# Achieving Low Latency in Public Edges by Hiding Workloads Mutual Interference

Weiwei Jia[*]
The University of Rhode Island

Jiyuan Zhang
New Jersey Institute of Technology

Jianchen Shan
Hofstra University

Jing Li
New Jersey Institute of Technology

Xiaoning Ding
New Jersey Institute of Technology

## Abstract

On multi-tenant platforms, such as public clouds and edges, workloads interfere with each other through shared resources. The performance degradation caused by such interference is a notoriously challenging problem. Though many solutions have been proposed for clouds, they can hardly help the application in edges, where workloads are mostly latency-critical, highly dynamic, and more sensitive to interference. Aggressive resource over-provisioning looks to be the only practical solution, albeit it causes significant resource waste.

The paper proposes dynamic asymmetric scheduling for edge computing (DASEC) as a unique approach to achieve low latency in public edges and improve resource utilization. DASEC makes application performance less sensitive to the interference between workloads by making the interference affect mostly the tasks on non-critical paths and rarely the tasks on critical paths. With DASEC, the interference is largely hidden from being reflected on the end-to-end performance observed by users.

The paper has investigated the techniques to implement DASEC in the task schedulers for edge workloads and tested its effectiveness in managing the interference caused by sharing CPU cores. For different types of edges that schedule tasks at different system levels, the paper implemented DASEC prototypes based on Linux/KVM vCPU scheduler, the completely fair scheduler (CFS) in Linux OS, and Google user-level scheduling framework. Extensive experiments with diverse real-world applications show that DASEC can reduce the latencies of the workloads consolidated on the same edge server by 32% $\sim$ 52%.

[*]Most work done while the author was a Ph.D. student at NJIT

*SoCC '22, November 7–11, 2022, San Francisco, CA, USA*
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9414-7/22/11...$15.00
https://doi.org/10.1145/3542929.3563459

## CCS Concepts

• **Software and its engineering**;

## Keywords

Edge Computing, Virtualization, Scheduling

**ACM Reference Format:**
Weiwei Jia, Jiyuan Zhang, Jianchen Shan, Jing Li, and Xiaoning Ding. 2022. Achieving Low Latency in Public Edges by Hiding Workloads Mutual Interference. In *Symposium on Cloud Computing (SoCC '22), November 7–11, 2022, San Francisco, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3542929.3563459

## 1 Introduction

By providing computing resources closer to end users, public edge computing can benefit a wide range of applications, such as autopilot, mobile or IoT services, online games, and augmented reality/virtual reality (AR/VR). Typical workloads in the edge are expected to be resource-demanding and latency-sensitive. These workloads can hardly run on the resource-constrained end devices or run in centralized clouds, because the network latencies are high between user end devices and the clouds.

As a multi-tenant computing platform, in a public edge, multiple workloads from different users can share the same edge server. Due to the resource sharing and resource contention, the execution of a workload interferes with the execution of other workloads. Such interference inevitably causes performance penalties to the workloads, which can be significant and difficult to predict if not well controlled.

Currently, edge computing platforms rely mainly on resource over-provisioning to mitigate resource contention and reduce performance degradation. To meet the strict quality of service (QoS) requirements of edge workloads on latencies and to deal with the workload fluctuation [1, 2], aggressive resource over-provisioning is required in the edge, causing significant resource waste. Studies have shown that resource utilization is much lower in edges than in conventional clouds. In particular, the mean CPU utilization is about 6x lower on edge servers than on cloud servers [2].

There are approaches that can eliminate/reduce the interference between workloads with relatively high resource efficiency. However, they are not applicable to edges. For example, one approach avoids co-locating the workloads that may interfere with each other; and another approach runs low priority best-effort workloads to consume the resources that are not being used by latency-critical workloads [3–7]. However, edges are dominated by latency-sensitive workloads and lack enough best-effort workloads. The resource usage patterns of latency-sensitive workloads may change dynamically, making it difficult to predict which workloads may interfere with each other. At the same time, the limited number of edge servers in the same site and the large number of workloads consolidated on each server make it very challenging to effectively distribute and separate interfering workloads across different servers.

Interference may also be mitigated by statically or dynamically partitioning resources among workloads. Resource partitioning essentially trades the capability of adapting resource allocation based on resource demand for the performance isolation between workloads. It fits the workloads with relatively stable resource demands and can barely help edge workloads that have very dynamic resource usages, as we will show in Section 6. Infrequent adjustment of resource partitions provides better isolation (i.e., less interference), but the resources may not be adapted to fit the resource demand of workloads, causing either performance degradation when a workload is short of resources or resource waste when the demand is low. Adjusting resource partitions frequently incurs high overhead and weakens the capability to reduce interference.

This paper presents DASEC that can effectively reduce the impact of the interference between edge workloads on their performance, such that these workloads can achieve low latency without aggressive resource over-provisioning. The main idea is to move the interference off the tasks on the critical paths of the workloads. The critical path in a workload is a set of tasks that must be finished as quickly as possible to avoid delaying the progress of the workload. By making interference affect mostly the tasks on non-critical paths and rarely the tasks on critical paths, the interference is largely hidden from being reflected on the latencies observed by users; thus the end-to-end performance is less impacted. Moving the interference off critical paths can be achieved by forcing the tasks on non-critical paths to yield resources or making them more willing to yield resources to the tasks on critical paths.

While the general idea in DASEC can be applied to manage the interference caused by many types of shared resources (e.g., CPU cores, memory space, CPU caches), this paper focuses on the interference caused by sharing CPU cores, because CPU resources, as the most important resource type, have the largest impact on performance. We leave the study of using DASEC in the management of other interference types as future work.

Through a shared CPU core, a task (in one workload) may interfere with another task (in another workload) in three ways: 1) It may delay the task, making it start late. 2) It may interrupt the task in the middle and delay its unfinished part. 3) The CPU share reserved for it reduces the CPU share available to the task and slows down the slow progress of the task. Thus, to remove the interference from the tasks on critical paths, DASEC identifies these tasks, makes them start early, allocates them with enough CPU share, and prevents them from being interrupted.

DASEC targets the edge workloads colocated on the same edge server. In each workload, tasks with dependencies are distributed into multiple threads or processes, and executed concurrently on multiple cores or virtual CPUs (vCPUs). DASEC aims to reduce the latency of each workload, instead of improving the performance of an individual thread/process. This makes it different from most of the other priority-based scheduling mechanisms, which schedule threads/processes as independent tasks.

Compared to various co-scheduling mechanisms designed for throughput-oriented systems, DASEC aims to improve performance on a platform dominated by latency-critical workloads. Co-scheduling improves the performance of a multi-threaded or multi-process workload by running collaborative tasks simultaneously. It boosts the priorities of these tasks indiscriminately, no matter whether they are on the critical-path or not. This design is a double-edge sword. On the one hand, it makes co-scheduling particularly effective; on the other hand, it causes notorious adverse effects, such as high overhead and CPU fragmentation [8], which can quickly negate the benefits of co-scheduling if not well controlled. Thus, co-scheduling is usually used on throughput-oriented systems, because it can be applied infrequently and/or applied discriminately only on a part of the colocated workloads that are the most communication intensive, in order to control the adverse effects. Compared to co-scheduling, DASEC only boosts the tasks on the critical path, and incurs minimal overhead. This makes DASEC fit the best public edges, where latency-critical workloads dominate, and task scheduling must be frequently involved to enable the fast response of all the workloads.

This paper makes the following major contributions. **First**, it proposes DASEC as an innovative approach to hide the inter-workload interference caused by resource sharing and reduce the impact on latencies. **Second**, following this approach, the paper investigates the techniques that can effectively hide the interference caused by sharing CPU cores. **Third**, it evaluates DASEC on six real-world applications, including two AI inference applications (`Image-classify` [9] and `Action-recognize` [10, 11]) and four latency sensitive applications (`Masstree` [12], `Img-dnn` [13], `Silo` [14], and `Memcached` [15]). The experiments show that:

1. `DASEC` can effectively reduce latencies. Compared to `Linux/KVM`, `DASEC` reduces mean latencies and 99th tail latencies by 46% and 52%, respectively. Compared to `PARTIES` [1] and `BVT` [4, 16], which are also designed for latency-sensitive workloads, `DASEC` reduces mean latencies and the 99th tail latencies by up to 44% and 32%, respectively.
2. The effectiveness of `DASEC` increases with the consolidation ratio (i.e., the number of workloads on the same edge server).
3. `DASEC` is portable. The paper shows that `DASEC` can be implemented and hide workload interference effectively at multiple system layers, including the hypervisor layer (implemented as a vCPU scheduler in Linux/KVM [17]) if conventional virtualization is used, the OS layer (implemented in Linux CFS thread/process scheduler) if containerization is used, or even at the application layer (implemented in Google user-level scheduling framework `ghOSt` [18]).

## 2 Background and Motivation

This section introduces the features of edge computing and edge workloads. Then it describes the performance issues caused by the interference between the latency-sensitive workloads colocated on the same edge server and sharing CPU cores. Finally, it explains the objectives of `DASEC`.

Edge clouds are tiny clouds deployed close to end users. They can benefit many new application types (e.g., IoT, AR/VR, gaming, and self-driving vehicles) that desire low processing and response times (e.g., millisecond scales) [19]. Public edges, such as AWS local zones [20], Google edge platforms [21], Azure edge zones [22], and Tencent edge clouds [23], are emerging quickly as the major computing platforms for the general public, individuals or businesses to run their latency-sensitive applications in edge clouds.

The servers in a public edge must support dense multi-tenancy. Given the huge number of geographically distributed public edges to serve end users locally, each public edge is expected to have only very limited computing resources, e.g., a small number of servers. These resources are shared by all local users to improve utilization.

When the latency-sensitive applications from different users are colocated on the same edge server, their executions can interfere with each other through resource sharing and contention. The interference may significantly increase the latencies observed by end users. We use a few experiments to demonstrate this problem. The experiments use two latency-sensitive benchmarks, `Masstree` and `Img-dnn`. We run multiple instances of the same benchmark in virtual machines

| **Workloads** | **Average CPU utilization** | |
|---|---|---|
| | `Linux/KVM` | `DASEC` |
| `Masstree` | 30% | 42% |
| `Img-dnn` | 43% | 56% |
| `Memcached` | 31% | 41% |
| `Silo` | 21% | 40% |

**Table 1:** ***Average CPU utilization of*** `Linux/KVM` ***and*** `DASEC` ***when the consolidation ratio is high.*** **VMs run the same workload. Each VM has 16 vCPUs and each workload has 16 threads.**

(VMs). Each VM has 16 vCPUs and runs an instance with 16 threads. We increase the number of benchmark instances colocated on a server and compare the performance with `Linux/KVM`[1] and our proposed solution `DASEC`. §6 details the experimental setup, including server/VM configurations and application descriptions.

The experiments focus on the contention for CPU cores. We use consolidation ratio to measure the level of resource contention, which is the ratio between the total number of threads (vCPUs since we run 1 thread on each vCPU) and the total number of physical cores in the system. For instance, with 128 virtual CPUs (vCPUs) sharing a 80-core physical server, the consolidation ratio is 1.6. To eliminate the contention for other resource types, including memory and I/O bandwidth, we control the experiment settings such that the main memory is not over-committed, and the benchmarks have minimal I/O operations.

Figure 1 (a) and (b) show the mean latencies and 99th tail latencies of `Masstree` for different consolidation ratios (0.4~3.2). When the consolidation ratio is low (0.4 and 0.8), each vCPU/thread can have a dedicated core, and there is little interference between the workloads. Thus, the latencies are low. However, as the consolidation ratio increases and the cores are over-committed, the mean latencies and 99th tail latencies increase quickly on vanilla Linux/KVM. Compared to those at a consolidation ratio of 0.4, they are 19% and 22% higher with the consolidation ratio increased to 1.6, and 70% and 4x higher when the rate is increased to 3.2. Note that the high latencies happen when the CPU utilization is still very low (around 30%, as shown in Table 1).

The performance of `Masstree` with `DASEC` shows that much of the latency increase can be avoided. With `DASEC`, when the consolidation ratio is increased to 3.2 from 0.4, the mean latencies are kept roughly stable, and the 99th tail latencies are increased by only 97%, which are much lower than those with vanilla Linux/KVM.

Similar observations can also be obtained from the performance of `Img-dnn` shown in Figures 1 (c) and (d). When the consolidation ratio is 1.6, with the vanilla Linux/KVM,

[1]We also use containers for experiments (see §6).

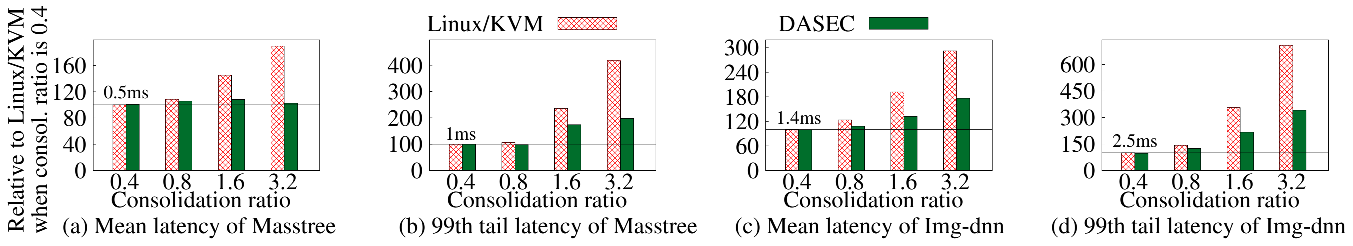

**Figure 1:** ***Mean latencies and 99th tail latencies of*** `Masstree` ***and*** `Img-dnn` ***with*** `DASEC` ***and vanilla Linux/KVM when the consolidation ratio increases from 0.4 to 3.2.*** **Consolidation ratio is the rate between the total number of vCPU and the total number of cores in the system.**

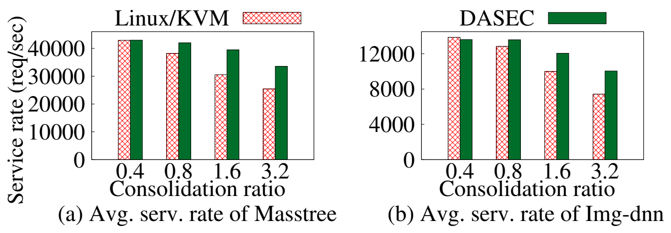

**Figure 2:** ***Service rates of*** `Masstree` ***and*** `Img-dnn` ***with*** `DASEC` ***and vanilla Linux/KVM as the consolidation ratio increases.*** **Relative to vanilla Linux/KVM,** `DASEC` **greatly improves the service rate when the consolidation ratio is bigger than 1. QPS of** `Masstree` **and** `Img-dnn` **is 1000 and 500, respectively.**

the mean latency and the 99th tail latency are increased by 59% and 138%, respectively, relative to the latencies when the consolidation ratio is 0.8; and, with `DASEC`, the mean latency and the 99th tail latency are 15% and 19% lower, compared to `Linux/KVM`. The latencies of `Img-dnn` increase more quickly with the consolidation ratio than `Masstree`. The reason is that `Img-dnn` has a much higher demand for CPU resources (Table 1). Thus, when the consolidation ratio is high, the interference between `Img-dnn` instances is more difficult to reduce than that with `Masstree`.

Latencies increase with the consolidation ratio, because the interference makes the benchmarks less capable in processing requests quickly. To show this, we measure the service rate, i.e., how many requests can be processed in each second. Figure 2 shows how the service rates of `Masstree` and `Img-dnn` change with the consolidation ratio. Generally, the service rates reduce with the consolidation ratio. This trend is more visible with `Linux/KVM`. On average, with `Linux/KVM`, the service rates of `Masstree` and `Img-dnn` are lower by 18% and 15% than those with `DASEC`. When the consolidation ratio is high, the interference becomes intense and can affect the capability of the benchmarks by a larger degree. Thus, we see the service rates with `Linux/KVM` are lower than `DASEC` by larger percentages (e.g., 32% and 35% for `Masstree` and `Img-dnn`, respectively with a consolidation rate of 3.2).

To understand how the interference affects benchmark executions, we sample the states of each VM, find the occasions when the execution on the VM is blocked, and study how the execution is blocked by the interference, i.e., the contention for CPU cores. Specifically, we find the occasions when a number of vCPUs in a VM are in the "ready" state (i.e., blocked due to the lack of available CPU cores) and other vCPUs are waiting for the "ready" vCPUs to make progress. We select to study these occasions, because they are the most discernible states that the benchmark execution is blocked by the interference — The "ready" vCPUs are considered to be blocked directly by the interference, causing a direct impact on performance, and the "waiting" vCPUs are considered to be blocked indirectly by the interference, causing indirect/extra impact on performance.

For each occasion, we count the number of "ready" vCPUs and the number of "waiting" vCPUs. The comparison of these numbers indicates how seriously the interference on a vCPU degrades performance. For example, compared to the occasion with 15 "ready" vCPU and one "waiting" vCPUs, in the occasion with one "ready" vCPU and 15 "waiting" vCPUs, the interference that blocks each "ready" vCPU is considered to be causing more serious performance degradation, because it blocks more vCPUs and causes more extensive indirect/extra performance impact.

We show the distribution of the occasions with different "waiting" vCPU counts for `Masstree` and `Img-dnn` using the CDF curves in Figures 3 (a) and (b). Under `Linux/KVM`, a substantial proportion of the occasions have most vCPUs in the "waiting" state (e.g., around 50% with more than 13 "waiting" vCPUs for `Masstree` and 60% for `Img-dnn`), indicating that the interference causes much extra performance degradation. `DASEC` reduces such occasions. For example, only around 20% (`Masstree`) and 40% (`Img-dnn`) of occasions have more than 13 "waiting" vCPUs under `DASEC`. Thus, with `DASEC`, the interference causes less performance degradation.

The experiments above show that consolidating multiple latency-sensitive workloads on the same serve may significantly increase latencies. This problem happens even when there are many idle resources. Thus, traditional methods,

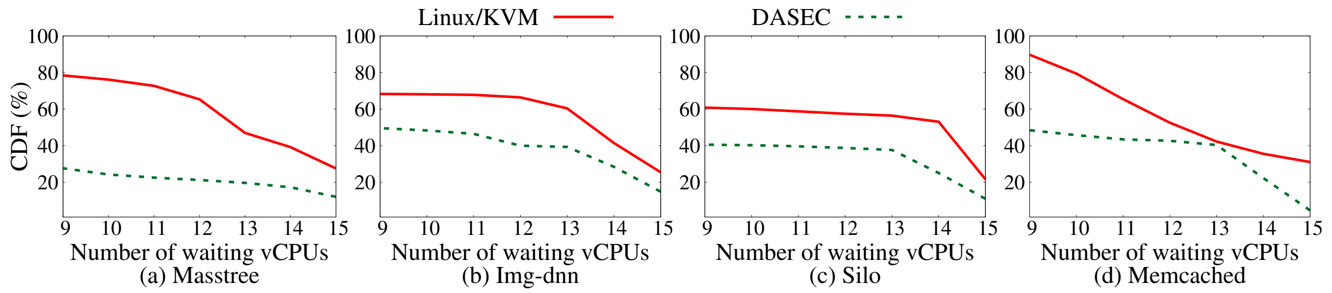

**Figure 3:** ***Distribution of occasions when a few vCPUs are in the "ready" state, and other vCPUs in the VM are waiting for the "ready" vCPUs.* We collocate 8 VMs in the 80-core physical server; all VMs run the same workload. Each VM has 16 vCPUs and each workload in the VM has 16 threads.**

such as resource over-provisioning [24–26] or collocating latency-sensitive workloads with best effort workloads [3–7], may not effectively solve this problem. At the same time, in an edge cloud, the effectiveness of these methods cannot be fully exploited, limited by the resource budget of the edge.

Given that this problem happens when there is a significant amount of idle resources, our idea for mitigating this problem is to make better utilization of the idle resource. Our method can be explained with the experiments and the analysis above: by reducing the vCPU "waiting" (i.e., the extra performance impact of the interference), idle CPU resources can be better utilized to improve performance. Since most workloads are now multi-threaded or multi-process and run collaborative tasks with dependencies, reducing "waiting" can be achieved by reducing the interference on the critical paths, as we will explain in the next section.

## 3 Main Idea: Hiding Mutual Interference

This section identifies the key factor affecting how the interference between workloads increases latencies. Then, targeting this factor, this section explains the main idea for reducing the performance impact caused by the interference.

• **Key Factor: interference on the critical path.** When multiple workloads are consolidated on the same server and time-sharing the cores, contention for the cores is often unavoidable, causing interference to the workload executions. If the interference happens to the tasks on critical paths, its performance impact is directly reflected by increased latencies. But, if the interference happens to the tasks that are not on critical paths, its performance impact may be hidden, and does not increase latencies.

We use Figure 4 to explain this factor. Figure 4 (a) shows the execution of a 2-thread application (denoted by *App*). The application processes the requests of a user. The user sends a new request after the previous request is processed. In *App*, *Thread0* is the main thread. Upon each request, the main thread communicates with a helper thread and assigns task to it.

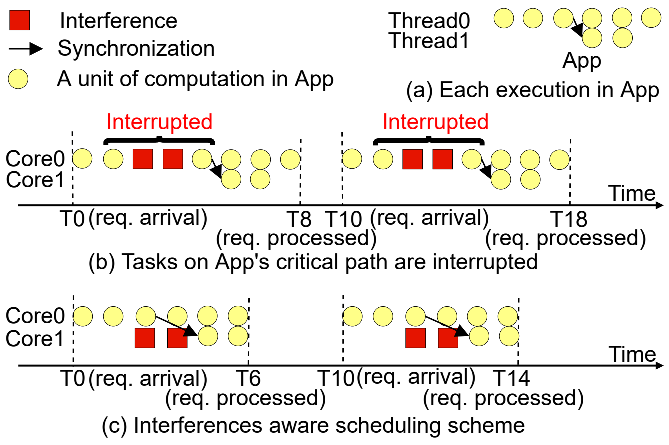

**Figure 4:** ***An illustrative example: Tasks on App's critical path are interrupted, causing longer latency.* A request arrives at *T0*; and the processing of the request finishes at *T8* in (b).** ***App*** **is waiting for a new request for fixed 2 time units: from *T8* to *T10* in schedule (b) and earlier at *T6*~*T8* in schedule (c) due to better scheduling by DASEC.**

Figure 4 (b) illustrates the performance impact on *App* caused by the interference on its critical execution path. The interference is caused at time *T2* and *T12* by the execution of another application that is sharing the same cores with *App*. Since the tasks on the critical path are delayed, it takes *App* 8 units of time to finish processing each request. The latency is increased by 2 units of time. If the interference is not on the critical path, as shown in Figure 4 (c). The latency is not increased.

• **Main Idea: hiding interference**

Our overall approach for reducing latency is to hide interference and make interference affect mostly the tasks that are not on critical paths. While this sounds intuitive, how to make it a viable approach is not. Our main idea is to prioritize tasks in an asymmetric way. The intuition is that the tasks with higher priorities are less likely to suffer from interference, because the interference is caused by tasks contending for CPU cores. With this intuition, the tasks on critical paths

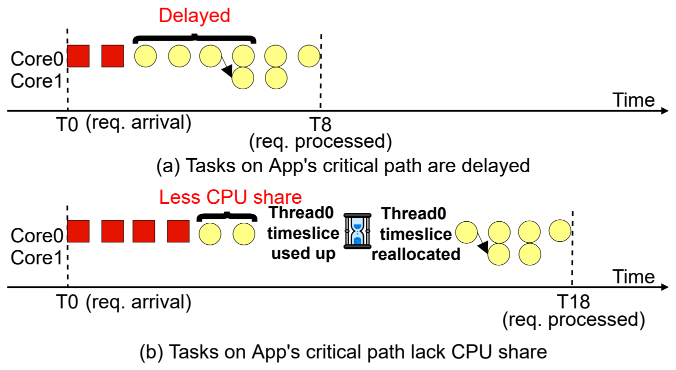
Delayed
Core0
Core1
Time
T0 (req. arrival)
T8
(req. processed)
(a) Tasks on App's critical path are delayed
Less CPU share
Thread0 timeslice used up
Thread0 timeslice reallocated
Core0
Core1
Time
T0 (req. arrival)
T18
(req. processed)
(b) Tasks on App's critical path lack CPU share

**Figure 5:** ***An illustration of possible scheduling problems: Tasks on App's critical path are delayed or lack CPU share to make progress, causing longer latency.***

should be prioritized to reduce the interference on critical paths. We want to reduce latencies of all workloads. Thus, we avoid prioritizing tasks indiscriminately and give the tasks that are not on critical paths normal priority, since the high-priority tasks in one workload would become the interference of other workloads.

Only prioritizing tasks in an asymmetric way is not sufficient. A task suffers from less interference only when its priority is relatively higher than other tasks competing resources with it. If tasks with high priorities are scheduled on the same core, the interference reduction through prioritization diminishes. For a high-priority task, the interference from another high-priority task may still affect the execution of the task in a few ways. For example, the execution of the task may be interrupted in the middle, similar to the example in Figure 4 (b), or may be scheduled late after the task is ready to run, as illustrated in In Figure 5 (a). Collocating high-priority tasks on the same core also reduce their time-shares if CPU time is shared based on priorities. Thus, a high-priority task receives less time-share than it collocated with low-priority tasks, and may cause extra pauses when it runs out of its time-share, as illustrated in In Figure 5 (b). Thus, another idea is to schedule high-priority tasks with low-priority tasks, in order to ensure the effectiveness of prioritization in reducing interference.

## 4 DASEC Design

We implemented the main ideas described in §3 into dynamic asymmetric scheduling for edge computing (DASEC), which is a task scheduler designed for edge clouds. This section introduces the main challenges, overall design, and major components of DASEC. DASEC's design is general and can be directly used in scheduling multi-threaded applications in multi-programming systems or containers. In our evaluation, we realized and tested DASEC in VMs, containers, and user-level scheduling (see §6). To facilitate our discussion, we introduce DASEC's design used in vCPU scheduling. We choose vCPU scheduling for illustration also because 1) VMs are prevalently used in edge clouds; 2) it is more challenging to implement the ideas at the VMM layer than at other layers.

### 4.1 Overall Design and Challenges

DASEC implements the main idea introduced in §3. Thus, the problem it targets is essentially how to prioritize vCPUs in each VM in an asymmetric and unaggressive way, so as to 1) make the workload in the VM achieve the best performance with the CPU time assigned to the VM and 2) keep adverse effects low at the same time. DASEC faces two challenges: how to effectively control the priorities of vCPUs, and how to achieve the above two goals?

To address the challenge with effective control of vCPU priorities, we identify/create a few system parameters that have the most influence on the relative progress of vCPUs, since the progress of vCPUs is the most important factor determining whether the vCPUs may spend much time on waiting for each other. Note that the priority used in DASEC mechanism is different from the system priority of the vCPUs (e.g., the "nice" values in Linux systems), and we choose not to use the system priorities in DASEC mechanism for two reasons: 1) they are used by the system for other purposes, which we do not want to interfere with; and 2) they cannot provide the fine-grained control over the relative progress of vCPUs. In our design, we choose the following parameters: 1) *rescheduling latency* to control the time that the computation starts. In system designs, rescheduling latency is the parameter determining when a vCPU can be scheduled after it becomes "ready". In some systems (e.g., Linux), there is a system-wide rescheduling latency for all the vCPUs; we need to modify the system to create a private rescheduling latency for each vCPU. 2) *time slice* of a vCPU to control how much progress a vCPU can make after it is scheduled to run. In some rare cases, when the scheduler finds that these two parameters cannot effectively control the priority of a vCPU, it also checks whether the execution of the vCPU may be interrupted by other vCPUs and takes this factor into account.

To address the second challenge, we use a step-by-step iterative method to adjust the parameters above for the two goals, i.e., high performance and low adverse effects. Specifically, for the first goal, it is not possible to directly measure the end-to-end performance of the workload in a VM. Instead, we use the CPU time consumed by the VM as an indicator of the amount of progress made by the workload. This indicator is reliable because the CPU time consumed by effective computation is roughly proportional to the amount of finished computation, idling does not consume CPU time, and vCPUs that perform excessive busy waiting are preempted promptly and consume little CPU time. For the adverse effects, it is

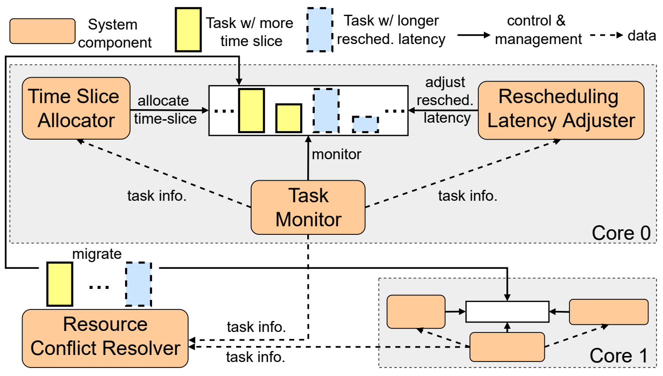

**Figure 6:** DASEC ***system architecture.*** **Key components are in orange.**

not realistic either to measure it in practice. Thus, instead of measuring it, we lower the priorities of the vCPUs under the condition that lowering the priorities will not reduce workload performance.

There are two challenging issues with adjusting the parameters to achieve the goals. One is whether the priority of the vCPU should be higher or lower, and the other one is which parameter should be adjusted. There are a few design choices for adjusting the parameters. For instance, the parameters can be adjusted based on the CPU utilization of the workload. If CPU utilization has been maximized within the CPU share of the VM, we adjust the parameters to lower the priorities; otherwise, we adjust the parameters to improve CPU utilization. However, this design has a scalability issue in maintaining a global CPU utilization for each VM, which might be high if each VM has a large number of vCPUs.

We choose to use a more scalable way, which adjusts the parameters of each vCPU based on the status of the vCPU: if increasing the priority of the vCPU helps improve performance, we increase the priority; otherwise, we reduce the priority to reduce adverse effects. We check the status of the vCPUs at the beginning of each period when the virtual machine monitor (VMM) is about to allocate new time-slices to the vCPUs of a VM, and based on the status adjust its parameters gradually. For example, if a vCPU has fully utilized its time-slice in the previous period and the vCPU is in a "ready" state, meaning that it could have made more progress if there were more CPU time, we will allocate more CPU time in the coming period to improve performance. We introduce the detailed design and the components for adjusting the parameters below.

## 4.2 Design Details

DASEC includes four key components as shown in Figure 6. (1) A time-slice allocator (TSA) component for dynamically adjusting the time-slice distribution between the vCPUs of each VM. (2) A rescheduling latency adjuster component for dynamically adjusting rescheduling latency. (3) A resource conflict resolver (RCR) component. (4) A task monitor component for collecting each vCPU's runtime information. The first two components reduce conflicts and the RCR component detects and resolves the conflicts that cannot be reduced by the first two components. For the first two components, we focus on introducing how the adjustment decisions are made, since enforcing the decisions is straightforward and system-dependent.

- **The time-slice allocation component** checks the amount of time-slice consumed by each vCPU periodically and uses the amount of time-slice consumed by the vCPUs in the previous period to adjust the amount of time-slice to be allocated to the vCPUs in the upcoming period[2]. Specifically, for a vCPU that has been preempted earlier due to the depletion of time slice, the component increases its time-slice. For other vCPUs, since they still have unused time-slice at the end of the period, there is no need to further increase their time-slice. The component assigns a weight to each vCPU. To increase the time-slice of a vCPU, the component increases the weight by 10%. The component keeps the total weight of the vCPUs in a VM fixed. Thus, it reduces the weight of other vCPUs accordingly based on their original weights.

- **The rescheduling latency adjustment component** looks at whether the vCPU has consumed its time-slice and whether the vCPU can still make progress at the end of each period. For a vCPU that is in a "ready" or "running" state at the end of the previous period, the vCPU cannot consume its time-slice quickly. This may be caused by the rescheduling delay. Thus, the component decreases the rescheduling latency of the vCPU by 10%. For the vCPUs that have consumed their time-slice and become idle at the end of the previous period, the component increases their rescheduling latencies.

There are scenarios, in which a vCPU with a low rescheduling latency has tasks depending on the completion of the tasks on other vCPUs with high rescheduling latencies. Since the tasks on the vCPUs with high rescheduling latencies complete late, the task on the vCPU with a low rescheduling latency cannot start early. Thus, it is possible that the vCPU with a low rescheduling latency still cannot consume its time-slice, no matter how its rescheduling latency is reduced. To detect such scenarios, when the rescheduling latency of a vCPU has been reduced to a minimal value allowed by the system, if a vCPU still cannot consume its time slice, the component assumes that the vCPU may be delayed by other vCPUs with high rescheduling latencies. To pin-point these vCPUs, the component uses wake-up inter-processor interrupts (IPIs) sent to the vCPU as indicators to find out the source vCPUs sending out the IPIs. Then it reduces the rescheduling latencies of these source vCPUs.

[2]The vCPU runtime information is provided by the TM component.

• **Resource Conflict Resolver.** The adjustment of time-slice distribution and rescheduling latencies of vCPUs effectively make the vCPUs have asymmetric and low priorities. They significantly reduce conflicts. However, conflicts cannot be completely avoided by these two components. The last component detects and tries to resolve such conflicts by migrating vCPUs between cores. For DASEC mechanism, conflicts arise when the total amount of time-slice allocated to the vCPUs scheduled on the same core exceeds the core's capacity. For example, a conflict arises when, in a time period of 80ms, each of two vCPUs scheduled on the same core is allocated with a 50ms time-slice. The vCPUs with low rescheduling latencies may also have conflicts. A conflict arises when a core is running a vCPU with low rescheduling latency and another vCPU with low rescheduling latency becomes ready to run. If the former vCPU is preempted promptly, its task is essentially delayed since the task cannot be finished quickly. If the former vCPU is not preempted promptly, the latter vCPU cannot be rescheduled quickly.

RCR tries to resolve conflicts by adjusting the layout of vCPUs on physical cores. Since adjusting vCPU layout is costly, RCR performs the adjustment in a conservative way. Specifically, to detect and resolve conflicts caused by high demands for CPU time, after the time-slice allocation component has adjusted the amounts of time-slice to be allocated to each vCPU, for each core, RCR calculates an aggregated amount of time-slice for the vCPUs scheduled on the core. Then, RCR finds out the core with the largest aggregated amount and the core with the smallest aggregated amount. If the largest aggregated amount is greater than the smallest aggregated amount by 10%, RCR tries to balance the aggregated amounts by swapping some of the vCPUs on the two cores.

To detect and resolve conflicts caused by the vCPUs with low rescheduling latencies, after the rescheduling latency adjustment component has adjusted the rescheduling latency of each vCPU, RCR categorizes the vCPUs into two groups based on their rescheduling latencies — vCPUs with low rescheduling latencies and vCPUs with high rescheduling latencies. In each period, RCR monitors the execution of the vCPUs with low rescheduling latencies. It counts the number of times that these vCPUs are preempted and the number of times that these vCPUs are not scheduled after they become ready and have waited a long time exceeding their rescheduling latencies. After the period, it uses the total number as the number of conflicts on the core caused by the vCPUs with low rescheduling latencies. Then, RCR finds out the core with the most conflicts and the core with the fewest conflicts. If the difference between the numbers of conflicts exceeds a threshold (2x in implementation), RCR selects half of the vCPUs with low rescheduling latencies on the core with the most conflicts and half of the vCPUs with high rescheduling latencies on the cores with the fewest conflicts, and then swaps the vCPUs. Low thresholds increase vCPU migration overhead. High thresholds "cripple" the conflict resolver. Thus, we measured how vCPU migrations reduce with increased thresholds, and selected the threshold values at knee points to make trade-off.

| App. | Workload description |
|---|---|
| `Image-classify` | Image classification on ImageNet [9]. |
| `Action-recognize` | Video action recognition [10, 11]. |
| `Img-dnn` | Handwriting recognition based on OpenCV [13]. |
| `Masstree` | In memory K/V store with 50% GET and 50% PUT [12]. |
| `Silo` | In-memory transactional database with TPCC [14]. |
| `Memcached` | Serve requests (random keys,50% SET,50% GET) [15]. |

**Table 2: Programs and workloads used to test DASEC.**

## 5 Implementation Details

DASEC incorporates four main components: Task Monitor (TM), Time Slice Allocator (TSA), Rescheduling Latency Adjuster (RLA), and Resource Conflict Resolver (RCR) as shown in Figure 6. TM periodically monitors the state and events of each task, such as remaining time slice, inter-process-interrupts (IPI), number of context switches, and state transitions. TSA assigns a time slice to each task for the upcoming time period. To implement TM and TSA, we change the Linux completely fair scheduler (CFS [27]) and leverage the Linux Proc file system interface [28]. Specifically, we collect each task's on-core execution time recorded in Linux CFS, obtain the task running state by checking the Linux run-queue and wait-queue, and read IPI/context-switch numbers by adding new IPI/context-switches parameters in the Proc file system. We adjust the weights in Linux CFS to give the tasks on critical paths higher weights. Linux CFS only provides a process-level interface. We add a thread-level interface to allow DASEC to adjust the weight of a thread and to collect thread-level statistics.

RLA adjusts thread rescheduling latencies. To implement RLA, we adjust the wakeup latency (sched_wakeup_granularity_ns [29]) of each task in Linux. Linux CFS only has a system-wide wakeup latency parameter. We extend the implementation in CFS to create a per-task wakeup latency parameter. RCR resolves time slice conflicts and rescheduling latency conflicts. It is implemented by setting the core affinities of the tasks.

## 6 Evaluation

We evaluated DASEC by answering the following questions.

§6.1 and §6.2: What is DASEC's performance?

§6.3: How much performance improvement can be achieved with DASEC, compared with related systems?

§6.4: How effective is each technique in DASEC?

§6.5: What is DASEC's applicability and overhead?

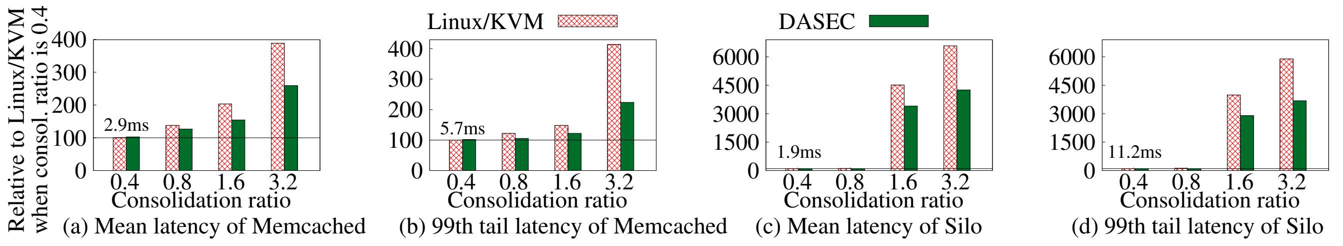

**Figure 7: *Latency of* `DASEC` *compared to vanilla Linux/KVM as consolidation ratio increases.* "Consolidation ratio" means the ratio between the total number of vCPU and the total number of pCPU in the system. For instance, consolidation ratio of "1.6" means there are 128 vCPUs (8 VMs) sharing the 80-core physical server; and each VM has 16 vCPUs, and each workload in the VM has 16 threads.**

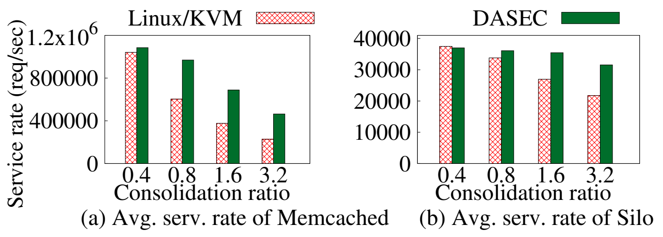

**Figure 8: *Service rate of* `DASEC` *compared to vanilla Linux/KVM as consolidation ratio increases.* Relative to vanilla Linux/KVM, `DASEC` greatly improves the service rate when the consolidation ratio is bigger than 1. QPS of `Memcached` and `Silo` is 10000 and 2000, respectively.**

**Experimental setup.** We conducted experiments on a HPE (Hewlett Packard Enterprise) ProLiant DL580 Gen10 server with four Intel Xeon Gold 6138 processors, 256GB memory, and two 2TB SSDs. Each processor has 20 cores. We created multiple VMs using Linux KVM [17] or multiple containers using docker [30]. Each VM has 16 vCPUs and 16GB memory. Each VM/container encapsulates one workload with 16 threads. Both host OS and guest OS are Ubuntu Linux 18.04 with the same Linux 5.3 kernel and software configuration, unless otherwise indicated. We implemented `DASEC` in the host OS and evaluated it with six real-world latency-sensitive workloads, including two AI inference workloads (`Image-classify` and `Action-recognize`) from `GluonCV` [31], three latency-sensitive workloads from `TailBench` [32], as well as `Memcached` workload, as summarized in Table 2.

Our experiments were conducted under two main settings: 1) homogeneous setting where VMs/containers ran the same workload; and 2) heterogeneous setting where VMs/containers ran different workloads. We chose these two settings as latency-sensitive workloads may be collocated with similar or different workloads in the edge server.

We compared `DASEC` with `PARTIES` [1] and `BVT` [4, 16]. `PARTIES` is the only system that coschedules multiple latency-sensitive workloads with best effort workloads on a physical server in data centers. For `BVT`, previous works [4, 33, 34] show that it can reduce the latency of one latency-sensitive workload when it is collocated with best effort workloads.

To show `DASEC`'s applicability, we ported `DASEC` into Google's user-level scheduler, `ghOSt`, and tested `DASEC`'s performance compared to vanilla `ghOSt` (see §6.5). In addition, we also tested `DASEC` when a container is used and compared `DASEC`'s performance to vanilla docker.

## 6.1 Same Workload in VMs

Figure 7 shows latencies of `Memcached` and `Silo` when `Linux/KVM` and `DASEC` are tested as the consolidation ratio increases under the homogeneous setting. Compared to `Linux/KVM`, `DASEC` reduces the mean latency and the 99th tail latency by 17% and 19% on average, respectively. As the consolidation ratio increases from 0.4 to 3.2, the performance advantage of `DASEC` also increases compared to `Linux/KVM`. When the consolidation ratio is low (e.g., 0.4), the physical core may not be shared by multiple vCPUs/threads, such that there is little chance for `DASEC` to improve performance. When the consolidation ratio is high (e.g., 3.2), `DASEC` offers 35% lower mean latency and 42% lower 99th tail latency on average, relative to the performance improvement of `DASEC` when the consolidation ratio is 0.4. This shows `DASEC`'s capability of reducing workloads' latencies when many latency-sensitive workloads are collocated in the edge server.

To understand how `DASEC` shows better performance compared to `Linux/KVM`, we collect service rates of `Linux/KVM` and `DASEC`, respectively, as the consolidation ratio increases. We show the results in Figure 8. On average, `DASEC` outperforms `Linux/KVM` by 42% in service rate. When the consolidation ratio is 0.4, `Linux/KVM` achieves almost the same service ratio as `DASEC` (3% higher on average). This explains how `DASEC`'s latency is almost the same as `Linux/KVM`'s latency when the consolidation ratio is low. As the consolidation ratio increases, `DASEC` improves the service rate by up to 204% compared to `Linux/KVM`. This explains why `DASEC` can further reduce latencies when the consolidation rate increases.

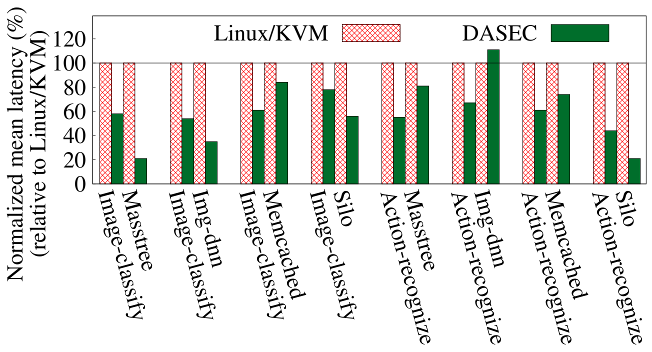

**Figure 9:** ***Mean latency of*** `DASEC` ***compared to vanilla Linux/KVM when heterogeneous workloads are collocated.*** **Relative to vanilla Linux/KVM,** `DASEC` **greatly improves the service rate when the consolidation ratio is 1.6.**

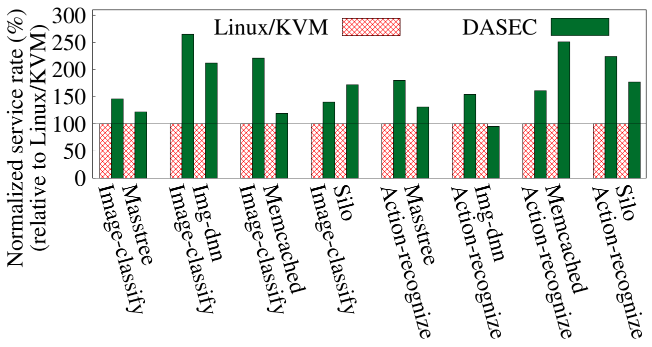

**Figure 10:** ***Service rate of*** `DASEC` ***compared to vanilla Linux/KVM when heterogeneous workloads are collocated.*** **Relative to vanilla Linux/KVM,** `DASEC` **greatly improves the service rate when the consolidation ratio is 1.6.**

To facilitate our understanding, we also collect the distribution when a few threads/vCPUs are in the ready state and all other threads of the workload are waiting for these threads. The distribution indicates how severe the critical path of the workload is delayed. We show the distribution in Figure 3. Figure 3 (c) and (d) show that the percentage of `Silo` and `Memcached`'s cumulative distribution in `Linux/KVM` is higher than it of `DASEC`. This indicates `Linux/KVM` suffers higher critical path delay and worse performance compared to `DASEC`. Figure 3 (c) also shows `Silo` suffers a 20% delay when there is one thread in the ready state. When there are two threads in the ready state, the percentage increases to around 60%. The big increase explains why the latency is significantly increased when the consolidation ratio increases from 0.8 to 1.6 as shown in Figures 7 (c) and (d), respectively.

## 6.2 Different Workloads in VMs

Figure 9 shows the mean latency of `Linux/KVM` and `DASEC` when the consolidation ratio is 1.6 under the heterogeneous

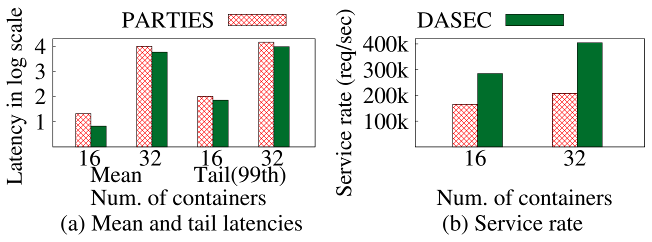

**Figure 11:** `DASEC` ***'s performance compared to*** `PARTIES`**. All containers run the same** `Masstree` **workload.**

setting. Under this setting, one VM executes the AI inference workload, while the other VMs execute the same latency-sensitive workload. We want to evaluate whether `Linux/KVM` may degrade the performance of the AI inference workload compared to `DASEC`. As shown in Figure 9, `DASEC` provides up to 56% (40% on average) lower mean latency compared to `Linux/KVM` on AI inference workload. For other latency-sensitive workloads, `DASEC` decreases the mean latency by 39% on average relative to `Linux/KVM`. On average, `DASEC` offers 23% lower mean latency under the heterogeneous setting compared to it under the homogeneous setting. This is because CPU contention is more severe under the heterogeneous setting, such that `DASEC` has more chance to improve performance. `DASEC` outperforms `Linux/KVM` in a similar trend when the consolidation ratio is 3.2. We only show the results when the consolidation ratio is 1.6. Since AI inference workloads do not report tail latencies, we only show the mean latencies under the heterogeneous setting.

To pinpoint why `DASEC` performs better than `Linux/KVM` on AI inference workloads under heterogeneous setting, we profile the GPU utilization during the execution of these workloads. We find that `DASEC` increases the GPU utilization by 25% on average compared to `Linux/KVM`. AI inference workloads usually preprocess some data set on CPUs before offloading tasks to GPUs. Once the preprocessing is delayed on CPUs due to CPU contention, the GPU utilization may be reduced. Since `DASEC` speedups preprocessing for AI inference workloads relative to `Linux/KVM`, it indirectly improves GPU utilization and AI inference workload performance. This is also reflected in Figure 10, which shows the service rates of AI inference workloads and other latency-sensitive workloads. On average, `DASEC` improves service rates of AI inference workloads and other latency-sensitive workloads by 86% and 60%, respectively, compared to `Linux/KVM`. In comparison to `Linux/KVM`, `DASEC`'s service rate improvement on latency-sensitive workloads other than AI inference workloads also explains why `DASEC` shows lower mean latency on these workloads as shown in Figure 9.

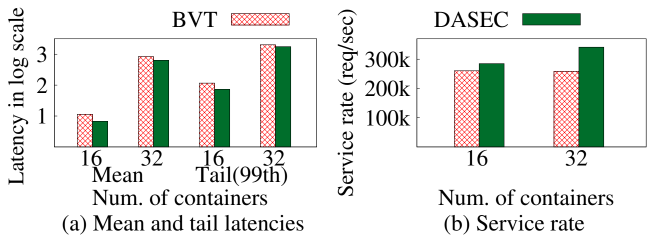

**Figure 12: DASEC's *performance compared to* BVT. All containers run the same Masstree workload.**

## 6.3 Comparison with Related Systems

Figure 11 shows the performance comparison results between PARTIES [1] and DASEC, as the number of containers collocated in the physical server increases. Relative to PARTIES, DASEC decreases the mean latency and the 99th tail latency by 44% and 31% on average, respectively. As the consolidation ratio increases, DASEC offers more performance improvement compared to PARTIES, up to 69% lower mean latency, 45% lower 99th tail latency, as well as 95% more service rate; and the service rate improvement trend explains the latency decrease trend. To pinpoint why DASEC shows more performance improvement as the consolidation ratio increases, we profile the context switches and CPU utilization per core during the execution of Masstree while running PARTIES and DASEC, respectively. As the number of containers increases from 16 to 32, PARTIES increases the context switches and CPU utilization per core by 30% and 26% on average; and DASEC achieves 59% lower context switches per core and 37% more CPU utilization per core compared to PARTIES.

Figure 12 shows DASEC's performance compared to BVT when the number of collocated containers in the server increases from 16 to 32. In comparison to DASEC, BVT increases the mean latency and the 99th tail latency by up to 27% and 32%, respectively. DASEC improves service rate by up to 30% compared to BVT. DASEC outperforms BVT for two main reasons. First, BVT cannot resolve interferences and causes long latencies when the consolidation ratio is high. This is also corroborated by the previous work [4]. Specifically, when multiple threads with very low rescheduling latencies (or very high time slice demands) are on the same core, BVT has no mechanisms to resolve the interferences. Second, BVT only provides interfaces to prioritize the thread with *the earliest effective virtual time* and borrow virtual time from its future CPU allocation, as well as weighted fair sharing and context switch allowance. However, it does not offer mechanisms to reduce interferences in the critical path like DASEC. This makes BVT less effective compared to DASEC, when multiple latency-sensitive workloads are collocated in the edge server.

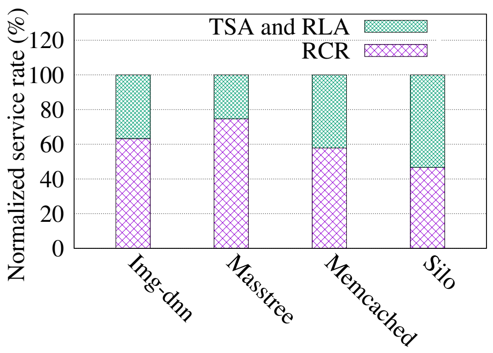

**Figure 13: *Performance breakdown of* DASEC. Service rate of each part is normalized to the total service rate of DASEC.**

## 6.4 Effectiveness of DASEC's Each Technique

Figure 13 shows the performance breakdown of DASEC under the homogeneous setting when the consolidation ratio is 1.6. DASEC's TSA (Time Slice Allocator) and RLA (Rescheduling Latency Adjuster) components contribute to the whole performance of DASEC by 40% on average. The remaining performance of DASEC is contributed by the RCR component. For some workloads (e.g., Memcached and Silo), TSA and RLA contribute more performance to DASEC's whole performance. This is because Memcached and Silo are quite vulnerable to the scheduling wait time. When using Linux/KVM Completely Fair Scheduler (CFS), these workloads suffer exceptionally long scheduling wait times as many Memcached or Silo workloads are collocated on the same edge server. This is also well corroborated by the previous work [4]. TSA and RLA can efficiently reduce the scheduling wait time and allocate more time slice for those workloads, such that these two components contribute more performance on those workloads. For other workloads (e.g., Masstree and Img-dnn), RCR contributes more performance to DASEC's whole performance. The reason is that when multiple Masstree workloads collocated in the same server, the main bottleneck comes from load imbalance, which may be caused by checkpoints run in parallel with request processing in each core as well as some limited resources contentions such as DRAM or interconnect bandwidth. RCR can help resolve such load imbalance among different cores and thus contributes more performance on those workloads.

## 6.5 Applicability and Overhead

Figure 14 shows the performance of docker and DASEC, respectively, as the number of containers collocated in the physical server increases. When the consolidation ratio is low (e.g., 16 containers), DASEC reduces the mean latency and the 99th tail latency by 39% and 38%, respectively, compared to docker. This is aligned with the service rate improvement, 35% compared to docker. As the consolidation

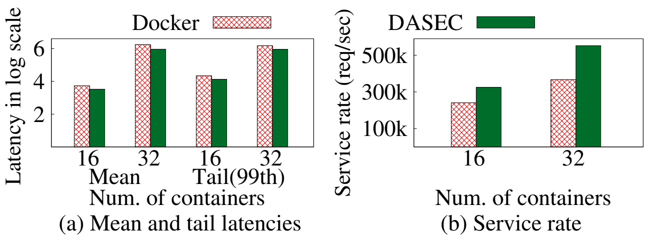

**Figure 14: DASEC*'s performance compared to* docker. We test the performance of docker and DASEC, respectively, when Masstree benchmark is running in each container.**

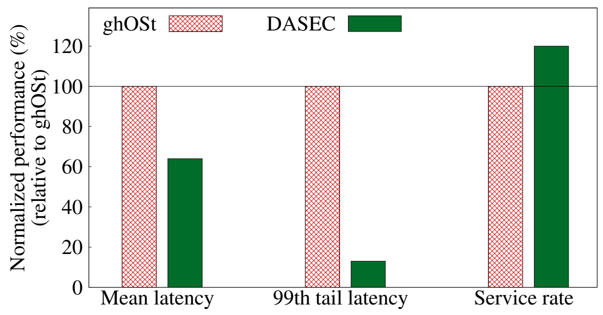

**Figure 15: DASEC*'s performance compared to* ghOSt. We test the performance of ghOSt and DASEC, respectively, when the consolidation ratio is 1.6. Eight containers are collocated in the physical server. In each container, we run Masstree benchmark with 16 threads.**

ratio increases, DASEC provides 48% lower mean latency, 41% lower 99th tail latency, and 51% more service rate compared to docker. This indicates that DASEC can decrease latencies when many latency-sensitive applications in containers are consolidated on the physical server in the edge clouds.

Figure 15 shows the performance of ghOSt and DASEC, respectively, when the consolidation ratio is 1.6. In comparison to ghOSt, DASEC offers 36% lower mean latency, 87% lower 99th tail latency, and 20% more service rate. ghOSt is a user-level scheduling, which only supports very simple scheduling mechanisms. We port DASEC's policies into ghOSt to make it dynamically allocate time slices, adjust rescheduling latency, as well as resolve conflicts, when multiple latency-sensitive applications are collocated on the same server in edge clouds.

Figure 16 shows the performance of DASEC compared to Linux/KVM, when the latency-sensitive workload is collocated with the background workload on the edge server. We run four Masstree instances; and in each instance, we run the Masstree workload in a 32-vCPU VM. We run MatrixMul [35] as the background workload. Compared to Linux/KVM, DASEC provides 13% more throughput for the background workload and 26% lower mean latency for the latency-sensitive workload. DASEC offers performance improvement for both the background workload and the

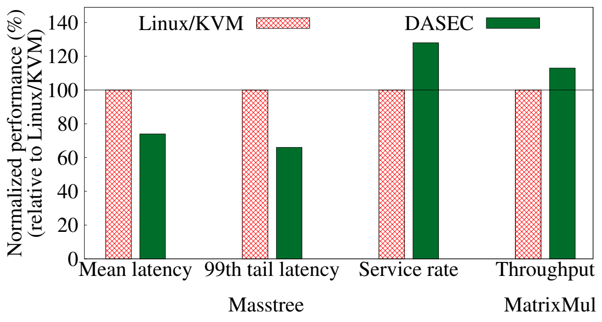

**Figure 16: DASEC*'s performance compared to* Linux/KVM *when latency-sensitive workload (*Masstree*) is collocated with background workload (*MatrixMul*) on the edge server.* We test the performance of Linux/KVM and DASEC, respectively, when the consolidation ratio is 1.6.**

latency-sensitive workload because they both perform communications in their executions. For Masstree, it is a lock intensive workload [12]. For MatrixMul, in each iteration of its execution, the main thread distributes computation tasks to other threads, which could not make progress if the main thread is delayed.

Figure 1, Figure 2, Figure 7, and Figure 8 show DASEC does not introduce much performance overhead (2% on average). When the consolidation ratio is low (e.g., 0.4), there is no space for DASEC to improve application performance as vCPUs/threads have dedicated cores. In these cases, DASEC's performance drops by up to 6%, compared to Linux/KVM. This shows DASEC introduces negligible overhead.

## 7 Related Work

**Workload collocation.** Workload collocation in public clouds and data centers has been extensively studied [1, 3, 4, 36–38]. Most of them at most coschedule one latency-sensitive workload with one or more best-effort workloads in a physical server. For instance, Leverich et al. [4] improves QoS of one latency-sensitive workload when it is collocated with best-effort workloads for high resource utilization in warehouse-scale data centers. Specifically, it identifies queuing delay, scheduling delay, and load imbalance as three key issues that can cause QoS degradation of the latency-sensitive workload. It adopts interference-aware provisioning to mitigate queuing delay, borrowed virtual time scheduling (BVT) [16] to minimize scheduling delay, and thread-pinning to solve threads load imbalance. Heracles [3] leverages hardware and software isolation mechanisms (e.g., resource partitioning) to improve server resource utilization by collocating one latency-sensitive workload with one or more best-effort workloads. It tries to guarantee the latency-sensitive workload' QoS and let the best-effort workloads utilize the idle resources.

Among all these related works, only PARTIES [1] allows multiple latency-sensitive workloads collocated with best-effort workloads on a physical server in data centers. PARTIES tries to improve server utilization without violating QoS of latency-sensitive workloads. Specifically, it detects and boosts allocation of one or more resources for the latency-sensitive workload whose latency suffers the most. It uses both OS and hardware level resource partitioning mechanisms available in modern platforms to allocate resources, such as thread pinning, cache partitioning, memory capacity partitioning, frequency scaling, and disk/network bandwidth partitioning. Once all latency-sensitive workloads meet their QoS targets, PARTIES reclaims excess resources from each sensitive workload and allocates to the background workloads to improve server resource utilization.

**User-level scheduling.** Many user-level scheduling mechanisms [18, 39–46] have been proposed to schedule latency-sensitive workloads in public clouds and/or data centers. Most of them allocate dedicated cores to one latency-sensitive workload collocated with one or more best-effort workloads in data centers and/or public clouds. Such resource partitioning approaches may lead to severe resource under-utilization in edge clouds [47], where resources are highly constrained. Moreover, in edge clouds, multiple latency-sensitive workloads may be consolidated in the edge server without collocating with best-effort workloads. DASEC targets how to collocate many latency-sensitive workloads on per edge server and reduces latencies of these workloads. Therefore, DASEC is orthogonal to user-level scheduling approaches. Meanwhile, DASEC's idea is general and can also work with user-level scheduling approaches. To prove this, we ported DASEC into Google's user-level scheduling (ghOSt [18]) and further improved its performance on scheduling multiple latency-sensitive workloads in edge clouds (see §6).

**Co-scheduling.** Co-scheduling [34, 48–52] is a widely-used approach for reducing the performance vulnerability of multi-threaded applications. Various coscheduling schemes [49, 51–53] have been designed for OSs and virtual machine monitors (VMMs). Coscheduling aims to reduce the execution delay of multi-threaded applications at their synchronization/communication points, which is the main cause of their performance vulnerability. The main idea is to maximize the co-running of collaborating threads; i.e., when a thread is scheduled to run, its collaborating threads should be scheduled as quickly as possible so as to run in parallel with the thread. To maximize the co-running of collaborating threads, most co-scheduling schemes temporarily prioritize these threads, such that they can preempt the execution of other threads to get the cores to co-run and are less likely to be preempted by other threads during the co-running.

Unfortunately, the effectiveness of existing coscheduling approaches is seriously limited when used in multi-tenant edge clouds due to the following three reasons. First, when a system has two or more multi-threaded applications, the effectiveness of the existing coscheduling approaches is limited, because there are conflicting demands for prioritizing different applications on the same set of hardware. Existing approaches focus mainly on one multi-threaded application and lack a mechanism to resolve the conflicts from multiple multi-threaded applications. Even worse, existing coscheduling approaches indiscriminately prioritize the threads to be co-scheduled in each application, significantly increasing the likelihood of conflicts.

Second, existing coscheduling schemes cannot address well the trade-off between improving the effectiveness and reducing the notorious adverse effects of coscheduling. They may unnecessarily sacrifice effectiveness when trying to reduce the adverse effects. For example, relaxed coscheduling [51] reduces CPU fragmentation by coscheduling fewer threads; this is at the cost of lower application performance.

Third, the performance vulnerability problem becomes even more pronounced when applications have dynamic changing workloads (e.g., the workload variation caused by changing parallelism). The scheduler usually provisions time slices periodically at a fixed rate, and does not allow unused time slices in the periods with the light workload to be accumulated and used later when the workload is heavy. Thus, such workload changes are penalized. Existing coscheduling approaches lack a mechanism to deal with such a performance penalty.

**Other works.** Other works on improving workload QoS in public clouds and/or data centers include resource provisioning [25, 36, 54–62] and VM placement [63–69]. Resource provisioning and VM placement approaches mainly consider QoS in a coarse-grained manner. For instance, XEN and Co. [70] presents workloads performance degradation caused by rescheduling delay in XEN hyppervisor (e.g., Domain0 is not scheduled on time to process send-out or received packets), and it develops a communication-aware CPU scheduler, expecting to schedule delayed domains timely and fairly to mitigate such delay and improve performance. These approaches are orthogonal to DASEC.

**DASEC's novelty.** Since resources are highly constrained in edge clouds [2, 47, 71, 72], multiple latency-sensitive workloads may be collocated on each edge server to improve resource utilization (e.g., time sharing the CPU resources [47]). To improve each workload's performance in edge clouds, DASEC proposes dynamic asymmetric CPU scheduling to allocate CPU resources to collocated edge workloads. Existing works mainly target how to collocate one sensitive workload with one or more best-effort workloads in public clouds and/or data centers. Only PARTIES targets how to collocate multiple latency-sensitive workloads with one or more best-effort workloads in data centers. However, PARTIES allocates dedicated hardware resources to each workload without allowing workloads time sharing CPU resources. This can significantly reduce CPU resource utilization in edge clouds [47]. This can also greatly reduce workload performance when the consolidation rate is high (confirmed in §6.3).

## 8 Conclusion

The interference caused by resource sharing and the performance issues caused by such interference is a long-standing problem. The emerging edge computing poses new challenges to this problem. Edges are expected to be dominated by resource-hungry and latency-critical workloads. This calls for new solutions which can effectively control the interference between latency-critical workloads, not the interference between latency-critical workloads and best-effort workloads as that in clouds. The resources in an edge site are very limited compared to those in a cloud data center. Thus, the solutions must increase resource efficiency. We have investigated many solutions designed for non-edge scenarios and have not seen such solutions that can satisfy these needs for edges.

DASEC provides a unique approach to this problem. Though the paper only explored the techniques and validated the effectiveness for hiding the interference caused by sharing CPU cores, the exploration has demonstrated that it is a promising direction and has a good potential to manage the interference caused by sharing many other resources, such as sharing CPU caches, memory bandwidth, and I/O bandwidth. As future work, we plan to design new techniques that can be used to control such interference and integrate them with the task scheduler techniques introduced in the paper.

## 9 Acknowledgments

We thank the anonymous reviewers and the shepherd Brian Kroth for the constructive suggestions to improve the paper.

This research was supported, in part, by the National Science Foundation (USA) under Grant Numbers CNS−1948457 and CCF−1617749.

## References

[1] 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.

[2] Mengwei Xu, Zhe Fu, Xiao Ma, Li Zhang, Yanan Li, Feng Qian, Shangguang Wang, Ke Li, Jingyu Yang, and Xuanzhe Liu. From cloud to edge: a first look at public edge platforms. In *Proceedings of the 21st ACM Internet Measurement Conference*, pages 37–53, 2021.

[3] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 450–462. ACM, 2015.

[4] 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*, page 4. ACM, 2014.

[5] Christina Delimitrou and Christos Kozyrakis. Bolt: I know what you did last summer... in the cloud. *ACM SIGARCH Computer Architecture News*, 45(1):599–613, 2017.

[6] Harshad Kasture, Davide B Bartolini, Nathan Beckmann, and Daniel Sanchez. Rubik: Fast analytical power management for latency-critical systems. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 598–610. IEEE, 2015.

[7] Daniel Sanchez and Christos Kozyrakis. Vantage: Scalable and efficient fine-grain cache partitioning. In *Proceedings of the 38th annual international symposium on Computer architecture*, pages 57–68, 2011.

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

[9] Image classification on ImageNet. https://github.com/dmlc/gluon-cv/tree/master/scripts/classification/imagenet.

[10] Yi Zhu, Xinyu Li, Chunhui Liu, Mohammadreza Zolfaghari, Yuanjun Xiong, Chongruo Wu, Zhi Zhang, Joseph Tighe, R Manmatha, and Mu Li. A comprehensive study of deep video action recognition. *arXiv preprint arXiv:2012.06567*, 2020.

[11] Video action recognition. https://github.com/dmlc/gluon-cv/tree/master/scripts/action-recognition.

[12] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196, 2012.

[13] A deep network handwriting classifier. https://github.com/xingdi-eric-yuan/multi-layer-convnet.

[14] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

[15] Memcached Key-Value Store. https://memcached.org.

[16] Kenneth J Duda and David R Cheriton. Borrowed-virtual-time (bvt) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. *ACM SIGOPS Operating Systems Review*, 33(5):261–276, 1999.

[17] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium*, pages 225–230, 2007.

[18] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. ghost: Fast & flexible user-space delegation of linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 588–604, 2021.

[19] Qiangyu Pei, Shutong Chen, Qixia Zhang, Xinhui Zhu, Fangming Liu, Ziyang Jia, Yishuo Wang, and Yongjie Yuan.

Cooledge: hotspot-relievable warm water cooling for energy-efficient edge datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 814–829, 2022.

[20] AWS local zones. https://aws.amazon.com/about-aws/global-infrastructure/localzones/.

[21] Google edge cloud platform. https://cloud.google.com/blog/topics/hybrid-cloud/announcing-google-distributed-cloud-edge-and-hosted.

[22] Azure edge zones. https://www.datacenterdynamics.com/en/news/microsoft-and-att-launch-azure-edge-zone-in-atlanta/.

[23] Tencent edge computing center. https://cntechpost.com/2020/10/15/tencent-cloud-enables-first-5g-edge-computing-center/.

[24] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 301–312. IEEE, 2014.

[25] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Eurosys 2013*, 2013.

[26] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 18:1–18:17, 2015.

[27] Chandandeep Singh Pabla. Completely fair scheduler. *Linux J.*, 2009(184), August 2009.

[28] Linux Proc file system. https://www.kernel.org/doc/html/latest/filesystems/proc.html.

[29] Linux kernel scheduler parameters. https://access.redhat.com/solutions/177953.

[30] Docker container. https://www.docker.com/.

[31] Jian Guo, He He, Tong He, Leonard Lausen, Mu Li, Haibin Lin, Xingjian Shi, Chenguang Wang, Junyuan Xie, Sheng Zha, et al. Gluoncv and gluonnlp: deep learning in computer vision and natural language processing. *J. Mach. Learn. Res.*, 21(23):1–7, 2020.

[32] 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.

[33] Ludmila Cherkasova, Diwaker Gupta, and Amin Vahdat. Comparison of the three cpu schedulers in xen. *SIGMETRICS Perform. Eval. Rev.*, 35(2):42–51, September 2007.

[34] 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. ACM, 2009.

[35] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA '07)*, pages 13–24, 2007.

[36] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. Cpi2: Cpu performance isolation for shared compute clusters. In *EuroSys 2013*, pages 379–391. ACM, 2013.

[37] Haishan Zhu and Mattan Erez. Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 33–47, 2016.

[38] Xi Yang, Stephen M Blackburn, and Kathryn S McKinley. Elfen scheduling: Fine-grain principled borrowing from latency-critical workloads using simultaneous multithreading. In *USENIX Annual Technical Conference*, pages 309–322, 2016.

[39] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 95–109, October 1991.

[40] 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.

[41] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-defined scheduling across the stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 605–620, 2021.

[42] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. When idling is ideal: Optimizing tail-latency for heavy-tailed datacenter workloads with perséphone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 621–637, 2021.

[43] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. Efficient scheduling policies for {Microsecond-Scale} tasks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1–18, 2022.

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

[45] 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.

[46] 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.

[47] Junjue Wang, Ziqiang Feng, Shilpa George, Roger Iyengar, Padmanabhan Pillai, and Mahadev Satyanarayanan. Towards scalable edge-native applications. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, pages 152–165, 2019.

[48] John K Ousterhout et al. Scheduling techniques for concurrebt systems. In *ICDCS*, volume 82, pages 22–30, 1982.
[49] Hwanju Kim, Sangwook Kim, Jinkyu Jeong, Joonwon Lee, and Seungryoul Maeng. Demand-based coordinated scheduling for SMP VMs. In *ACM ASPLOS 2013*, pages 369–380, 2013.
[50] Orathai Sukwong and Hyong S Kim. Is co-scheduling too expensive for SMP VMs? In *EuroSys 2011*, pages 257–272. ACM, 2011.
[51] Scott Drummonds. Co-scheduling SMP VMs in VMware ESX server, 2008. http://communities.vmware.com/docs/DOC-4960.
[52] Eitan Frachtenberg, Dror G Feitelson, Fabrizio Petrini, and Juan Fernandez. Flexible coscheduling: Mitigating load imbalance and improving utilization of heterogeneous resources. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 10–pp. IEEE, 2003.
[53] Andrea C. Dusseau, Remzi H. Arpaci, and David E. Culler. Effective distributed scheduling of parallel workloads. *SIGMETRICS Perform. Eval. Rev.*, 24(1):25–36, May 1996.
[54] Pradeep Padala, Kang G Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant, and Kenneth Salem. Adaptive control of virtualized resources in utility computing environments. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 289–302. ACM, 2007.
[55] Pradeep Padala, Kai-Yuan Hou, Kang G Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 13–26. ACM, 2009.
[56] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European conference on Computer systems*, pages 237–250. ACM, 2010.
[57] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.
[58] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.
[59] Christina Delimitrou and Christos Kozyrakis. HCloud: Resource-Efficient Provisioning in Shared Cloud Systems. In *Proceedings of the Twenty First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2016.
[60] Bhuvan Urgaonkar, Prashant Shenoy, and Timothy Roscoe. Resource overbooking and application profiling in shared hosting platforms. *ACM SIGOPS Operating Systems Review*, 36(SI):239–254, 2002.
[61] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 5. ACM, 2011.
[62] Nedeljko Vasić, Dejan Novaković, Svetozar Miucin, Dejan Kostić, and Ricardo Bianchini. Dejavu: accelerating resource allocation in virtualized environments. In *ACM SIGARCH computer architecture news*, volume 40, pages 423–436. ACM, 2012.
[63] Dejan Novakovic, Nedeljko Vasic, Stanko Novakovic, Dejan Kostic, and Ricardo Bianchini. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *Proceedings of the 2013 USENIX Annual Technical Conference*, number EPFL-CONF-185984 in USENIX ATC'13, 2013.
[64] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *ACM SIGPLAN Notices*, volume 48, pages 77–88. ACM, 2013.
[65] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, pages 248–259. ACM, 2011.
[66] Ron C Chiang and H Howie Huang. TRACON: Interference-aware scheduling for data-intensive applications in virtualized environments. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 47. ACM, 2011.
[67] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding long tails in the cloud. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 329–341, Lombard, IL, 2013. USENIX.
[68] 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. ACM, 2017.
[69] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: reconciling scheduling speed and quality in large shared clusters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 97–110. ACM, 2015.
[70] Sriram Govindan, Arjun R Nath, Amitayu Das, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Xen and co.: communication-aware cpu scheduling for consolidated xen-based hosting platforms. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 126–136. ACM, 2007.
[71] Yuxin Ren, Guyue Liu, Vlad Nitu, Wenyuan Shao, Riley Kennedy, Gabriel Parmer, Timothy Wood, and Alain Tchana. {Fine-Grained} isolation for scalable, dynamic, multi-tenant edge clouds. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 927–942, 2020.
[72] Antonio Barbalace, Mohamed L Karaoui, Wei Wang, Tong Xing, Pierre Olivier, and Binoy Ravindran. Edge computing: the case for heterogeneous-isa container migration. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 73–87, 2020.