Workload Consolidation in Alibaba Clusters: The Good, the Bad, and the Ugly

Yongkang Zhang*
HKUST
Hong Kong, China

Yinghao Yu
Alibaba Group
Hangzhou, China

Wei Wang*
HKUST
Hong Kong, China

Qiukai Chen
Alibaba Group
Hangzhou, China

Jie Wu
Alibaba Group
Hangzhou, China

Zuowei Zhang
Alibaba Group
Hangzhou, China

Jiang Zhong
Alibaba Group
Hangzhou, China

Tianchen Ding
Alibaba Group
Hangzhou, China

Qizhen Weng*
HKUST
Hong Kong, China

Lingyun Yang*
HKUST
Hong Kong, China

Cheng Wang
Alibaba Group
Hangzhou, China

Jian He
Alibaba Group
Sunnyvale, CA, USA

Guodong Yang
Alibaba Group
Hangzhou, China

Liping Zhang
Alibaba Group
Hangzhou, China

ABSTRACT

Web companies typically run latency-critical long-running services and resource-intensive, throughput-hungry batch jobs in a shared cluster for improved utilization and reduced cost. Despite many recent studies on workload consolidation, the production practice remains largely unknown. This paper describes our efforts to efficiently consolidate the two types of workloads in Alibaba clusters to support the company's e-commerce businesses.

At the cluster level, the host and GPU memory are the bottleneck resources that limit the scale of consolidation. Our system proactively reclaims the idle host memory pages of service jobs and dynamically relinquishes their unused host and GPU memory following the predictable diurnal pattern of user traffic, a technique termed *tidal scaling*. Our system further performs node-level micro-management to ensure that the increased workload consolidation does not result in harmful resource contention. We briefly share our experience in handling the surging traffic with flash-crowd customers during the seasonal shopping festivals (e.g., November 11) using these "good" practices. We also discuss the limitations of our current solution (the "bad") and some practical engineering constraints (the "ugly") that make many prior research solutions inapplicable to our system.

CCS CONCEPTS

• **Information systems → Enterprise resource planning**.

KEYWORDS

workload consolidation, cluster management, scheduling

ACM Reference Format:
Yongkang Zhang, Yinghao Yu, Wei Wang, Qiukai Chen, Jie Wu, Zuowei Zhang, Jiang Zhong, Tianchen Ding, Qizhen Weng, Lingyun Yang, Cheng Wang, Jian He, Guodong Yang, and Liping Zhang. 2022. Workload Consolidation in Alibaba Clusters: The Good, the Bad, and the Ugly. 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/10.1145/3542929.3563465

*Work done while the author was visiting Alibaba, Hangzhou, China.

SoCC '22, November 7–11, 2022, San Francisco, CA, USA

ACM ISBN 978-1-4503-9414-7/22/11…$15.00
https://doi.org/10.1145/3542929.3563465

1 INTRODUCTION

Alibaba Group [22] is a web giant with businesses in e-commerce, cloud computing, digital economy, etc. Like other IT companies, Alibaba runs two types of workloads in large clusters to support its businesses: (1) latency-critical (LC) services deployed in long-running containers (e.g., web search, e-commerce, machine learning (ML) inference), and (2) resource-intensive batch processing jobs that emphasize high throughput over low latency (e.g., big data analytics and ML training).

Instead of running LC and batch workloads in separate clusters, the common practice is to colocate them into *shared clusters* (i.e., *workload consolidation*) for improved utilization and reduced cost [47, 49]. Yet, our operational experience in Alibaba clusters highlights the following challenges.

Stable memory footprints of diurnally changing LC services. Most of the LC services running in our clusters are to support the company's e-commerce business whose traffic exhibits a clear diurnal pattern with a large peak-to-valley ratio. This results in the diurnally changing CPU and GPU utilization of LC services. The underutilized computing resources can then be used to run batch jobs at night. However, the host and GPU memory footprints of LC services remain at a high level even at night, leaving only a limited amount of memory for the colocated batch jobs.

Frequent micro-scale utilization fluctuations. Frequent jitters in CPU and memory bandwidth utilization are widely observed in our applications. In the context of workload consolidation, the fluctuating resource utilization of one service not only undermines its own performance, but could also interfere with the other colocating jobs.

Surging traffic brought by large seasonal sales. Alibaba hosts several large-scale *seasonal shopping festivals* (SSF) every year, with the largest one on November 11. Timed promotions are offered during the shopping festival to attract more customers to place orders online. This can create flash crowd traffic in a few minutes with peak load orders of magnitude higher than the usual time.

Improving resource efficiency by means of auto-scaling and workload consolidation is a hot topic that has been extensively studied in recent years [26, 27, 32, 44, 51, 55, 59]. Yet, these studies cannot be applied to our system:

Practical issues that limit the design options. Our system must be generally applicable to the diverse applications running in Alibaba clusters, which are developed based on different technology stacks for a range of computing tasks. The applications are also evolving rapidly to support fast-growing businesses. These constraints preclude most profiling-based, application-specific solutions [3, 31, 37, 39, 50].

Making fast and quality scheduling decisions at scale. In our clusters, a large portion of jobs has complex placement constraints [1, 54], making scheduling an NP-hard problem. Nevertheless, we still require the scheduler to make high-quality decisions for a large number of jobs in sub-seconds. Existing works solve the constrained scheduling problem using integer linear programming [20, 48], feedback control [32], and learning-based approaches [3, 12, 14, 31, 37, 39, 41, 50, 56]. However, none of them can handle the scheduling problem at our scale, especially during SSFs when the flash crowd traffic arrives within a few minutes.

Despite the challenges above, Alibaba's business scenario has many *predictable* events and patterns that can be exploited for improved resource planning and scheduling, such as the diurnal pattern of resource utilization in the usual time and the arrival of flash crowd traffic in SSFs.

In this paper, we describe the key techniques we developed to address the aforementioned challenges, and share our experiences in achieving high resource utilization while maintaining good service quality at scale in our clusters. This paper is organized as follows:

In §2, we briefly introduce the LC and batch workloads running in Alibaba's clusters to support its e-commerce businesses, as well as the cluster management system.

In §3, we show that the user traffic of LC services follows a diurnal pattern with a large peak-to-valley ratio. We then describe how to exploit this pattern to overcommit the idle compute resources to batch jobs at night, and more importantly, how we address the resource bottleneck on the main and GPU memory in overcommitment by means of *proactive memory reclamation* and *tidal scaling*.

In §4, we illustrate how the load jitters commonly observed in our clusters may harm the applications' performance in terms of two key resources: CPU and memory bandwidth. Accordingly, we present two solutions deployed to address these problems: the shared CPU pool with burstable CPU quota and memory bandwidth isolation using Intel's Dynamic Resource Control (DRC) available on IceLake processors [59].

In §5, we briefly explain workload scheduling in SSFs, a special but important business scenario, and its unique challenges. We then describe how we use the previously developed techniques to address those challenges. We briefly survey related work in §6 and conclude the paper in §7.

2 BACKGROUND

In this section, we give an overview of Alibaba's workloads and its cluster management system. We also explain our principles for designing and optimizing our infrastructure.

2.1 Workloads in Alibaba's Clusters

As one of the largest IT companies in the world, Alibaba Group has built dozens of large clusters (data centers) around the globe, where the number of machines in each cluster ranges from a few hundred to more than 10k. There are hundreds of thousands of machines in total, with tens of millions of CPU cores and tens of thousands of GPUs. These clusters run millions of service instances. Fig. 1 gives an overview of Alibaba's businesses and infrastructure. There are mainly two types of workloads running in Alibaba's clusters:

1) *Latency-critical (LC) services.* These services run in long-lived containers and provide user-facing services for

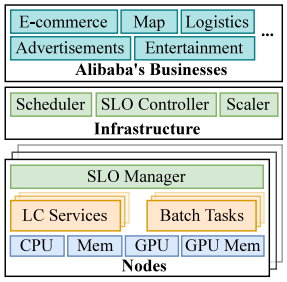


Figure 1: An overview of Alibaba's businesses and infrastructure.

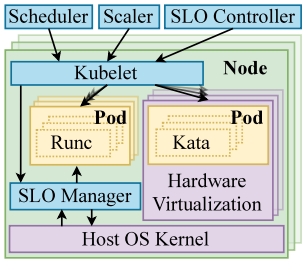


Figure 2: Illustration of Alibaba's cluster management system.

Alibaba's businesses, including e-commerce, web search, promotion, advertisements, autonomous driving, logistics, maps, entertainment, etc. We deploy LC services typically as microservices, and develop user-facing applications mainly based on Java and SpringBoot.

2) *Throughput-hungry batch processing jobs.* These are data-parallel jobs that emphasize high throughput over low latency. Examples include MapReduce and Spark query jobs for big data processing, and distributed machine learning.

Both LC services and batch jobs have complex *placement constraints*. These constraints may specify a fixed set of CPU cores allocated to an instance, limit the number of instances of an application on each machine, and disallow an instance to run on multiple CPU cores at different sockets. Our measurement shows that around 70% of application instances can meet their placement constraints on only 20% of the cluster nodes. LC services and batch jobs are often colocated on the same machines for improved utilization and reduced cost. Our applications are rapidly evolving, driven by the fast-changing market.

2.2 Alibaba's Cluster Management System

Overview. We use Alibaba's cluster management system to manage the compute resources in a cluster, including CPUs, host memory, GPUs, self-designed accelerators, etc. The system is developed based on Kubernetes [18], with many customized features and optimizations. Similar to Kubernetes, an instance of an application runs in clusters as a set of containers (aka, a *pod*). Specifically, there are two types of containers running in our clusters: Runc container, a conventional container based on Linux's `cgroup` and namespace, and Kata container [19], a secure container running in a lightweight virtual machine. The system encompasses all of Alibaba's core businesses and provides the following services: 1) workload scheduling, 2) auto-configuration for instances' resource specifications (e.g., memory size, number of CPU cores, etc.), 3) load balancing, 4) performance monitoring, 5) managing applications' Service-Level Objectives (SLOs), and 6) resource provisioning with diverse SLOs.

System architecture. Fig. 2 shows an architectural overview of Alibaba's cluster management system. Similar to Google's Borg [47, 49], our system is based on a master-slave architecture. Each cluster runs a few replicas of the master with three main components, a scheduler, a scaler, and an SLO controller. They handle the requests submitted by users and manage the status of all objects in the cluster (e.g., pods, nodes, etc.). Each computing node runs a local agent, Kubelet, which is in charge of 1) collecting and reporting the status, resource usage, and heartbeats of pods running on the node, and 2) forwarding master's requests (e.g., launching a pod on the node).

Scheduler. Similar to Kubernetes, the scheduler matches and places pods to a node based on their resource demands, label constraints, and other specifications. It supports both online scheduling (sequentially placing each pod following its arrival order) and offline orchestration of a large number of instances (§5.2).

Scaler. The scaler elastically adjusts the number of instance replicas and/or the resource quota of running applications based on relevant metrics (e.g., resource utilization, latency) or some pre-specified plans (§3.3 and §5.2).

SLO manager. The SLO manager ensures that each application can meet its SLOs without using more resources than allocated. It dynamically adjusts each application's OS parameters (e.g., CFS quota), and evicts pods on demand (e.g., when the machine runs out of memory), which we will explain in detail in §3.2, §4.1, §4.2, and §5.3.

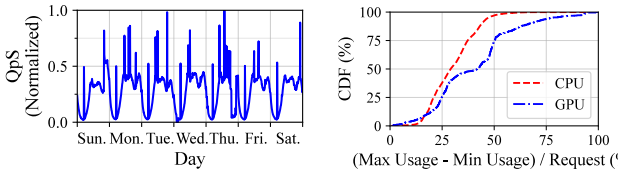
2.3 Design Principles

The goal of workload management is to reduce the resource provisioning cost without violating the performance SLOs of applications. In the context of Alibaba's scenario, a practical solution must follow the following design principles:

Transparent to applications. Our clusters run a large number of applications developed by dozens of business units. Optimizations intrusive to applications (e.g., change of code) are difficult to adopt in practice. Furthermore, given the frequent updates of applications, establishing an accurate performance profile for each application becomes elusive – in fact, many LC applications have no clear SLOs, and their performance cannot be easily measured. Therefore, optimizations of workload management must be made transparent to applications.

Generally applicable. Applications running in our clusters are built atop a variety of technology stacks. Therefore, any optimization used in the cluster management system should not target a specific application, but generally apply to a range of services and frameworks.

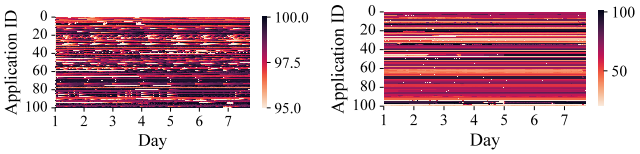
Following these principles, our system performs macro- and micro-management at the cluster and node levels, respectively, which we explain in detail in the following sections.



(a) The 7-day trace of an LC service's QPS (normalized by the peak QPS in the 7-day window).

(b) The diurnal inequality of CPU and GPU utilization of LC services with seasonality > 0.95.

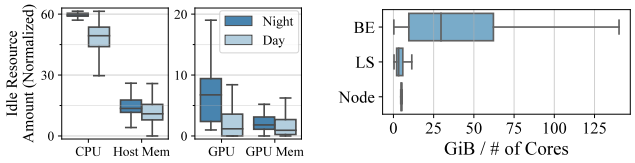
Figure 3: Illustration of diurnal pattern in LC services.



(a) Host Memory

(b) GPU Memory

Figure 4: 7-day utilization ($\frac{Usage}{Request}$, %) of host memory and GPU memory of LC services.



(a) The amount of idle CPU, GPU, host memory and GPU memory, normalized by the average amount of corresponding resource requested by batch jobs.

(b) Distribution of (Requested memory / # requested CPU cores) of LC services (*LS*) and batch jobs (*BE*), and (memory capacity / # of CPU cores) of machines (*Node*).[3]

Figure 5: Host memory and GPU memory are the bottlenecks of overcommitment. The box represents the 25th to 75th percentile, and the line in the middle of the box represents the median. The whiskers extend to 1.5 times the interquartile range.

3 CLUSTER-WIDE MACRO-MANAGEMENT

In this section, we show through workload analysis that LC services have diurnally changing loads during the off-season (no SSF). However, their memory footprints stay relatively stable, making memory the bottleneck resource that limits the scale of workload consolidation with batch jobs. We present our solutions that address this problem with cluster-wide macro-management and evaluate their effectiveness in production clusters.

3.1 The Problem of Overcommitment

Diurnally changing LC services. To illustrate the diurnal pattern of LC services, we refer to Fig. 3a, which shows the request load (queries per second or QPS) of a core e-commerce service in a 7-day trace. We measure a large peak-to-valley ratio of over 10x for this service. Accordingly, its CPU utilization also changes diurnally following the same pattern. To demonstrate the prevalence of this observation, we measure the *diurnal inequality*[1] and *seasonality*[2] of the CPU and GPU utilization of all instances of LC service in a cluster. Around 62% of CPU instances and 21% of GPU instances have a seasonality larger than 0.95. For those instances, the median values of diurnal inequality of CPU and GPU utilization are around 27% and 45%, respectively (Fig. 3b).

The prevalent diurnal pattern and the large peak-to-valley difference suggest significant utilization benefits by means of overcommitting the underutilized compute resources of LC services to batch jobs at night. Borrowing the ideas from Borg [49], we estimate the future CPU and memory utilization (i.e., *resource reservation*) of each LC instance and overcommit the remaining idle resources to batch jobs.

The memory bottleneck. Unlike CPUs and GPUs, the host and GPU memory footprints of LC services stay largely stable (Fig. 4): in our clusters, almost all LC services hold a constantly large amount of host memory (GPU memory) throughout a day. More than 90% of the available host memory space requested by LC services is occupied. As a result, there is no significant temporal variation in *resource reservation* of the host memory (and GPU memory) of LC services. The explanations for this phenomenon are: 1) the majority of Alibaba's LC services are written in Java (§2.1), where JVM tends to reserve a large amount of memory capacity for data caching; and 2) a portion of the GPU LC services always occupy a fixed amount of GPU memory regardless of how heavy the load is.

We estimate the *overcommitable space* of CPU, host memory, GPU, and GPU memory during the day and night by ($\frac{The\ amount\ of\ idle\ resources}{The\ average\ amount\ of\ resources\ requested\ by\ batch\ jobs}$), and depict the results in Fig. 5a. The overcommitable space of the CPU and GPU has a large gap between day and night, whereas the host memory and GPU memory do not. Moreover, Fig. 5a shows that, compared to the host memory, the CPU can be overcommitted to 3–4x more batch jobs during

[1]The diurnal inequality is defined as the peak-to-valley difference, i.e., (*the maximum utilization in the daytime*) - (*the minimum utilization at night*). A larger value indicates more underutilized resources at night.

[2]The seasonality [52] of a time series Y_t is measured by $1 - \frac{Var(E_t)}{Var(E_t+S_t)}$, where S_t and E_t are the seasonal component and the remainder component after the seasonality decomposition of Y_t respectively. Its value is within $[0, 1]$, and a larger value indicates a stronger seasonal temporal pattern.

[3]Although machines are heterogeneous in Alibaba's clusters, they generally have the same memory–CPU ratio (e.g., 128 GiB : 20 Cores).

both the day and night. This is because batch jobs usually demand more memory than LC services. In particular, we measure the requested amount of host memory normalized by the requested number of CPU cores for LC services and batch jobs, as well as the total capacity of host memory per CPU core of all machines. The results are depicted in Fig. 5b. Compared to LC services, batch jobs request 5–10x more memory, far exceeding the memory-CPU ratio of the machine. This makes memory a severe bottleneck when colocating the two types of workloads.

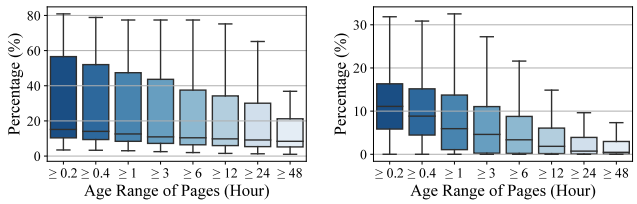
Fine-grained vertical scaling is insufficient. A simple solution to address the memory bottleneck is vertical scaling: it reduces the allocation quotas of the host and GPU memory for LC services at night and increases them to the normal level in the daytime. However, unlike CPUs, memory is non-elastic, in that the host and GPU memory, once allocated, cannot be safely relinquished without interrupting the application execution.

Horizontal scaling cannot be directly applied. An alternative solution is horizontal scaling, which dynamically reduces the number of LC instances at night and increases it during the daytime. However, our experience suggests that frequently scaling LC instances up and down results in increasingly fragmented resources, especially when there are complex placement constraints.

3.2 Memory Reclamation

Motivation. Our LC services are mainly written in Java and occupy a large number of memory pages that are infrequently accessed (as described in §3.1). Those cold memory pages should be timely reclaimed to make room for batch jobs. Although directly overcommitting more memory to batch jobs can trigger the Linux kernel to reclaim more pages from LC services, this is undesirable as it makes the system frequently undergo memory pressure and indiscriminately reclaim memory pages from running applications, significantly increasing the page fault rate and harming the application performance. Borrowing the ideas from Lagar-Cavilla et al. [29] and Corbet [9], our system keeps track of the idleness of each page frame and proactively reclaims all inactive pages.

Tracking memory idleness. Following Google's `kstaled` [30], we define the *age* of a memory page frame as the time elapsed since its last access. An older *age* indicates that the corresponding page frame is less frequently accessed by processes. We added a module, `kidled` [24], into the Linux kernel to periodically mark the age of reclaimable pages (i.e., swappable anonymous pages and clean file pages). Each memory cgroup maintains a histogram to count the number of pages within different age intervals. We collect a one-week trace of the age of reclaimable memory occupied by



(a) Swappable Anonymous Page (b) Clean File Page

Figure 6: The distribution of reclaimable idle memory (the amount of reclaimable memory / the total memory usage) of LC services running on each machine in a cluster by the age (last access time) of memory pages. The box and the whiskers have the same meaning as in Fig. 5.

LC services in a production cluster, and depict their distribution in Fig. 6. For anonymous pages (file pages), in the median, around half of the reclaimable pages have an age older than 48 (3) hours, indicating a significant number of inactive pages.

Proactive memory reclamation. We implemented a kernel module, `kreclaimd`, to proactively reclaim idle pages at regular intervals. Specifically, it periodically reclaims the page cache with an age older than T minutes. Our system reclaims swappable anonymous pages and clean file pages of all running LC services and batch jobs: it swaps anonymous pages to the swap space of a non-volatile storage device (which is cheaper but slower than DRAM) and directly drops the clean file pages.

Determining the reclamation threshold. Dynamically tuning the reclamation threshold in the production environment is risky as it may harm the system's stability. Therefore, we select a group of representative, memory-sensitive workloads and apply memory reclamation with different thresholds. Specifically, we measure the page fault rate, disk read cost[4], and application response time when swappable anonymous pages and clean file pages are reclaimed with different thresholds. The experimental result, depicted in Fig. 7, shows that as the reclamation threshold decreases, the page fault rate increases by 15.3×, while the amount of reclaimable anonymous page and file page increase from 14.7% to 29.8%, and from 2.2% to 12.6%, respectively (Fig. 6). Furthermore, compared with the reclamation of clean file pages (which does not require swapping memory pages back to the disk), reclaiming swappable anonymous pages significantly increases the reading cost, causing a more negative impact on running applications.

Our experiment also suggests that aggressive, dynamic thresholds only lead to limited benefits. As enabling dynamic

[4] *Disk read cost* = (*Averaged disk read latency*) × (*Averaged disk read operation count per second*).

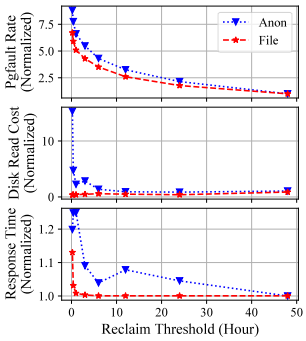


Figure 7: Reclaiming the anonymous and clean file pages with different thresholds. Results are normalized by the baseline (no reclamation).

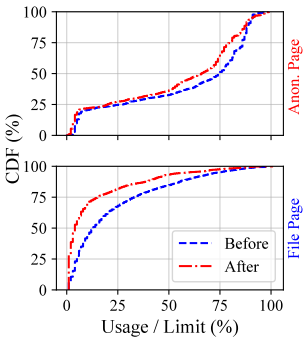


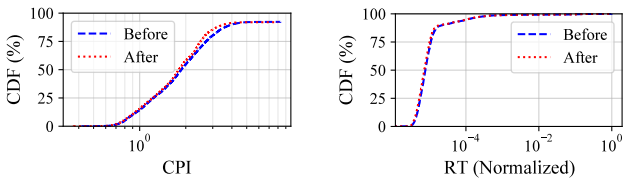
Figure 8: The anonymous page and file page utilization (usage / memory limit) of all LC instances in a cluster before and after memory reclamation.

threshold requires extensive workload profiling and monitoring, its performance gains cannot justify the overhead and risks. Therefore, we choose to set static reclamation thresholds, and to reclaim clean file pages more aggressively. In practice, our system only reclaims swappable anonymous pages with an age older than 6 hours, and clean file pages with an age older than 1.5 hours.

Detecting memory pressure. While memory reclamation enables the colocation of more batch jobs, the increased memory pressure may harm the LC services. It is hence necessary to accurately detect memory pressure and quickly react to it. When the memory pressure occurs, the system evicts and reschedules batch jobs to avoid OOM (out-of-memory). In addition to memory utilization, we use another kernel metric, memory pressure stall information (PSI) [10] of the root cgroup, which measures the amount of time a process waits on the slow path of memory allocation. Intuitively, when the host machine is running out of memory, its memory utilization is high, and its processes stall when applying new memory pages. Therefore, we use the combination of memory PSI and memory utilization to detect when applications stall at the allocation of new memory pages.

Evaluation. We deployed the memory reclamation on a cluster consisting of more than 3000 machines. Fig. 8 compares the memory utilization distributions of LC services before and after the reclamation. In the median, the utilization of anonymous pages and file pages was reduced from 74% to 67%, and from 13% to 4%, respectively.

Fig. 9 compares the distribution of the CPI (cycles per instruction, indicating the application performance [58]) of all LC services and the average response time of around 1000 memory-sensitive LC services before and after memory reclamation. We observe *no noticeable change* in CPI (Fig. 9a).



(a) Cycles per instruction (CPI) of all LC services, before and after memory reclamation.

(b) Average response time (RT) of memory-sensitive LC instances (normalized by the longest RT).

Figure 9: CPI and the average response time (RT) of LC services, before and after memory reclamation.

Besides, the average response time[5] slightly increased by 1.26% (Fig. 9b). These results confirm that memory reclamation has a negligible negative impact on LC services.

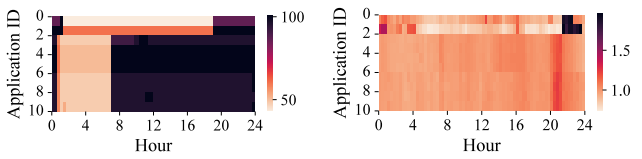
3.3 Tidal Scaling

While memory reclamation reduces the memory footprints of LC services, it is still insufficient as 1) it only applies to the host memory but not GPU memory, and 2) the resulting memory utilization still has no clear diurnal pattern, limiting the space for resource overcommitment at night. We further propose *tidal scaling* to address these problems.

Bimodal instance. In our cluster, LC services are deployed as *bimodal instances* with two states: *running* and *dormant*. An instance serves user requests only when it is running. When turning to the dormant mode, the instance is suspended and relinquishes all the allocated resources. A dormant instance can be quickly activated into the running state by executing the service processes in its container; similarly, a running instance can be set to the dormant mode by killing all its running processes. For each LC service with a diurnally changing load, tidal scaling deploys a number of bimodal instances on clusters. The instance number is decided by the service capacity as well as the operator's experience, which is usually maintained the same as before without tidal scaling.

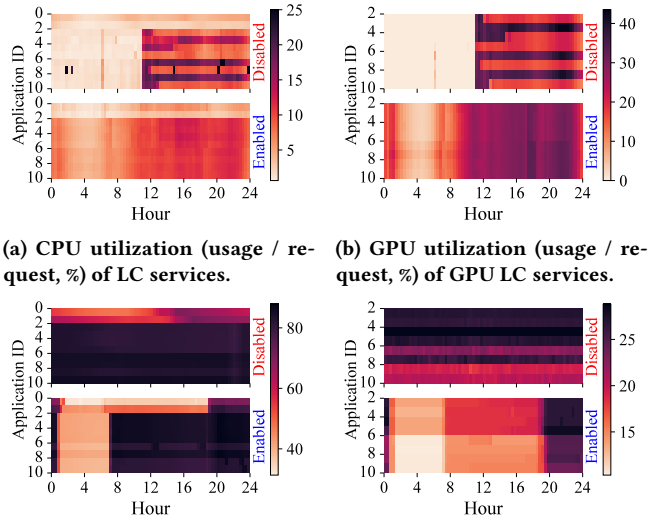
Scaling policy. Tidal scaling uses a similar scaling policy to that of horizontal scaling. Specifically, the controller dynamically adjusts the number of running instances based on the overall resource utilization of each service, so as to approach the target utilization. The target utilization is determined based on the average resource utilization of the corresponding LC services before tidal scaling is enabled. Unlike horizontal scaling, tidal scaling does not make scheduling decisions but simply activates the dormant instances

[5]Our monitoring system only records the average response time per instance at a second-level granularity, so we cannot calculate a precise tail latency (e.g., P95 latency per request) but can only calculate the exact average latency by averaging these values.



(a) Percentage of active instances when tidal scaling is enabled. (b) Service RT (normalized by each application's daily average).

Figure 10: Deployment result of tidal scaling.



(a) CPU utilization (usage / request, %) of LC services.

(b) GPU utilization (usage / request, %) of GPU LC services.

(c) Host memory utilization (usage / limit, %) of LC services.

(d) GPU memory utilization (usage / limit, %) of GPU LC services.

Figure 11: Comparison of resource utilization with and without tidal scaling.

that are already deployed on the machines following the offline optimized placement scheme.

Evaluation. We evaluate tidal scaling through comparison studies on 10 representative LC services(of which services 2 to 9 demand GPUs) in a production cluster. Fig. 10 depicts the percentage of active instances and their response time (RT). We also compare the resource utilization of these LC services in 24 hours with and without tidal scaling in Fig. 11. For all 10 services, tidal scaling awakens all their dormant instances during the peak time (e.g., from 17:00 to 1:00 for Apps. 0 and 1) and suspends some running instances in the off-peak time. The tidal scaler dynamically adjusts the number of active instances of these LC services (Fig. 10a), controlling their average RT at around the daily average (Fig. 10b). As a result, no diurnal utilization pattern is observed in CPUs and GPUs (Figs. 11a and 11b). In the off-peak time, the tidal scaler puts some running instances into dormancy. We hence observe a significant drop in both the host and GPU memory utilization in Figs. 11c and 11d.

3.4 Open Challenges

Proactive reclamation of idle GPU memory. Although we have added a memory idleness tracking module into the Linux kernel and implemented proactive reclamation of idle memory pages, we currently lack an infrastructure-level instrumentation tool for proactive GPU memory reclamation. Many deep learning frameworks already provide interfaces to proactively release GPU memory. Yet, they require laborious modifications of the application's source code, which is impractical in our system.

Proactive reclamation of idle memory in Kata containers. Currently, idle memory pages can be reclaimed in Runc containers, but not in Kata containers. The reason is that Runc containers share the host Linux kernel, and the host kernel's `kidled` and `kreclaimd` can recognize and reclaim the idle memory pages of cgroups belonging to these containers. In contrast, Kata containers run in separated guest kernels, and their memory usage is opaque to the host kernel. Although Kata containers support memory ballooning, there is no way to proactively track and reclaim idle memory pages. In a cluster with more than 5k nodes, there are around 3k Kata containers (all being batch jobs) running at any given time, occupying around 24% memory capacity. Proactively reclaiming the idle memory of Kata containers can thus produce huge benefits.

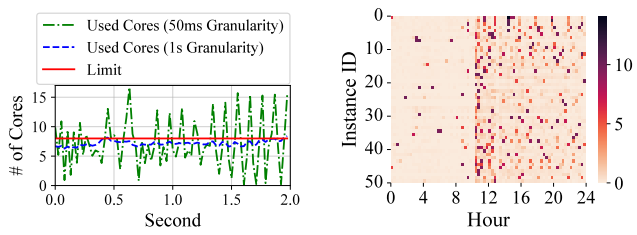
Warm-up latency. While tidal scaling can quickly start up a dormant instance, some applications undergo a warm-up period, during which the user requests experience a significant slowdown or even timeout. The warm-up latency varies dramatically from one application to another. It remains open how this problem can be addressed efficiently for reduced system hiccups and improved user experience. We currently workaround this problem by predicting the service's resource utilization, and based on which we activate instances in advance.

4 NODE-LEVEL MICRO-MANAGEMENT

In addition to cluster-wide macro-management, node-level micro-management is also needed as we observe frequent resource variations of cluster applications which are often unpredictable and large in magnitude. In this section, we illustrate this phenomenon on CPU and memory bandwidth that harms the performance of LC services and present our solution to it.

4.1 CPU Jitters

At Alibaba, engineers set CPU quotas for applications based on the CPU utilization traces, which are collected by the monitoring system as the result of smooth downsampling at one-second intervals. However, the second-level trace cannot reflect the CPU jitters that occurred at a finer time



(a) The CPU usage trace of a bursty LC service in different time scales.

(b) The CPU throttling rate of 50 instances of an LC service with bursty CPU usage in 24 hours.

Figure 12: An illustration of CPU jitters.

scale. Specifically, code hotspots, memory garbage collection, and other program behaviors cause some applications to have tiny CPU load spikes, each lasting no more than 100 milliseconds. Fig. 12a shows the CPU utilization traces of a CPU-bursty LC instance collected in 1 s and 50 ms intervals.

CPU load spikes in millisecond intervals. The Linux kernel's *Completely Fair Scheduler* (CFS) [36] controls the amount of CPU time allocated to normal processes in each cgroup for fair scheduling, thus preventing processes from overusing CPU time. The CFS controller manages the limits of any given cgroup in an accounting period of `cfs_period_us`. In each accounting period, once the process has used up the quota of CPU time (`cfs_quota_us`), it is throttled until the next accounting period begins. Although this mechanism works well for workloads with stable CPU demand, it becomes problematic for bursty workloads. These workloads use far less CPU time than their quota in most periods, but occasionally a burst of work arrives which requires more CPU time than the quota permits. The application's CPU usage is thus suppressed, causing performance degradation.

The CPU load spikes usually have a temporal pattern that correlates with the application's CPU usage and its traffic load. To see this, we refer to Fig. 12b, which shows the distribution of CPU throttling rate (measured by the number of throttled CFS periods per second) of 50 instances of an LC service with bursty CPU usage in 24 hours. All 50 instances have diurnally changing CPU throttling rates. For these instances, allocating CPU quotas based on the maximum load spike, though avoiding performance degradation caused by CPU throttling, could waste a lot of CPU cycles in most time periods.

Shared CPU pool for CPU-bursty applications. To address the aforementioned performance issues, CPU-bursty processes should be allowed to borrow the idle CPU time from the other colocating processes upon the arrival of their load spikes. However, CPU-bursty hyper-threads running on paired logical cores (mapped to the same physical core) could contend for resources at the micro-architecture level, affecting the other non-bursty threads. To minimize this interference, we set up a *shared CPU pool* on each node for CPU-bursty applications. Specifically, we classify LC applications into two categories: *exclusive* and *shared*. Applications in the exclusive class are sensitive to CPU contention and need to run exclusively on physical cores, whereas those in the shared class are either CPU-insensitive or CPU-bursty, and can only run on CPU cores in the shared pool. We scale up `cfs_quota_us` for shared-class applications to allow them to borrow CPU time slices from idle CPU resources in the pool.

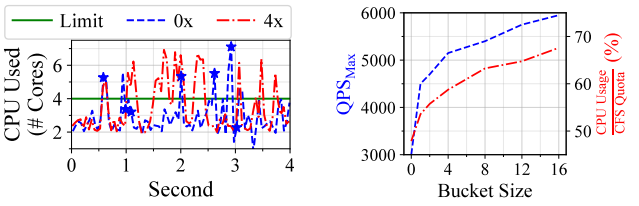
Management of the shared CPU pool. Note that CPU-bursty hyper-threads colocated on the same physical core can also interfere with each other when their logical CPU utilization is high. To minimize such interference, the utilization of the shared CPU pool should be kept below 45% based on our experience, i.e., $U < U_H = 45\%$.[6]

We use two strategies in this regard: 1) at the cluster level, the scheduler places shared-class LC instances (pods) evenly on different machines based on the principle of load balancing and ensures that $U < U_H$; and 2) at the node level, we use the AIMD (Additively Increase and Multiplicatively Decrease) algorithm, borrowed from TCP congestion control, which evenly increases the CFS quota of all shared-class pods under the constraint that $U < U_H$ and the average utilization of any shared-class pod should be maintained below its original CPU limit. This algorithm is executed every 5 seconds. If a pod was throttled in the last period, its CFS quota increases linearly. If the average utilization of the shared CPU pool in the last period exceeds the threshold $U > U_H$, or the average CPU utilization of the pod in the last period exceeds its original CFS quota (overusing the quota), the algorithm reduces its quota by half until the utilization of the shared pool drops below U_H.

Burstable CFS controller. To further alleviate the impact of CPU throttles for bursty workloads, we add a token bucket mechanism to the CFS controller for each cgroup [6]. It allows the processes running in each cgroup to carry over some of their unused quotas from one CFS period to the next. For each cgroup, the controller exposes a configurable parameter, `cfs_burst_us`, which sets the maximum amount of time that can be accumulated in the bucket.

We demonstrate the efficacy of our CFS controller through microbenchmark experiments. We run Shore, an on-disk transactional database provided by Tailbench [28] with a CPU-bursty workload, and depict the CPU utilization trace in Fig. 13a. Compared with the original CFS controller without a token bucket (bucket size set to 0), our controller (bucket

[6]We ran various benchmarks on two logical cores mapped to the same physical core. Extensive experiments show that when the utilization is below 45%, the interference between the two logical cores is negligible.



(a) The CPU utilization with bucket size set to 4×`cfs_quota` (4x) and zero (0x), where `cfs_quota` is set to *Limit*. A "⋆" indicates a CPU throttle.

(b) Peak throughput (QPS_{Max}) and the average CPU usage attained with different bucket sizes.

Figure 13: Running Shore with our burstable CFS controller with varying bucket sizes.

size set to 4× `cfs_quota`) allows the bursty workload to use spared quota carried over from previous periods when needed, dramatically reducing CPU throttles. We also evaluate the peak throughput (i.e., the maximum QPS without CPU throttle and queuing delay) and the corresponding CPU utilization with different bucket sizes in Fig. 13b. As the CFS bucket size increases, the peak throughput grows logarithmically. Accordingly, the CPU utilization also improves but never exceeds the quota limit.

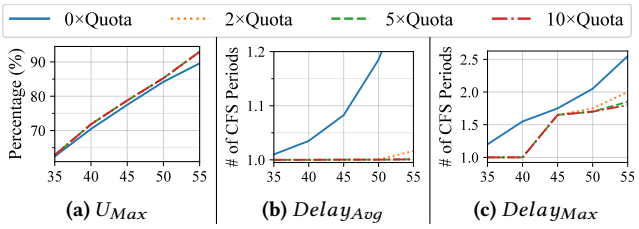
Parameter tuning. Although configuring a larger CFS bucket size B reduces CFS throttles for bursty workloads, it risks violating the utilization constraint of the shared pool: when multiple running instances have CPU load spikes arriving at the same time, the short-term average utilization of the shared CPU pool can be significantly higher than U_H. Therefore, it is necessary to configure a "right" bucket size. To do so, we collected millisecond-level CPU utilization traces of bursty workloads running in production clusters and performed a trace-driven simulation to experiment with different bucket sizes. The simulation generates mocked applications' requests and consumes the CPU time slices based on a simple queuing model. It uses the following metrics to evaluate the short-term utilization of the shared pool and the resulting performance:

1) $Delay_{Avg}$: The average request queuing delay of all applications, which measures their average performance.

2) $Delay_{Max}$: The average of the maximum queuing delay of each application, which measures the worst performance.

3) U_{Max}: The maximum average utilization of the shared CPU pool for 5 consecutive CFS periods, which is used to measure the extent to which the utilization constraint of the shared CPU pool is violated.

In our simulation, we run 20 bursty applications (the average number of applications running in the shared CPU pool per machine is also set around this value). Each application is allocated the same CFS quota. Given the target average utilization U of the shared CPU pool, the average



(a) U_{Max} **(b) $Delay_{Avg}$** **(c) $Delay_{Max}$**

Figure 14: U_{Max}, $Delay_{Avg}$ and $Delay_{Max}$ (Y-axis) under different shared CPU pool utilization (%, X-axis) and CFS bucket size B. "$k \times Quota$" denotes that the CFS bucket size $B = k\times$`cfs_quota_us` of each pod.

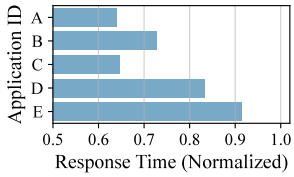
CPU utilization of each application follows a normal distribution $\mathcal{N}(U, 0.25 \times U)$. Fig. 14 shows the simulation results. As expected, configuring a larger CFS bucket size can significantly reduce the request queuing delay (smaller $Delay_{Avg}$ and $Delay_{Max}$ as shown in Figs. 14b and 14c). When the average utilization of the shared CPU pool is within $[35, 55]$%, setting a large bucket size slightly increases the peak utilization of the shared pool (U_{Max}) by up to 3.8% (Fig. 14a), which is negligible. We therefore set the bucket size of each running LC instance to 10× `cfs_quota_us` to reduce CPU throttles as much as possible.

Production deployment. We deployed the shared CPU pool and the burstable CFS controller on a cluster with over 160k running LC instances labeled as 'shared'. We set the CFS bucket size at 10×`cfs_quota_us`. Fig. 15 shows the deployment results. Before the deployment of our solution, around 73.4% of 'shared' LC instances were bursty and being throttled during peak time. After the deployment, only 0.12% of 'shared' LC instances were throttled. This improvement is mostly due to the efficient management of the shared CPU pool, with which only 0.75% of 'shared' instances needed to borrow time slices from the burstable CFS bucket. Fig. 15a further compares the average response time (RT) of the five most bursty applications with and without our solution. We observe a 10–35% reduction in the average RT enabled by our approach. Also, Fig. 15b shows that our solution successfully keeps the utilization of the shared CPU pool of 99% of machines below the threshold value $U_H = 45$%.

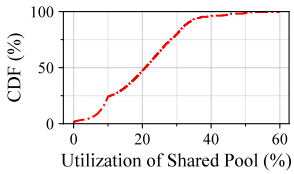
4.2 Variations on Memory Bandwidth

In addition to CPU jitters, variations in memory bandwidth are frequently observed, especially in batch jobs with different computing phases. Fig. 16 shows the memory bandwidth usage (estimated by the number of L3 cache misses per second) trace of three batch jobs in 30 minutes.

Excessive memory bandwidth utilization of batch jobs. In our clusters, around 12% of the machines have memory access latency 1.5-8x longer than the average (~130 ns). All these machines have high memory bandwidth utilization. In



(a) Average RT of 5 bursty LC services (*A* - *E*) after the deployment. Each application's RT is normalized by its average before the deployment (*Baseline*).



(b) Utilization distribution of the shared CPU pool on each machine (*the CPU cores used by all shared LC services / the capacity of the shared CPU pool, %*).

Figure 15: Deployment results of the shared CPU pool with the burstable CFS controller.

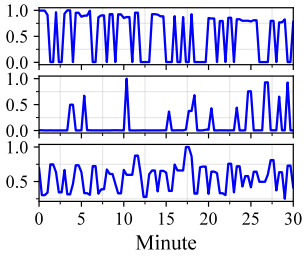


Figure 16: L3 cache miss trace of 3 instances of different batch jobs (normalized by the maximum).

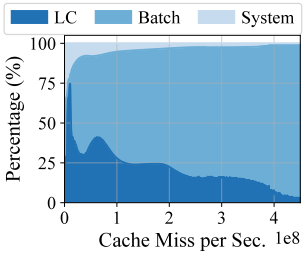
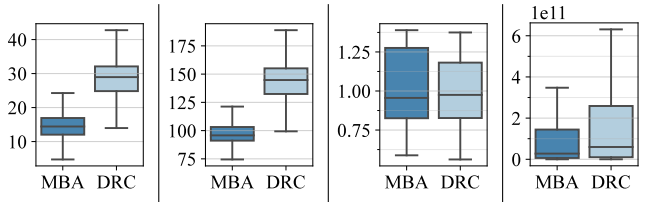


Figure 17: Area plot of estimated memory bandwidth usage of different workloads in a cluster.

addition, Fig. 17 shows the distribution of the total memory bandwidth utilization on each machine and the usage (approximated by the average number of L3 cache misses per second) of batch, LC, and system applications in a one-day trace, where the X-axis is the total memory bandwidth usage of a host machine, and Y-axis breaks down the usage into different applications in percentage. It is clear from the figure that the high memory bandwidth utilization of a machine is mainly contributed by the running batch jobs.

Memory bandwidth control using Intel's DRC. Efficiently and accurately controlling memory bandwidth consumption is technically challenging. Existing mechanisms, such as Intel's Memory Bandwidth Allocation (MBA) [23] and AMD's Quality of Service Extensions [17], provide indirect and approximate control over memory bandwidth available per core, which is inflexible and may result in undesirable system performance. In collaboration with Alibaba, Intel introduced a new dynamic resource control (DRC) module to the latest IceLake server processor architecture [59] for autonomous memory bandwidth management based on a PID feedback control algorithm.

DRC automatically throttles the memory requests of low-priority tasks when their memory read request rate exceeds



(a) Mem. b/w utilization (%). **(b) Mem. access latency (ns).** **(c) CPI of LC services.** **(d) IPS of batch jobs.**

Figure 18: Performance comparison between MBA and DRC in a production cluster.

a pre-specified threshold P. This threshold can be automatically tuned using existing techniques [59], but it requires heavy profiling of applications, which is not applicable in our clusters given the vast amount and rapidly evolving LC services (§2.3). We therefore use a simple heuristic to tune P based on the observed memory bandwidth utilization U and memory access latency L per socket. In our experience, when $U \leq 20\%$ or $L \leq 200$ ns, setting $P = 120$ allows batch jobs to fully utilize memory bandwidth for improved throughput. For higher memory bandwidth utilization and longer access latency, i.e., $U > 20\%$ and $L > 200$ ns, batch jobs could interfere with the colocating LC services, and we linearly scale down P, with the minimum value being 20 when $U \geq 60\%$.

Production deployment. We have recently deployed the DRC-based memory bandwidth control with the aforementioned threshold tuning heuristic in a production cluster, and share our operational experience as follows. Originally, the cluster management system suppressed the memory bandwidth contention from batch jobs using Intel's MBA. Specifically, given that the overall memory bandwidth utilization always exceeded 40% in the cluster without any bandwidth regulation, we set the parameter of MBA to 10%, the minimum value, to guarantee the performance of LC services. This means that batch jobs can only use up to 10% of the memory bandwidth capacity when colocated with LC services. Fig. 18 compares the memory bandwidth utilization and the performance of LC services and batch jobs before and after the deployment of the DRC-based solution. Overall, the performance of LC services, measured by cycles per instruction (CPI) [58], sees no noticeable changes (Fig. 18c). Despite a slight increase in median memory access latency from 100ns to around 140ns (Fig. 18b), the median memory bandwidth utilization doubles from around 15% to near 30% (Fig. 18a), and the throughput of batch jobs, measured by the number of instructions executed per second (IPS), also sees an order-of-magnitude improvement (Fig. 18d). In summary, the DRC-based solution significantly outperforms MBA even with a simple heuristic tuning algorithm.

4.3 Open Challenges

Side effects of burstable CFS controller. Our burstable CFS controller is not without problems. On the one hand, it provides more agile, fine-grained CFS quota management, which can reduce throttles in the presence of more tiny CPU spikes. On the other hand, it is a kernel feature not controlled by the SLO manager. Compared to CPU quota management in user space, it brings more uncertainties when the CPU utilization of the shared pool is too high.

Batch job scheduling made aware of memory bandwidth contention. Although DRC can alleviate the interference on LC services caused by the excessive memory bandwidth usage of batch jobs, it inevitably harms the throughput of the latter when both LC services and batch jobs have high utilization of memory bandwidth. The scheduling of batch jobs should hence be aware of such contention. For example, the scheduler may reschedule those batch jobs with high memory bandwidth usage to some other nodes with less contention on memory bandwidth.

Adaptive fine-tuning of DRC's parameter. Our current system uses a simple heuristic to tune parameter P of DRC. This can be improved with an adaptive, application-agnostic algorithm that automatically tunes the best parameter without heavy profiling or prior knowledge, which we leave for future work.

5 HANDLING SEASONAL SHOPPING FESTIVALS: A CASE STUDY

Alibaba's e-commerce platform hosts a number of *Seasonal Shopping Festivals* (SSFs) around the year. Among all SSFs, the largest one is on November 11. On that day, user-facing services see flash crowd traffic roaring in just a few seconds. Compared to the off-season, the peak load can be orders of magnitude higher, as illustrated in Fig. 19. This creates an extreme load pressure on our system. In this section, we describe how we address this challenge through judicious capacity planning and macro- and micro-management of cluster resources. Due to the space constraint, our descriptions are given at a high level.

5.1 Extreme, yet Predictable Load in SSFs

To attract customers to place orders, merchants usually hold flash sales (e.g., offering first-come, first-served limited free order opportunities) and issue coupons at the beginning of an SSF (e.g., 0:00, November 11), resulting in the instantaneous flash crowd traffic at that moment. When a promotion starts, user traffic spikes instantly in a few seconds and falls back afterward in minutes. As all the sales events are scheduled in advance, the peak traffic of all e-commerce services has predictable start and end times. Fig. 20 shows how the QPS

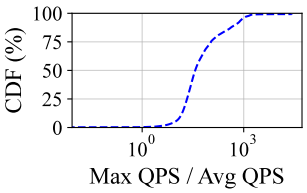


Figure 19: CDF of peak QPS / daily average of e-commerce services in an SSF. Peak QPS can be orders of magnitude higher than the daily average.

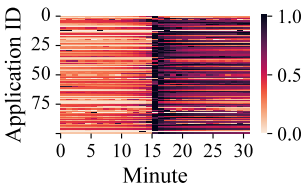


Figure 20: The QPS (normalized by the peak) of 100 e-commerce services at the beginning of an SSF, where the peak traffic arrived at the 15th minute.

of 100 e-commerce services surged and then dropped in an SSF held on November 11, 2021.

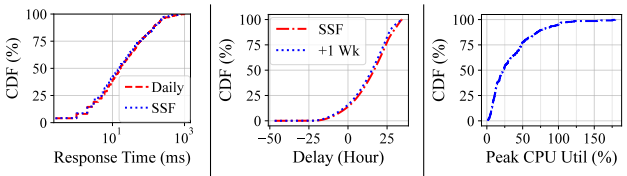
5.2 Capacity Planning and Resource Macro-management

As mentioned in §3.1, horizontal scaling is too slow to handle the surging flash crowd and cannot make quality scheduling decisions at our scale. Leveraging the predictable nature of the SSF traffic, we developed a solution consisting of four steps as follows: 1) predict the peak load and estimate the required service capacity; 2) reserve resources for the serving instances in advance by suspending and postponing the execution of non-critical batch jobs; 3) offline compute the optimal placement of the scale-up service instances and place them onto the corresponding nodes as dormant instances in advance; 4) activate dormant instances and warm them up at the scheduled time before the SSF starts.

Capacity planning. For each SSF, Alibaba predicts the peak load (the number of transactions per second or TPS_{Max}) and the required capacity for each user-facing service based on the historical data and load testing results. This is usually done a few months before the SSF. During the SSF, exceeding requests beyond the planned peak TPS_{Max}, if any, will be dropped automatically to maintain the system stability.

Offline instance orchestration and placement. Based on the planned capacity for the SSF, the system scales up the corresponding user-facing services and places all the scale-up instances in the cluster as dormant instances, similar to tidal scaling (§3.3). The optimal instance placement is computed offline by solving a complex bin-packing problem. The entire process is known as *offline orchestration*.

Resource reservation. Based on the offline orchestration plan, the system needs to reserve sufficient resources so that the dormant scale-up instances can be quickly activated and provisioned when the SSF begins. To minimize the resource provisioning cost, we choose not to add more compute resources to the cluster but to postpone the execution of non-critical batch jobs and use the relinquished resources to run

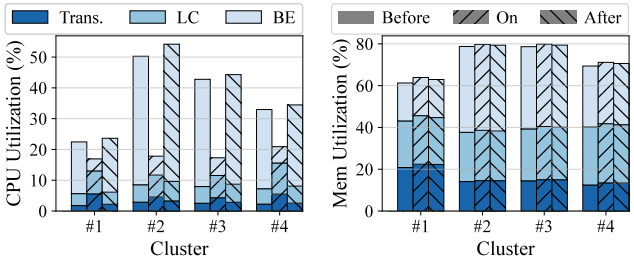


(a) CDF of the average response time (ms) of e-commerce applications during the SSF and the off-season (daily).

(b) Completion delay (*completion time - deadline*) of batch jobs submitted on the SSF day and one week later (+1 Wk).

(c) CDF of the peak CPU utilization (*# used cores / # requested cores*) of LC applications during the SSF.

Figure 21: The performance of e-commerce applications and the peak CPU utilization on Nov. 11, 2021.



(a) CPU utilization. **(b) Memory utilization.**

Figure 22: Resource utilization of different classes of applications (transaction-related services (*Trans.*), other LC services (*LC*) and batch jobs (*best-effort* or *BE*)) at different time moments: one day before the SSF (*Before*), during the peak traffic of the SSF (*On*), and 3 hours after the SSF (*After*) in four production clusters.

LC instances during the SSF. As an SSF lasts for one day, the completion time of the suspended batch jobs is expected to be delayed by 24 hours. The scheduler also stops scheduling newly arrived batch jobs to the clusters. To provide more CPU resources to LC services, we reduce the `cfs_quota_us` of all running batch jobs to throttle their CPU usage.

Activating and warming up dormant LC instances. The dormant scale-up instances are activated and warmed up at the scheduled time before the SSF starts (usually hours before the SSF). To avoid delaying the batch jobs for a long time, once the SSF ends, all the scale-up LC instances are instantly terminated and their relinquished resources are given back to the previously suspended batch jobs for resumed execution. Newly submitted batch jobs are also accepted by the scheduler and can run on clusters.

Performance of LC services during the SSF. To demonstrate the effectiveness of our solution, we depict in Fig. 21a the distribution of the average response time (RT) of e-commerce applications during the off-season and an SSF on November 11, 2021. We observe no significant difference between the distributions of the average RTs during the SSF and the off-season, confirming that our approach can effectively handle the extreme load pressure created by the SSF.

Impact on batch jobs. We also evaluate the impact of delayed execution on batch jobs during the SSF. In our clusters, each batch job specifies a desired deadline for its completion. We measure the *completion delay* of a batch job as the difference between its completion time and the specified deadline. We measure the completion delays of the batch jobs submitted on the SSF day and those submitted one week later (off-season), and depict their distributions in Fig. 21b. Note that for the former jobs, their deadlines were extended by 24 hours as they were suspended during the SSF. Fig. 21b shows that their completion delays follow the same distribution as those submitted in daily time.

Imbalanced peak CPU load during the SSF. Our current solution is not without a problem. One issue we observed is the *uneven* peak CPU utilization of different LC services during the SSF, ranging from close to 0% to 190%, as shown in Fig. 21c. About 75% of e-commerce applications have a peak CPU usage lower than 59%. This indicates that the peak CPU pressure created by the flash crowd traffic is *non-uniform* across applications. As mentioned in §2.2, our businesses rapidly evolve and the LC applications are mainly deployed as microservices, making it challenging to accurately estimate the required service capacity in the SSF.

Cluster resource utilization. Fig. 22 depicts the average utilization of CPU and memory for different classes of applications in four production clusters at three time moments, one day before the SSF (Nov. 10, 2021), during the SSF with peak traffic (Nov. 11, 2021), and 3 hours after the SSF (Nov. 12, 2021). We observe a severely underutilized CPU despite the close to 100% allocation ratio of both CPU and memory (not shown in the figure). The reasons are two-fold: 1) our capacity planning algorithm overestimated the required CPU quota for a large number of LC applications (Fig. 21c); 2) although we delayed the execution of non-critical batch jobs and reduced their CPU usages during the SSF, we did not proactively reclaim more memory pages from the remaining running jobs, thus making memory a bottleneck in the SSF.

5.3 Handling Resource Usage Jitters

We next examine the effectiveness of our node-level micro-management solutions (§4) in the SSF scenario.

Shared CPU pool in the SSF. Like in the off-season periods, the shared CPU pool (§4.1) plays an important role in handling CPU jitters in the SSF. The surging traffic during the SSF can significantly increase the CPU utilization of LC applications. Take an SSF held in June 2022 as an example, in the clusters where the shared CPU pools were not configured, the percentage of CPU-throttled LC instances grew

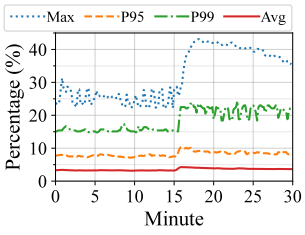


Figure 23: The maximum (*Max*), P99, P95, and average (*Avg*) shared pool utilization in a cluster of machines during an SSF in Jun. 2022. Peak traffic arrived at the 15th minute.

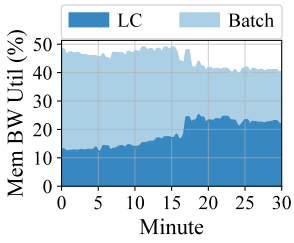


Figure 24: The memory bandwidth utilization of LC services and batch jobs on a machine during an SSF in Jun. 2022. Peak traffic arrived at the 15th minute.

from 10% (daily average) to 25.85% upon the arrival of the flash crowd traffic in the SSF. This is not the case in the clusters where the shared CPU pool is enabled. Fig. 23 shows the maximum (*Max*), the 99th and 95th percentile (*P99* and *P95*), and the average (*Avg*) CPU utilization of the share pools on a cluster of machines upon the arrival of the peak traffic in this SSF. Across all machines, the shared pool utilization was maintained below $U_H = 45\%$. Only 1.74% of LC instances experienced CPU throttles during the SSF (not shown in the figure), which was far less than that in the clusters where the shared CPU pools were not enabled (25.85%).

DRC-based memory bandwidth control. While non-critical batch jobs are suspended during SSFs, the critical ones are still running, which consume a small amount of CPUs and memory bandwidth. Fig. 24 shows how the memory bandwidth utilization of LC services and batch jobs on a machine (DRC enabled) changed during the aforementioned SSF in June 2022. When the peak traffic arrived (at the 15th minute), LC services' utilization ramped up, at which point DRC immediately reduced the memory bandwidth limit set to the batch jobs and ensured that the overall memory bandwidth utilization remained at around 40%. This is expected because our DRC management algorithm prevents the memory bandwidth utilization from exceeding 60%, which would cause bandwidth contention among the colocated applications (§4.2).

5.4 Discussion and Open Challenges

More accurate capacity planning. During SSFs, applications have diverse peak-to-normal load ratios, as shown in Fig. 19, demanding highly divergent capacity increases. However, our current capacity planning algorithm is coarse-grained, leading to an uneven distribution of CPU utilization of applications (Fig. 21c). More accurate capacity estimations are hence desired to enable fine-grained planning for improved resource efficiency and service quality in SSFs.

Memory reclamation for postponed batch jobs. Proactively reclaiming more idle memory pages from running batch jobs can improve resource utilization during SSFs. Before the arrival of SSF, we should configure a larger memory reclamation threshold to aggressively reclaim more idle memory pages of batch jobs. Moreover, as a large number of batch jobs are running in Kata containers, proactively reclaiming Kata containers' idle memory (discussed in §3.4) can also improve the resource utilization in SSFs.

Potential bottlenecks. Ideally, the utilization of each node's shared CPU pool should strictly stay below 45%. But the unbalanced utilization distribution (Fig. 15b) leads to around 1% shared pools approaching 25% when the peak traffic arrived in the SSF (Fig. 23). Memory bandwidth is also a potential bottleneck: referring to Fig. 24, the memory bandwidth utilization of the LC applications running on a machine reached around 25% when the peak traffic arrived. To ensure that the memory access latency remained low, DRC kept the overall memory bandwidth utilization below 45% during this period.

How far are we away from the optimum CPU utilization? Without considering the limitations brought by the shared CPU pool and memory bandwidth, we estimate that the maximum achievable CPU utilization in SSFs is around 48% in the aforementioned four production clusters. Despite a potentially large improvement space of CPU utilization in SSFs, we believe that the current utilization level meets our business expectations. We therefore leave it as a future work to further improve CPU utilization without compromising the service quality.

Can batch jobs experience less delay? Although we currently do not have a mature solution to further improve resource utilization during SSFs, we can temporarily (within a few hours) loan some idle resources from the other non-core clusters located in different geographical regions to run the queued batch jobs for reduced completion delay. We have validated the feasibility of this workaround in an SSF held in summer 2022: the delays of the batch jobs were reduced from one day to 1–1.5 hours.

6 RELATED WORK

Memory reclamation and overcommitment. Proactive memory reclamation offers an effective solution to address the memory bottleneck in large clusters. In Google's clusters, infrequently used memory pages are evicted to far memory [29], where the reclamation parameters are auto-tuned using reinforcement learning. This approach requires operators to estimate the maximum page miss ratio that is tolerable to each application, and does not address how to overcommit memory to low-priority tasks. Meta's TMO [53] periodically reclaims cold memory pages based on the PSI

metric [10]. Compared to our approach, TMO does not differentiate high-priority LC services from low-priority batch jobs, and does not allow the latter to loan idle resources from the former.

As for resource overcommitment, in [4], Google proposes a clairvoyant overcommitment policy based on the predicted future resource usage. This approach does not solve our problem: given the consistently high memory footprints of LC instances, there is little space to overcommit the other underutilized resources to batch jobs at night, which we address with tidal scaling. Prior works also studied how to detect and eliminate memory pressure in the context of overcommitment. Notably, OA Killer [8] detects memory pressure and evicts the low-priority jobs in the Linux kernel to achieve a low eviction latency. When the available memory space becomes small, resource deflation [45] compresses the memory pages of low-priority jobs to avoid out-of-memory errors. Compared to these works, we report our operational experiences of memory reclamation and overcommitment in production clusters running e-commerce workloads.

Auto-scaling. Auto-scaling plays a key role in workload consolidation, which can be broadly categorized into two approaches. The first is horizontal scaling, which automatically adjusts the number of instance replicas in response to the load changes. Facebook's Twine [46] and Microsoft's SEAGULL [40] make horizontal scaling decisions based on the prediction of future resource demands. However, horizontal scaling is too slow to handle the surging traffic volumes in SSFs, and cannot make quality placement decisions quickly at our scale. The second auto-scaling approach is vertical scaling, which dynamically adjusts an instance's resource allocation to reduce resource slack, OOMs, and CPU throttles. For example, Amazon EC2 Auto-scaling [2] and Netflix's Scryer [38] vertically scale the VM instances based on the estimation of future traffic load. Google's Autopilot [43] uses reinforcement learning to right-size a container instance. However, vertical scalers cannot handle large-scale diurnal traffic variations, load spikes in SSFs, frequent, micro-scale CPU bursts, or memory bandwidth contentions.

QoS management with shared resources. Managing applications' QoS by reducing contention on shared resources in the context of colocation has been an active research topic in recent years. Some cluster schedulers predict the interference caused by colocation, and then adjust resource allocations at runtime or disallow resource sharing [5, 11, 13, 15, 16, 25, 35, 56–58, 60]. Another approach is to leverage OS- and hardware-level fine-grained resource partitioning to eliminate interference [7, 26, 27, 32, 44, 51, 55, 59]. However, these approaches cannot manage bursty CPU load spikes in CFS periods. Apart from Intel's DRC [59], there is no hardware solution to dynamically partition memory bandwidth between the colocated applications.

Constraint-aware scheduling. Container placement in production clusters needs to meet a number of constraints such as hardware requirements, fault tolerance, resource contentions, and incremental deployment [1, 21]. Existing works address this problem using integer linear programming [20, 48], feedback control [32], and learning-based approaches [3, 12, 14, 31, 37, 39, 41, 50, 56]. Many of these works avoid harmful interference by preventing the contending jobs from being colocated [12, 14, 33, 34, 37, 39, 41, 42]. However, they do not consider scheduling CPU-bursty workloads and minimizing their interference, nor do they explore how to rapidly scale up services when there is a large volume of traffic surging in a short period of time.

7 CONCLUSION

This paper documents the operational practice of workload consolidation in Alibaba's clusters for reduced cost and uncompromised service performance. A number of macro- and micro-management techniques have been developed. At the cluster level, the consistently high memory footprints of LC services limit the idle resources from being utilized by the collocated batch jobs. Our system addresses this problem by proactively reclaiming the cold memory pages of LC services and diurnally awakening/suspending the LC instances with tidal scaling. At the node level, frequent fluctuations of resource usage harm the collocated LC and batch workloads. Our system uses a shared CPU pool together with a burstable, throttle-aware CFS scheduler to handle frequent CPU jitters of LC instances. It also uses the newly introduced DRC module to control the contention of memory bandwidth. We have also shared our operational experiences in handling the extreme load pressure created by the flash crowd traffic during seasonal shopping festivals.

ACKNOWLEDGMENTS

We thank our shepherd Malte Schwarzkopf and the anonymous reviewers for their valuable comments that helped improve the quality of this work. The authors of this paper collected the production traces, performed the experiments, and wrote the paper, while many more colleagues at Alibaba have implemented and maintained the cluster management system along with us. We cannot list all of them but those who most actively participated in the system development and provided suggestions to this paper: Yu Chen, Kingsum Chow, Guoyao Xu, Mou Sun, Wei He, Rougang Han, Xiaofen Yang, Bin Huang, Lida Shen, Xingxing Zhao, Kangjin Wang, and Zhezheng Lin. This work was supported in part by the Alibaba Innovative Research Program and RGC GRF Grant 16213120. Yongkang Zhang and Qizhen Weng were supported in part by the Hong Kong PhD Fellowship Scheme.

REFERENCES

[1] Alibaba. 2022. Alibaba production cluster data. https://github.com/alibaba/clusterdata.

[2] Amazon. 2022. Amazon EC2 Auto Scaling Introduces Predictive Scaling as a Native Scaling Policy. https://aws.amazon.com/about-aws/whats-new/2021/05/amazon-ec2-auto-scaling-introduces-predictive-scaling-native-scaling-policy/.

[3] Yixin Bao, Yanghua Peng, and Chuan Wu. 2019. Deep Learning-based Job Placement in Distributed Machine Learning Clusters. In *Proc. IEEE INFOCOM*. 505–513.

[4] Noman Bashir, Nan Deng, Krzysztof Rzadca, David E. Irwin, Sree Kodak, and Rohit Jnagal. 2021. Take it to the limit: peak prediction-driven resource overcommitment in datacenters. In *Proc. ACM EuroSys*. 556–573.

[5] Sergey Blagodurov, Alexandra Fedorova, Evgeny Vinnik, Tyler Dwyer, and Fabien Hermenier. 2015. Multi-objective job placement in clusters. In *SC15: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE/ACM, 66:1–66:12.

[6] Huaixin Chang. 2022. Burstable CFS bandwidth controller. https://lwn.net/ml/linux-kernel/20210202114038.64870-1-changhuaixin@linux.alibaba.com/.

[7] Shuang Chen, Christina Delimitrou, and José F. Martínez. 2019. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *Proc. ACM ASPLOS*. 107–120.

[8] Wei Chen, Aidi Pi, Shaoqi Wang, and Xiaobo Zhou. 2019. OS-Augmented Oversubscription of Opportunistic Memory with a User-Assisted OOM Killer. In *Proc. ACM Middleware*. 28–40.

[9] Jonathan Corbet. 2022. Proactively reclaiming idle memory. https://lwn.net/Articles/787611/.

[10] Jonathan Corbet. 2022. Tracking pressure-stall information. https://lwn.net/Articles/759781/.

[11] Christina Delimitrou and Christos Kozyrakis. 2013. iBench: Quantifying interference for datacenter applications. In *Proc. IEEE IISWC*. 23–33.

[12] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proc. ACM ASPLOS*. 77–88.

[13] Christina Delimitrou and Christos Kozyrakis. 2013. QoS-Aware scheduling in heterogeneous datacenters with paragon. *ACM Trans. Comput. Syst.* 31, 4 (2013), 12:1–12:34.

[14] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: resource-efficient and QoS-aware cluster management. In *Proc. ACM ASPLOS*. 127–144.

[15] Christina Delimitrou and Christos Kozyrakis. 2016. HCloud: Resource-Efficient Provisioning in Shared Cloud Systems. In *Proc. ACM ASPLOS*. 473–488.

[16] Christina Delimitrou, Daniel Sánchez, and Christos Kozyrakis. 2015. Tarcil: reconciling scheduling speed and quality in large shared clusters. In *Proc. ACM SoCC*. 97–110.

[17] Advanced Micro Devices. 2018. AMD64 Technology Platform Quality of Service Extensions. https://developer.amd.com/wp-content/resources/56375_1.00.pdf.

[18] The Linux Foundation. 2022. Kubernetes. https://www.kubernetes.io/.

[19] The Open Infrastructure Foundation. 2022. Kata Containers - Open Source Container Runtime Software. https://katacontainers.io/ https://katacontainers.io/.

[20] Panagiotis Garefalakis, Konstantinos Karanasos, Peter R. Pietzuch, Arun Suresh, and Sriram Rao. 2018. Medea: scheduling of long running applications in shared production clusters. In *Proc. ACM EuroSys*. 4:1–4:13.

[21] Google. 2022. Google production cluster data. https://github.com/google/cluster-data.

[22] Alibaba Group. 2022. Alibaba Group's website. https://www.alibabagroup.com/en/global/home.

[23] Andrew J Herdrich, Marcel David Cornu, and Khawar Munir Abbasi. 2022. Introduction to Memory Bandwidth Allocation. https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-memory-bandwidth-allocation.html.

[24] Alibaba Inc. 2022. kidled. https://github.com/alibaba/cloud-kernel/blob/linux-next/mm/kidled.c.

[25] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew V. Goldberg. 2009. Quincy: fair scheduling for distributed computing clusters. In *Proc. ACM SOSP*. 261–276.

[26] Harshad Kasture, Davide B. Bartolini, Nathan Beckmann, and Daniel Sánchez. 2015. Rubik: fast analytical power management for latency-critical systems. In *Proc. IEEE/ACM MICRO*. 598–610.

[27] Harshad Kasture and Daniel Sánchez. 2014. Ubik: efficient cache sharing with strict QoS for latency-critical workloads. In *Proc. ACM ASPLOS*. 729–742.

[28] Harshad Kasture and Daniel Sánchez. 2016. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *Proc. IEEE IISWC*. 3–12.

[29] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. 2019. Software-Defined Far Memory in Warehouse-Scale Computers. In *Proc. ACM ASPLOS*. 317–330.

[30] Michel Lespinasse. 2022. kstaled. https://lore.kernel.org/lkml/20110922161448.91a2e2b2.akpm@google.com/T/.

[31] Suyi Li, Luping Wang, Wei Wang, Yinghao Yu, and Bo Li. 2021. George: Learning to Place Long-Lived Containers in Large Clusters with Operation Constraints. In *Proc. ACM SoCC*. 258–272.

[32] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: improving resource efficiency at scale. In *Proc. ACM ISCA*. 450–462.

[33] Jason Mars and Lingjia Tang. 2013. Whare-map: heterogeneity in "homogeneous" warehouse-scale computers. In *Proc. ACM ISCA*. 619–630.

[34] Jason Mars, Lingjia Tang, and Robert Hundt. 2011. Heterogeneity in "Homogeneous" Warehouse-Scale Computers: A Performance Opportunity. *IEEE Comput. Archit. Lett.* 10, 2 (2011), 29–32.

[35] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011. Bubble-Up: increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proc. IEEE/ACM MICRO*. 248–259.

[36] Ingo Molnar. 2022. Linux Completely Fair Scheduler. https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt.

[37] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. 2010. Q-clouds: managing performance interference effects for QoS-aware clouds. In *Proc. ACM EuroSys*. 237–250.

[38] Netflix. 2022. Scryer: Netflix's Predictive Auto Scaling Engine. https://netflixtechblog.com/scryer-netflixs-predictive-auto-scaling-engine-a3f8fc922270.

[39] Dejan M. Novakovic, Nedeljko Vasic, Stanko Novakovic, Dejan Kostic, and Ricardo Bianchini. 2013. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *Proc. USENIX ATC*. 219–230.

[40] Olga Poppe, Tayo Amuneke, Dalitso Banda, Aritra De, Ari Green, Manon Knoertzer, Ehi Nosakhare, Karthik Rajendran, Deepak Shankargouda, Meina Wang, Alan Au, Carlo Curino, Qun Guo, Alekh Jindal, Ajay Kalhan, Morgan Oslake, Sonia Parchani, Vijay Ramani, Raj Sellappan, Saikat Sen, Sheetal Shrotri, Soundararajan Srinivasan, Ping Xia,

Shize Xu, Alicia Yang, and Yiwen Zhu. 2020. Seagull: An Infrastructure for Load Prediction and Optimized Resource Allocation. *Proc. VLDB Endow.* 14, 2 (2020), 154–162.

[41] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *Proc. USENIX OSDI*. 805–825.

[42] Francisco Romero and Christina Delimitrou. 2018. Mage: online and interference-aware scheduling for multi-scale heterogeneous systems. In *Proc. ACM PACT*. 19:1–19:13.

[43] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. 2020. Autopilot: workload autoscaling at Google. In *Proc. ACM EuroSys*. 16:1–16:16.

[44] Daniel Sánchez and Christos Kozyrakis. 2011. Vantage: scalable and efficient fine-grain cache partitioning. In *Proc. ACM ISCA*. 57–68.

[45] Prateek Sharma, Ahmed Ali-Eldin, and Prashant J. Shenoy. 2019. Resource Deflation: A New Approach For Transient Resource Reclamation. In *Proc. ACM EuroSys*. 33:1–33:17.

[46] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. 2020. Twine: A Unified Cluster Management System for Shared Infrastructure. In *Proc. USENIX OSDI*. 787–803.

[47] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: the next generation. In *Proc. ACM EuroSys*. 30:1–30:14.

[48] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. 2016. TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proc. ACM EuroSys*. 35:1–35:16.

[49] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proc. ACM EuroSys*. 18:1–18:17.

[50] Luping Wang, Qizhen Weng, Wei Wang, Chen Chen, and Bo Li. 2020. Metis: learning to schedule long-running applications in shared container clusters at scale. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE/ACM, 68.

[51] Xiaodong Wang, Shuang Chen, Jeff Setter, and José F. Martínez. 2017. SWAP: Effective Fine-Grain Management of Shared Last-Level Caches with Minimum Hardware Support. In *Proc. IEEE HPCA*. 121–132.

[52] Xiaozhe Wang, Kate A. Smith, and Rob J. Hyndman. 2006. Characteristic-Based Clustering for Time Series Data. *Data Min. Knowl. Discov.* 13, 3 (2006), 335–364.

[53] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. 2022. TMO: Transparent Memory Offloading in Datacenters. In *Proc. ACM ASPLOS*. 609–621.

[54] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In *Proc. USENIX NSDI*. 945–960.

[55] Carole-Jean Wu and Margaret Martonosi. 2008. A Comparison of Capacity Management Schemes for Shared CMP Caches. In *Proc. of the 7th Workshop on Duplicating, Deconstructing, and Debunking*, Vol. 15. 50–52.

[56] Hailong Yang, Alex D. Breslow, Jason Mars, and Lingjia Tang. 2013. Bubble-flux: precise online QoS management for increased utilization in warehouse scale computers. In *Proc. ACM ISCA*. 607–618.

[57] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2010. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proc. ACM EuroSys*. 265–278.

[58] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. 2013. CPI^2: CPU performance isolation for shared compute clusters. In *Proc. ACM EuroSys*. 379–391.

[59] Ying Zhang, Jian Chen, Xiaowei Jiang, Qiang Liu, Ian M. Steiner, Andrew J. Herdrich, Kevin Shu, Ripan Das, Long Cui, and Litrin Jiang. 2021. LIBRA: Clearing the Cloud Through Dynamic Memory Bandwidth Management. In *Proc. IEEE HPCA*. 815–826.

[60] Yunqi Zhang, Michael A. Laurenzano, Jason Mars, and Lingjia Tang. 2014. SMiTe: Precise QoS Prediction on Real-System SMT Processors to Improve Utilization in Warehouse Scale Computers. In *Proc. IEEE/ACM MICRO*. 406–418.