metric, designing a proper algorithm and an effective controller to support dynamic core adjustment.

eCS [18, 36] annotates critical sections in the guest OS so that the host OS can adjust scheduling priority accordingly. The host states are further transmitted to the guest OS alleviating LWP and BWW problems. PLE-KVM [32] annotates vCPU status in the guest OS, and adjusts vCPU scheduling in the host OS, to mitigate excessive pause-loop-existing (PLE) events and excessive spinning. Similar to vCPU ballooning, eCS and PLE-KVM do not study LC workloads. But since they are open-sourced, we are able to conduct a quantitative comparison with eCS and PLE-KVM in Section 3.6.

3 Characterization

In this section, we characterize three typical LC cloud applications and compare them with BE applications. We illustrate the limitations of current CPU core isolation mechanisms, and show the potential of core management in virtualized and oversubscribed public clouds.

3.1 Methodology

We launch four 8-vCPU VMs on a 16-core host, i.e., 32 vCPUs on 16 pCPUs. This simulates an oversubscription ratio of two.[1] We enable hyper-threading because real public clouds always enable it, so a pCPU is essentially a hyperthread of a physical core. We avoid different VMs sharing the same physical core. Therefore, an 8-vCPU VM can be allocated on 2, 4, 6 or 8 pCPUs. We mainly experiment with three LC applications, Memcached, NGINX, and MySQL. Section 5 includes more details on the evaluated applications, the test bed, and the testing strategy. For simplicity, we run the same workloads in the four VMs such that all the VMs perform homogeneously. This makes core allocation decisions trivial: all the VMs should be allocated with the same number of cores. This allows us to focus on observing the performance differences under various core allocation mechanisms. For each LC application, we experiment with increasing amount of load measured in request-per-second (RPS). Each RPS is tested three times, each lasting for 60s. The geometric mean of the 99^{th} percentile of latency of the four VMs is reported after each test. We then report the average result of the three tests for any given RPS.

3.2 Guest OS Should Coordinate with Host OS

Conventional approaches [22, 44, 51] manage core allocation completely on the host OS. The guest OS is unaware of the host OS; it does not know how many physical cores are actually allocated to the VM. We show that this is insufficient; guest being aware of the host allocations can achieve much higher resource efficiency.

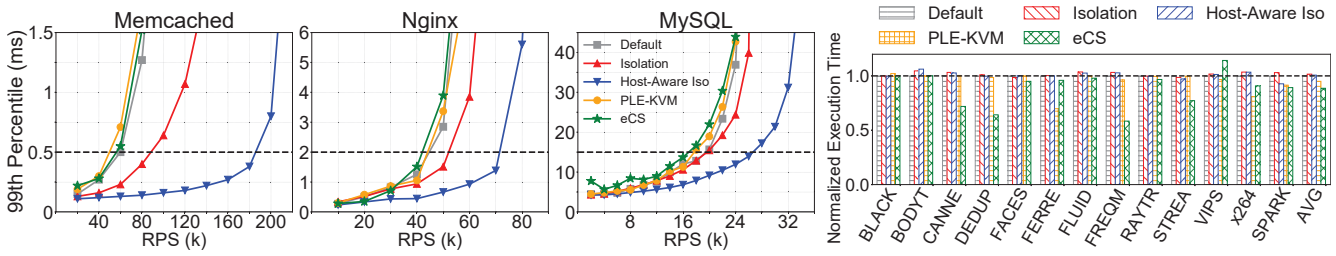
We compare three core allocation mechanisms:

- **Default:** We rely on the scheduling policy of OS to schedule VMs. All the VMs share the same 16 pCPUs, and can be freely scheduled on these pCPUs.
- **Isolation:** We isolate the four VMs, each assigned four pCPUs on the host.
- **Host-Aware Isolation:** On top of *Isolation*, the guest OS is aware that the VM is allocated with only four pCPUs , and schedules jobs to only four vCPUs. We implement this by statically hot-unplugging four vCPUs in the guest OS.

Figure 2 shows how VMs perform when running different workloads under the three mechanisms. We find that under QoS targets (horizontal dotted lines in Figure 2), isolation achieves up to 33% (average of 18%) higher load than default, and host-aware isolation furthers increases the maximum load under QoS by up to 25%-125% than isolation. This is because under host-aware isolation:

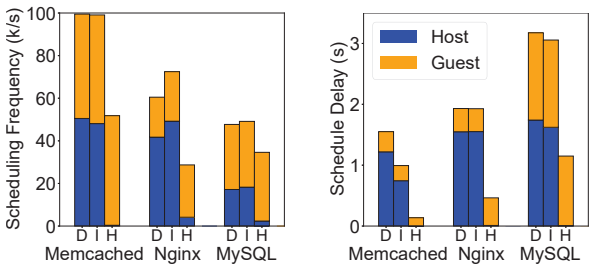
- **Host-side context switches are reduced by 92%.** Figure 3a shows the number of context switches per second (i.e., scheduling frequency) happening in the guest and the host OS. Memcached, NGINX and MySQL run at RPS of

[1] Higher oversubscription ratio usually signals higher resource efficiency, e.g., VMware suggests an oversubscription ratio of at least three [10].



(a) LC **(b) BE**

Figure 2: Performance under five core allocation mechanisms. For LC applications, we show the 99^{th} percentile tail latency with increasing input load (RPS). Horizontal dotted lines represent applications' QoS targets. For BE applications, we show the execution time of each benchmark normalized to that under the *Default* manager. Lower is better.



(a) Scheduling Frequency **(b) Scheduling Delay**

Figure 3: Scheduling frequency and scheduling delay under default (D), isolation (I), and host-aware isolation (H), decomposed into host-side (blue bars) and guest-side (orange bars).

60k, 50k and 20k, respectively, which are the maximum RPS under QoS with the default core manager (see the cross points of gray curves and the horizontal dotted lines in Figure 2a). Because host-aware isolation ensures #vCPUs=#pCPUs, the host OS rarely needs to schedule vCPU threads off a physical core. Note that users' applications are unchanged and the number of application threads does not decrease with fewer vCPUs, so context switches in the guest OS are not reduced. Since a context switch of a vCPU thread on the host involves additional switches of the VMCS structure [55, 60], which is more expensive than a normal context switch of an application thread, we can obtain huge benefits from the reduction of only host-side context switches in virtualized environments.

- **Host-side scheduling delay is reduced by 99%**. Scheduling delay of a process is defined by the time between the process wake-up and its actual running.[2] Figure 3b shows scheduling delay on the host and the guest. As a consequence of reduced context switches, host-side scheduling delay decreases to almost 0 under host-aware isolation. Reduced scheduling delay is critical for LC applications because scheduling delay significantly affects requests' latency. When a request comes in, the application threads inside the guest OS are woken up, leading to schedule events on the guest OS, which may further lead to schedule events on the host. Scheduling delay inside and outside the VM both affect the degree of QoS satisfaction.

[2] We obtain scheduling delay of the guest/host OS by executing `perf sched record` and then `perf sched timehist`, and reporting the sum of scheduling delay of all the processes.

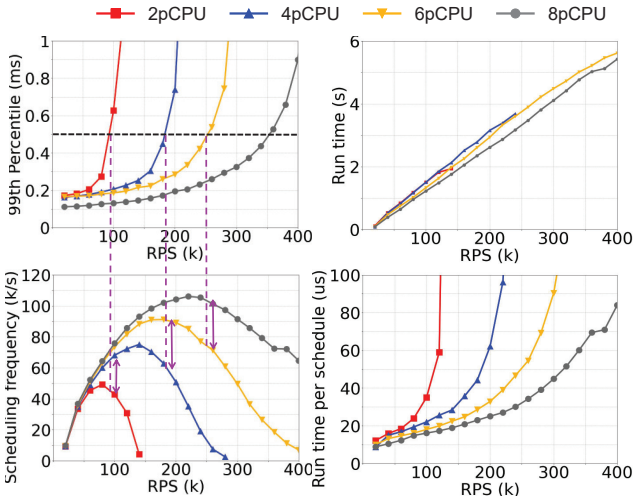


Figure 4: Memcached with increasing input load under various #pCPUs. The top left figure shows Memcached's tail latency, and the horizontal line at $y = 0.5ms$ represents the QoS target. The other three figures show guest-side process runtime, scheduling frequency, and runtime per schedule (division of the previous two metrics).

Appendix A.1 and A.2 include additional analysis on VM exits and caches. In summary, *for LC applications, the guest should be aware of the actual physical core allocations on the host, and dynamically adjust the number of runnable vCPUs.*

3.3 Host OS Should Learn from Guest OS

In public clouds, VMs are black-box to the cloud provider, and application-level performance metrics are hard to obtain. It is therefore difficult to use application-level metrics as indicators to guide resource management under a QoS target in public cloud. This has been an open question in industry for a long time. We successfully solve the open question by identifying a set of great indicators from the guest OS.

- **Guest-side scheduling frequency is a concave function with input load.** As the bottom left subfigure in Figure 4 shows, under a certain #pCPUs, guest-side scheduling fre-

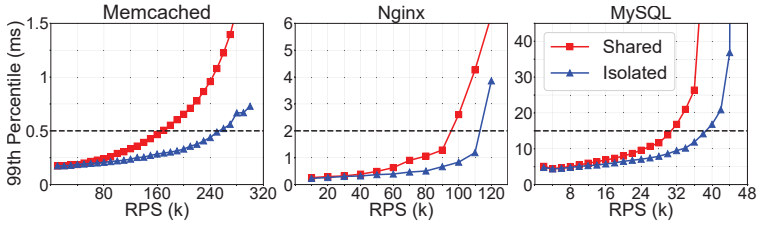


Figure 5: vCPU and emulator threads share 8 pCPUs (Shared), or are isolated with 6 and 2 cores (Isolated, #vCPU is hot-unplugged to 6).

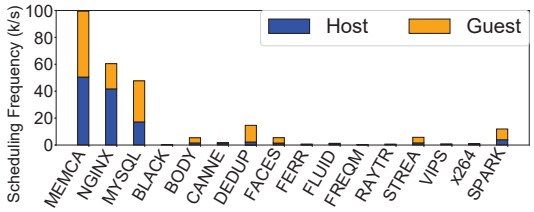
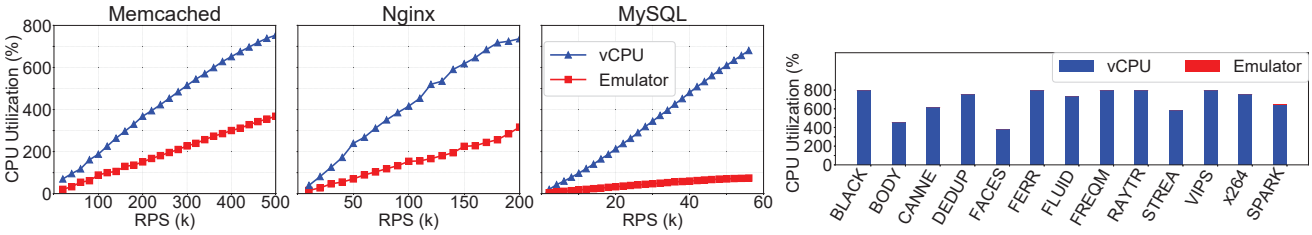


Figure 6: LC applications experience orders of magnitude more scheduling frequency than BE applications.



(a) LC **(b) BE**

Figure 7: CPU utilization of vCPU and emulator threads. BE applications barely use emulator threads.

quency first increases and then decreases along with input load increasing. This is because at low input load (i.e., few requests per second), request inter-arrival time is longer than request processing time, meaning that the application is idle in between requests. As input load increases to a certain extent, new requests come when the processing for old requests has not been finished, and the application threads are active for longer to process consecutive requests. This leads to longer runtime per schedule and fewer context switches. The right two figures in Figure 4 show the total runtime, and the average runtime per schedule, i.e., we divide the total runtime by scheduling count. We see the total time scales linearly with input load. In contrast, the runtime curve per schedule is flat at the beginning, and increases super-linearly after a certain threshold.

- **Guest-side scheduling frequency is a great indicator of application's tail latency.** Comparing the left two figures in Figure 4, we find that the peak point of each curve of scheduling frequency is also the point where application's tail latency starts to increase super linearly with input load. This is because the peak point implies that requests' queuing delay starts to accumulate, which will lead to increased tail latency. Meeting QoS in public clouds essentially means avoiding queuing delay to grow exponentially. This means that we should try staying to the left, or around the peak point of the scheduling frequency curve, and avoid deviating too much to the right of the peak point.
- **Guest-side scheduling frequency can help guide QoS-aware core management decisions on the host OS.** As Figure 4 shows, when *RPS* is less than 100*k*, 2 pCPUs are sufficient to meet QoS. We find that scheduling frequencies under various #pCPUs all overlap. When *RPS* increases to 100k, 4 pCPUs are needed. Guest-side scheduling frequencies under 2 and 4 pCPUs have a 40% difference, while the difference between 4 and 6 pCPUs is only 5%. This motivates us to compare scheduling frequencies between adjacent pCPU counts to obtain the best core allocation.

We show the generality of the findings above to more LC applications in Appendix A.3. It shows that *by comparing guest-side scheduling frequencies between adjacent pCPU counts, we can find the best core management decisions.*

3.4 A Single VM Needs Core Isolation Too

A QEMU-KVM process has two main thread groups, vCPU and emulator threads [31]. Emulator threads are responsible for handling interrupt requests for VM hardware emulation [17]. We find the two thread groups have different core demands, and interfere with each other when sharing cores.

Figure 5 shows how each application in a VM performs when its vCPU and emulator threads (1) share the 8 pCPUs (Shared), or (2) partition the 8 pCPUs into 6 and 2 cores (Isolated), and adopt host-aware isolation in the vCPU core group. We find that:

- Compared with Shared, Isolated achieves 15%-50% higher RPS under the same core count while meeting QoS.
- Emulator threads of LC applications are quite active. Figure 7 shows that CPU utilization of emulator threads is 15%-50% of that of vCPU threads. This is because LC applications are naturally networked applications; requests and responses are all transmitted through network, and part of these network operations are handled by emulator threads. Since request processing time is at the scale of tens of microseconds to a few milliseconds, the network interrupt processing time is a non-negligible part of the end-to-end request latency of LC applications.
- CPU utilization of both vCPU and emulator threads increase linearly with input load.

The observations suggest that *(1) CPU utilization of either vCPU or emulator threads is a great indicator of application's input load, and (2) core allocation of both vCPU and emulator threads should be dynamically adjusted based on input load.*

3.5 How about BE applications?

We have clarified three needs for LC applications: (1) host-aware isolation, (2) monitoring of guest-side scheduling frequency, and (3) isolation between vCPU and emulator threads. However, the three techniques are not necessary for BE applications. In this section, we reveal the fundamental differences between LC and BE applications that lead to the differences in the effectiveness of the three techniques.

We experiment with 12 BE applications from the PARSEC benchmark suite [21] (each benchmark is abbreviated by its first five characters), and 99 queries from TPC-DS [16] deployed in Spark [20]. The spark cluster consists of three 8-vCPU VMs, one master VM and two slave VMs. We report the total execution time of all the 99 queries in Figure 2b, and the average statistics of the three VMs in Figure 6 and 7b.

First, *LC applications are fundamentally different from BE applications in that LC applications are comprised of a number of short requests.* Request latency of LC applications is at sub-millisecond to millisecond granularity (Figure 2a). A thread only runs for tens of microseconds before being scheduled out (Figure 4). On the contrary, BE applications typically consist of fewer and longer tasks. When scheduled, a thread runs for the entire timeslice before being scheduled out. This difference is reflected in context switches that have orders of magnitude difference (Figure 6). Section 3.2 shows that host-aware isolation mainly reduces host-side context switches, which consequently reduces host-side scheduling delay and HLT VM exit handled time. Since BE applications do not suffer from host-side context switches like LC ones, they do not benefit much from host-aware isolation. Figure 2b shows that host-aware isolation reduces execution time by merely 0.7% on average compared with the default core manager. Spark achieves the most reduction (i.e., 7%) because of its relatively higher host-side scheduling frequency (Figure 6).

Second, BE applications have almost no usage of the emulator threads. As shown in Figure 7b, for BE applications, the average CPU utilization of emulator threads is less than 1%. Utilization of each query in Spark is detailed in Appendix A.4. As a result, it is not necessary to separate vCPU and emulator threads for BE applications.

3.6 Comparison with Prior Work

Section 2.2.2 mentioned that prior work on guest-host coordination evaluates only BE workloads. We study the effectiveness of PLE-KVM [32] and eCS [36] in Figure 2, and find that they indeed perform well for BE applications. PLE-KVM and eCS reduce execution time by up to 30% and 42% (average of 5% and 12%), respectively. However, they behave similarly as the default mechanism for LC applications, and significantly under-perform host-aware isolation. This is because PLE-KVM and eCS do not attempt to reduce host-side context-switches or scheduling delay, a severe issue when running LC applications. Also, we find that when prioritizing certain vCPU thread, PLE-KVM and eCS sometimes introduce significant unfairness between vCPUs and VMs.

3.7 Summary

In summary, we make the following key observations:

- Partitioning cores on the host OS is insufficient. The guest OS should be aware of the core allocation decision made by the host, and adjust core management in the guest OS as well such that the number of usable vCPU count is no more than the allocated pCPU count for the VM.
- Partitioning resources between different VMs is insufficient. Threads belonging to the same VM contend for resources as well; vCPU and emulator threads should be isolated within the same VM as well.
- Without application-level performance metrics, the guest OS can still provide useful hints like scheduling frequency and scheduling delay to guide core management on the host (and the guest), meeting application-level QoS targets.
- Prior work considers only BE applications, and fails to unlock the full potential of guest-coordination for LC applications.

4 UFO Design

UFO is a feedback-based controller that dynamically manages core allocation for virtualized public clouds, aiming to accommodate more VMs on a single host without violating applications' QoS. In this section, we describe UFO in detail.

4.1 Design Principles

UFO is designed with three design principles in mind:

- **Prioritize for LC applications:** Our primary goal is to meet QoS for LC applications. In public clouds, there is no user-specified QoS target. So our goal is to avoid LC applications suffering from extensive queuing delay. Since prior works on guest-host coordination all focus on conventional throughput-oriented batch jobs [32, 36, 57], UFO focuses on LC applications. It simply lets all BE applications share the idle pool, and does not explicitly handle core allocation for each BE application.
- **Optimize for virtualized and oversubscribed environments:** Virtualization and oversubscription are ubiquitous in public clouds. This means that double scheduling always exists, and VMs usually do not get as many physical cores as their vCPU size. We tackle the challenges brought by virtualization and oversubscription by guest-host coordination and vCPU-emulator isolation.
- **Focus on core management:** Despite many other shared hardware resources, UFO is currently designed for only

Algorithm 1: UFO's main function.

```
while TRUE do
    monitor for 1s;
    // Adjust the emulator pool first.
    if emuUtil <50% then
        remove cores from the emulator pool as long as
          emuUtil is under 70%;
    else
        if emuUtil >70% then
            assign more cores to the emulator pool to make
              emuUtil under 50%;
            continue;
    // Adjust the vCPU pool for each VM
    for i = 1..N do
        // Record the sample; fit model upon sufficient
           samples; adjust model upon inaccuracy detection
        updateModel(i, pCPU[i], cpuUtil[i], schedFreq[i]);
        // Find the best core count
        p = predict(i, pCPU[i], cpuUtil[i]);
        if p ≠ pCPU[i] then
            // Adjust core allocations on the guest& host
            modify(i, p);
            sleep 3s;
            pCPU[i] = p;
```

core management. This is because other resources such as last-level cache and memory bandwidth do not require special treatment in virtualized and oversubscribed environments, and there are many prior work that specifically targets management of these resources [38, 50, 61, 64].

4.2 Resource Pool

UFO splits all the physical cores on each host into three pools:

- **vCPU Pool** consists of N core groups for N running VMs, and one group is assigned for one VM's vCPU threads. For VM i, UFO dynamically expands or shrinks its pCPU cores ($pCPU[i]$) based on UFO's core predictor (Section 4.5).
- **Emulator Pool** consists of cores allocated for running VMs' emulator threads. All the VMs share the emulator pool. The emulator pool dynamically expands or shrinks based on its CPU utilization (Section 4.3).
- **Idle Pool** consists of all the leftover cores on the host. It provides the source of free cores. An increasingly larger idle pool signals capability to host more VMs, while a shrinking pool signals system overload (Section B.4). The vCPU and emulator pools interact with the idle pool constantly. We introduce how cores flow between pools below. Note that idle VMs (i.e., vCPU utilization is consistently below 1%) are also placed in the Idle Pool. When UFO detects an increase in vCPU utilization, it will move the active VM to the vCPU pool.

4.3 UFO Controller

As shown in Algorithm 1, UFO consists of three stages:

1. **Monitor:** UFO monitors the CPU utilization of the emulator pool on the host OS, and the CPU utilization and scheduling frequency on each guest OS every second.
2. **Adjust the emulator pool:** UFO always makes sure the emulator pool is large enough to satisfy the need of emulator threads. This is achieved by keeping CPU utilization of the emulator pool under 70%. Upon detection of low utilization (i.e., $emuUtil < 50\%$) of the emulator pool, more cores will be moved from the emulator pool to the idle pool. The thresholds can all be adjusted based on practical needs. UFO checks and adjusts the emulator pool before the vCPU pool. This is because the resource demand of emulator threads is only a fraction of the demand of vCPU threads. However, if emulator threads do not get enough cores, vCPU threads will have to be assigned more cores to meet application's performance target (Section 3.4), leading to lower resource efficiency.
3. **Adjust the vCPU pool:** Figure 8 shows the system components for vCPU adjustment in UFO. CPU utilization and scheduling frequency are continuously monitored inside each guest OS, and are fed to the collector in the host OS. The collector in the host OS is responsible for collecting data samples and fitting models upon enough samples ($updateModel()$, Section 4.4, detailed algorithm in Appendix B.1). Then, UFO calculates the best core allocation through the core predictor ($predict()$, Section 4.5, detailed algorithm in Appendix B.2). If differing from the current core allocation, UFO will adjust core allocations through the enforcer components on the host and the guest OS ($modify()$, Section 4.6).

UFO is monitored and scheduled at a fine-grained level of seconds. Refer to Appendix B.3 for more details.

4.4 Modeling of Scheduling Frequency

As discussed in Section 3.3, guest-side scheduling frequency is a great indicator of application's QoS level, and can be used to guide core management decisions. However, it is impractical to obtain the entire curve of scheduling frequency under any input load and any pCPU count. Therefore, UFO leverages VM's CPU utilization as a proxy of application's input load, and builds a model to predict guest-side scheduling frequency under any CPU utilization given a number of (guest-side CPU utilization, guest-side scheduling frequency) pairs.

Figure 10a shows an example of the fitted curves of Nginx's guest-side scheduling delay under eight pCPUs. We find quadratic function fits the bests, represented by $f(u) = Au^2 + Bu + C$ where u is CPU utilization ranging from $[0, 100 * \#pCPU]$. R^2 [6] is a goodness-of-fit measure for predictions. R^2 closer to 1 represents higher accuracy.

The collector in the host OS continuously collects guest-side scheduling frequency and CPU utilization. As each VM

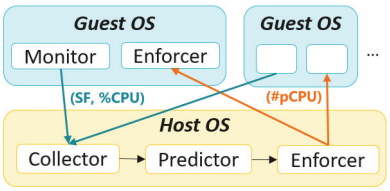
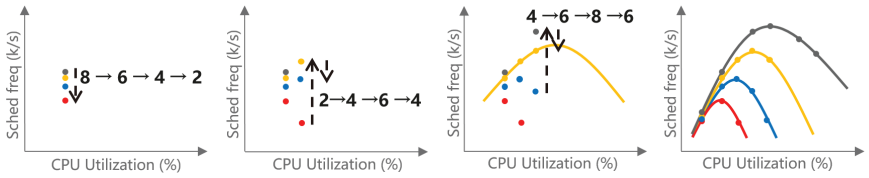
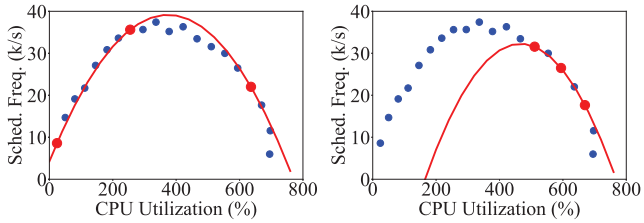


Figure 8: Guest-host coordination in UFO. Monitored events from the guest OSes are fed to the host OS for core prediction. Core adjustment is enforced in the guest and host at the same time.



(a) Initial stage. **(b) Collection stage.** **(c) Fitting stage.** **(d) Full profile.**

Figure 9: Four stages in UFO's vCPU adjustment. (a) and (b) are early stages with few samples, so UFO has to try various core counts. (c) starts model fitting as long as three samples are accumulated for a certain core count. (d) obtains more samples under each core count, fitting more curves and adjusting previous fittings. The best core count can be directly predicted in this stage.



(a) Good samples: $R^2 = 0.962$ **(b) Bad samples:** $R^2 = -0.029$

Figure 10: Fitted curves of Nginx's guest-side scheduling delay under eight pCPUs. All the samples are marked blue, while the samples used for the fitting are marked red. Both fitted curves use three samples. The left figure has samples spanning both low and high CPU utilization, while the right figure is fitted with samples only from high CPU utilization.

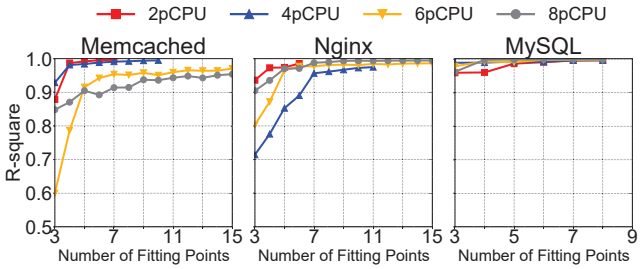


Figure 11: Model accuracy (R^2) of scheduling frequency with increasing number of fitting points. Seven points are enough to reach $R^2 > 0.95$ in most cases.

runs, more samples under various core counts and various CPU utilization are collected. For any given pCPU count, when sufficient samples are collected (e.g., Figure 9c), the curve of scheduling frequency with CPU utilization can be fitted. Ideally, three points determine a quadratic function. However, due to data instability, we find three may not be enough generally. Figure 10 shows a good and a bad case using three fittings points. In Figure 10a, the three samples span both sides of the concave function, and we can successfully fit a model with $R^2 > 0.95$. However, if the three samples all fall into the same side of the curve as shown in Figure 10b, the fitted model can be totally inaccurate. In general, more samples lead to higher accuracy. Practically, we find eight samples to be more than enough for a fitted model to achieve $R^2 > 0.9$, shown in Figure 11. Therefore, UFO maintains the last eight samples under each core count for each VM.

For a given pCPU count, when three samples are recorded, UFO starts fitting the model, though the model may be inaccurate due to skewed samples. The inaccuracy will eventually come to light as the VM continuously runs. Upon detection of inaccuracy (i.e., the difference between the fitted value and the true value is more than 5%), the model will be refitted using the latest and more samples. Storing the latest eight samples and refitting models upon detection of inaccuracy allow UFO to adjust to workload churn inside the VM. For instance, if a VM previously runs Memcached and later runs Nginx, the models will all be updated with the latest samples.

4.5 Core Adjustment

The goal of UFO's core predictor on the host OS is to find the best core allocation for each VM. After the best core count is predicted, the enforcer components on the host and the guest OS will adjust pCPU count for the VM.

As discussed in Section 3.3, c pCPUs is the best core allocation if and only if:

- Scheduling frequency (SF) does not increase much under $c+2$ cores, i.e., $\frac{SF[c+2]-SF[c]}{SF[c+2]} < x$.
- Scheduling frequency drops significantly under $c-2$ cores, i.e., $\frac{SF[c]-SF[c-2]}{SF[c]} > x$.

In this paper, we set the threshold x to 30%. Figure 4 shows why. The vertical lines on the left pictures show the maximum RPS that each pCPU count can sustain while meeting QoS, which are 98k/180k/260k under 2/4/6 pCPUs. The vertical lines run through the figure of guest-side scheduling frequency, such that we can compare scheduling frequency of different #pCPUs under these input loads. We find the difference in scheduling frequency between 2&4/4&6/6&8 pCPUs under 98k/180k/260k RPS is 30%/29%/27% (double-headed arrows in Figure 4). Considering other applications (elaborated in Appendix A.3), we find 30% to be a practically good threshold that universally works for most LC applications.

In reality, UFO will go through four stages in the core predictor, shown in Figure 9. Initially, each VM is given #vCPU pCPUs (8 in the example). Without any model fitting, UFO

has to iteratively try fewer pCPUs, obtain and record SF, check if SF decreases by more than 30%, and finally find the best core count (2 in Figure 9a). When the input load increases in Figure 9b, UFO has to iteratively try increasing #pCPUs until SF increases by less than 30%, and reverts back to last #pCPUs (2->4->6->4 in Figure 9b). To reduce these trials, UFO adopts some heuristics based on CPU utilization when models are not completely fitted, to keep per-core CPU utilization (PCU) roughly within 40%-80%. Suppose current pCPU count is p and current PCU is u. This means that UFO will stop reducing cores if $u * \frac{p}{p-2} > 80\%$, and will stop increasing cores if $u * \frac{p}{p+2} < 40\%$. As the VM runs, more samples are collected, and UFO will start fitting curves when 3 samples are accumulated under a certain pCPU count (Figure 9c), so that UFO can directly infer SF under this pCPU count next time. UFO fits more curves and continuously adjusts the fitting with the latest samples. When the full profile is obtained (Figure 9d), UFO can directly predict the best count without any trial.

4.6 UFO Implementation

Sharing information between guests and the host. Similar to the `kvm_steal_time` implementation [14, 36], when launching a new VM, the guest OS allocates a 32B read-write memory segment that is shared between the guest and the host OS. After the memory allocation, the guest OS sends the memory address of the memory segment to the host OS through a hypercall. The host OS handles the hypercall from each guest OS, recording the memory address.

This segment consists of three 8B integers: two integers are written by the guest-side kernel module, including the latest guest-side CPU utilization and scheduling frequency; another integer is written by the host-side kernel module, denoting the number of pCPUs allocated for this VM.

Guest-side kernel module. A kernel module is installed on each guest OS, recording the guest-side CPU utilization and scheduling frequency to the shared memory every second. It also monitors if #pCPUs in the shared memory changes. If so, it will offline/online some vCPUs in the guest OS.

Host-side kernel module. A kernel module is also installed on the host OS. It periodically does the following tasks: it first reads all the guest-side CPU utilization and scheduling frequency samples, and maintains the latest 8 samples under each pCPU count for each VM. It then fits or adjusts the model, and predicts the pCPU count for each VM. Finally, it writes the prediction result to the shared memory, and adjusts the core allocation of each VM on the host OS accordingly by executing `virsh vcpupin`.

Host-side emulator management. There is another script running continuously on the host OS. This script reads the aggregate CPU utilization of emulator threads of all the VMs every second, and adjusts the emulator pool using `virsh emulatorpin`.

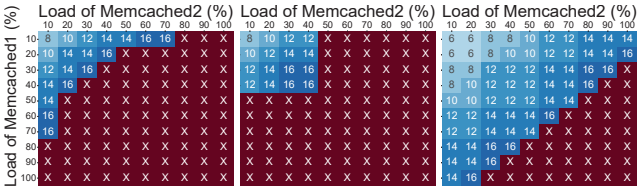


Figure 12: Colocation of 2 Memcached VMs. Each cell represents the minimum required #pCPUs to meet QoS when each Memcached run at the fraction of their *max loads* indicated in the x and y axes. A brown cell with cross mark represents unreachable load combination, meaning that 16 #pCPUs are still insufficient to meat QoS target of both Memcached instances.

5 Experimental Setup

5.1 Hardware Platform

Table 1 shows the specifications of our experimented platform. Both Hyperthreading and turbo boost are enabled to emulate real cloud setups. Since UFO is an intra-node manager, we only experiment with one server node. Clients (i.e., load generators) run on another Intel Xeon machine, with a 10Gbps network link to the server node.

5.2 Applications

We follow testing strategies from prior work [22] to set up our applications, load generators, constant and dynamic load experiments. Table 2 shows the details of our experimented LC applications and the testing strategy. These applications are widely used in industry, and cover different application domains. All the load generators use the default exponential inter-arrival time distributions to simulate a Poisson process, where requests are independent with each other [42]. This means that even under a given constant request-per-second, requests are not generated uniformly. Load still fluctuates at sub-second granularity as shown in Figure 20.

We set the QoS target of each application as the 99^{th} percentile latency of the knee of the latency-with-RPS curve, as marked in Figure 2a and Figure 5, and listed in Table 2 [22]. We define *max load* of each application (also listed in Table 2) as the maximum RPS under the QoS target when 8 vCPUs exclusively run on 8 pCPUs, without emulator interference (emulator threads are allocated on another 8 pCPUs). Note that this means the actual resource need to reach 100% of max load under QoS is more than 8 pCPUs.

We sweep the input load in 10% increments from 10% to 100% of the max load, and record the number of total pCPUs allocated while meeting QoS targets.

5.3 Baselines

We compare UFO with two baselines:

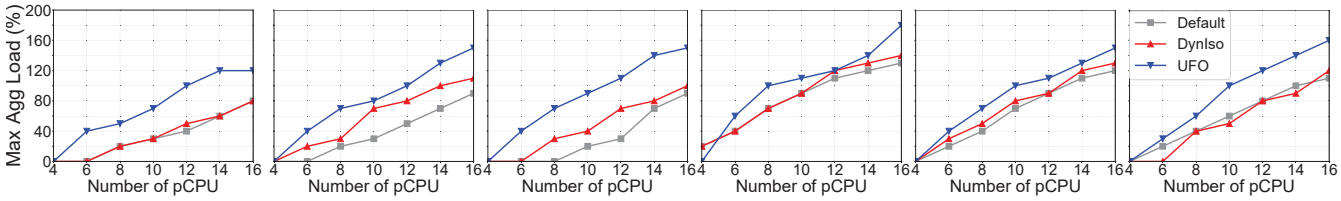
- **Default:** We rely on the OS to schedule VMs. For a given colocation scenario, we increase the number of pCPUs until all the colocated VMs meet QoS target. This is an ideal version of the default manager. Default does not isolate between vCPU and emulator threads.

Table 1: Platform Specification

Model	Intel Xeon Platinum 8378
Microarchitecture	Icelake
Cores/Socket	32 (2 sockets in total)
Threads/Core	2
Hyperthreading	Enabled
Turbo Boost	Enabled
Virtualization Technology	QEMU-KVM
Host and Guest Kernel	5.10
VM Size	8 vCPU, 16 GB memory

Table 2: Latency-critical applications.

Application	**Memcached [11]**	**Nginx [13]**	**MySQL [12]**
Domain	Key-value store	Web server	Database
QoS Target	0.5ms	2ms	15ms
Max Load under QoS	350k	120k	50k
Load Generator	Mutated [7]	wrk2 [9]	sysbench [8]
Dataset	One million <key,value> pairs	10,000 html files of 4KB each	20 tables, each with one million entries
Request Type	100% GET requests	Get file content	OLTP transactions, each with 18 select and 2 update queries



(a) Mem$+Mem$ **(b) Mem$+Nginx** **(c) Mem$+MySQL** **(d) Nginx+Nginx** **(e) Nginx+MySQL** **(f) MySQL+MySQL**

Figure 13: Colocation of 2 VMs. A+B means that VM_1 and VM_2 run application A and B, respectively. The y-axis represents the maximum aggregated load that each core manager is able to sustain while meeting QoS of both VMs under a certain #pCPU.

- **DynIso:** Similar to the Isolation mechanism in Section 3.2, this mechanism does core isolation of vCPU threads between VMs. Since the input load may vary with time, we leverage the feedback-control loop like PARTIES [22] to dynamically decide #pCPUs for each VM, by comparing the application's tail latency with the QoS target. It is an ideal version of the Isolation mechanism. DynIso does not isolate between vCPU and emulator threads.

6 Results and Analysis

In this section, we first evaluate UFO with constant load, and then with dynamic load. We also decompose UFO into vCPU management only and isolation of emulator threads, to show the effectiveness of each component in UFO. Finally, we quantify the overhead of UFO.

6.1 Constant Load

Figure 12 shows colocations of 2 VMs, both running Memcached, under the three core managers. The x and y axes denote the load of Memcached run at the fraction of *max loads*, and the values in the heatmap denote the minimum number of pCPUs required to meet both VMs' QoS targets. Smaller values (or lighter colors) indicate higher resource efficiency. The brown cells with "x" marked indicate that 16 pCPUs are still insufficient to meet QoS Target of both Memcached instances under certain load combinations.We find that:

- DynIso is superior to Default when both VMs run at medium loads, e.g., 40% of max load. This is because DynIso physically separates the two VMs, and is able to eliminate interference at a higher extent.
- Default outperforms DynIso when VMs' loads are imbalanced for two reasons. First, Default manages resources at the granularity of timeslices (i.e., tens of milliseconds). It allows a VM to utilize only a fraction of a pCPU under very low load, and leave the rest of the pCPU to other colocated VMs. Second, DynISO has the restriction to allocate at most #vCPU pCPUs to a VM, while Default could potentially assign more than this amount when any VM is under very high load. For example, when the two Memcached VMs run at 70% and 10% of their max loads, VM2 requires only 2 pCPUs to meet QoS, so the remaining 14 pCPUs cores are all used by VM1, including 8 pCPUs for VM1's vCPU threads, and 4 pCPUs for its emulator threads.
- UFO significantly outperforms Default and DynIso. The number of blue cells is 64 under UFO, meaning that UFO is able to meet QoS of both Memcached instances under 64% of all the load combinations, while Default and DynIso can only achieve 19% and 16%, respectively.

For easier quantitative comparison, for each pCPU count, we record the maximum aggregated load (MAL) that each core manager is able to sustain while meeting QoS of both VMs. Figure 13 plots MAL of six 2-VM co-location mixes (Appendix C.1 shows all the heatmaps). Comparing with DynISO, UFO achieves up to 60% (average of 27%) higher MAL under the same number of pCPUs. On the other hand, under the same input load of each VM, UFO saves up to 50% (average of 22%) cores than DynISO.

In summary, UFO is able to achieve higher MAL under the same pCPU count, and save pCPUs under the same load.

6.2 Dynamic Load

In this section, we evaluate how UFO reacts to various load fluctuation patterns. Figure 14 shows a 3-VM co-location mix, each VM running a different application with a different fluctuation pattern. We assume no prior samples and no previously fitted model in UFO in this experiment. In order to focus

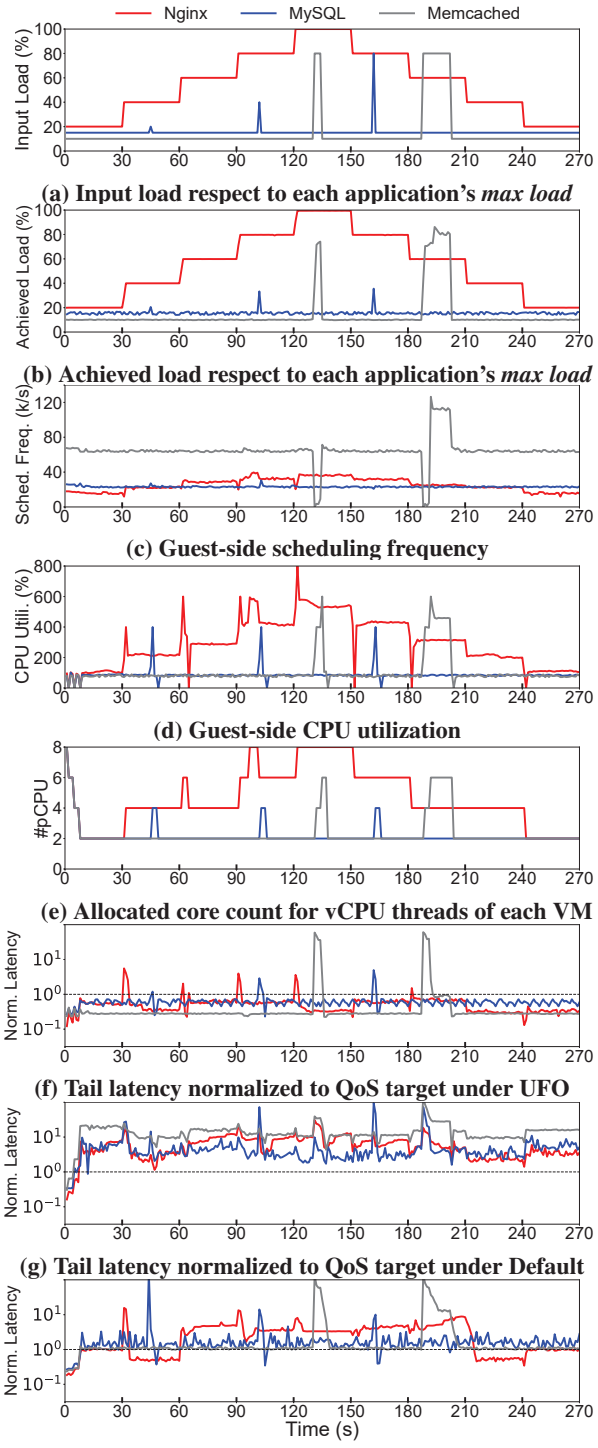


Figure 14: UFO's reactions to fluctuating load patterns. Horizontal dotted lines in Figure 14f, 14g and 14h represent QoS targets.

on evaluating the effect of dynamic core adjustment for vCPU threads among different core managers, we exclude the interference of emulator threads by also applying vCPU-emulator isolation to Default and DynIso, and keep the emulator pool large enough. To compare tail latency, we keep #pCPUs the same on the host OS under all the core managers.

Initially, each VM is given 8 pCPUs. UFO detects low per-core CPU utilization (PCU), and gradually reduces #pCPU. After each adjustment, UFO checks (1) if PCU is higher than 80%, and (2) if guest-side scheduling frequency decreases by more than 30%. Due to the super low load, UFO successfully reduces #pCPU to 6, 4, and 2 at 2s, 6s, and 9s, respectively. QoS is consistently satisfied in this phase. We notice that every time vCPU offlining is performed, CPU utilization of the corresponding VM suddenly drops to 0, then recovers in 1-3s (see the first 10s in Figure 14d).

We also compare UFO with Default and DynIso without guest-host coordination. Default and DynIso are both unable to meet QoS if allocating the same number of cores as UFO.

6.2.1 Diurnal Load Fluctuations

Cloud applications typically have diurnal load fluctuations [30], i.e., load gradually increases during the day, and decreases during the night. Nginx (red lines in Figure 14) mimics such a pattern. The load gradually increases from 20% to 100% in units of 20%, and then drops back to 20%, each step lasting for 30s, and the total duration is 270s.

At 30s, Nginx's load increases and is reflected in an increase in CPU utilization to 184% (PCU of 92%). UFO increases its #pCPUs to 4 and waits for 3s. At 34s, UFO finds that increasing #pCPUs to 6 will lead to PCU lower than 40% (55%*4/6=37%), so it stops increasing #pCPUs further. In this phase, QoS violation only lasts for 3s (Figure 14f) because it takes 1s for UFO to observe load surge and increase #pCPUs, and it takes an additional 2s for system to stabilize and for tail latency to recover. Note that every time vCPU is hot-plugged, CPU utilization bursts and then recovers in 1-3s.

At 60s, UFO detects increase in CPU utilization, and increases #pCPUs to 6. However, after waiting for 3s, UFO finds that guest-side scheduling frequency increases by only 21%, less than the 30% threshold. This means that 6 #pCPUs is too much. Therefore, UFO reverts #pCPUs back to 4 at 64s.

Similarly, UFO increases #pCPUs at 90s and 120s, and successfully resolves the QoS violations of Nginx in 1-3s. At 150s, UFO detects load drop. Since UFO has experienced the same load before, it quickly decreases #pCPUs to the right count without any reversion of #pCPUs.

In summary, UFO is capable of handling diurnal load changes. It reacts to one second after any load change is detected, and performs better as more samples are collected. On the contrary, Default and DynIso consistently violate QoS under the same core count.

6.2.2 Sub-Second Load Bursts

Load bursts refer to a sudden load increase, followed by a load drop. We test two types of bursts that last for varying amounts of time, to test UFO's responsiveness to bursty loads.

We follow testing strategy in Shenango [49] to test sub-second load bursts. The load of MySQL (blue lines in Figure 14a) suddenly changes from the default 15% to 20%,

40% and 80% at 45s, 100s and 160s, respectively. Each burst lasts for only 1s. UFO is able to observe the load burst, and it increases #pCPUs accordingly. However, soon after UFO reacts, the burst has disappeared and application's tail latency recovers by itself. In fact, there is no need for UFO to take reaction. Also, because MySQL is assigned 2 pCPUs initially, if the load burst exceeds the processing capability of 2 pCPUs (i.e., 30% in this case), requests would be dropped, showing in the difference between input load and achieved load in Figure 14a and 14b. Note that latency suffers more under Default (Figure 14g) and DynIso (Figure 14h) despite the reaction time. This shows the effectiveness of UFO's guest-host coordination.

UFO is designed to operate at second-granularity like prior work [22, 44]. Recent studies [28, 34, 49, 53] show the ability to resolve microsecond-scale QoS violations. However, as discussed in Section 2.1, they have to intrude users' applications, and are applicable only in private clouds. Achieving sub-second QoS in public clouds remains an open problem.

6.2.3 Bursts with Increasing Duration

We then increase the burst duration to 5s and 15s. As the grey lines in Figure 14a show, Memcached bursts from 10% to 80% at 130-135s, and 187-202s. Note that this vast degree of load burst is not realistic; loads usually gradually increase or decrease with small spikes [30]. This stress test aims to show the boundary of UFO's responsiveness.

When bursting to 80% of load, Memcached in fact needs 6 pCPUs to meet QoS. Since UFO adjusts 2 pCPUs at a time (with no prior samples and model fitting), it takes two steps to reach the desired core count. At 130s, UFO detects load increase and increases #pCPUs to 4 and 6 at 131s and 134s, respectively, reaching the desired core count in 4s. The second spike lasts longer for 15s. UFO still reaches the desired core count after two steps, and resolves the QoS violation in 4s. This shows that the responsiveness of UFO depends on the number of steps for UFO to adjust. Upon vast load increases, the longer the burst, the better UFO handles.

6.3 Decomposition of UFO

Figure 15 shows the effectiveness of each component in UFO. We experiment with 16-vCPU VMs to show more #pCPU options. UFO-vCPU does not separate emulator and vCPU threads, while UFO-emulator does not online/offline vCPUs to match pCPU count. We find that:

- Compared with the *max load* under Default, UFO-vCPU and UFO-combined are able to achieve an average of 19.8% and 69.8% higher load under QoS, respectively.
- UFO-emulator sometimes even underperforms Default. This is because after reserving some cores for emulator threads, vCPUs are stacked onto fewer pCPUs, causing severe double scheduling symptoms. This is especially detrimental for VMs with high emulator activity. Therefore, it is important to apply vCPU-emulator separation and vCPU-pCPU coordination at the same time.

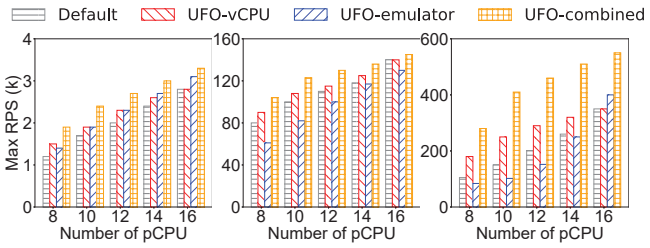


Figure 15: Decomposition of UFO's effectiveness.

6.4 UFO Overhead

UFO mainly involves the following overhead:

- **Guest-side monitoring**: We compare CPU and memory resource utilization, and application's latency with and without guest-side monitoring. The CPU and memory resource consumption is barely observable, and application's tail latency remains the same.
- **Host-side kernel module**: The host OS is responsible for data collection and core prediction. This kernel module takes less than 5% of CPU utilization on average, since it is only active for a few tens of milliseconds every second. Memory consumption scales linearly with the number of VMs and the total number of pCPUs on the host. Assuming a maximum of 50 VMs and 200 pCPUs on the host, memory usage is less than 2MB.
- **Time to online or offline vCPUs in the guest OS**: It takes 20-30ms to online or offline a single vCPU for LC applications. This overhead increases almost linearly (i.e., slightly sublinear) with the number of vCPUs to online or offline (more details in Appendix C.2). Prior work [24] shows that the overhead can be reduced to tens of microseconds per vCPU. We hope similar techniques could be merged to mainline of Linux kernel in the future.
- **Performance impact of onlining/offlining vCPUs in the guest OS**: As shown in Figure 14, onlining vCPUs cause a sudden burst in guest-side CPU utilization, and offlining vCPUs cause a sudden decrease in CPU utilization down to 0. It takes up to 3s for CPU utilization to recover. This explains why we wait for 3s after each vCPU adjustment.

7 Conclusion

We have presented UFO, a core manager for latency-critical applications in virtualized and oversubscribed public clouds. UFO leverages guest-host coordination and inner-VM core isolation, to push core management to the extreme. UFO outperforms state-of-the-art core managers by up to 50% in core saving under the same colocation scenario. It also increases the aggregate system load by up to 60% under the same amount of core resources while still ensuring QoS.

References

[1] Alibaba cloud elastic compute service. https://www.alibabacloud.com/product/ecs.

[2] Alibaba cloud linux. https://alibaba.github.io/cloud-kernel/os.html.

[3] Amazon ec2. https://aws.amazon.com/ec2/.

[4] Amazon linux. https://github.com/amazonlinux/amazon-linux-2023.

[5] Centos end of life date. https://endoflife.date/centos.

[6] Coefficient of determination. https://en.wikipedia.org/wiki/Coefficient_of_determination.

[7] Github page of mutated load generator. https://github.com/scslab/mutated.

[8] Github page of sysbench load generator. https://github.com/akopytov/sysbench.

[9] Github page of wrk2 load generator. https://github.com/sc2682cornell/wrk2.

[10] Guidelines for overcommitting vmware resources.

[11] Memcached official website. https://memcached.org/.

[12] Mysql official website. https://www.mysql.com/.

[13] Nginx official website. https://www.nginx.com/.

[14] Steal time for kvm. https://lwn.net/Articles/449657/.

[15] Tecent cloud virtual machine. https://cloud.tencent.com/product/cvm.

[16] Tpc-ds homepage. https://www.tpc.org/tpcds/.

[17] Using virsh emulatorpin in virtual environments with nfv. https://access.redhat.com/documentation/en-us/red_hat_openstack_platform/10/html/ovs-dpdk_end_to_end_troubleshooting_guide/using_virsh_emulatorpin_in_virtual_environments_with_nfv.

[18] Jeongseob Ahn, Chang Hyun Park, Taekyung Heo, and Jaehyuk Huh. Accelerating critical os services in virtualized systems with flexible micro-sliced cores. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[19] Maryam Amiri and Leyli Mohammad-Khanli. Survey on prediction models of applications for resources provisioning in cloud. *Journal of Network and Computer Applications*, 82:93–113, 2017.

[20] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1383–1394, 2015.

[21] 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.

[22] Shuang Chen, Christina Delimitrou, and José F Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 107–120, 2019.

[23] Shuang Chen, Angela Jin, Christina Delimitrou, and José F Martínez. Retail: Opting for learning simplicity to enable qos-aware power management in the cloud. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 155–168. IEEE, 2022.

[24] 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.

[25] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 153–167, 2017.

[26] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.

[27] 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, Philadelphia, PA, June 2014. USENIX Association.

[28] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297, 2020.

[29] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces. In *Proceedings of the International Symposium on Quality of Service*, pages 1–10, 2019.

[30] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629. IEEE, 2018.

[31] SR Jiri Herrmann, Dayle Parker, and Scott Radvan. Red hat enterprise linux 7 virtualization tuning and optimization guide, 2015.

[32] 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*, pages 139–152, 2021.

[33] 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. USENIX Association, July 2020.

[34] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for {μsecond-scale} tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, 2019.

[35] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Opportunistic spinlocks: Achieving virtual machine scalability in the clouds. *ACM SIGOPS Operating Systems Review*, 50(1):9–16, 2016.

[36] 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.

[37] Harshad Kasture, Davide B Bartolini, Nathan Beckmann, and Daniel Sanchez. Rubik: Fast analytical power management for latency-critical systems. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 598–610, 2015.

[38] Harshad Kasture and Daniel Sanchez. Ubik: Efficient cache sharing with strict qos for latency-critical workloads. *ACM SIGPLAN Notices*, 49(4):729–742, 2014.

[39] Harshad Kasture and Daniel Sanchez. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10. IEEE, 2016.

[40] Hwanju Kim, Sangwook Kim, Jinkyu Jeong, and Joonwon Lee. Virtual asymmetric multiprocessor for interactive performance of consolidated desktops. *SIGPLAN Not.*, 49(7):29–40, mar 2014.

[41] Hwanju Kim, Sangwook Kim, Jinkyu Jeong, Joonwon Lee, and Seungryoul Maeng. Demand-based coordinated scheduling for smp vms. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, pages 369–380, 2013.

[42] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

[43] Yuhang Liu, Xin Deng, Jiapeng Zhou, Mingyu Chen, and Yungang Bao. Ah-q: Quantifying and handling the interference within a datacenter from a system perspective. pages 471–484, 2023.

[44] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 450–462, 2015.

[45] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible colocations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, pages 248–259, 2011.

[46] Paul Marshall, Kate Keahey, and Tim Freeman. Improving utilization of infrastructure clouds. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 205–214. IEEE, 2011.

[47] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Willy Zwaenepoel. Diagnosing performance overheads in the xen virtual machine environment. VEE '05, page 13–23, New York, NY, USA, 2005. Association for Computing Machinery.

[48] Rajiv Nishtala, Vinicius Petrucci, Paul Carpenter, and Magnus Sjalander. Twig: Multi-agent task management for colocated latency-critical cloud services. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 167–179. IEEE, 2020.

[49] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, 2019.

[50] Jinsu Park, Seongbeom Park, Myeonggyun Han, Jihoon Hyun, and Woongki Baek. Hypart: A hybrid technique for practical memory bandwidth partitioning on commodity servers. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, pages 1–14, 2018.

[51] Tirthak Patel and Devesh Tiwari. Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 193–206. IEEE, 2020.

[52] Yaqiong Peng, Song Wu, and Hai Jin. Robinhood: Towards efficient work-stealing in virtualized environments. *IEEE Transactions on Parallel and Distributed Systems*, 27(8):2363–2376, 2016.

[53] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne:{Core-Aware} thread management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 145–160, 2018.

[54] Jia Rao and Xiaobo Zhou. Towards fair and efficient smp virtual machine scheduling. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, page 273–286, New York, NY, USA, 2014. Association for Computing Machinery.

[55] 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.

[56] 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, 2017.

[57] 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.

[58] Orathai Sukwong and Hyong S Kim. Is co-scheduling too expensive for smp vms? In *Proceedings of the sixth conference on Computer systems*, pages 257–272, 2011.

[59] 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.

[60] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L Santoni, Fernando CM Martins, Andrew V Anderson, Steven M Bennett, Alain Kagi, Felix H Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.

[61] Xiaodong Wang, Shuang Chen, Jeff Setter, and José F Martínez. Swap: Effective fine-grain management of shared last-level caches with minimum hardware support. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 121–132. IEEE, 2017.

[62] Chuliang Weng, Zhigang Wang, Minglu Li, and Xinda Lu. The hybrid scheduling framework for virtual machine systems. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 111–120, 2009.

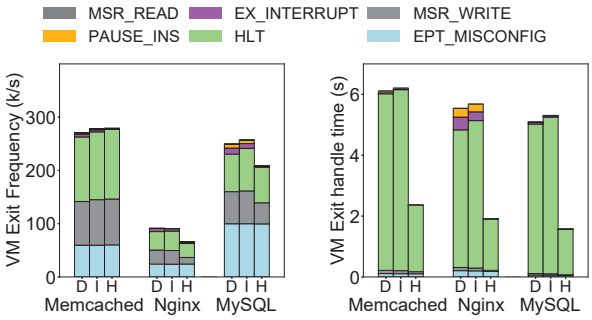
[63] Song Wu, Zhenjiang Xie, Haibao Chen, Sheng Di, Xinyu Zhao, and Hai Jin. Dynamic acceleration of parallel applications in cloud platforms by adaptive time-slice control. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 343–352, 2016.

[64] Ying Zhang, Jian Chen, Xiaowei Jiang, Qiang Liu, Ian M Steiner, Andrew J Herdrich, Kevin Shu, Ripan Das, Long Cui, and Litrin Jiang. Libra: Clearing the cloud through dynamic memory bandwidth management. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 815–826. IEEE, 2021.

Appendix

A Extended Characterization

A.1 Impact on VM Exits under Host-Aware Isolation



(a) #VM Exits **(b) VM Exit Handled Time**

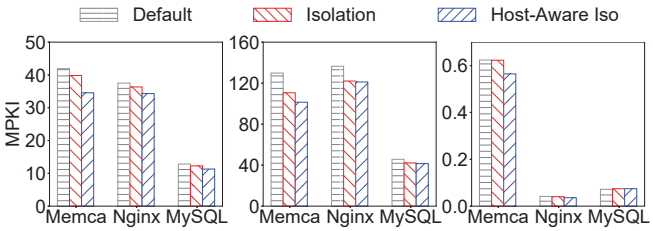
Figure 16: VM exit frequency and VM exit handled time under default (D), isolation (I), and host-aware isolation (H), decomposed by VM exit reason.

VM exits are handled 2x faster on the host under host-aware isolation. Figure 16 shows the number of VM exits and the handling time of VM exits per second under the three core managers, decomposed by exit reason, collected and reported by `perf kvm`. We find that the number of VM exits does not vary much among the three mechanisms, but the handling time is reduced by more than 60%.

For LC applications whose requests come and go, a HLT VM exit is triggered every time when the application becomes idle in between requests, and a VM entry is triggered every time when a new request is received. HLT handling time is the time between a HLT vm exit and the subsequent vm entry. It signals how fast the system reacts to new requests. Under host-aware isolation, running vCPU count never exceeds pCPU count, so that there are always free pCPUs to handle VM entries as quickly as possible. However, under other mechanisms, vCPU count is more than pCPU count, and VM entries cannot be consumed immediately since there may be no idle pCPU.

A.2 Impact on Caches under Host-Aware Isolation

Caches are better utilized under host-aware isolation. Figure 17 shows misses-per-kilo-instructions (MPKI) of L1 instruction cache (L1I), L1 data cache (L1D), and last-level cache (LLC). Compared with Default, Isolation reduces L1D and L1I MPKI by up to 5% and 15% (average of 4.1% and 11%), respectively. This is because under Default, a vCPU can freely move around between 16 pCPUs, while Isolation restricts a vCPU to a fixed set of 4 pCPUs, eliminating cache pollution and context switches with other VMs' vCPUs. This explains why Isolation outperforms Default in Figure 2, despite occasionally higher guest-side scheduling frequency. Host-aware isolation further reduces L1D, L1I and LLC MPKI by up to 12.5%, 7% and 10% (average of 8.5%, 2.7%, and 6%), respectively. This is because our host-aware isolation can reduce the context switches and in turn decrease the times of cache line evictions.



(a) L1 dcache **(b) L1 icache** **(c) Last-level cache**

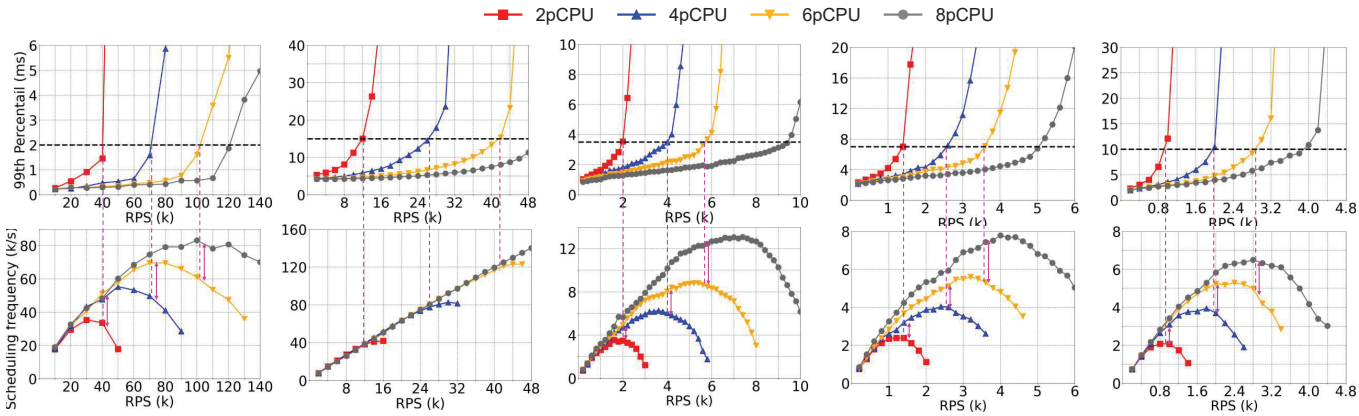
Figure 17: Cache misses-per-kilo-instructions (MPKI) under three core managers.

A.3 Indications of Guest-Side Scheduling Frequency

To show the generality of guest-side scheduling frequency, we experiment with three more LC applications from Tailbench [39], including ImgDNN, Xapian, and Moses. Adding Nginx and MySQL, Figure 18 shows tail latency and guest-side scheduling frequency with increasing RPS under various pCPU count of the five LC applications (Memcached is already shown in Figure 4).

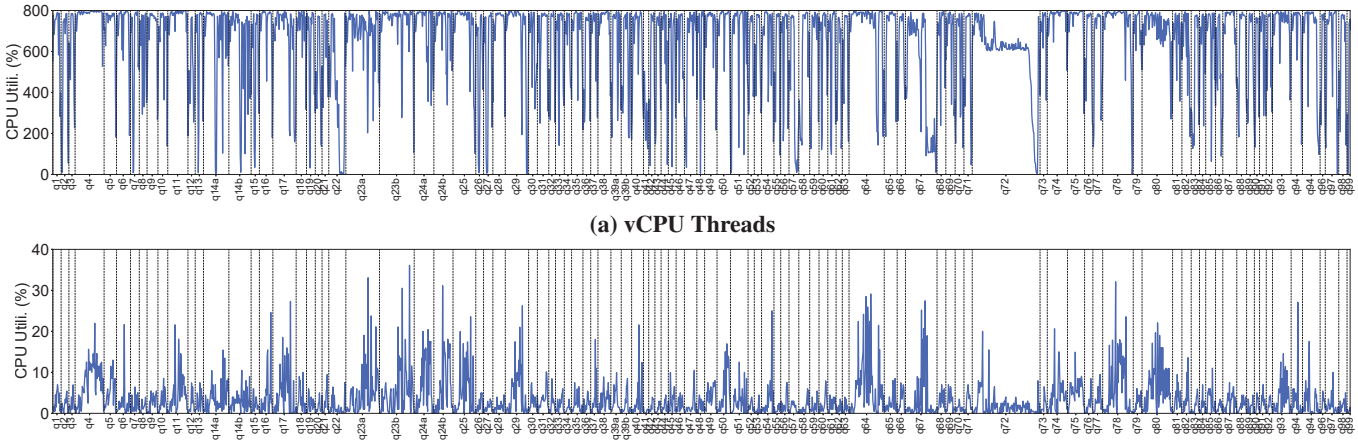
For all the evaluated applications except for MySQL, guest-side scheduling frequency is a concave function with RPS. Given a certain input load, we can compare guest-side scheduling frequency between adjacent pCPUs, and check if the difference is above a certain threshold. 30% universally work for these LC applications. For instance, for Nginx at RPS=80k, scheduling frequency under 8 and 6 pCPUs are very similar. However, when reducing pCPU count to 4, guest-side scheduling frequency is reduced by 43%. Therefore, we have to assign 6 pCPUs for Nginx at RPS=80k.

MySQL is the only exception: scheduling frequency increases with RPS and then reaches a short plateau. We find that MySQL is fundamentally different from other applications in that a single MySQL request involves multiple application threads and thus multiple guest-side context switches. Suppose one MySQL request triggers X context switches and the current RPS is R, there will be $X * R$ context switches per second regardless of how small or large R is. On the contrary, a request of the other five LC applications is only processed by a single application thread. At low load, the application thread will be scheduled in upon request arrival, and scheduled out after request processing. When load increases and there is no free time between consecutive requests, the application thread will not be immediately scheduled out after a single



(a) Nginx (b) MySQL (c) ImgDNN (d) Xapian (e) Moses

Figure 18: Relationship between application's tail latency and guest-side scheduling frequency of five LC applications, apart from Memcached which is already shown in Figure 4. The x-axis represents increasing input load. The horizontal dotted lines represent QoS targets. The vertical purple lines represent the maximum RPS under QoS under each pCPU count. Purple arrows represent the difference in guest-scheduling frequency between adjacent pCPUs.



(a) vCPU Threads

(b) Emulator Threads

Figure 19: CPU utilization of vCPU threads and emulator threads when running the 99 queries in Spark. The x-axis represent time, but is marked by query index to differentiate different queries. Vertical dotted lines also split all the queries.

request, and will keep processing the next request, resulting in reduced context switches at higher load.

This potentially means that we need to set a different threshold for MySQL, e.g., 5%. Fortunately, we find that this is not necessary. As introduced in Section 4.5, UFO also adopts heuristics based on CPU utilization to guide core adjustment. This is primarily to reduce trials when models are not fitted for most applications. However, it also sets a bottom line for core adjustment: when 30% is too relaxed for some applications, UFO will not keep reducing pCPUs which may cause QoS violations. UFO will ensure that per-core CPU utilization is under 80%.

There are some alternative methods to handle cases like MySQL. UFO does not adopt these methods currently, but we leave more options for comprehensiveness. The first method is to reduce the threshold from 30% to 5%. While this is conservative for the other five applications, it is a simple and universal approach, and we can achieve lower tail latency (at the cost of more resources). For instance, as shown in Figure 18a, if the threshold is set to 30%, Nginx at RPS=40k would request only 2 pCPUs to reach a tail latency of 2ms. If setting the threshold to 5%, Nginx at RPS=40k would be assigned 4 pCPUs, reaching a tail latency of 0.5ms. The second approach is to differentiate the two types of LC applications, and set different thresholds for each category. As shown in Figure 18, when the full profile of guest-side scheduling frequency is obtained, it is very easy to distinguish MySQL-like applications from other applications since scheduling frequency does not drop at higher load. We set the threshold to 5% once identifying these applications, and keep the threshold of 30% for all other LC applications.

Table 3: Symbols in UFO

N	Number of virtual machines
$vCPU[i]$	Requested vCPU count of VM_i
$pCPU[i]$	Allocated pCPU count of VM_i
$emuUtil$	Current CPU utilization of the emulator pool
$cpuUtil[i]$	Current CPU utilization of VM_i
$schedFreq[i]$	Current scheduling frequency of VM_i
$SF[i][c]$	Recorded scheduling frequency of VM_i under c pCPUs A dictionary of (u, f) pairs, where u and f are CPU utilization and scheduling frequency, respectively.
$Model[i][c]$	Fitted model of scheduling frequency of VM_i under c pCPUs using $SF[i][c]$, a quadratic function denoted by (A_{ic}, B_{ic}, C_{ic}) Default value is $NULL$ if not fitted yet
$Pred[i][c]$	Max CPU utilization that can be sustained for VM_i under c pCPUs, predicted using $Model[i][c]$ and $Model[i][c+2]$ $Pred[i][0] = 0$. If $c > 0$, default is -1 if not predicted yet.

A.4 CPU Utilization of Spark SQL

Figure 19 shows CPU utilization of vCPU and emulator threads when running each query in Spark. vCPU threads may run up to 800% of CPU utilization (8-vCPU VMs), while emulator threads use at most 40% of CPU utilization. The relative low usage of emulator threads makes isolation of emulator threads ineffective for Spark.

B Extended UFO Design

In this section, we show the detailed algorithms in the UFO design. The algorithms cover all the four stages in UFO's vCPU adjustment, shown in Figure 9. Table 3 includes all the symbols used in the algorithms.

B.1 Update Modeling of Scheduling Frequency

Algorithm 2 shows how and when the modeling of scheduling frequency is updated. For VM_i, when it just gets launched on the host, no samples have been recorded (i.e., $len(SF[i][c]) = 0$ for any c — the number of cores), and no models have been fitted yet (i.e., $Model[i][c] = NULL$ is for any c).

As the VM runs, an increasing number of samples at various CPU utilization and pCPUs are recorded, and we maintain the last eight samples in $SF[i][c]$ for each c. For a given pCPU count, when three samples are recorded, we will start fitting the model, though the model may be inaccurate due to skewed samples. The inaccuracy will eventually come to light as the VM runs and more samples are collected. Upon detection of inaccuracy (i.e., the difference between the fitted value and the true value is more than 5%), the model will be refitted using the latest and more samples. Storing the latest eight samples and refitting models upon detection of inaccuracy allow UFO to adjust to workload churn inside the VM. For instance, if a VM previously runs Memcached and later runs Nginx, SF and $Model$ will all shortly be updated with the latest samples.

Algorithm 2: $updateModel(i,c,u,f)$: Update samples and the fitted model of VM_i under c pCPUs. VM_i's current CPU utilization and scheduling frequency are u and f, respectively.

```
if u is not in SF[i][c] then
    // The first time for VM_i to reach util of u under c cores
    add (u, f) to SF[i][c];
    if len(SF[i][c]) > 8 then
        // Too many samples. Maintain the latest 8 samples for modeling
        remove the eldest element in SF[i][c] ;
if Model[i][c] == NULL then
    if len(SF[i][c]) == 3 then
        // First time to collect enough samples for model fitting
        Fit Model[i][c];
        if Model[i][c+2] ≠ NULL then
            // Record the maximum CPU utilization that c pCPUS can sustain under QoS
            Pred[i][c] = max_{getSF(i,c,u)/getSF(i,c+2,u) ≥ 0.7} u;
else
    if |f − SF[i][c][u]|/f > 5% or
      |f − (A_ic * u^2 + B_ic * u + C_ic))|/f > 5% then
        // The current recorded value or the fitted value is inaccurate
        Fit Model[i][c];
        if Model[i][c+2] ≠ NULL then
            // Update the max CPU utilization
            Pred[i][c] = max_{getSF(i,c,u)/getSF(i,c+2,u) ≥ 0.7} u;
return;
```

When models of consecutive pCPUs are fitted, i.e., both $Model[i][c]$ and $Model[i][c+2]$ are obtained, we start calculating $Pred[i][c]$. We clarify the reasoning behind $Pred[i][c]$ in Appendix B.2.

B.2 Core Predictor

The goal of UFO's core predictor $predict(i,c,u)$ is to output the best core allocation for VM_i, given its current pCPU count c and current CPU utilization u. Note that its scheduling frequency is monitored and fed to the module of $updateModel()$ (see Algorithm 1), so we do not need to explicitly pass it as an argument in the core predictor.

As discussed in Section 3.3, we can leverage guest-side scheduling frequency to guide core allocation decisions on the host OS. Comparing the relationship between scheduling frequency and application's tail latency, we find two conditions to help determine the best core allocation.

Suppose VM_i current runs at CPU utilization u with scheduling frequency f under c pCPUs, c is the best core allocation if and only if:

Algorithm 3: $predict(i,c,u)$: Predict core allocation for VM_i whose current pCPU count is c and CPU utilization is u.

```
for p == 2, 4 .. vCPU[i] do
    // Check if we can directly predict based on u
    if Pred[i][p−2] == −1 then
        continue;
    if Pred[i][p−2] < u and Pred[i][p] ≥ u then
        return p;
p = c;
if p+2 ≤ vCPU[i] and getSF(i,p,u)/getSF(i,p+2,u) < 0.7 then
    // c pCPUs is not enough, check more pCPUs
    while p+2 ≤ vCPU[i] and getSF(i,p,u)/getSF(i,p+2,u) < 0.7 do
        p=p+2;
else
    // c pCPUs is already enough, check fewer pCPUs
    while p−2 > 0 and getSF(i,p−2,u)/getSF(i,p,u) > 0.7) do
        p=p-2;
return p;
```

Algorithm 4: $getSF(i,c,u)$: Get scheduling frequency of VM_i under c pCPUs and u CPU utilization.

```
if u in SF[i][c] then
    // A previously recorded pair
    return SF[i][c][u];
if Model[i][c] ≠ NULL then
    // A previously fitted model
    return A_ic * u^2 + B_ic * u + C_ic;
// Adjust pCPUs to collect scheduling frequency
modify(i, c);
sleep for 3s;
updateModel(i, c, cpuUtil[i], schedFreq[i]);
return schedFreq[i];
```

- $c+2$ is too much, i.e., scheduling frequency does not increase much with more cores.
- $c-2$ is not enough, i.e., scheduling frequency drops significantly with fewer cores.

UFO checks the two conditions above by comparing f with scheduling frequency f' under adjacent pCPU counts, and checks if the difference between f and f' is within a certain threshold.

B.3 Monitoring and Reaction Frequency

As discussed in Section 4.3, UFO operates at second granularity, monitoring every second, and waiting for three seconds after each core adjustment. This is because:

- As shown in Figure 20, monitoring at finer granularity causes unstable results. As introduced in Section 5.2, even under constant load, there are many load and latency fluctuations due to the exponential request inter-arrival distribution. UFO is designed for black-box public clouds and does not assume any application-level knowledge like prior fine-grained approaches [28, 49]. It aims to meet QoS in the long run (i.e., at least over tens of seconds like Heracles [44] and PARTIES [22]).

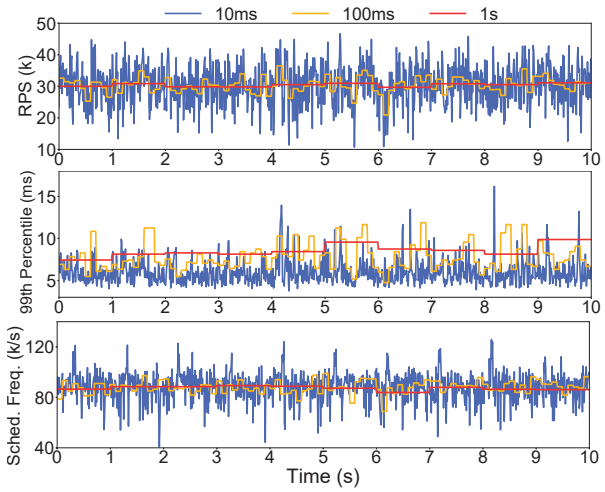


Figure 20: Data stability with increasing monitoring interval when MySQL runs at RPS of 30000. Despite constant RPS, request inter-arrival time follows an exponential distribution, causing significant load and latency fluctuations at sub-second granularity.

- We find that after vCPU onlining/offlining in the guest OS, it will be up to 3s for guest-side CPU utilization to stabilize (more discussion in Section 6.4). We evaluate how this affects UFO's responsiveness to bursty loads in Section 6.2.

B.4 Interaction with the Cluster Manager

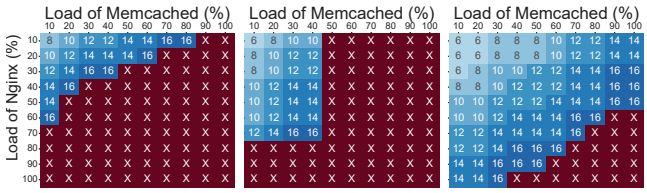
UFO is an intra-node core management, and should interact with the high-level cluster manager based on the size of the idle pool. There are three cases of the idle pool size:

- If the idle pool consistently has more than a handful of idle cores (e.g., $\#idle\ cores > 2$), this signals system underloading, and UFO will signal the cluster manager to schedule more tasks to the node to make use of the idle pool.
- If the idle pool is constantly small (e.g., $0 \leq \#idle\ cores \leq 2$), this means there is *just* the right amount of system load, and the cluster manager should stop scheduling more tasks to the node.
- If the idle pool consistently fails to supply the vCPU or the emulator pool with more needed cores, UFO will signal the cluster manager to migrate some tasks to other nodes.

C Extended Evaluation

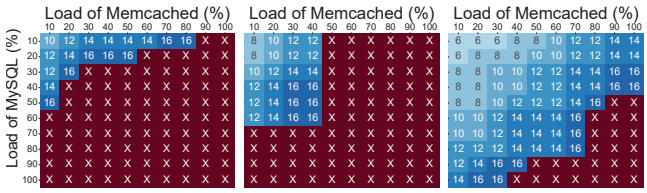
C.1 Heatmaps of 2-VM Colocation Mixes

We show heatmaps of all the 2-VM colocation mixes in Figure 21, 22, 23, 24 and 25. Heatmaps of colocation of Memcached and Memcached is already shown in Figure 12.



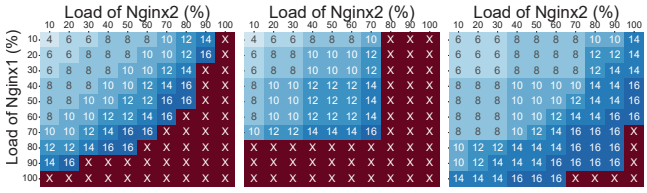
(a) Default **(b) DynIso** **(c) UFO**

Figure 21: Colocation of Memcached and Nginx.



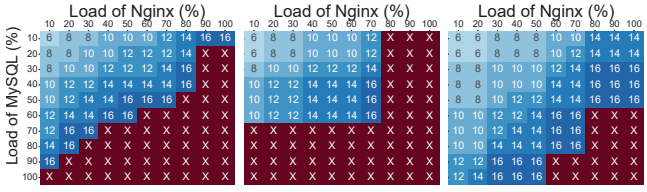
(a) Default **(b) DynIso** **(c) UFO**

Figure 22: Colocation of Memcached and MySQL.



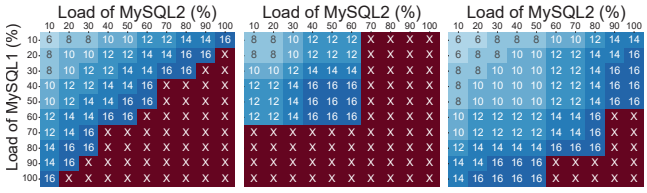
(a) Default **(b) DynIso** **(c) UFO**

Figure 23: Colocation of Nginx and Nginx.



(a) Default **(b) DynIso** **(c) UFO**

Figure 24: Colocation of Nginx and MySQL.



(a) Default **(b) DynIso** **(c) UFO**

Figure 25: Colocation of MySQL and MySQL.

C.2 UFO Overhead

Figure 26 shows the time required for onlining and offlining increasing number of vCPUs in the guest OS under various workloads. For each experiment, we consecutively online and offline vCPUs for 1000 times, obtain the total overhead, and report the average overhead per adjustment. For LC workloads, the overhead is about 20-30ms per vCPU. Overhead is higher under BE workloads. This is because BE applications tend to operate at very high CPU utilization (Figure 7). When offlining a certain vCPU, application threads running on the vCPU will all be migrated to other vCPUs. Therefore, higher CPU utilization usually results in higher overhead.

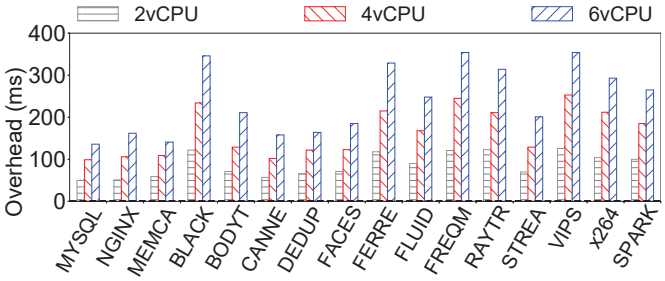


Figure 26: Average time spent on onlining or offlining 2, 4 and 6 vCPUs under various workloads. LC applications operate at their *max load*.