Component-distinguishable Co-location and Resource Reclamation for High-throughput Computing

LAIPING ZHAO, YUSHUAI CUI, YANAN YANG*, XIAOBO ZHOU, TIE QIU, and KEQIU LI†, College of Intelligence and Computing, Tianjin University, Tianjin Key Lab. of Advanced Networking, China
YUNGANG BAO, Inst. of Computing Technology, CAS, China

Cloud service providers improve resource utilization by co-locating latency-critical (LC) workloads with best-effort batch (BE) jobs in datacenters. However, they usually treat multi-component LCs as monolithic applications and treat BEs as "second-class citizens" when allocating resources to them. Neglecting the inconsistent interference tolerance abilities of LC components and the inconsistent preemption loss of BE workloads can result in missed co-location opportunities for higher throughput.

We present `Rhythm`, a co-location controller that deploys workloads and reclaims resources rhythmically for maximizing the system throughput while guaranteeing LC service's tail latency requirement. The key idea is to differentiate the BE throughput launched with each LC component, that is, components with higher interference tolerance can be deployed together with more BE jobs. It also assigns different reclamation priority values to BEs by evaluating their preemption losses into a multi-level reclamation queue. We implement and evaluate `Rhythm` using workloads in the form of containerized processes and microservices. Experimental results show that it can improve the system throughput by 47.3%, CPU utilization by 38.6%, and memory bandwidth utilization by 45.4% while guaranteeing the tail latency requirement.

CCS Concepts: • **Computer systems organization → Cloud computing**.

Additional Key Words and Phrases: Datacenters, Resource Utilization, Tail latency, Co-locating.

1 INTRODUCTION

The multi-tenant sharing nature of cloud computing exacerbates contention for shared resources, including the CPU cores, memory, cache, memory bandwidth, and networks. This resource contention leads to disordered execution of cloud services, wherein the resource consumption becomes challenging to manage, resulting in long tail latency [21]. For example, in Google's latency-critical (LC) service, the fluctuation range of tail latency can be as wide as 0 and 500ms, with the highest variation difference exceeding 600× [47].

To mitigate the disorder caused by the contention, prior work seeks to isolate tenants through two approaches: *hardware methods* and *software methods*. *Hardware methods*, such as the Intel RDT [68] and PARD [54], provide open control interfaces that allow differentiation of services at the hardware level for resourcing-on-demand. Although they are effective in achieving performance isolation, their adoption requires new hardware support.

*Corresponding author: ynyang@tju.edu.cn
†Corresponding author: keqiu@tju.edu.cn

Authors' addresses: Laiping Zhao, laiping@tju.edu.cn; Yushuai Cui, cuiys@tju.edu.cn; Yanan Yang, ynyang@tju.edu.cn; Xiaobo Zhou, xiaobo.zhou@tju.edu.cn; Tie Qiu, qiutie@tju.edu.cn; Keqiu Li, keqiu@tju.edu.cn, College of Intelligence and Computing, Tianjin University, Tianjin Key Lab. of Advanced Networking, Tianjin, China; Yungang Bao, baoyg@ict.ac.cn, Inst. of Computing Technology, CAS, Beijing, China.

ACM 0734-2071/2023/11-ART
https://doi.org/10.1145/3630006

The *software methods* commonly rely on resource overprovisioning (i.e., provisioning more capacity than they actually need) to reduce the interference. However, this approach leads to low resource utilization, resulting in increased costs for cloud services. For example, traces from Aliyun [50] show that they merely achieve aggregate CPU utilization of < 40% and aggregate memory utilization of < 60%. The average CPU utilization of Google's cluster is higher, but still < 60% [87].

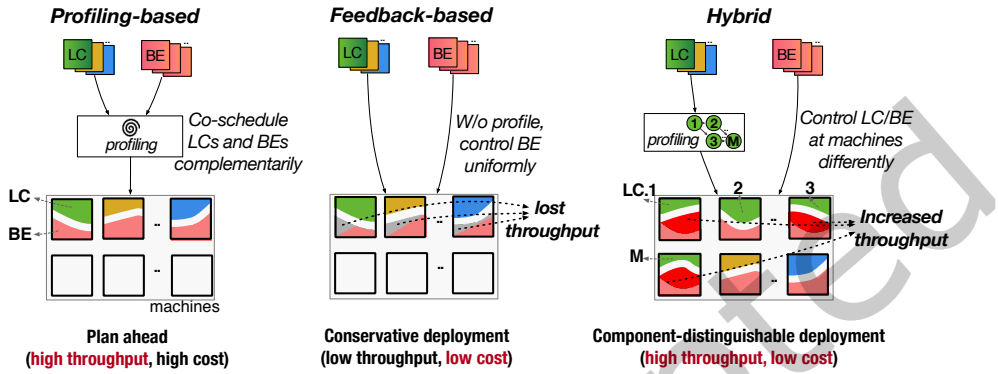


Fig. 1. Schematic overview of the workload deployment method.

It is possible to improve resource utilization by co-locating as many workloads as possible. However, it needs more control from the management system to avoid violating the tail latency Service Level Objective (SLO) of the LC service. We have identified the two prevalent control approaches shown in Figure 1. *Profiling-based* control strategy consolidates non-competitive applications based on their resource usage profiles [20, 24, 25, 59, 84, 92, 94, 100]. This strategy analyzes the resource needs of different applications and groups them together based on their compatibility, thus optimizing resource allocation. *Feedback-based* control strategy deploys workloads directly onto machines while continuously monitoring SLO violations. If violations occur, this strategy employs actions such as system reconfiguration [56] or resource reallocation [16, 51, 63] to address the issues and maintain SLO compliance. However, the *profiling-based* strategy, while generating high throughput, incurs significant profiling overhead. This makes it challenging to deploy this strategy widely in large-scale cloud systems, especially considering the continuous emergence of new applications and potential consolidations in the cloud. *Feedback-based* strategy does not involve profiling overhead. However, the existing *feedback-based* approaches [16, 51, 56, 63] manage workload co-locations at the granularity of the entire LC service, disregarding the variations among different parts of the service when it is distributively deployed. This coarse-grained design results in low throughput. Furthermore, best-effort (BE) jobs are often treated as "second-class citizens" in this approach, with limited resource allocation and the possibility of being reclaimed and rescheduled arbitrarily to avoid SLO violations [35]. This conservative strategy further leads to wasted computation.

We suggest launching BE jobs more aggressively in a feedback way while capturing a little more profile information about the LC service (i.e., the *hybrid* strategy in Figure 1). Leveraging an aggressive policy would highly risk SLO violations; Therefore, it is crucial to carefully determine where and how to launch these jobs. With virtualization techniques evolving from VMs to containers, the controllable objects on cloud providers' side are becoming more varied and lightweight, offering greater flexibility in control methods. This shift allows for control at a finer level of granularity. In line with this, we propose a fine-grained controller built around the hybrid strategy. It differentiates the BE throughput launched with each LC *Servpod*, which is a new abstraction representing a collection of LC service parts deployed on the same physical machine. A *Servpod* acts as a unit for service deployment, defining the mappings between the LC structure and physical machines. By employing

this fine-grained controller, we can optimize the allocation and management of resources within the cloud environment, taking into account the specific characteristics and requirements of different *Servpods*.

To guide the launch of BE jobs, we conduct an analysis of the tail-latency contribution of each *Servpod*. This allows us to measure the weights of each *Servpod* in terms of overall tail latency. There are several challenges associated with this approach:

(1) How can we track a request and extract its sojourn time at each *Servpod* to achieve *Servpods* differentiation? As it is difficult to track a request in the public cloud due to the black-box design of VMs, we only consider private clouds where the sojourn time at each *Servpod* can be measured. An LC workload often includes multiple *Servpods*, and user requests may follow different paths within the service call. While program instrumentation can provide precise measurements of the sojourn time for each request in each *Servpod*, it typically requires a significant development cost. Therefore, we opt for a non-intrusive method, which involves deriving latencies in each *Servpod* from the large number of system events generated by processes.

(2) How can we transform the sojourn times in each *Servpod* into BE launch decisions? When analyzing the call path of an LC request, each *Servpod* plays a distinct role in the overall end-to-end tail latency. To guide the deployment of BE jobs, we define the *contribution* of each *Servpod* by considering its sojourn time mean, variance, and correlation coefficient. *Servpod* with a small contribution to the tail latency can be deployed along with more BE jobs.

As "second-class citizens", BE workloads may be suspended or even reclaimed whenever SLO violations occur. To prevent unnecessary computation resulting from arbitrary reclamation, we propose selectively controlling BEs based on their runtime behavior. Specifically, we prioritize the reclamation of BEs with lower preemption loss. There are also two challenges to this problem:

(1) How to differentiate the reclamation-priorities of BEs? Useless computation increases over the job progress and fault tolerant costs. It is generally not desirable to reclaim a job that is nearly completed, as it would result in wasted computation. However, certain types of jobs, such as distributed deep learning training, can tolerate a few worker failures without significant throughput loss. Therefore, we leverage the varying job progress and fault-tolerant abilities of BEs to determine their reclamation priorities.

(2) How can we implement the controller by making use of both the LC *Servpod* contributions and BE reclamation priorities? By monitoring the QPS (Query per Second) and the slack between the current tail latency and SLO target, the controller continuously makes decisions regarding resource allocation and release for the BEs based on the contribution analysis of *Servpods*. Under SLO violations, BEs are reclaimed according to their reclamation priorities.

To address these challenges, we present `Rhythm`, a cloud controller that maximizes the system throughput while guaranteeing LC service's tail latency SLO. `Rhythm` supports tracking the requests in all *Servpods*, and derives the contribution of each *Servpod* to the end-to-end tail latency. It implements a runtime agent at each Servpod, enabling the aggressive launch of BE jobs at *Servpods* with less contributions. For BE workloads, it monitors the job progress of BEs and evaluates their preemption losses into a MLRQ (Multi-Level Reclamation Queue) for discriminative BE reclamation. `Rhythm` carefully isolates interference between LC and BE jobs utilizing the hardware features, including cache isolation and DVFS (Dynamic Voltage and Frequency Scaling) and software isolation mechanisms (including core isolation, DRAM isolation and network traffic isolation). In summary, the contributions of this paper include:

- The insight of varying interference-tolerance abilities among different components of an LC service. (§ 2)
- Introducing a new abstraction called *Servpod* and analyzing the tail-latency contributions of *Servpods* enables the co-location of distinguishable workloads in a cloud system. (§ 3)
- A new reclamation mechanism, referred to BE-distinguishable reclamation, has been introduced to address the resource reclamation of BE jobs when there is a significant risk of SLO violations. (§ 3)

- A prototype system, Rhythm, designed following the hybrid strategy: "profiling LC once, feedback control BE." This design approach can be adopted in private clouds and offers high extensibility and flexibility. (§ 3 and § 4)
- A detailed comparative evaluation between Rhythm and non-component-distinguishable systems has been conducted, highlighting the performance improvements achieved by Rhythm in terms of system throughput while avoiding SLO violations. (§ 5)

2 BACKGROUND AND MOTIVATION

In this section, we first study the interference sensitivity of LC components. Then, we evaluate the preemption loss experienced by BEs when SLO violations occur due to interference.

2.1 Inconsistent Interference Tolerance Ability

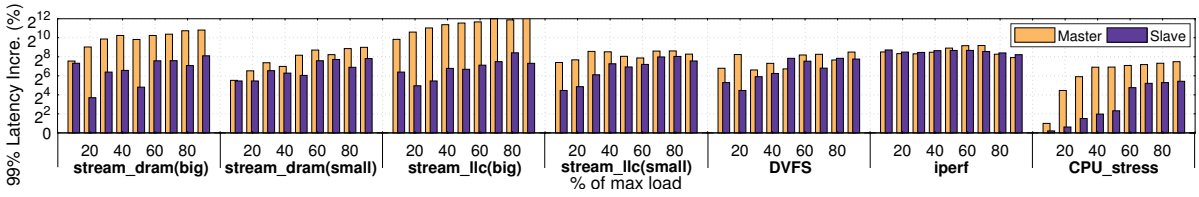
We study the interference sensitivity of LC components utilizing two typical LCs: the multi-tier *E-commerce website* [62] consists of four components, namely, *HAProxy*, *Tomcat*, *Amoeba* and *MySQL*, and the fan-out *Redis* [74] consists of two components, *Master* and *Slave*. In order to generate sufficient interference, we use five synthetic microbenchmarks as BE jobs, namely, *CPU-stress* [57], *stream-llc* [23], *stream-dram* [23], *DVFS* and *iperf* [88], which can put strong pressure on various shared resources. For evaluating the interference at different intensities of pressure on DRAM bandwidth and LLC, we also extend *stream_dram* (or *stream-llc*) to two intensity levels: *big* and *small*, where *big* means saturating the corresponding DRAM bandwidth (or LLC), while *small* means occupying only half of the whole capacity.

Each LC service component is deployed together with a BE job on the same machine to measure the impact on the overall $99th$ percentile latency. For measuring the contention on cores, we pin the component and CPU-stress on the cores from the same socket. For measuring LLC interference, we pin the component and stream-llc to different cores from the same CPU socket since they have separate L1/L2 cache but share L3. For measuring interference on DRAM bandwidth, we use `numactl` [66] to place the component and stream-dram on the same socket without CPU core usage overlap. In addition, we use DVFS to adjust the frequency of processors holding the component to evaluate its impact on tail latency. All experiments share the same settings with those in § 5, and each run is repeated 5 times for reducing errors.

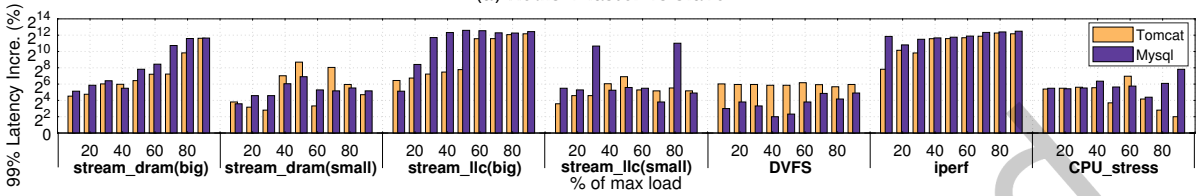
We evaluate the performance degradations of two LC services under the interference over increased request load, the characterization results are presented in Figure 2.

Redis: Figure 2a shows the increase in latency when we co-schedule the Master or Slave of Redis with BE jobs. We see that the performance degradation under interference generally increases over the request load. Master is more sensitive than Slave in most interference groups, and their difference varies with the BE jobs. In particular, since the Master strongly relies on LLC, memory and network bandwidth for both requests distribution and data operation, it is particularly more sensitive to interference caused by stream-dram (big), stream-llc (big) and CPU-stress than Slave. The difference between Master and Slave even exceeds 28× under the same interference from stream-llc (big). For interference from stream-dram (small), stream-llc (small) and DVFS, the differences of the latency increase between Master and *Slave* also reach 155.1%, 181.1% and 122%, respectively. CPU-stress generates the least interference, which increases the latency by an average of 113.1% at the Master and 22% at the Slave, resulting in 91.1% difference.

E-commmerce website: Figure 2b shows the increase in latency when we co-schedule the Tomcat or MySQL of E-commmerce with BE jobs. MySQL is more sensitive to interference generated by stream-dram (big), stream-llc (big), CPU-stress and iperf. The differences of the latency increase between Tomcat and MySQL reach 435.8% and 35.1% under the interference from stream-dram (big) and CPU-stress. In case of stream-llc (big), the difference between Tomcat and MySQL reaches more than 35×. Tomcat is more sensitive to interference generated by



(a) Redis: Master vs Slave



(b) E-commerce website: Tomcat vs MySQL

Fig. 2. Impact of interference on the 99th percentile latency of LC service: the X-axis represents the interference groups of LC service components and BE jobs under different percent of the maximum request load. The Y-axis represents the corresponding increase in 99th percentile latency normalized to the solo-run performance (presented in logarithmic scale).

DVFS, and the differences between Tomcat and MySQL is 416.7%. In groups of stream-dram (small) and stream-llc (small), the differences are 71% and 13.2%, respectively.

Hence, the impact of interference on different components of LC service exhibits significant inconsistency. While resource contention affects highly sensitive components and leads to SLO violations, less sensitive components can accommodate more BE deployment. This highlights the inefficiency of existing approaches that uniformly control the co-location of LC workload and BE jobs, as they overlook the varying interference tolerance abilities of individual components (aka "**Law of the Minimum**" [33]).

2.2 Inconsistent Preemption Loss of BEs

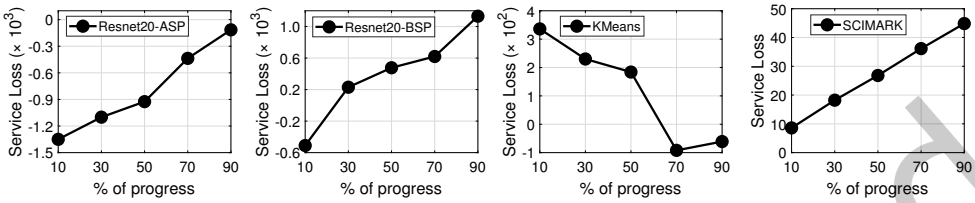
The inconsistent preemption loss of BEs refers to the variation or disparity in the level of negative impact experienced by different BEs when they are preempted whenever SLO violations occur. It means that some BEs may suffer more severe consequences or incur higher costs when resources are reclaimed from them compared to others. A typical reclamation process can be divided into three stages: (1) If the computing cluster still has idle resources available, the controller will reschedule the co-located BEs to utilize these idle resources. This allows the BEs to continue running without interruption. (2) If there are no idle resources in the cluster, the controller will choose to reduce the resource allocation of the currently running BEs instead of terminating them. This means that the resources assigned to the BEs will be reduced, which will result in the BEs running at a lower speed or with limited capacity. (3) If the SLO is still being violated even after reducing the resources allocated to the BEs, the controller will take a more aggressive approach. It will preempt the resources assigned to the co-located BEs, effectively stop their execution, and allocate all those resources to the LCs. By doing so, the controller prioritizes the performance of the LC service and ensures that it receives sufficient resources to meet the SLO requirements.

For evaluating the preemption losses, we deploy four real BE benchmarks with different fault tolerance ability on our testbed (§ 5). We evaluate their preemption losses by terminating one of its tasks at midway ranging in [10%, 90%]. The preemption loss is defined as the lost service as below:

$$L = S_{pmtn} - S_{ognl} = t_{pmtn} r_{pmtn} - t_{ognl} r_{ognl} \quad (1)$$

BEs	Description	configuration
Resnet20-ASP [4]	Deep learning model for image classification in asynchronous training mode	Two workers (5 CPUs/worker)
Resnet20-BSP [4]	Deep learning model for image classification in synchronous training mode	Two workers (5 CPUs/worker)
KMeans [82]	Bigdata application implemented on Spark	Three exectuors (2 CPUs/exectuor)
SciMark [2]	A Java benchmark for scientific and numerical computing	Single binary file (2 CPUs)

(a) BE workloads Description



(b) Service Loss ($Time \times CPU$) of different BEs under resource reclaim

Fig. 3. BE benchmarks and their preemption losses.

where t_{pmtn} (or t_{ognl}) represents the makespan of the BE under preemption (or without preemption). r_{pmtn} (or r_{ognl}) represents the resources (e.g., number of CPU cores) occupied by the BE under preemption (or without preemption).

Figure 3 illustrates the significant variability in preemption loss among different BEs. In the case of Resnet20-ASP, preemption actually benefits the workload because terminating a worker in asynchronous mode does not result in job failure or the need for worker rescheduling. Despite the reduction in occupied resources, the makespan of Resnet20-ASP remains relatively stable under preemption. Therefore, task preemption in Resnet20-ASP improves service efficiency in our configuration. However, for Resnet20-BSP, where workers need to be synchronized and any failed worker requires restarting from the latest checkpoint, terminating a worker after 30% progress leads to service loss. It is common for a later preempted task to incur higher loss for a job. For example, the preemption loss of Scimark grows linearly with progress (Figure 3b). Since we did not provide any fault tolerance mechanism for Scimark, each preemption triggers resubmission and rerunning from the beginning. Conversely, in the case of KMeans, we observe that later preemptions result in less loss. This can be attributed to two factors: (1) The Resilient Distributed Datasets (RDDs) [98] offer high fault tolerance for Spark applications, enabling the Spark scheduler to quickly recover from task failures regardless of when they occur. (2) Applications often execute in phases [78]. We find that preemption at 70% progress generates minimal contention among Spark executors. Thus, reclaiming resources at this stage has little effect on the overall makespan.

According to the varying preemption losses of BEs, it is feasible to design a BE-distinguishable reclamation mechanism to further enhance system throughput. We deploy Redis as the LC on our testbed (§ 5) and co-locate it with 20 different BE workloads, comprising 5 bigdata analysis jobs, 5 distributed deep learning training jobs (following the parameter-server architecture), and 10 scientific computing jobs. We compare our `Rhythm` with two BE-indistinguishable reclamation mechanisms: *Resource deflation* [77] which aims to delay the preemption as much as possible using a cascading reclamation technique across multiple levels (applications, operating systems, and hypervisors); and *Heracles* [51] which rapidly restores LC's latency by directly disabling co-located BEs when the latency approaches the SLO target.

Figure 4 illustrates the changes in EMU, CPU utilization, and the $99th$ percentile latency of Redis over time. EMU represents the effective machine utilization, which is the sum of LC throughput and BE throughput (§ 5). During the time intervals $[261, 486]$ and $[657, 855]$, the request load towards Redis increases from 65% to 85% of the maximum QPS for creating significant interference. For example, at time 261, Redis experiences a sudden increase in access load, requiring more resources to maintain the desired tail latency. In this scenario, *Heracles*

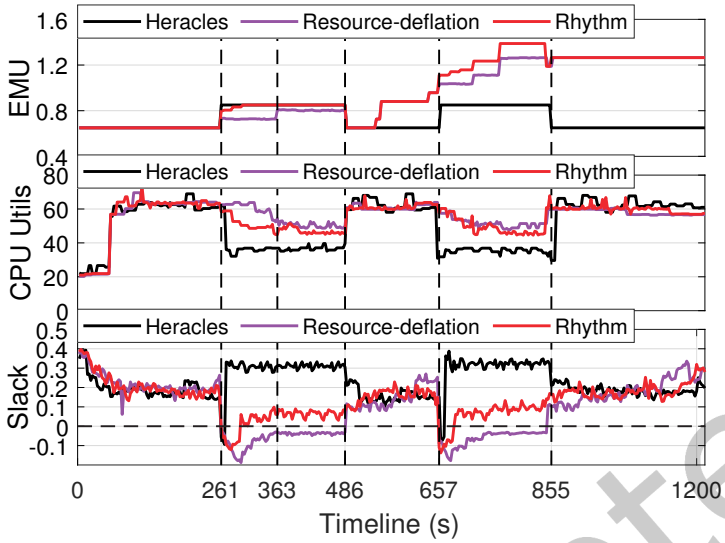


Fig. 4. A scheduling scheme that does not distinguish between BE loads is inefficient.

chooses to preempt all running BEs, prioritizing fast SLO recovery at the expense of resource utilization. On the other hand, *Resource Deflation* adopts a gradual but slow reclamation process, maintaining high CPU utilization but failing to ensure the SLO during this period. In contrast, our `Rhythm` distinguishes between BE workloads based on their preemption costs and selectively preempts the ones with lower costs. This approach achieves a high EMU and CPU utilization while achieving a quicker recovery of tail latency.

2.3 Implications

In summary, by leveraging the inconsistent interference tolerance abilities of LC components, we can control the co-locations at LC components differently and aggressively. Specifically, co-locating more BE workloads with the low-sensitive components of an LC is less likely to result in SLO violations. However, quantifying the interference tolerance abilities of LC components can introduce overhead to the controller. To mitigate this overhead, we propose quantifying the tail latency contribution of each LC component, which only requires a single solo-run profiling process.

Similarly, by leveraging the inconsistent preemption loss characteristics of BEs, we can control the reclamation process of BEs differently when SLO violations occur. Specifically, BEs with lower preemption loss are given higher priority for reclamation. To reduce the evaluation cost of preemption losses for BEs, we only consider their runtime available information. This approach helps optimize the resource reclamation process while minimizing the computational overhead.

3 RHYTHM DESIGN

In this section, we present the design of `Rhythm` and show where and how to launch or reclaim BEs making use of the features of LC *Servpods* and BE workloads.

3.1 The *Servpod* Abstraction

We introduce *Servpod*, which refers to a collection of service components from a single LC workload that are deployed together on the same physical machine. In the context of a directed acyclic graph (DAG) representation of an LC service, where vertices represent LC components and edges indicate the precedence relation among them,

a *Servpod* can comprise multiple LC components that are scheduled on the same physical machine. Consequently, *Servpod* provides insights into the mappings between physical machines and the service structure of the LC workload. The number of *Servpods* corresponds to the number of deployed physical machines.

A *Servpod* could be one or multiple processes, containers or microservices [31]. We do not discuss the scheduling problem of LC components here, but assume that an LC has already been scheduled on physical machines, generating multiple *Servpods*. Given the distributed *Servpods*, we next explore how to deploy the number of BE jobs differently along with them.

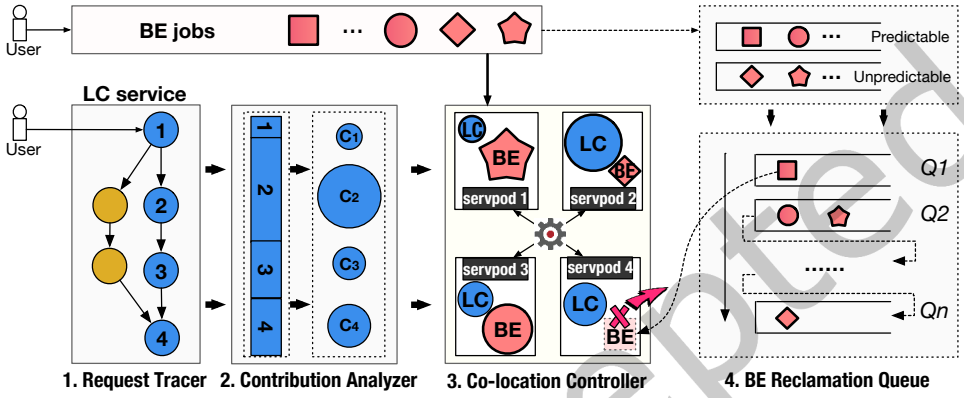


Fig. 5. Design of Rhythm.

3.2 System Overview

The insight of Rhythm is that, the system throughput can be significantly improved through distinguishable control over LC *Servpods* and BE workloads: (1) *Servpods* with larger contributions to the tail latency need to be controlled conservatively to ensure adherence to the SLO. On the other hand, *Servpods* with lesser contributions can be allowed to launch BE jobs aggressively, maximizing resource utilization. (2) In the event of SLO violations, BEs with lower preemption losses are prioritized for suspension or termination. This approach helps reduce unnecessary computation, improving overall system efficiency.

Figure 5 highlights the overall design of Rhythm. Quantifying the *contribution* of an LC *Servpod* can be done through an offline profiling of *Servpods* or an online analysis of real-time monitoring. We choose the offline profiling way for three reasons: (1) As the contribution of a *Servpod* depends on various factors, like the sojourn time, the access load, an online exploration process may take a very long time until collecting sufficient data, resulting in frequent SLO violations during this period. (2) Offline profiling can be conducted along with the necessary stress test before the launch of a service, saving much profiling cost. (3) Unlike short BEs, LCs are commonly long-running services, and the profiling cost can be amortized over time. As shown in Figure 5, we first characterize the contributions of LC *Servpods* using two modules: *request tracer* and *contribution analyzer*. Then, we manage the running of BE jobs using a *controller* at each physical machine.

The offline profiling of a *Servpod*'s contribution to the overall tail-latency includes two ways: The *directed way* involves collecting the sojourn times of requests in each *Servpod*. The contribution can then be derived through statistical modeling and analysis of these sojourn times. The *indirected way*, like "bubble pressure" [59], applies a tunable pressure on individual *Servpods* to measure the performance degradation of the LC service (e.g., increased tail latency or IPC). The contribution is defined as the "bubble size" that a *Servpod* can tolerate while still guaranteeing the SLO. However, as shown in Figure 2, a *Servpod* exhibits varying sensitivities when subjected to interferences from different bubbles [69]. In addition, the workload variation also significantly affects the performance of the LC service. Relying solely on the "bubble pressure" approach, which generates

one-dimensional interference, is inadequate. For example, a CPU-intensive *Servpod* that contributes significantly to the overall tail latency might be able to tolerate strong interferences from an I/O-intensive bubble. Furthermore, designing a single bubble suite that can represent all types of BE jobs is impractical. Therefore, we choose the *directed way* to characterize the contributions of *Servpods*.

The *request tracer* of Rhythm identifies the service call paths of requests and records their sojourn time at each *Servpod* when the LC service runs independently. This data is then used by the *contribution analyzer* to calculate the contribution of each *Servpod* to the tail latency. The *contribution* is derived using statistical metrics such as mean, variance, and the Pearson correlation coefficient of the sojourn times. This characterization is based solely on the LC service itself, and its cost increases linearly with the number of *Servpods*. Compared to *profiling-based* approaches that measure interference in combinations of M LC services and N BE jobs (resulting in $M \times N$ measurements), Rhythm significantly reduces the cost to just M. The *controller* utilizes a *contribution*-based threshold methodology to control resource allocation for BE jobs on each machine. It calculates control thresholds for different *Servpods* using a thresholding algorithm and then employs a trial-and-error approach to deploy or reclaim BEs.

The *controller* maintains a reclamation priority queue for all BEs and updates it periodically. When an SLO violation occurs, the *controller* selects the BE with the highest priority from the queue for reclamation. The BE workloads are categorized into two groups: *predictable* and *unpredictable*. *Predictable* BEs are those whose makespan can be easily and accurately estimated. For example, the makespan of MapReduce or Spark applications can be estimated based on the proportion of processed data. Rhythm can utilize offline profiling-free prediction models like [71, 77] to estimate the makespan of these BEs. With the predicted makespan, the preemption losses of the predictable BEs can be derived using Formula 1. The opposite of the preemption loss is considered as the reclamation priority for these BEs. On the other hand, *unpredictable* BEs have makespans that are difficult to estimate accurately. In such cases, it is not possible to derive their preemption losses under reclamation. Instead, the *unpredictable* BEs are prioritized based on the direct computation loss, which is the amount of recomputation required after preemption. To guide the reclamation process, the *predictable* and *unpredictable* priority queues are combined using a Multi-Level Reclamation Queue (MLRQ) method based on the Borda count. This method assigns weights to the priority queues and combines them to determine the overall reclamation priority.

For a newly deployed LC service, we activate both the *request tracer* and *contribution analyzer* only once for characterizing its *Servpods*' contributions. While each server *controller* continues running independently for controlling BE jobs along with each *Servpod*, its thresholding algorithm (i.e., the only step that requires coordination among *Servpods*) also runs only once to derive thresholds. Hence, the characterization cost is low and Rhythm has good scalability.

3.3 Request Tracer

Each request towards an LC service may pass a number of different *Servpods*. The *request tracer* identifies the causal path of a request and constructs a *causal path graph* (CPG), which is a directed acyclic graph $G(V, E)$ describing the request process. Vertices in V are event sets of *Servpods*, and edges E represent causal relations between events. To record the sojourn time in which a request stays at a *Servpod*, we also need to record the arrival and departure time at each *Servpod*. When there are multiple components in a *Servpod*, we only record the arrival time at the entry component and the departure time at the exit component.

The key challenge is to capture the system events in V and find the causality of them. We collect the relevant system calls at LC *Servpods*. However, the calling stack in a *Servpod* could be associated with a depth of more than hundreds of system calls due to the frequent switches between the user space and kernel. Many of them are generated by other unrelated processes, including operating system processes or other applications. To filter the unrelated events, we record four specific events in each LC *Servpod*: *syscall_accept* indicates the acceptance of a

request; *tcp_sendmsg* represents the sending of data package; *tcp_rcvmsg* represents the receiving of data package; and *syscall_close* is the close of a request call, where we denote them as ACCEPT, RECV, SEND and CLOSE, respectively. Each event is structured with four attributes: *event type*, *timestamp*, *context identifier* and *message identifier*. In particular, the *context identifier* is defined as a quad: $< hostIP, programName, processID, threadID >$, which can be used to filter out noise system calls from unrelated processes. The *message identifier* is defined as a five-tuple: $< senderIP, senderPort, receiverIP, receivePort, messageSize >$, which can be used to filter out noises from unrelated communications.

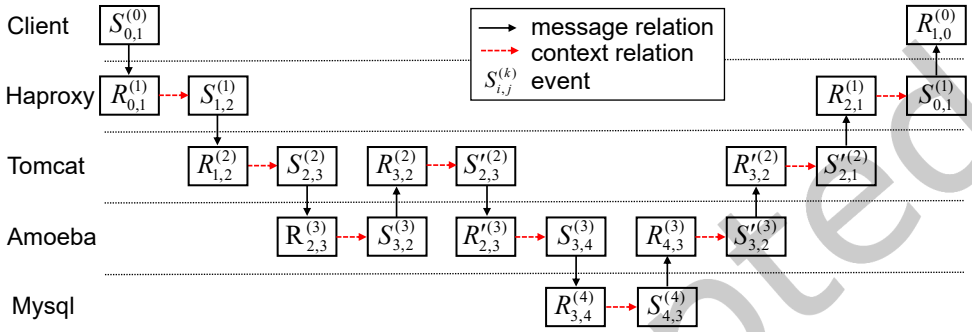


Fig. 6. The CPG constructed by a request to e-commerce.

Next, we show how to identify the causality of these events, including *intraServpod causality* and *interServpod causality*. *IntraServpod causality* denotes the causality of a pair of RECV and SEND events inside a Servpod. We use the *context identifier* to identify their causality. That is, a RECV event happens before a SEND event if they share the same *hostIP*, *program name*, *process ID* and *thread ID*. *InterServpod causality* denotes the causality of a pair of SEND and RECV events between neighbor *Servpods*. A SEND event happens before a RECV event at the neighbor *Servpod* if they share the same *message identifier*.

Denote by $S_{i,j}^{(k)}$ (or $R_{i,j}^{(k)}$) the SEND (or RECV) event recorded in node k; i, j represents the data flow from node i to j. Figure 6 shows an example CPG constructed by a request to E-commerce. Note that there may be hundreds of system events in the process, we only list part of them here.

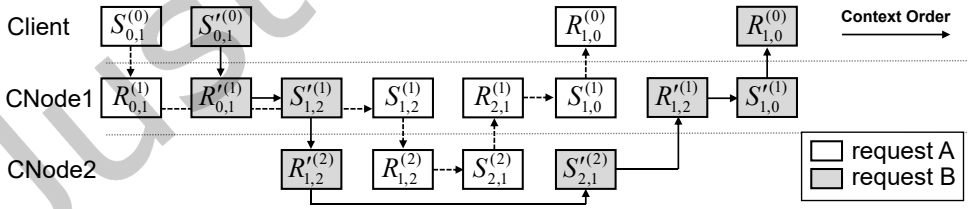


Fig. 7. IntraServpod causality: request B is issued earlier by Servpod1, but returned later than request A.

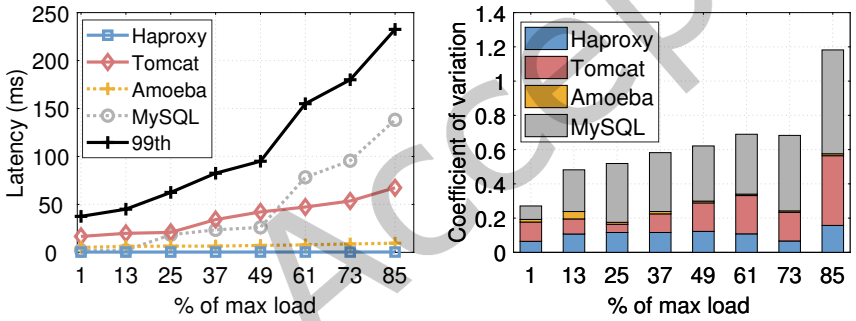
How is the intraServpod causality of nonblocking threads identified? Each RECV event is matched with a SEND event with respect to their order of occurrence (i.e., timestamp). If the LC thread runs in blocking mode, this order can be detected easily using the *context identifier*. In case of nonblocking threads, a later-issued request may return earlier than an earlier-issued request (Figure 7).

Since they may share the same *context identifier*, the mappings of RECV and SEND of them would be incorrect, resulting in an incorrect calculation of sojourn time at the *Servpod*. In this case, we do not identify the accurate

intraServpod causality directly, but avoid its side effect on the calculation of the sojourn time through analyzing the mean sojourn time of all requests. For example, the mean sojourn time of request A and B at *Servpod* 1 in Figure 7 is not affected by the mismatching, because we have $(S^{(1)}_{1,2} - R^{(1)}_{0,1}) + (S^{(1)}_{1,0} - R^{(1)}_{2,1}) + (S^{'(1)}_{1,2} - R^{'(1)}_{0,1}) + (S^{'(1)}_{1,0} - R^{'(1)}_{2,1}) = (S^{(1)}_{1,2} - R^{'(1)}_{0,1}) + (S^{(1)}_{1,0} - R^{(1)}_{2,1}) + (S^{'(1)}_{1,2} - R^{(1)}_{0,1}) + (S^{'(1)}_{1,0} - R^{'(1)}_{2,1})$.

How is the interServpod causality of persistent TCP connections identified? If the communication between LC neighbor *Servpods* is implemented using persistent TCP connections, multiple requests may share the same *message identifier*. In this case, pairing SEND and RECV events with respect to their order of occurrence could also lead to mismatching (i.e., incorrect interServpod causality). Similar to the calculation in the intraServpod causality, we avoid its side effect by analyzing the mean sojourn time of communications of all requests in the design of the *contribution analyzer*. That is, the mean sojourn time of communications between *CNode 1* and *CNode 2* in Figure 7 is $(R^{'(2)}_{1,2} - S^{'(1)}_{1,2}) + (R^{(2)}_{1,2} - S^{(1)}_{1,2}) + (R^{(2)}_{2,1} - S^{(2)}_{2,1}) + (R^{'(1)}_{2,1} - S^{'(2)}_{2,1}) = (R^{'(2)}_{1,2} - S^{(1)}_{1,2}) + (R^{(2)}_{1,2} - S^{'(1)}_{1,2}) + (R^{(1)}_{2,1} - S^{'(2)}_{2,1}) + (R^{'(1)}_{2,1} - S^{(2)}_{2,1})$. Note that errors still can be caused by unsynchronized clocks between machines. Fortunately, it can be compensated if the *Servpods* are called following a *nested chain model* (i.e., a *Servpod* invokes a callee *Servpod* and waits for the callee's results before it can return). Otherwise, there will be a skew in the derived communication sojourn time.

3.4 Contribution Analyzer



(a) Average sojourn time (b) Normalized coefficient of variations

Fig. 8. The average sojourn time of *Servpods* in E-commerce website and their normalized coefficient of variations collected in solo-run.

Figure 8a shows the average sojourn time of the four *Servpods* of E-commerce and the overall 99*th* percentile latency under different request loads. Figure 8b shows the normalized coefficient of variation of their sojourn times. We find that, *HAProxy* contributes less than 5% of the overall latency, while its variance takes more than 20% among four *Servpods*. *Amoeba*'s sojourn time is also small but very stable, i.e., its coefficient of variance is the smallest. For the *MySQL* and *Tomcat*, when the load is less than 50% of the *max load*, *MySQL* has a smaller average sojourn time than *Tomcat*, and when the load exceeds 50%, its sojourn time increases much faster than that of *Tomcat*. However, *MySQL*'s variance is always much larger than *Tomcat*. Based on the observation, we consider three principles that guide our definition of *contribution*:

(1) *Servpods* with a higher average sojourn time contribute more to tail latency. The first principle highlights the average sojourn time of each Servpod. Tail latency surely increases over each *Servpod*'s average sojourn time. For example, *MySQL* contributes the most on 99*th* percentile latency when the load is high in Figure 8a.

(2) *Servpods* with higher sojourn time variance contribute more to tail latency. This principle relates tail latency to the fluctuation characteristic of each *Servpod*, since the fluctuations constitute the "heavy-tail" of overall latency. For example, while *Tomcat* and *MySQL* have a similar average sojourn time when the request load is in

the range [25%, 49%]. However, the $99th$ percentile latency increases significantly due to the high variance of *MySQL* (Figure 8b).

(3) *Servpods* that are highly correlated with the tail latency contribute more to tail latency. Suppose there exists a *Servpod X* which has a constant mean and coefficient of variance of sojourn times over different loads, then the vary of tail latency would be independent of *X* if the *contribution* is derived merely based on the mean and normalized and coefficient of variance. Hence, we also analyze the correlation between each *Servpod*'s sojourn time and the tail latency, and take it as an important factor of the *contribution*.

Following the principles above, we next show how to derive the *contribution* of a *Servpod*. Since the request sojourn time at each *Servpod* may be incorrect due to the mismatch of SEND and RECV, we use the mean sojourn time in the definition. Denote $\overline{T_i}$ as the average sojourn time of *Servpod i* under all load levels and T_i^j as the average sojourn time of *Servpod i* under load *j*; then, we have $\overline{T_i} = \sum_{j=1}^{m} T_i^j / m$, where *m* is the number of loads we used. We derive the weight of average sojourn time by *Servpod i*'s as follows, where *n* is the number of Servpods.

$$P_i = \frac{\overline{T}_i}{\sum_{k=1}^{n} \overline{T}_k} \tag{2}$$

We use the *Pearson Correlation Coefficient* ($\rho_{T_i,T_{tail}}$) to evaluate the correlation between *Servpod i* and the overall tail latency of the LC service (T_{tail} denotes the overall tail latency, and $tail$ could be the $99th$, $99.9th$ percentile, etc.). Let T_{tail}^j be the $tail$ latency under load j, then, we have,

$$\rho_{T_i,T_{tail}} = \frac{\sum_{j=1}^{m} (T_i^j - \overline{T}_i)(T_{tail}^j - \overline{T}_{tail})}{\sqrt{\sum_{j=1}^{m} (T_i^j - \overline{T}_i)^2} \sqrt{\sum_{j=1}^{m} (T_{tail}^j - \overline{T}_{tail})^2}} \tag{3}$$

We denote *V* as the normalized coefficient of variation (*V*) to derive the contribution by the *Servpod i*'s variance as follows,

$$V_i = \frac{1}{\overline{T}_i} \sqrt{\frac{1}{m(m-1)} \sum_{j=1}^{m} (T_i^j - \overline{T}_i)^2} \tag{4}$$

Finally, we define the contribution of *Servpod i* using their product:

$$C_i = f(\rho_{T_i,T_{tail}}, P_i, V_i) = \rho_{T_i,T_{tail}} P_i V_i \tag{5}$$

If there exists fan-out in a request, the end-to-end latency is determined by the latency of the critical path, i.e., the path (denoted by *R*) with the longest time. A *Servpod i* not on *R* can tolerate stronger interference than those on *R*, and its contribution can be scaled down to:

$$C_i = \alpha_i \rho_{T_i,T_{tail}} P_i V_i \tag{6}$$

where $\alpha_i = \sum_{j \in R_i} \overline{T_j} / \sum_{k \in R} \overline{T_k}$, and R_i denotes the *Servpod* set on the path that is non-critical but longest among all paths through *Servpod i*.

Note that Equation 6 may not be the only way to define the contribution. We validate its rationality though a comparative analysis between *Servpod* sensitivity and contribution. Figure 9 shows their correlation: The x-axis depicts the contributions of the four *Servpods* of E-commerce, and the y-axis shows the sensitivity of them, which is defined as the increase in the $99th$ percentile latency under interference compared to that under the solo-run. We find that the sensitivity is positively correlated with the contribution no matter what the BE is, proving that a *Servpod* with higher contribution is usually more sensitive to interference. We implement the Servpod-level

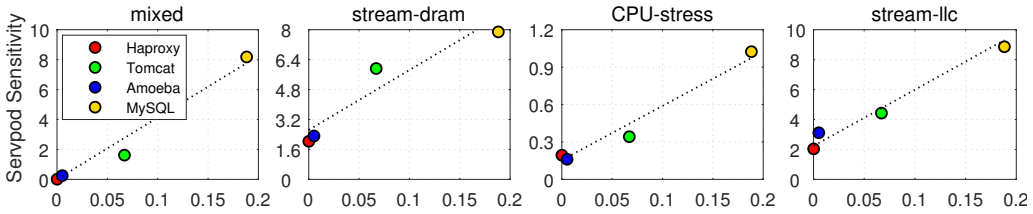


Fig. 9. *Servpod* sensitivity vs contributions: the increase in the 99*th* percentile latency of E-commerce when a single *Servpod* is interfered by different BEs: (1) mixed BEs of wordcount, imageClassify, LSTM, CPU-stress, stream-dram and stream-llc, (2) Stream-dram, (3) CPU-stress, (4) Stream-llc.

control in algorithm 1 based on this contribution, and the experimental results show it works well. (See Figure 13-15).

3.5 Co-locating Controller

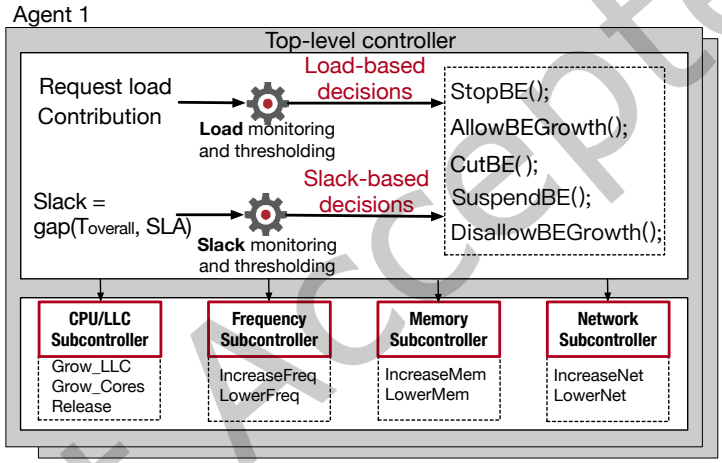


Fig. 10. The architecture of a Rhythm agent

Given the contributions of LC *Servpods*, we next present how the controller operates to control the resource allocation and reclamation for BE workloads. We design a coordinated controller running as an agent at every server holding the LC *Servpod*. Figure 10 shows its hierarchical architecture: a top-level controller and four subcontrollers. The top-level controller makes decisions on the BE jobs, including a load-based decision and a slack-based decision. The four subcontrollers increase or decrease the resources allocated to BE jobs following the control instructions by the top-level controller. In particular, top-level controllers first coordinate with each other to derive two thresholds: *loadlimit* and *slacklimit* using the threshold mechanism. Then, each controller runs independently to control BE jobs according to thresholds. If the top-level controller decides to cut or stop BEs, it reads the BE reclamation queue and selects the highest priority one for operation. The controller scales well as the number of *Servpods* increases because they do not have interactions anymore after finding thresholds.

3.5.1 ***Thresholding Mechanisms.*** Under the solo-run of the LC service, we derive two thresholds of *loadlimit* and *slacklimit* in each machine using the request load and *contributions* of *Servpods*. Since *contributions* vary over LC *Servpods*, the thresholds are also different. In particular, *loadlimit* denotes the "switch" determining whether or not to run BE jobs; *slacklimit* decides how many resources are allocated to BE jobs.

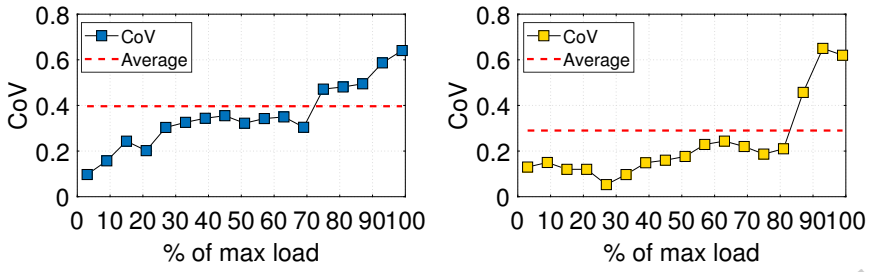


Fig. 11. The *CoV* of *Servpod* sojourn times increase over request loads. Determine the *loadlimit* of *Servpods* in E-commerce using the first load point whose fluctuation is greater than the average. (CoV: normalized coefficient of variation).

Loadlimit: The threshold *loadlimit* denotes the upper bound of the request load of the LC service for allowing the running of BE jobs along with an LC *Servpod*. We configure this threshold using the CoV of sojourn times across different requests at each *Servpod*. Figure 11a shows the volatility of *CoV* by the *MySQL Servpod* over the request load. We see that the fluctuation tends to increase significantly when the request load exceeds 76% of the maximum allowable load. We choose *loadlimit* as the first load point whose fluctuation is greater than the average. That is, we have $loadlimit = 76\%$ for the *MySQL* of the E-commerce website, meaning that if the load towards *MySQL* exceeds 76% of the maximum allowable load, we have to suspend all BE jobs on this MySQL machine. In the case of *Tomcat*, the *loadlimit* is 87% (Figure 11b).

Slacklimit: Let *slack* be the gap between the current tail latency and latency target in SLO. The threshold *slacklimit* denotes the lower bound of *slack* for allowing the growth of BE jobs. It is inversely related to the *Servpod*'s interference-tolerance factor. If a *Servpod* has a small contribution to the overall latency, we only need a small *slacklimit* so that more BE jobs can be deployed in this machine, or the subcontrollers can allocate more resources to BE jobs. We design an iterative algorithm to find the best *slacklimit* for each machine based on LC *Servpods*' *contributions*.

Algorithm 1 presents the details. We first normalize the contribution of each *Servpod* among all LC *Servpods* and initialize the *slacklimit* of each *Servpod* with 1.0. The normalized contribution will be used as a stepsize for updating the *slacklimit*. The algorithm proceeds in a while loop until finding the minimum *slacklimit* with the SLO guarantee. In each loop, we gradually decrease the value of the *slacklimit* of each *Servpod* by their respective *stepSize*. Then, we run the LC workload at this configuration for a while. If the SLO is violated, we step backward and update the *slacklimit*.

Algorithm 1 may have different outputs of *slacklimit* depending on the BE used during *run_system* (*curLimit*). We recommend to run the algorithm with representative, mixed-intensive BEs and run multiple times to increase its accuracy. In our experiment, the best *slacklimit* for *Tomcat* and *HAProxy* are 0.078 and 0.032, respectively, whereas for *MySQL* and *Amoeba*, they are 0.347 and 0.04, respectively. Hence, we can launch many more BE jobs on *Amoeba*, *Tomcat* and *HAProxy* than on *MySQL*.

3.5.2 Control Operation. *Loadlimit* and *slacklimit* define a reliable boundary for startup and adjustment of BE jobs, enabling `Rhythm` to implement more precise control over BEs based on *Servpod* while ensuring SLO compliance.

Top controller: `Rhythm` compares the real-time request load and slack with the *Loadlimit* and *slacklimit* and manages the running of BE jobs through the five decisions released: StopBE, CutBE, DisallowBEGrowth, AllowBEGrowth and SuspendBE. In particular,

(1) *StopBE* immediately kills the top job in BE reclamation queue and releases all its resources.

Algorithm 1: *findSlacklimit(C_i)*

Input: *contribution* of *Servpod:* C_i, $\forall i \in [1..n]$;
Output: *slacklimit* for *Servpod* i;

```
1  stepSize = 1 − C_i / Σ_{i=1}^{m} C_i ;
2  slacklimit = curLimit = 1.0; // Initialization ;
3  SLO_violation = false ;
4  while curLimit > 0 do
5      curLimit = curLimit − stepSize ;
6      run_system (curLimit) ;
       // Running for 10 minutes ;
7      SLO_violation=SLO_evaluation() ;
8      if SLO_violation = true then
9          slacklimit = Record.pop() ;
10         break;
11     else
12         Record.push(curLimit) ;
```

Algorithm 2: Decision making by top controller

```
1  Estimate slack based on monitored latency and SLO;
2  while True do
3      slack = (T_tail^SLO − T_tail) / T_tail^SLO ;
4      if slack < 0 then
5          StopBE();
6      else if workload > loadLimit then
7          SuspendBE();
8      else if 0 < slack < slackLimit/2 then
9          CutBE();
10     else if slackLimit/2 < slack < slackLimit then
11         DisallowBEGrowth();
12     else
13         AllowBEGrowth();
14     sleep(2 seconds);
```

(2) *SuspendBE* pauses all of the running BE jobs, but they can still keep their memory space.
(3) *CutBE* allows the existing BE jobs to continue running, but reclaims part of the allocated resources from the top job in BE reclamation queue.
(4) *DisallowBEGrowth* does not allow the number of BE jobs increase, but the existing BE jobs can still hold their resources and run continually.
(5) *AllowBEGrowth* allows subcontrollers to allocate more resources to BE jobs and increase the number of BE jobs.

Denote T_{tail}^{SLO} as the tail latency requirement stated in SLO. The decision-making algorithm is shown in Algorithm 2.

Subcontroller: There are four subcontrollers in each machine. They periodically adjust the resource allocations between LC service and BE jobs following instructions from top controller and BE reclamation queue. While frequent monitoring and adjustment are effective to detect the load burst and protect the SLO of LC workload, it also causes more runtime overhead. To assess the tradeoffs between efficiency and performance, we set the operation period of each controller thread to 2 seconds. The experimental results also validate the reliability of our design. (Figure 19).

(1) CPU/LLC subcontroller: We adopt the same control as in *Heracles* [51] for allocating cores, LLC, and memory bandwidth. When it is allowed to deploy BE jobs, a BE job is activated and configured with one core, 10% memory bandwidth, and 10% LLC. Both *CutBE* and *AllowBEGrowth* adjust the cores, MB and LLC of BE jobs at the granularity of one core, 10% memory bandwidth and 10% LLC, until no more resources are available or all BE's resources have been released.
(2) Frequency subcontroller: It monitors the power of the CPU periodically and adjusts the frequency using DVFS. If the power has exceeded 80% of TDP (thermal dissipation power) and the frequency of the LC service is less than the minimum allowable frequency (for meeting SLO), it will reduce the operating frequency of BE jobs at the stepsize 100 MHz to ensure sufficient power for the LC service.
(3) Memory subcontroller: It monitors the memory utilization of the LC service. A newly started BE job is initialized with 2 GB of memory, and the adjustment stepsize for *CutBE* and *AllowBEGrowth* is 100 MB.
(4) Network subcontroller: It continuously monitors the bandwidth of LC services (B_{LC}), and allocates bandwidth of $B_{link} - 1.2B_{LC}$ to BE jobs.

3.6 BE Reclamation

While operations *StopBE* and *CutBE* are activated by Algorithm 2, the reclaimed resources from BEs are redistributed to LCs in order to restore the SLO. However, selecting the appropriate BE to reclaim can be challenging, as it requires finding a balance between quick SLO recovery and minimal throughput degradation. We have observed that the majority of running BEs in datacenters can be categorized into three groups: *big data*, *artificial intelligence (AI)* and *scientific computing*. *Big data* applications, such as MapReduce and Spark [3, 36, 82, 90], involve processing large datasets, and their running time can be estimated based on the progress of data processing. *AI* training [4, 61, 85] focuses on achieving a desired level of accuracy for neural models, and their running time depends on the convergence rate of the models rather than data processing. *Scientific computing* [1, 53] primarily involves short-term computations that don't involve extensive data processing.

Different BE applications have varying structures, with some being monolithic and others consisting of multiple components. Reclaiming resources from different BE components can have diverse effects on the BE throughput. It may simply reduce the processing speed or, in some cases, even prevent the BE from running altogether. To minimize these negative impacts, we evaluate how resource reclamation affects the *occupied service* of each BE. The *occupied service* is calculated using $resource \times time$. In Formula 1, if the *occupied service* of a BE increases after reclamation, it indicates a preemption loss ($L > 0$). This preemption loss quantifies the negative impact on the BE due to resource reclamation.

To derive L, we need to know the running time of the BE, i.e., t_{pmtn} and t_{ognl}. We classify BEs into two categories based on their predictability. If a specific BE has a predicting model that can accurately estimate its running time, we classify it as *predictable*. Otherwise, it is classified as *unpredictable*. Here are examples of predicting models for different types of BEs: (1) Spark-based *big data* applications: These applications can adopt the slowdown model in [77] to estimate the job completion time after reclamation. The model takes inputs such as job completion statistics (c), elapsed time (t), and the proportion of resource (p) under preemption. The expected completion

time can be calculated using $t_{pmtn} = ((t/c) \times (1-c)/p) + t$. The job completion statistics c can be obtained through the HTTP API exposed by Spark. (2) Deep learning training: These workloads can employ the white box model proposed in [49, 71] to predict the completion time under different resource configurations. The model takes inputs such as the remaining number of training steps (s), elapsed time (t), and step processing speed (q). The expected completion time can be calculated using $t_{pmtn} = (s/q) + t$. The remaining number of steps can be updated according to the real-time loss value of the training job, while s and q can be estimated through model fitting. Other predicting models, such as those proposed in [91, 104], can be used for estimating the completion time of MapReduce and GPU workloads. It is important to note that `Rhythm` does not design its own predicting models but instead utilizes existing models if available for accurate estimation of running times.

For *unpredictable* BEs that cannot be accurately predicted, we prioritize them for reclamation based on the amount of *useless computation* they generate. *useless computation* (U) refers to the repeated computation caused by reclamation. If a task becomes slower after resource reclamation but does not require recomputation, $U = 0$. In cases where multiple BEs have $U = 0$, we break the tie using their *attained service*, i.e., *BE elapsed time* $\times$ *resource*. If a task fails, a portion of its computation becomes useless, resulting in $U > 0$. *Useless computation* is related to the fault-tolerant mechanism of BE. The existing fault-tolerant mechanisms can be broadly categorized into two types:

- *Temporal redundancy-based mechanisms* involve rescheduling failed tasks on a backup server to resume execution, which can result in delayed task execution [105]. To mitigate the repeated computation caused by rescheduling, checkpointing is often employed. Checkpointing allows a failed task to be restarted from the latest saved checkpoint instead of starting from the beginning [28]. Thus, we derive the *useless computation* as follows,

$$U_{temp} = t_{ckpt} r_{ognl} \tag{7}$$

  where t_{ckpt} represents the computation time since the latest checkpoint time.
- *Space redundancy-based mechanisms* involve creating multiple replicas of the same task to improve efficiency [13, 14]. These replicas run concurrently, and the task is considered successful if at least one of the replicas completes successfully. In this case, if a task has more than one replica, reclamation does not generate any repeated computation, resulting in $U_{space} = 0$. However, if all replicas of a task fail, the task needs to be rescheduled, and the useless computation is calculated using the same formula as in Formula 7, i.e., $U_{space} = U_{temp}$.

`Rhythm` adopts a priority-based reclamation algorithm for BE tasks to reduce the system preemption loss. For *unpredictable* BE tasks, we mainly consider their *useless computation* (U) after resource reclamation. For *predictable* BE tasks, the priority is defined based on its estimated preemption loss (L). *Predictable* and *unpredictable* BEs are categorized into two separate reclamation queues, which poses a challenge when selecting the "best" BE for reclamation. To address this issue, we utilize the Borda count voting method [8] to merge the *Predictable* and *unpredictable* queues into a single BE reclamation queue. The Borda count is a voting mechanism that considers multiple influencing factors. Each voter ranks the candidates based on their preferences, and the final ranking is determined by integrating the rankings from different voters. In our case, each BE receives points based on its ranking in each sequence, and the BE with the lowest sum of points across all sequences is preempted first.

Figure 12 shows our design of how to merge the predictable and unpredictable BE queues. We use the Borda counting method [9], a voting system used for winner elections in which each voter ranks the list of candidates in order of preference. A job i receives a number of points (denoted by n_{points}) from a sequence. It is derived as follows,

$$n^i_{points} = J - r_i \tag{8}$$

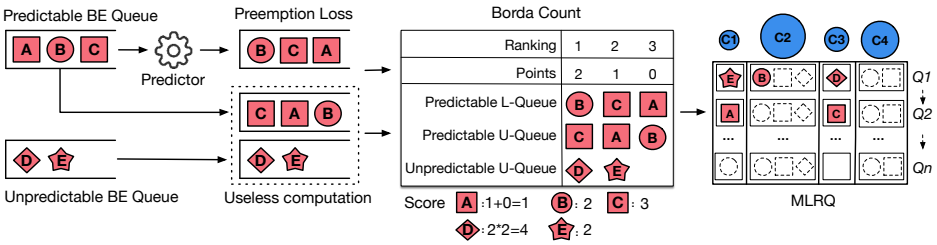


Fig. 12. Queue merging through the Barda count voting method.

where r_i denotes the ranking of i and J denotes the number of jobs in the sequence. The sum of i's points from different sequences decides the winner. That is, the task with the highest n_{score} is selected first for reclamation.

$$n^i_{score} = \sum_{L-sequence,U-sequence} n^i_{points} \tag{9}$$

Since it is not possible to derive the preemption loss (L) for unpredictable BEs (i.e., unpredictable BEs will not appear in the L-sequence), it would be unfair to directly apply the Borda counting method. Therefore, we also maintain a useless computation sequence for predictable tasks. Similar to unpredictable tasks, the BEs in the sequence are sorted in ascending order of useless computation (U). The predictable useless computation sequence (U-sequence) is then combined with the preemption loss sequence (L-sequence) to calculate scores for all predictable tasks using Formula 9. Since unpredictable BEs only appear in one sequence while predictable BEs appear in two sequences, we double the score obtained by unpredictable BEs for a fair comparison, i.e., $n_{score} = 2{\times}n_{points}$. Then, the rankings of predictable and unpredictable BEs are combined and sorted in descending order of their scores.

Given the global BE reclamation queue, the operations *StopBE* or *CutBE* always select the top entry in the queue to execute under SLO violation. If the first entry is not available on the local server, the BEs are sequentially substituted until a match is found. To expedite the SLO recovery process, we further organize the BE reclamation queue into a Multi-Level Reclamation Queue (MLRQ), where BEs are categorized into multiple levels, and *StopBE* (or *CutBE*) always chooses the top level for reclamation. In Figure 12, each server maintains a local MLRQ and has a long subqueue in each MLRQ level if its local *Servpod* contributes more to tail latency. When executing *StopBE* (or *CutBE*), resources are reclaimed simultaneously from all BE workloads in the top MLRQ level. This means that more resources are reclaimed from the BEs that are co-located with the *Servpod* making larger contributions. The number of BEs in an MLRQ level (q_{MLRQ}) is determined based on the *Servpod* contributions. That is, we have,

$$q^i_{MLRQ} = n_{BE} \times \frac{C_i}{\sum_{i=1}^{m} C_i} \tag{10}$$

where n_{BE} is the number of BEs in the system and C_i is the contribution of *Servpod* i. In the example of Figure 12, the four servers keep 1, 3, 1, 2 BEs in each MLRQ level respectively, corresponding to the *Servpod* contributions.

4 IMPLEMENTATION

We have developed a prototype of `Rhythm` using approximately 6.6K lines of code (KLOC) of C, Java, Python, and Linux Shell. It runs on the Linux operating system, and supports the automatic profiling of *Servpods* using a load generator for generating a broad spectrum of access loads and a *SystemTap* [40]-based system events analysis tool. It cooperates with Linux container technology to manage the resource allocation for LC services and BE jobs. It also provides APIs on latency and system status monitoring, contribution analysis, parameters exchanging, BE deploying and resource allocation updating in each *Servpod* agent. The interactions with the operating system are mainly implemented through JDK runtime library and Linux shell interface.

Table 1. LC workloads and BE jobs.

LC Workloads						BE Jobs		
Applications	Domain	*Servpods*	MaxLoad	SLO	Containers	Workload	Domain	-intensive
Redis [74]	Key-Value database	Master, Slave	86K QPS	1.15ms	18	Stream-llc [23]	LLC-benchmark in iBench	LLC
Elasticsearch [27]	Index engine	Index, Kibana	750 QPS	200ms	12	Stream-dram [23]	DRAM-benchmark in iBench	DRAM
E-commerce[62]	TPC-W website	Haproxy, Tomcat, Amoeba, MySQL	1300 QPS	250 ms	16	CPU-stress [57]	CPU stress testing tool	CPU
						LSTM	Time series prediction service	mixed
Solr [81]	Web search	Apache+Solr, Zookeeper	400 QPS	350ms	15	ImageClassify	Image classification on CycleGAN[107]	mixed
Elgg[29]	Social network	Nginx+PHP-FPM, Memcached, MySQL	200 QPS	320ms	8	Spark-Bench [82]	Wordcount, KMeans, LogisticRegression, LinearRegression	mixed
SNMS[32]	Microservice	UserService,Frontend, MediaService	1500 QPS	380ms	30	TensorFlow-Bench[4]	Resnet50, Lenet, Alexnet	mixed
Hotel Reservation[22]	Microservice	Fontend, Reserver, Database	1400 QPS	300ms	19	SciMark[2]	Scientific computing	mixed

Isolation: For mitigating the performance interference between the LC service and BE jobs, we utilize resource isolation mechanisms as follows: (1) *Core/thread isolation*: `Rhythm` uses the `cpuset cgroups` of the Linux operating system to bind LC and BE jobs on different physical cores to reduce the interference caused by thread contention. (2) *LLC/Memory Bandwidth isolation*: `Rhythm` uses Intel CAT (cache allocation technology) and MBA (Memory Bandwidth Allocation) to partition the LLC and Memory Bandwidth resources. By dividing these resources, `Rhythm` dedicates a portion to the LC service and reserves the remaining portion for BE jobs. (3) *Network isolation*: `Rhythm` uses the `qdisc` in the Linux operating system to control bandwidth allocation for the traffic flows of both LC and BE jobs. (4) *Power isolation*: `Rhythm` uses the running average power limit (RAPL) to monitor the CPU power consumption in each CPU socket and DVFS to redistribute power among different cores.

Interact with scheduler: When an LC arrives, the scheduler first decides the schedule for its components. Components that are co-located form *Servpods*, which are profiled using a load generator. This profiling helps derive *loadlimit* and *slacklimit*. Then, the thresholding mechanism decides whether it is appropriate to deploy or reclaim BEs. If deployment is allowed, the scheduler checks the waiting queue of BE jobs and dispatches them to physical machines that have sufficient available resources. When resources need to be reclaimed from BEs, the top-controller selects BEs in the top MLRQ level for reclamation. Once the decision about which BEs to reclaim is made, the subcontrollers at the local physical machines take charge of determining the specific resource allocation for those BEs. They ensure that the reclaimed resources are appropriately allocated to other tasks or components as needed.

System integration: `Rhythm` can be integrated into container management framework like Kubernetes [48] or serverless cloud system. To integrate Rhythm into a container management framework like Kubernetes, several steps need to be taken: (1) Extend the *cAdvisor* in kubelet of Kubernetes to support the measurement of utilization of memory bandwidth, frequency&power and network traffic; (2) Enhance the configuration module of Kubernetes to support the runtime control of the resource allocation as `Rhythm`'s controller agent; (3) Associate the scheduler of Kubernetes with the top-controller agent for providing feedback to scheduling algorithms.

5 EVALUATION

5.1 Methodology

Workloads: Table 1 summarizes the LC and BE workloads we used to evaluate the efficiency of `Rhythm`. In particular, the maximum allowable request load (i.e., *max load*) is measured when the arrival speed approaches the maximum processing speed. Their SLOs are not defined arbitrarily, but following the principle: *each LC service runs at its maximum allowable request load without interference over 30 minutes, and we record the* 99*th percentile latency per second and set the worst one as the SLO.* We also deploy multiple BE workloads on our testbed, including *synthetic microbenchmarks that put strong pressure on a specific resource (i.e., LLC, CPU and DRAM),*

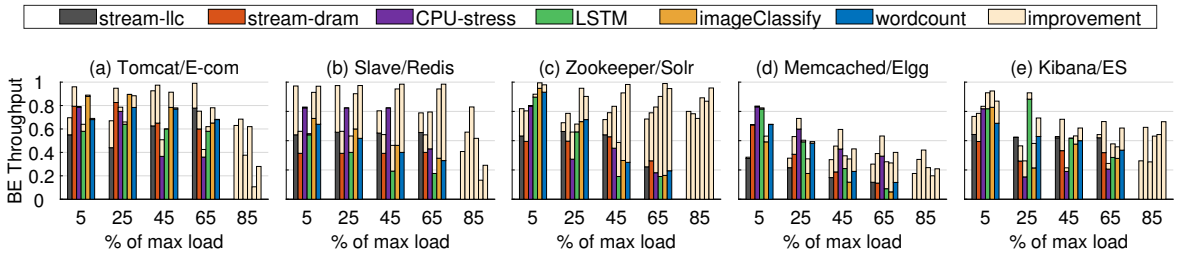


Fig. 13. The BE throughput at *Servpods* under different loads.

big data, *AI*, and *scientific computing* benchmarks. In particular, we use the synthetic microbenchmarks, LSTM and imageClassify for evaluating the co-location efficiency. We further extend BEs with more big data, AI and scientific computing workloads for evaluating the efficiency of BE-distinguishable reclamation.

Metrics: We co-locate an LC service with the BE jobs, and measure the system's CPU and memory utilization, and power consumption. We also use the metric of EMU to measure the overall system throughput. In particular, $EMU = LC\ Throughput + BE\ Throughput$, where *LC Throughput* denotes the request load for LC service normalized to its maximum allowable load, and *BE Throughput* denotes the average number of BE jobs successfully finished per hour normalized to when it runs alone on a machine. Note that EMU may exceed 100% due to the resource sharing between the LC service and BE jobs.

Testbed: LC service and BE jobs are deployed on a cluster with seven physical machines, four of which are configured with 40 cores of a quad-socket Intel Xeon E7-4820 v4 @ 2.0 GHz and 64 GB of DRAM per socket and the other three are configured with 40 cores of a dual-socket Intel(R) Xeon(R) Gold 6230 @ 2.0 GHz and 64 GB of DRAM per socket. The operating system is Ubuntu 14.04 with kernel version 4.4.0-31. We utilize containers for deploying multiple instances for LC workloads. The detailed configurations of workloads are shown in Table 1. Although each container is configured with a specific capacity initially, its unused resources can be allocated to BE jobs through the container resource control interface.

Overhead: After deploying `Rhythm` in the system, we measure its overhead and find that the request tracer only consumes approximately 6% of the CPU and 3 MB of memory, and each controller runs every 2 seconds only consumes 3.6% of the CPU and less than 50 MB of memory. `Rhythm` collects request sojourn time in each *Servpod* by solo-run LC service only once, the off-line profiling takes negligible overhead. For the collection of BE information and the maintenance of the BE reclamation queue, `Rhythm` only needs 2 logical cores for supporting both predicting and reclamation. Among them, big data workloads such as Spark are profiled by invoking the Restful interface; for sciMark workloads, `Rhythm` only needs to record the timestamp of the start and the *StopBE* operation; for AI jobs, `Rhythm` needs to scan its output log to check its progress.

5.2 Component-distinguishable Co-location

We firstly evaluate the component-distinguishable co-location ability of `Rhythm`, i.e. `Rhythm-CDC`. That is, whenever SLO violation occurs, we just reclaim resources from all the co-located BE workloads without distinguishing their priorities. All the experimental results are compared against *Heracles* [51], which is a feedback-based method, but does not distinguish between Servpods: (1) It disables BE jobs at all machines whenever the load exceeds 85%. (2) It disallows the growth of BE jobs whenever the slack between the current tail latency and SLO target is less than 10%. We observe that the measured SLO on our testbed is larger than those in *Heracles* because of the different software configurations and hardware environments we used. To make a fair comparison, our implementation of *Heracles* [51] also conducts its control using the same SLO as in Table 1.

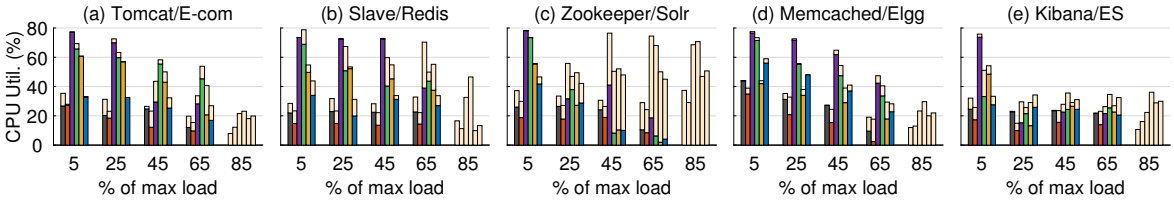


Fig. 14. The CPU utilization at *Servpods* under different loads.

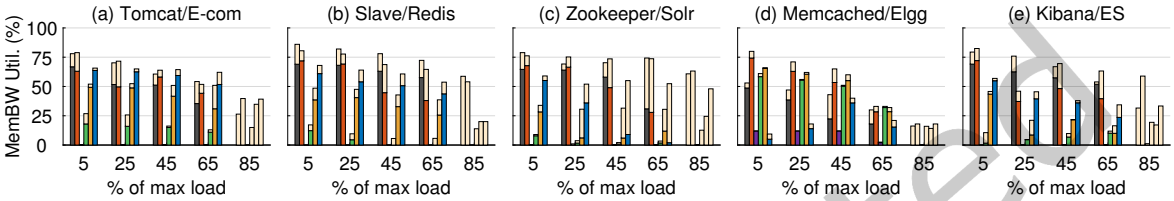


Fig. 15. The Memory bandwidth utilization at *Servpods* under different loads.

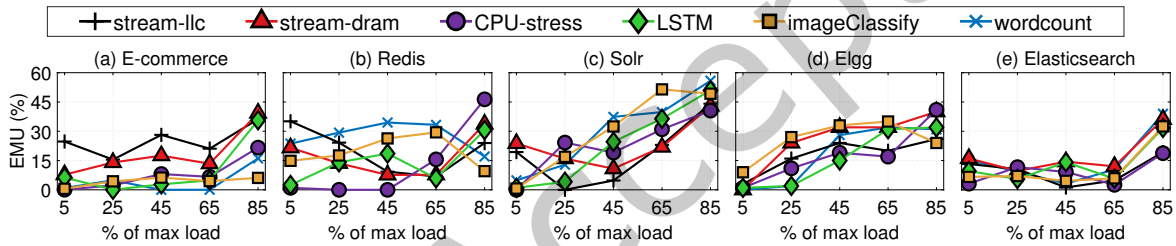


Fig. 16. EMU improvements $((EMU_{Rhythm-CDC} - EMU_{Heracles})/EMU_{Heracles})$ under different loads.

5.2.1 Servpod *Analysis under Constant Load.* Figures 13-15 show the BE throughput, CPU utilization and memory bandwidth utilization at the *Servpods* of Tomcat/E-commerce, Slave/Redis, Zookeeper/Solr, Memcached/Elgg, and Kibana/Elasticsearch. We see that the `Rhythm-CDC` is particularly effective when the load exceeds 65% of the *max load*. While *Heracles* can launch BE jobs at a lower load, no co-location exists when the load is set as 85% of the *max load* because *Heracles* does not allow co-location when the load > 0.85. Hence, in this case, the BE throughput, CPU utilization and memory bandwidth utilization by BE jobs are all zero. In contrast, `Rhythm-CDC` allows deploying BE jobs at the load > 0.85 since the *loadlimits* of Tomcat, Slave, Zookeeper, Memcached and Kibana are 0.87, 0.91, 0.93, 0.87 and 0.9, respectively.

5.2.2 *Overall Performance under Constant load.* In Figure 13, we see that `Rhythm-CDC` increases BE throughput by an average of 0.196, 0.296, 0.41, 0.185, and 0.194 compared with that of *Heracles* for the five LC Servpods. In particular, Zookeeper is deployed with the most BE jobs due to its large $loadlimit = 0.93$ and small $slacklimit = 0.035$. As we increase the load of the LC service, the BE throughput is reduced due to the operation control by the *controller*. For the CPU utilization in Figure 14, when we co-schedule *Servpods* with CPU-stress, the CPU utilization at all machines could approach 80% at the 5% load due to the CPU-intensive nature of CPU-stress. LSTM can also utilize CPU resources at more than 70%, because the training phase of LSTM heavily consumes CPU resources. When they are co-located with other BE jobs, although they cannot achieve the same CPU utilization, `Rhythm-CDC` still improves CPU utilization by an average of 7.98%, 11.44%, 27.59%, 8.4%, and 10.44%. For the memory bandwidth

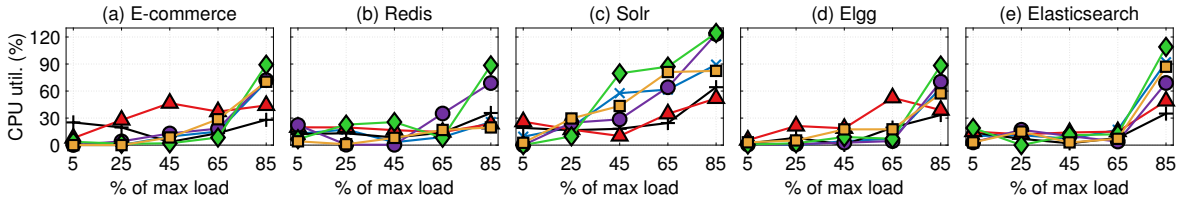


Fig. 17. CPU utilization improvements $((CPU_{Rhythm-CDC} - CPU_{Heracles})/CPU_{Heracles})$ under different loads.

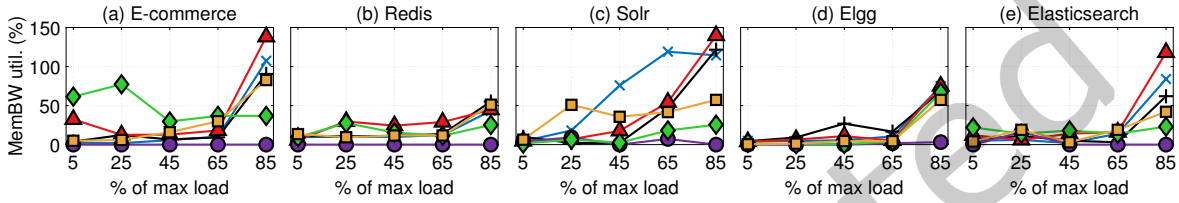


Fig. 18. Memory bandwidth utilization improvements $((MeB_{Rhythm-CDC} - MeB_{Heracles})/MeB_{Heracles})$ under different loads.

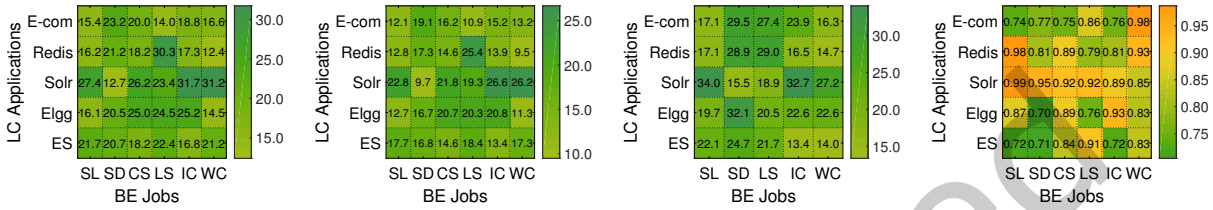
utilization in Figure 15, we see that Rhythm-CDC can drive the utilization up to 82% when co-scheduling stream-llc and stream-dram with LC *Servpods*. CPU-stress does not require much memory bandwidth, so the utilization is quite low. Generally, Rhythm-CDC can improve memory bandwidth utilization by an average of 11.4%, 13.1%, 18.9%, 10.44%, and 10.57% compared with that of *Heracles*.

Rhythm-CDC improves the throughput and resource utilization not only when LC is scheduled together with the extreme BEs, such as stream-llc, stream-dram, and CPU-stress, but also when BE jobs are normal ones (LSTM, imageClassify and wordcount). Specifically, the average improvements on BE throughput by extreme BEs and normal ones are 17.56% and 21.7%, respectively. Improvements in CPU utilization are 25.54% and 29.53%, and improvements in memory bandwidth are 21.03% and 39.13%, respectively.

We next show the overall improvements in EMU and resource utilization by Rhythm-CDC. Figures 16-18 show that it generates a much higher EMU and resource utilization than *Heracles* in all interference groups. Since both Rhythm-CDC and *Heracles* can deploy BE jobs at low load, the improvements generally increase over the load, indicating that Rhythm-CDC is more effective when the load towards the LC service is intensive.

In Figure 16, we see that Rhythm-CDC generates 11.6%, 18.4%, 24.6%, 14%, and 12.7% more EMU on average than *Heracles* in E-commerce, Redis, Solr, Elgg, and Elasticsearch, respectively. In particular, when Solr is co-located with imageClassify/TensorFlowBench and wordcount/SparkBench, improvements of up to 57% can be achieved because of the significant improvements in Zookeeper. Figure 17 shows the CPU utilization improvements for the five LC services. Rhythm-CDC can improve the CPU utilization by 22.2%, 19.1%, 35.3%, 20.6%, and 23% on average compared with that of *Heracles*. co-locating LSTM and CPU-stress with the LC service performs much better utilization than others because they both require CPU resources heavily, and an improvement of up to 112% can be achieved in the case of Elasticsearch. Figure 18 shows that Rhythm-CDC can improve memory bandwidth utilization by 28.1%, 16.8%, 33.4%, 28.9%, and 19.5% on average compared with that of *Heracles*. co-locating stream-dram (or wordcount/SparkBench) with the LC service shows much higher improvements than the other BE jobs since they both consume considerable memory bandwidth. The improvement even reaches 120% when co-locating stream-dram with Elasticsearch.

5.2.3 Overall Performance under Production Load. We also evaluate Rhythm-CDC using a production request load from ClarkNet [19] to capture its improvement on resource utilization. The request load illustrates clear periodicity (see the top in Figure 17), and the period length is 24 hours. In our experiment, we scale down five days of the ClarkNet trace to six hours of workload for shortening the experimental period, and the traffic load and fluctuating pattern are kept the same. Then, we collect the resource efficiency in the period.



(a) EMU (%) (b) CPU Utilization (%) (c) MemBW Utilizaiton (%) (d) 99*th* latency/SLO

Fig. 19. The average performance improvements by Rhythm-CDC on EMU (a), CPU utilization (b), and membw utilizaiton (c) under production load. (d) represents the 99*th* percentile latency normalized to the latency stated in SLO. (E-com: E-commerce, ES: Elasticsearch, SL: Stream-llc, SD: Stream-dram, CS: CPU-stress, LS: LSTM, IC: ImageClassify/TensorFlowBench, WC: Wordcount/SparkBench.)

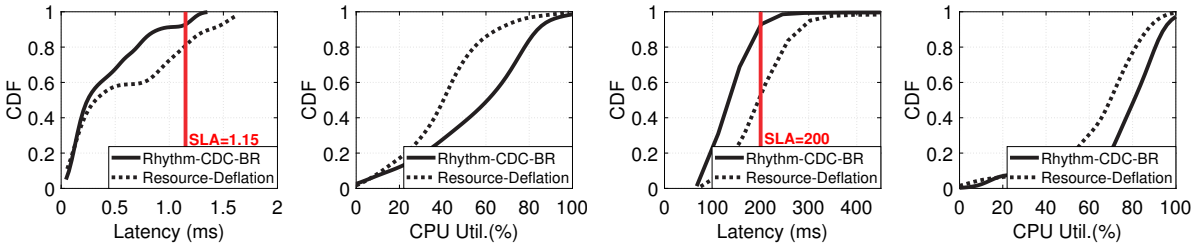
Figures 19a-19c show the average performance improvements compared to those of *Heracles* on EMU, CPU utilization and memory bandwidth utilization under the production load, respectively. We see that Rhythm-CDC can improve EMU by at least 12.4% in the Redis-Wordcount group and at most by 31.7% in the Solr-ImageClassify group. For CPU utilization, Rhythm-CDC can achieve an improvement of 26.2% in the Solr-Wordcount group. For memory bandwidth utilization, Rhythm-CDC can achieve an improvement of 34% in the Solr-Wordcount group. Generally, while Rhythm-CDC can improve the performance in all interference groups, Solr benefits the most on EMU, CPU utilization and memory bandwidth utilization among all of the five LC services.

Figure 19d presents the worst 99*th* percentile latency normalized to the SLO latency of Rhythm-CDC in production request loads. The actual 99*th* latency increases with the request load due to the increasing pressure in server end. Meanwhile, interference from the co-located BE jobs will also cause performance degradation of LC service. But we see that Rhythm-CDC can strictly guarantee the SLO in all cases (the worst case is 0.99× SLO). The result shows the effectiveness of Servpod-level control in Rhythm-CDC, which can improve throughput without hurting the SLO.

5.3 BE-distinguishable Reclamation

We further evaluate the BE-distinguishable reclamation ability of Rhythm, i.e., Rhythm-CDC-BR, which supports both component-distinguishable co-location and BE-distinguishable reclamation. We choose the *resnet50, lenet, and alexnet* from TensorFlow-Bench [4] as *AI* workloads, the *kMeans, logisticRegression, and linearRegression* from Spark-Bench [82] as *big data* workloads, and *sciMark* [2] as *scientific computing* workloads. They are submitted to the system following a Poisson process, meaning they are randomly selected for submission whenever there is a new request. When the SLO of an LC service is violated, Rhythm-CDC-BR uses a predictor (if available) to obtain the BE preemption loss. As introduced in Section 3.6, our predictor currently supports the prediction models proposed by Optimus [71] and *Resource Deflation* [77], which can estimate the completion time of Spark and AI workloads without relying on offline characterization.

We firstly compare Rhythm-CDC-BR with *Resource Deflation* [77], a state-of-the-art approach that employs a dynamic, multi-level cascading reclamation technique. We evaluate them using the two LC services of Redis



(a) Redis (b) Elastic-Search

Fig. 20. The performance comparison on latency, CPU utilization under the resource management of Rhythm-CDC-BR and Resource-Deflation methods.

and Elastic-search. Figure 20 shows that although *Resource Deflation* can achieve a higher CPU utilization than Rhythm-CDC-BR, it can only guarantee the SLO within 80% and 58.4% of the time. Meanwhile, Rhythm-CDC-BR can achieve 92.1% and 94.5% of the time SLO guarantee. Clearly, as the intensity of co-location increases, the risk of SLO violations also increases. This situation can be addressed by adjusting the *loadlimit* and *slacklimit* to enhance the guarantee of SLO. In a real production environment, the determination of SLO is based on business requirements. For example, LC services like search engines typically set the SLO as the 99*th* percentile latency not exceeding 100 milliseconds. Administrators can adjust the *loadlimit* and *slacklimit* based on the actual business SLO to achieve full SLO guarantee (see § 5.6). *Resource Deflation* [77] nearly impossible to provide a guarantee for SLOs due to its overly conservative approach. It always aims to reclaim as few resources as possible, which can lead to SLO violations. The strategy iteratively checks resources that can be reclaimed from higher layers, such as the application and the guest OS, before considering lower layers like the hypervisor. The lower layer is only checked if the reclamation at the previous layer is insufficient to recover the latency of the LC service. However, since it takes time for new configurations to take effect, this conservative strategy often leads to SLO violations. Considering the significant degradation in user experience caused by *Resource Deflation*, we exclude it from the following experiments.

5.3.1 Servpod *Analysis under Constant Load.* Figures 21-23 demonstrate the effectiveness of Rhythm's reclamation technique in terms of Servpod-level improvements in BE throughput, CPU utilization, and memory bandwidth utilization. Rhythm-CDC-BR, compared to Rhythm-CDC, achieves further enhancements in BE throughput with average increases of 0.107, 0.153, 0.162, 0.112, and 0.091 on the five LC *Servpods*, respectively. While BE throughput tends to decrease as the access load on the LC increases, Rhythm-CDC-BR exhibits increased improvements because the BE reclamation strategy becomes more active under high load conditions. Notably, at 85% of the maximum load, Rhythm-CDC-BR achieves a 0.193 increase in BE throughput for the co-located Zookeeper service. This result is attributed to Zookeeper's low contribution rate of 0.075, which leads to the reclamation of only one BE at a time during *StopBE* or *CutBE* operations. In contrast, Kibana/ES maintains three BEs in each MLRQ level due to its high contribution rate of 0.32. Alongside the improvements in BE throughput, Rhythm-CDC-BR also demonstrates average increases of 9%, 11.3%, 10.7%, 12.7%, and 8.7% in CPU utilization, as well as average increases of 11.4%, 12.6%, 10.3%, 8.8%, and 9.3% in system memory bandwidth utilization compared to Rhythm-CDC.

5.3.2 Overall Performance under Constant load. Figures 24-26 illustrate the overall performance improvements achieved by BE-distinguishable reclamation in Rhythm-CDC-BR. The results demonstrate that Rhythm-CDC-BR effectively enhances system throughput and resource utilization. When the load is low, the LC service experiences fewer SLO violations, and the improvements brought by Rhythm-CDC-BR are less apparent. However, as the load

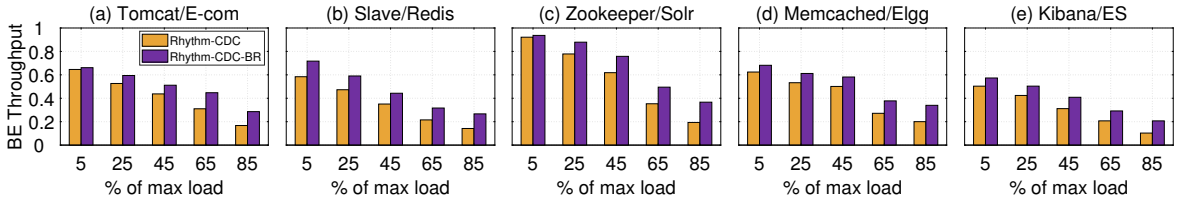


Fig. 21. The BE throughput at *Servpods* under different loads.

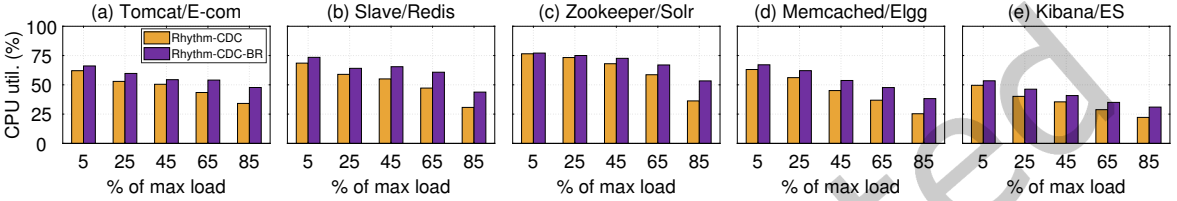


Fig. 22. The CPU utilization at *Servpods* under different loads.

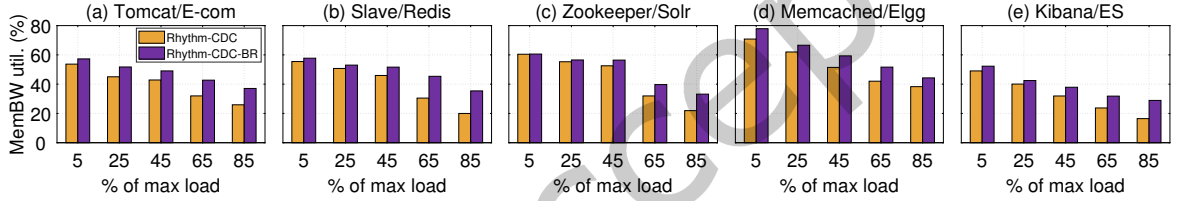


Fig. 23. The memory bandwidth utilization at *Servpods* under different loads.

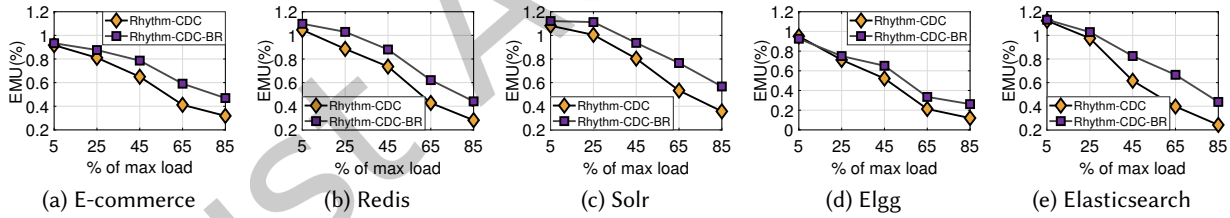
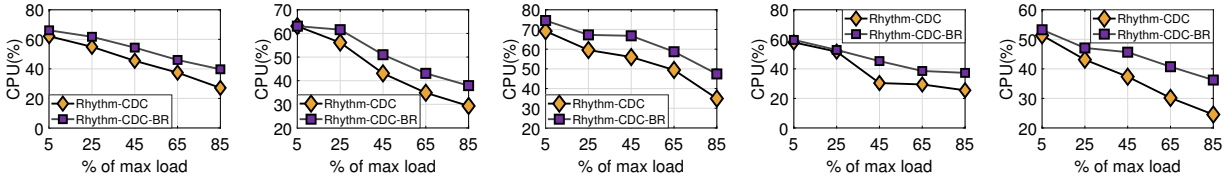


Fig. 24. The EMU improvements under different loads.

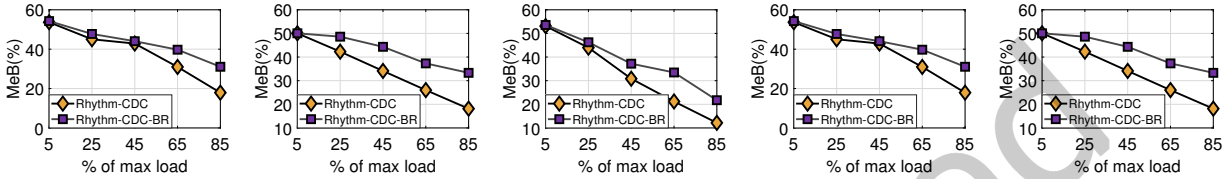
increases, more *CutBE* and *StopBE* operations are executed. At this point, the BE-reclamation strategy significantly reduces preemption losses, thereby enhancing resource utilization and throughput.

For instance, at 85% load for the Zookeeper service, `Rhythm-CDC` triggers a total of 11 *CutBE* and 8 *StopBE* operations to protect latency SLOs for the Solr service. Enabling the BE-distinguishable reclamation technique leads to smoother preemption of BE tasks. Although this may result in more *CutBE*/*StopBE* operations, the number of stopped BE tasks decreases from 17 to 9 (as a *StopBE* operation can terminate multiple tasks), reducing throughput loss by approximately 50%. Notably, `Rhythm-CDC-BR` achieves further improvements of 11.4%, 13.7%, 17.4%, 10.7%, and 16.1% in EMU for E-commerce, Redis, Solr, Elgg, and Elasticsearch, respectively. Regarding resource utilization, Figure 25 demonstrates that `Rhythm-CDC-BR` increases average CPU utilization by an additional 9.4%, 11.2%, 12.7%, 10.8%, and 13.8%. Figure 26 shows that `Rhythm-CDC-BR` further enhances memory bandwidth utilization by 9.1%, 14.6%, 10.1%, 9.8%, and 11.4% for the respective services.



(a) E-commerce (b) Redis (c) Solr (d) Elgg (e) Elasticsearch

Fig. 25. The CPU utilization improvements under different loads.



(a) E-commerce (b) Redis (c) Solr (d) Elgg (e) Elasticsearch

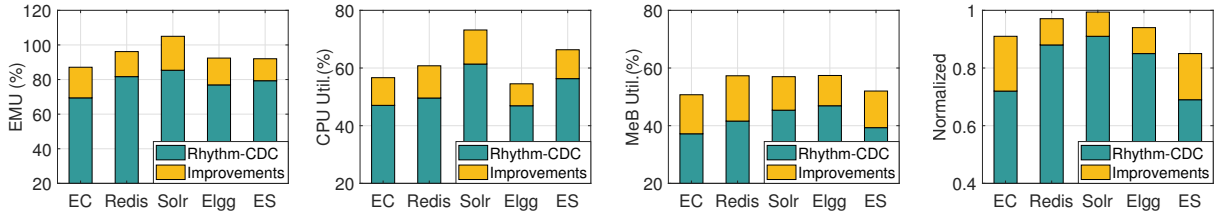
Fig. 26. The memory bandwidth utilization improvements under different loads.

5.3.3 Overall Performance under Production Load. Figure 27 demonstrates the improvements in resource utilization and SLO guarantee achieved by `Rhythm-CDC-BR` using the production load of Clarknet [19]. It is observed that `Rhythm-CDC` provides a better SLO guarantee, as shown in Figure 27d, due to its non-discriminatory reclamation strategy for BEs. However, this also results in a lower resource utilization rate for `Rhythm-CDC` over an extended period. Under the production load, `Rhythm-CDC-BR` significantly increases EMU, CPU utilization, and memory bandwidth utilization by 17.7%, 14.5%, and 19.6%, respectively (Figure 27a-27c). In summary, compared to *Heracles*, `Rhythm-CDC-BR`, which incorporates both component-distinguishable co-location and BE-distinguishable reclamation, improves the average system throughput by 47.3%, CPU utilization by 38.6%, and memory bandwidth utilization by 45.4%.

5.4 Running with Microservice

`Rhythm` demonstrates its effectiveness in managing processes, containers, or microservices. In this section, we evaluate its efficiency using two representative microservice benchmarks from DeathStarBench [32]: SNMS (social network in microservice) and HR (hotel reservation). SNMS consists of 30 unique microservices that communicate through RPC (Remote Procedure Call). These microservices are divided into three *Servpods*: *media service* (13 microservices for media data processing), *frontend* (3 microservices including nginx-thrift, media-frontend, and jaeger), and *user service* (14 microservices for user operations). Similarly, HR consists of 19 unique microservices, grouped into three *Servpods*: *frontend* (3 microservices), *reservation* (7 microservices related to the reservation process), and *data service* (9 microservices for data storage and processing). Each *Servpod* is allocated 20 CPU cores and 64 GB of memory, and they are deployed in a distributed manner. For the evaluation of these microservice benchmarks, `Rhythm` does not require its own request tracer. Instead, the benchmarks support a built-in jaeger [41], which is a distributed tracing system similar to Dapper [79]. Jaeger can record the sojourn time of each request at each microservice, providing the necessary data for analysis and evaluation.

Figure 28 shows the overall performance evaluation results. Since the contributions of the three *Servpods* are 0.295, 0.14, and 0.565, respectively, and their *slackLimits* are 0.189, 0.054, and 0.381, respectively, the improvements mainly benefit from the *media service* and *frontend Servpods*. Compared with *Heracles*, `Rhythm` achieves an average improvement of 14.3%, 30.2%, and 45.8% in the EMU, CPU utilization, and memory bandwidth utilization, respectively. In particular, `Rhythm` achieves an EMU improvement of 23.27% in the wordcount group because wordcount performs many computations and IO operations, which affects the tail latency significantly. The



(a) EMU (%) (b) CPU Utilization (%) (c) MemBW Utilizaiton (%) (d) 99%ile/SLO

Fig. 27. The average performance improvements by Rhythm-CDC-BR on EMU (a), CPU utilization (b), and MemBw utilizaiton (c) under production load. (d) represents the 99*th* percentile latency normalized to the latency stated in SLO. (EC: E-commerce, ES: Elasticsearch)

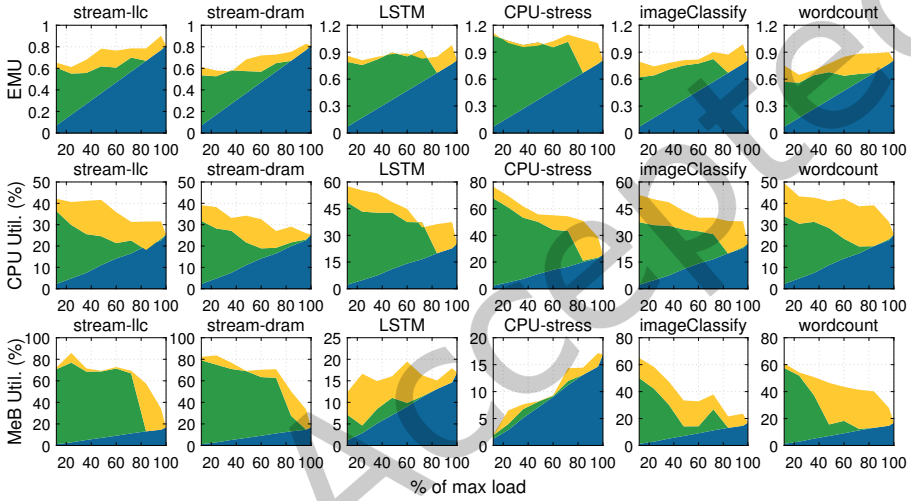
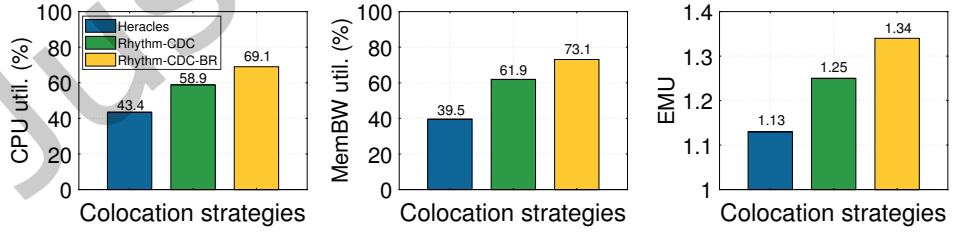


Fig. 28. Performance comparison when running with SNMS microservice. Improvements are color-coded as follows: ■ represents the EMU or resource utilization of LC itself; ■ represents the improvements achieved by *Heracles*; ■ represents the further improvements achieved by Rhythm.



(a) CPU utilization (b) Memory bandwidth (c) EMU

Fig. 29. Performance evaluation when running with hotel reservation microservice.

sciMark group also shows the best improvements in CPU utilization, but the least in memory bandwidth utilization for the same reason as in previous experiments.

Figure 29 presents a comprehensive performance comparison among *Heracles*, Rhythm-CDC, and Rhythm-CDC-BR when co-locating HR (hotel reservation) with BE tasks. The results show that Rhythm-CDC launches more BEs compared to *Heracles*, resulting in a significant improvement in CPU utilization and memory bandwidth utilization by 35.7% and 56.7% respectively. Furthermore, Rhythm-CDC-BR enhances system efficiency even further, with a notable increase in CPU utilization, memory bandwidth utilization, and EMU by approximately 17.3%, 18.4%, and 7.2% respectively. These findings demonstrate the effectiveness of the component-distinguishable co-location mechanism and the BE-distinguishable reclamation technique in scenarios involving multiple microservices with varying workload behaviors and resource dependencies in each component.

5.5 Example of Running Process

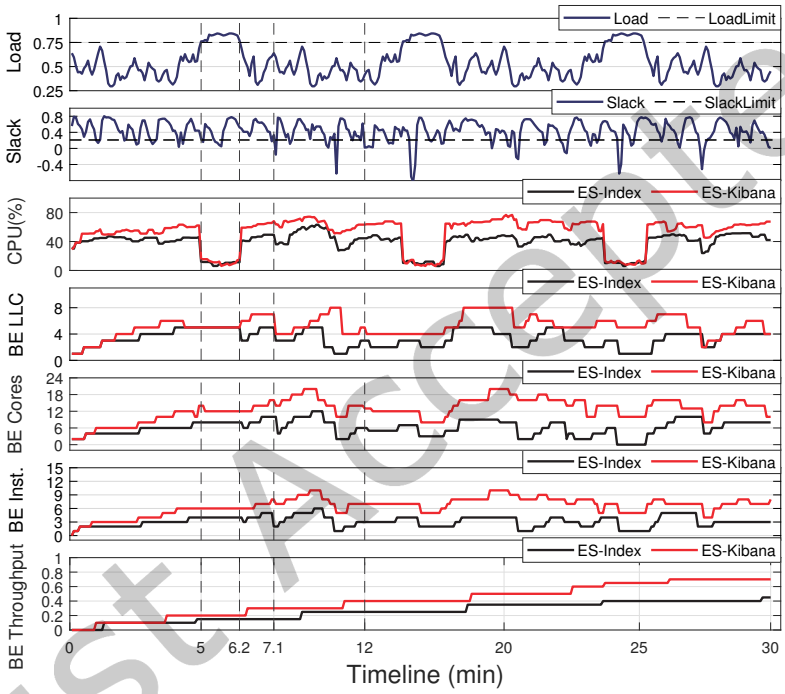


Fig. 30. The timeline of Rhythm's running process.

Figure 30 illustrates the timeline of Rhythm's operation on two *Servpods* (Index and Kibana) of Elasticsearch when they are co-located with 20 BEs under the production load. Initially, Rhythm allows the BE workload to grow as there is sufficient slack between the actual latency and the SLO target. This results in continuous increases in BE throughput, BE instances, BE cores, BE LLC, and CPU utilization. At time 5, Rhythm triggers the *SuspendBE* operation because the request load exceeds the *loadlimit*. As a result, even though the allocated resources for the BE jobs remain unchanged, CPU utilization rapidly decreases, and the BE throughput stops increasing. During the *SuspendBE* period, the memory occupied by the BE may still cause SLO violations, leading to the activation of the *StopBE* operation. At time 6.2, when the request load drops below the *loadlimit*, the BE jobs resume their growth until time 7.1. However, a sudden decrease in slack prompts Rhythm to initiate the *StopBE* operation. During this operation, Rhythm selects the BEs with fewer preemption losses for reclamation. Since the Index *Servpod* has a higher contribution rate than Kibana, three BEs co-located with Index are reclaimed. At time 12,

Table 2. SLO violations and BE kills when varing the loadlimit (slacklimit).

	Fixed Loadlimit=0.76			Fixed Slacklimit=0.347		
Level	Slacklimit	SLO Violation	BE kills	Loadlimit	SLO violation	BE kills
70%	0.243	22	7	0.532	0	0
80%	0.278	16	5	0.608	0	0
90%	0.312	13	3	0.684	0	0
100%	0.347	0	0	0.760	0	0
110%	0.382	0	0	0.836	12	5
120%	0.416	0	0	0.912	14	8
130%	0.451	0	0	-	-	-

the *CutBE* operation is triggered. Although the number of BE instances remains unchanged, their LLC and core allocations are reduced. This operation helps optimize resource allocation and management in the system.

5.6 Loadlimit and Slacklimit Analysis

We also evaluate the impact of *loadlimit* and *slacklimit* on the BE throughput. By fixing the *slacklimit* and *loadlimit* of HAProxy, Tomcat, Amoeba, but varying the ones of MySQL, Figure 31 shows how the BE throughput varies over the *loadlimit* and *slacklimit*. We see that the BE throughput peaks when the *loadlimit* is at the 90% level (i.e., 90% of the actual derived values). In the case of "fixing loadlimit, varying slacklimit", the BE throughput at the 80% and 90% levels are both higher than that at the 100% level. However, Table 2 shows that setting the *slacklimit* at 90% also causes 13 SLO violations and kills 3 BE jobs in the period. For *loadlimit*, the number of SLO violations and BE kills at the 90% level is the same as that at the 100% level, indicating that the 90% level is a better choice.

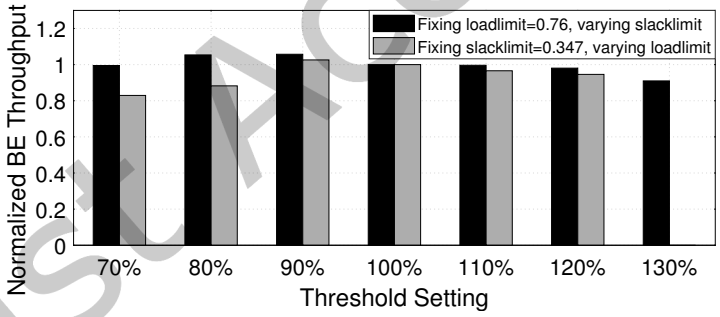


Fig. 31. Trade-off between loadlimit (slacklimit) and BE throughput.

6 RELATED WORK

Request Tracer: Tracking the service path of a request has been extensively studied in earlier work. They can be classified into two categories: *execution tracing* and *communication tracing*. *Execution tracing* [5, 11, 18, 30, 43, 75, 79, 86, 93, 108] records the low-level system events (e.g., system calls) or log messages generated during execution, and identifies the request path through pairing analysis. *Communication tracing* [6, 10, 15, 17] discovers the dependency between components by analyzing the network traffic. While the *communication tracing* cannot identify the intraServpod causality easily, we choose the *execution tracing*, which is further implemented through either the *intrusive method* [11, 30, 43, 79, 86, 108] or the *non-intrusive method* [5, 18, 40, 52, 75, 93]. As the

intrusive method causes high instrumentation cost, we simply use the easy-to-use systemTap [40] for deriving the mean sojourn time of each request at each Servpod.

Interference analysis: A significant body of work has studied the interference in cloud computing systems. The studies show that the performance of cloud services varies significantly due to multiple reasons [12, 38, 76], including hardware heterogeneity [67], virtualization [89], or the contention on various resources [44, 72, 80]. In particular, contention on cache [20, 34, 58, 83] and I/O [80] are two main sources for performance interference. These contentions are not only from the same core [97, 100] but also possibly from cross-cores [102, 103]. However, these works mainly focus on the evaluation of the overall performance of an application, e.g., the latency of a web application [72], a multimedia service [12], or the execution time of a bigdata analysis job [26]. They never study the performance variation of any one of *Servpods* under interference, while most applications consist of multiple *Servpods*.

Profiling-based QoS management: Given the precisely characterized interference features of cloud services, previous work can guarantee QoS through interference-aware QoS management. For example, Bubble-Up [60] and Bubble-Flux [96] predict the impact of interference from potential corunners through the instantaneous pressure generated by a dynamic bubble. DeepDive [65] infers performance loss due to interference by clustering low-level metrics. Dirigent [106] and Wrangler [95] supports to control QoS based on the prediction of execution time. Stay-away [73] throttles the batch jobs to avoid contention by predicting any progression towards a QoS violation at runtime. SMiTe [101] achieves precise interference prediction on real-system architectures. Quasar [25] and Paragon [24] use classification techniques to quickly estimate the impact of interference, and improve resource utilization while guaranteeing QoS. Pythia [94] predicts the combined contention of multiple applications using a simple linear regression model to improve utilization. Harvest VM [7] only considers the unallocated resources in cluster. It characterizes the time-varying features of unallocated resources and proposes to leverage them for more workloads. These approaches rely on accurate prediction models, which may be costly in practice.

Other related work like Ubik [46], CQoS [39], CPI2 [99], iAware [92] and [42] present how to guarantee the QoS with cache or memory bandwidth isolation mechanisms. For core isolation, PerfIso [37] co-locates batch jobs with production latency-sensitive services using CPU binding isolation to protect SLO from burst workloads. Retro [55] presents resource management framework to improve efficiency using these isolation mechanisms.

Feedback-based QoS management: Although the interference-aware QoS management works well in many scenarios, it's impossible to characterize the interference behaviors of all applications. Moreover, it is hard to achieve zero error on interference predictions. Hence, another approach for improving the resource utilization is using the feedback-based method, that is, response immediately after a possible SLO violation is detected. ICE [56] works in the application layer, and it improves the web server performance during interference by reconfiguring the balancer and middleware to reduce the load on the impacted server. In the system layer, Q-Clouds [63] uses online feedback to capture interference, and tunes resource allocations to mitigate performance interference effects. *Heracles* [51] enables the safe co-location of BE jobs and LC service through a conservative thresholding method. PARTIES [16] and CLITE [70] further increases the number of co-scheduled LC services per server to improve the throughput. CoPart [69] analyzes the characteristics of workloads and allocates the LLC and memory bandwidth for BE jobs to improve fairness. Twig [64] employs a deep reinforcement learning model for improving the energy-efficiency of co-located latency-critical services. *Resource deflation*[77] extends to dynamically shrink resource allocation in response to resource pressure, instead of being preempted outright. It helps to reduce the useless computation yielded by preempted BEs, but cannot guarantee SLO well. The feedback-based method may cause oscillations in the control loop especially when the tail latency is unstable. While the use of *loadlimit* can reduce such oscillations, we can also mitigate this problem by introducing buffer resources, like in PerfIso [37].

GrandSLAm [45] is another work considering the different characteristics of each service component. It enables consolidated execution of requests belonging to multiple jobs in a microservice-based computing framework. There are four differences between `Rhythm` and GrandSLAm: (1) GrandSLAm co-locates multiple LCs, while

Rhythm co-locates an LC and multiple BEs. (2) GrandSLAm is effective when multiple LCs share microservices, while Rhythm is also effective when LC has no shared microservices with BEs. (3) The execution time of each microservice is highly predictable in GrandSLAm given the batch size at each microservice, while it is difficult to predict the execution time in each *Servpod* due to the uncertain interference. (4) GrandSLAm uses the end-to-end latency in SLO, while Rhythm considers the tail latency, which is a statistical result over all latencies. Hence, GrandSLAm is orthogonal to Rhythm.

7 DISCUSSION

Application Scenarios: Rhythm can be adopted in the private Infrastructure-as-a-Service (IaaS) cloud or public serverless computing. Serverless computing is essentially a Platform-as-a-Service (PaaS) where the user's application is split into multiple stateless functions, and the user only needs to focus on their business code. In serverless, cloud providers offer runtime environments for tenant functions, expanding the system's scope of control compared to IaaS. Function execution times and even the end-to-end latency of a workflow can be easily measured at the cloud provider side, giving serverless computing an inherent advantage for integrating Rhythm.

The deployment of Rhythm in the serverless computing scenario is similar to the one in the microservices scenario. Rhythm's subcontrollers can be deployed within each server to manage local functions. In particular, functions located in the same physical server can also be grouped as a *Servpod*, while latency-insensitive functions can be treated as BE tasks. Note that Rhythm's control algorithm incurs low overhead and is scalable with the function scales in a serverless platform. This enables Rhythm to be highly adaptive in serverless computing.

Co-location in multi-LC scenarios: The current Rhythm can only support the co-location of a *Servpod* with multiple BEs. It is possible that multiple *Servpods* from different LCs are deployed in the same machine. In this case, Rhythm can be combined with existing methods such as CLITE[70] and Twig [64] to address the problem. Specifically, we can employ the methods in CLITE or Twig to address the service-level workload co-location challenge among different LCs. In addition, Rhythm's component-level workload collocation approach can be used to achieve finer-grained resource management and SLO guarantees within each LC.

8 CONCLUSIONS

In this paper, we present Rhythm, a system that manages resource allocation between LC service and BE jobs in the profiling-feedback hybrid way. Rhythm allows the aggressive deployment of BE jobs on machines contributing less to tail latency based on Servpod-level control. Rhythm also adopts a BE-distinguishable reclamation scheme that reduces the useless computation yielded by BE reclamation under SLO violations. We evaluate Rhythm with typical LC services and BE jobs under different load scenarios, and find it can improve the resource efficiency significantly. Rhythm can be deployed easily in a private cloud, where we can conduct deep analysis on LC services. The characterization cost is low, as (1) it only relies on the LC service itself, (2) the request tracing and performance monitoring have always been an important component in the cloud, even without deploying Rhythm.

In the future, we would like to further improve system resource efficiency through co-locating multi-tenant LCs and BEs. For the public cloud where we know little about the LC service, we will explore the design space of co-locations using the evolved software and hardware isolation mechanisms.

ACKNOWLEDGMENTS

This work is supported by the National Key Research and Development Program of China No.2022YFB 4500702; project ZR2022LZH018 supported by Shandong Provincial Natural Science Foundation; the National Natural Science Foundation of China under grant 62141218, 62372322 and the open project of Zhejiang Lab (2021DA0AM01/003).

REFERENCES

[1] 2020. https://parsec.cs.princeton.edu/.

[2] 2020. Scimark:A benchmark for scientific and numerical computing. https://openbenchmarking.org/test/pts/scimark2-1.3.2.

[3] 2020. The SPEC Cloud IaaS 2018 benchmark is SPEC's second benchmark suite to measure cloud performance. https://www.spec.org/.

[4] 2020. Tensorflow-Bench: A benchmark framework for TensorFlow. https://github.com/tensorflow/benchmarks.

[5] S. Agarwala, F. Alegre, K. Schwan, and J. Mehalingham. 2007. E2EProf: Automated End-to-End Performance Management for Enterprise Systems. In *The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. 749–758.

[6] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. 2003. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 74–89.

[7] Pradeep Ambati, Íñigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. 2020. *Providing SLOs for Resource-Harvesting VMs in Cloud Platforms*. USENIX Association, USA.

[8] Javed A. Aslam and Mark Montague. 2001. Models for Metasearch. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (New Orleans, Louisiana, USA) *(SIGIR '01)*. Association for Computing Machinery, New York, NY, USA, 276–284.

[9] Javed A. Aslam and Mark H. Montague. 2001. Models for Metasearch. In *SIGIR 2001: Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, September 9-13, 2001, New Orleans, Louisiana, USA*, W. Bruce Croft, David J. Harper, Donald H. Kraft, and Justin Zobel (Eds.). ACM, 275–284.

[10] Paul Barham, Richard Black, Moises Goldszmidt, Rebecca Isaacs, John MacCormick, Richard Mortier, and Aleksandr Simma. 2008. *Constellation: automated discovery of service and host dependencies in networked systems*. Technical Report MSR-TR-2008-67. 1–14 pages.

[11] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. 2004. Using Magpie for request extraction and workload modelling. In *Proceedings of the Sixth USENIX Symposium on Operating Systems Design and Implementation (OSDI) 2004* (proceedings of the sixth usenix symposium on operating systems design and implementation (osdi) 2004 ed.). 259–272.

[12] Sean Kenneth Barker and Prashant Shenoy. 2010. Empirical Evaluation of Latency-sensitive Application Performance in the Cloud. In *Proceedings of the First Annual ACM SIGMM Conference on Multimedia Systems* (Phoenix, Arizona, USA) *(MMSys '10)*. ACM, New York, NY, USA, 35–46.

[13] Anne Benoit, Mourad Hakem, and Yves Robert. 2008. Fault tolerant scheduling of precedence task graphs on heterogeneous platforms. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. 1–8.

[14] Anne Benoit, Mourad Hakem, and Yves Robert. 2009. Contention awareness and fault-tolerant scheduling for precedence constrained tasks in heterogeneous systems. *Parallel Comput.* 35, 2 (2009), 83–108.

[15] P. Chen, Y. Qi, and D. Hou. 2019. CauseInfer: Automated End-to-End Performance Diagnosis with Hierarchical Causality Graph in Cloud Environment. *IEEE Transactions on Services Computing* 12, 2 (March 2019), 214–230.

[16] Shuang Chen, Christina Delimitrou, and José F. Martínez. 2019. 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* (Providence, RI, USA) *(ASPLOS '19)*. ACM, New York, NY, USA, 107–120.

[17] Xu Chen, Ming Zhang, Z. Morley Mao, and Paramvir Bahl. 2008. Automating Network Application Dependency Discovery: Experiences, Limitations, and New Solutions. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 117–130.

[18] MIchael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. 2014. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 217–231.

[19] The Internet Traffic Archive ClarkNet. 2017. http://ita.ee.lbl.gov/html/traces.html.

[20] Henry Cook, Miquel Moreto, Sarah Bird, Khanh Dao, David A Patterson, and Krste Asanovic. 2013. A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 308–319.

[21] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (Feb. 2013), 74–80.

[22] DeathStarBench. 2019. https://github.com/delimitrou/DeathStarBench.

[23] Christina Delimitrou and Christos Kozyrakis. 2013. ibench: Quantifying interference for datacenter applications. In *2013 IEEE international symposium on workload characterization (IISWC)*. IEEE, 23–33.

[24] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 77–88.

[25] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: resource-efficient and QoS-aware cluster management. *ACM SIGPLAN Notices* 49, 4 (2014), 127–144.
[26] Christina Delimitrou and Christos Kozyrakis. 2016. HCloud: Resource-Efficient Provisioning in Shared Cloud Systems. *SIGPLAN Not.* 51, 4 (March 2016), 473–488.
[27] Elasticsearch. 2021. Elasticsearch: a search engine based on the Lucene library. https://lucene.apache.org/solr/.
[28] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. 2002. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Comput. Surv.* 34, 3 (Sept. 2002), 375–408.
[29] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. *SIGPLAN Not.* 47, 4 (March 2012), 37–48.
[30] Rodrigo Fonseca, George Porter, Randy H. Katz, and Scott Shenker. 2007. X-Trace: A Pervasive Network Tracing Framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*. USENIX Association, Cambridge, MA.
[31] Yu Gan and Christina Delimitrou. 2018. The Architectural Implications of Cloud Microservices. *IEEE Computer Architecture Letters* 17, 2 (July 2018), 155–158.
[32] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. ACM, New York, NY, USA, 3–18.
[33] Alexander N. Gorban, Lyudmila I. Pokidysheva, Elena V. Smirnova, and Tatiana A. Tyukina. 2011. Law of the Minimum Paradoxes. *Bulletin of Mathematical Biology* 73 (2011), 2013–2044.
[34] Sriram Govindan, Jie Liu, Aman Kansal, and Anand Sivasubramaniam. 2011. Cuanta: Quantifying Effects of Shared On-chip Resource Interference for Consolidated Virtual Machines. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing* (Cascais, Portugal) *(SOCC '11)*. ACM, New York, NY, USA, 22:1–22:14.
[35] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. 2019. Who Limits the Resource Efficiency of My Datacenter: An Analysis of Alibaba Datacenter Traces. In *Proceedings of the International Symposium on Quality of Service* (Phoenix, Arizona) *(IWQoS '19)*. ACM, New York, NY, USA, Article 39, 10 pages.
[36] HiBench. 2020. HiBench: HiBench is a big data benchmark suite that helps evaluate different big data frameworks in terms of speed, throughput and system resource utilizations. https://github.com/Intel-bigdata/HiBench.
[37] Calin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. 2018. PerfIso: Performance Isolation for Commercial Latency-Sensitive Services. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 519–532.
[38] Alexandru Iosup, Nezih Yigitbasi, and Dick Epema. 2011. On the Performance Variability of Production Cloud Services. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 104–113.
[39] Ravi R. Iyer. 2004. CQoS: a framework for enabling QoS in shared caches of CMP platforms. In *International Conference on Supercomputing*. 257–266.
[40] Bart Jacob, Paul Larson, B Leitao, and SAMM Da Silva. 2008. SystemTap: instrumenting the Linux kernel for analyzing performance and functional problems. *IBM Redbook* (2008).
[41] jaeger. 2019. https://www.jaegertracing.io/.
[42] M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver. 2012. A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC. In *DAC Design Automation Conference 2012*. 850–855.
[43] Jonathan Kaldor, Jonathan Mace, Michal Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. 2017. Canopy: An End-to-End Performance Tracing And Analysis System. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 34–50.
[44] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim. 2012. Measuring interference between live datacenter applications. In *High PERFORMANCE Computing, Networking, Storage and Analysis*. 1–12.
[45] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. 2019. GrandSLAm: Guaranteeing SLAs for Jobs in Microservices Execution Frameworks. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 34, 16 pages.
[46] Harshad Kasture and Daniel Sanchez. 2014. Ubik: efficient cache sharing with strict qos for latency-critical workloads. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 729–742.
[47] Darja Krushevskaja and Mark Sandler. 2013. Understanding latency variations of black box services. In *Proceedings of the 22nd international conference on World Wide Web*. ACM, 703–714.

[48] Kubernetes. 2019. https://kubernetes.io/.
[49] Wenyu Qu Kunlin Zhan Laiping Zhao, Fangshu Li and Qingman Zhang. 2021. AITurbo: Unified Compute Allocation for Partial Predictable Training in Commodity Clusters. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '21)*. ACM, USA.
[50] Qixiao Liu and Zhibin Yu. 2018. The Elasticity and Plasticity in Semi-Containerized Co-locating Cloud Workload: A View from Alibaba Trace. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. ACM, New York, NY, USA, 347–360.
[51] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving resource efficiency at scale. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 450–462.
[52] LTTng. 2019. https://lttng.org/.
[53] Piotr Luszczek, Jack J Dongarra, David Koester, Rolf Rabenseifner, Bob Lucas, Jeremy Kepner, John McCalpin, David Bailey, and Daisuke Takahashi. [n.d.]. Introduction to the HPC Challenge Benchmark Suite. ([n. d.]).
[54] Jiuyue Ma, Xiufeng Sui, Ninghui Sun, Yupeng Li, Zihao Yu, Bowen Huang, Tianni Xu, Zhicheng Yao, Yun Chen, Haibin Wang, Lixin Zhang, and Yungang Bao. 2015. Supporting Differentiated Services in Computers via Programmable Architecture for Resourcing-on-Demand (PARD). *SIGARCH Comput. Archit. News* 43, 1 (March 2015), 131–143.
[55] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. 2015. Retro: Targeted Resource Management in Multi-tenant Distributed Systems. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 589–603.
[56] A. K. Maji, S. Mitra, and S. Bagchi. 2015. ICE: An Integrated Configuration Engine for Interference Mitigation in Cloud Services. In *2015 IEEE International Conference on Autonomic Computing*. 91–100.
[57] Haroon Malik, Hadi Hemmati, and Ahmed E Hassan. 2013. Automatic detection of performance deviations in the load testing of large scale systems. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 1012–1021.
[58] Raman Manikantan, Kaushik Rajan, and Ramaswamy Govindarajan. 2012. Probabilistic shared cache management (PriSM). In *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*. IEEE, 428–439.
[59] 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 *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 248–259.
[60] 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 *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (Porto Alegre, Brazil) *(MICRO-44)*. ACM, New York, NY, USA, 248–259.
[61] Peter Mattson, Christine Cheng, Gregory F. Diamos, Cody Coleman, Paulius Micikevicius, David A. Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debo Dutta, Udit Gupta, Kim M. Hazelwood, Andy Hock, Xinyuan Huang, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Carole-Jean Wu, Lingjie Xu, Cliff Young, and Matei Zaharia. 2020. MLPerf Training Benchmark. In *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*, Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze (Eds.). mlsys.org.
[62] D. A. Menasce. 2002. TPC-W: A Benchmark for E-Commerce. *IEEE Internet Computing* 6 (05 2002), 83–87.
[63] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. 2010. Q-clouds: managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European conference on Computer systems*. ACM, 237–250.
[64] Rajiv Nishtala, Vinicius Petrucci, Paul Carpenter, and Magnus Sjalander. 2020. Twig : Multi-Agent Task Management for Colocated Latency-Critical Cloud Services. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 167–179.
[65] Dejan Novakovic, Nedeljko Vasic, Stanko Novakovic, Dejan Kostic, and Ricardo Bianchini. 2013. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (San Jose, CA) *(USENIX ATC'13)*. USENIX Association, Berkeley, CA, USA, 219–230.
[66] Numactl. 2019. https://github.com/numactl/numactl.
[67] Zhonghong Ou, Hao Zhuang, Jukka K. Nurminen, Antti Ylä-Jääski, and Pan Hui. 2012. Exploiting Hardware Heterogeneity Within the Same Instance Type of Amazon EC2. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing* (Boston, MA) *(HotCloud'12)*. USENIX Association, Berkeley, CA, USA, 4–4.
[68] Ioannis Papadakis, Konstantinos Nikas, Vasileios Karakostas, Georgios Goumas, and Nectarios Koziris. 2017. Improving QoS and Utilisation in modern multi-core servers with Dynamic Cache Partitioning. In *Proceedings of the Joined Workshops COSH 2017 and VisorHPC 2017*, Carsten Clauss, Stefan Lankes, Carsten Trinitis, and Josef Weidendorfer (Eds.). Stockholm, Sweden, 21–26.
[69] Jinsu Park, Seongbeom Park, and Woongki Baek. 2019. CoPart: Coordinated Partitioning of Last-Level Cache and Memory Bandwidth for Fairness-Aware Workload Consolidation on Commodity Servers. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. ACM, New York, NY, USA, Article 10, 16 pages.

[70] Tirthak Patel and Devesh Tiwari. 2020. CLITE : Efficient and QoS-Aware Co-location of Multiple Latency-Critical Jobs for Warehouse Scale Computers. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 193–206.

[71] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 3, 14 pages.

[72] Xing Pu, Ling Liu, Yiduo Mei, Sankaran Sivathanu, Younggyun Koh, and Calton Pu. 2010. Understanding Performance Interference of I/O Workload in Virtualized Cloud Environments. In *Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing (CLOUD '10)*. IEEE Computer Society, Washington, DC, USA, 51–58.

[73] Navaneeth Rameshan, Leandro Navarro, Enric Monte, and Vladimir Vlassov. 2014. Stay-Away, Protecting Sensitive Applications from Performance Interference. In *Proceedings of the 15th International Middleware Conference* (Bordeaux, France) *(Middleware '14)*. ACM, New York, NY, USA, 301–312.

[74] Redis. 2019. Redis: an open source, in-memory data structure store. https://redis.io.

[75] B. Sang, J. Zhan, G. Lu, H. Wang, D. Xu, L. Wang, Z. Zhang, and Z. Jia. 2012. Precise, Scalable, and Online Request Tracing for Multitier Services of Black Boxes. *IEEE Transactions on Parallel and Distributed Systems* 23, 6 (June 2012), 1159–1167.

[76] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. 2010. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 460–471.

[77] Prateek Sharma, Ahmed Ali-Eldin, and Prashant Shenoy. 2019. Resource Deflation: A New Approach For Transient Resource Reclamation. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 33, 17 pages.

[78] Timothy Sherwood, Erez Perelman, Greg Hamerly, Suleyman Sair, and Brad Calder. 2003. Discovering and Exploiting Program Phases. *IEEE Micro* 23, 6 (Nov. 2003), 84–93.

[79] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure.* Technical Report. Google, Inc.

[80] S. Sivathanu, X. Pu, L. Liu, X. Dong, and Y. Mei. 2013. Performance Analysis of Network I/O Workloads in Virtualized Data Centers. *IEEE Transactions on Services Computing* 6 (01 2013), 48–63.

[81] Solr. 2021. Solr is the popular, blazing-fast, open source enterprise search platform built on Apache Lucene. https://solr.apache.org/.

[82] Spark-Bench. 2020. Spark-Bench: A Benchmark Suite for Apache Spark. https://github.com/codait/spark-bench.

[83] Shekhar Srikantaiah, Mahmut Kandemir, and Qian Wang. 2009. SHARP control: controlled shared cache management in chip multiprocessors. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 517–528.

[84] Christopher Stewart and Kai Shen. 2005. Performance modeling and system management for multi-component online services. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 71–84.

[85] Fei Tang, Wanling Gao, Jianfeng Zhan, Chuanxin Lan, Xu Wen, Lei Wang, Chunjie Luo, Zheng Cao, Xingwang Xiong, Zihan Jiang, Tianshu Hao, Fanda Fan, Fan Zhang, Yunyou Huang, Jianan Chen, Mengjia Du, Rui Ren, Chen Zheng, Daoyi Zheng, Haoning Tang, Kunlin Zhan, Biao Wang, Defei Kong, Minghe Yu, Chongkang Tan, Huan Li, Xinhui Tian, Yatao Li, Junchao Shao, Zhenyu Wang, Xiaoyu Wang, Jiahui Dai, and Hainan Ye. 2021. AIBench Training: Balanced Industry-Standard AI Training Benchmarking. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2021, Stony Brook, NY, USA, March 28-30, 2021*. 24–35.

[86] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R. Ganger. 2006. Stardust: Tracking Activity in a Distributed Storage System. *SIGMETRICS Perform. Eval. Rev.* 34, 1 (June 2006), 3–14.

[87] 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 *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 30, 14 pages.

[88] A Tirumala, F Qin, J Dugan, J Ferguson, and K Gibbs. 2005. Iperf: The TCP/UDP bandwidth measurement tool. *http.dast.nlanr.net/Projects* 38 (2005).

[89] Guohui Wang and T. S. Eugene Ng. 2010. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In *Proceedings of the 29th Conference on Information Communications* (San Diego, California, USA) *(INFOCOM'10)*. IEEE Press, Piscataway, NJ, USA, 1163–1171.

[90] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, Chen Zheng, Gang Lu, Kent Zhan, Xiaona Li, and Bizhu Qiu. 2014. BigDataBench: A big data benchmark suite from internet services. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 488–499.

[91] Fei Xu, Fangming Liu, and Hai Jin. 2016. Heterogeneity and Interference-Aware Virtual Machine Provisioning for Predictable Performance in the Cloud. *IEEE Trans. Comput.* 65, 8 (2016), 2470–2483.

[92] Fei Xu, Fangming Liu, Linghui Liu, Hai Jin, Bo Li, and Baochun Li. 2014. iAware: Making Live Migration of Virtual Machines Interference-Aware in the Cloud. *IEEE Trans. Comput.* 63, 12 (Dec. 2014), 3012–3025.

[93] H. Xu, X. Ning, H. Zhang, J. Rhee, and G. Jiang. 2016. PInfer: Learning to Infer Concurrent Request Paths from System Kernel Events. In *2016 IEEE International Conference on Autonomic Computing (ICAC)*. 199–208.

[94] Ran Xu, Subrata Mitra, Jason Rahman, Peter Bai, Bowen Zhou, Greg Bronevetsky, and Saurabh Bagchi. 2018. Pythia: Improving Datacenter Utilization via Precise Contention Prediction for Multiple Co-located Workloads. In *Proceedings of the 19th International Middleware Conference* (Rennes, France) *(Middleware '18)*. ACM, New York, NY, USA, 146–160.

[95] Neeraja J Yadwadkar, Ganesh Ananthanarayanan, and Randy Katz. 2014. Wrangler: Predictable and faster jobs using fewer resources. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 1–14.

[96] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. 2013. Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. *ACM SIGARCH Computer Architecture News* 41, 3 (2013), 607–618.

[97] Xi Yang, Stephen M. Blackburn, and Kathryn S. McKinley. 2016. Elfen Scheduling: Fine-Grain Principled Borrowing from Latency-Critical Workloads Using Simultaneous Multithreading. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 309–322.

[98] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for in-Memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (San Jose, CA) *(NSDI'12)*. USENIX Association, USA, 2.

[99] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. 2013. CPI 2: CPU performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*. 379–391.

[100] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang. 2014. SMiTe: Precise QoS Prediction on Real-System SMT Processors to Improve Utilization in Warehouse Scale Computers. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 406–418.

[101] 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 *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*. IEEE, 406–418.

[102] Jiacheng Zhao, Huimin Cui, Jingling Xue, and Xiaobing Feng. 2016. Predicting Cross-Core Performance Interference on Multicore Processors with Regression Analysis. *IEEE Trans. Parallel Distrib. Syst.* 27, 5 (May 2016), 1443–1456.

[103] Jiacheng Zhao, Huimin Cui, Jingling Xue, Xiaobing Feng, Youliang Yan, and Wensen Yang. 2013. An Empirical Model for Predicting Cross-core Performance Interference on Multicore Processors. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques* (Edinburgh, Scotland, UK) *(PACT '13)*. IEEE Press, Piscataway, NJ, USA, 201–212.

[104] Wenyi Zhao, Quan Chen, Hao Lin, Jianfeng Zhang, Jingwen Leng, Chao Li, Wenli Zheng, Li Li, and Minyi Guo. 2019. Themis: Predicting and Reining in Application-Level Slowdown on Spatial Multitasking GPUs. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 653–663.

[105] Qin Zheng, Bharadwaj Veeravalli, and Chen-Khong Tham. 2009. On the Design of Fault-Tolerant Scheduling Strategies Using Primary-Backup Approach for Computational Grids with Low Replication Costs. *IEEE Trans. Comput.* 58, 3 (2009), 380–393.

[106] Haishan Zhu and Mattan Erez. 2016. Dirigent: Enforcing QoS for latency-critical tasks on shared multicore systems. *ACM SIGARCH Computer Architecture News* 44, 2 (2016), 33–47.

[107] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. 2017. Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks. In *Proceedings of the IEEE international conference on computer vision*. 2223–2232.

[108] Zipkin. 2019. https://zipkin.io/.