Rhythm: Component-distinguishable Workload Deployment in Datacenters

Laiping Zhao
Tianjin Key Lab. of Advanced Networking (TANKLab), College of Intelligence and Computing (CIC), Tianjin University
laiping@tju.edu.cn

Yanan Yang
TANKLab, CIC, Tianjin University
ynyang@tju.edu.cn

Kaixuan Zhang
TANKLab, CIC, Tianjin University
kingxzhang@gmail.com

Xiaobo Zhou
TANKLab, CIC, Tianjin University
xiaobo.zhou@tju.edu.cn

Tie Qiu
TANKLab, CIC, Tianjin University
qiutie@tju.edu.cn

Keqiu Li
TANKLab, CIC, Tianjin University
keqiu@tju.edu.cn

Yungang Bao
Inst. of Computing Technology, CAS
baoyg@ict.ac.cn

Abstract

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 an LC workload as a whole when allocating resources to BE jobs and neglect the different features of components of an LC workload. This kind of coarse-grained co-location method leaves a significant room for improvement in resource utilization.

Based on the observation of the inconsistent interference tolerance abilities of different LC components, we propose a new abstraction called *Servpod*, which is a collection of a LC parts that are deployed on the same physical machine together, and show its merits on building a fine-grained co-location framework. The key idea is to differentiate the BE throughput launched with each LC Servpod, i.e., Servpod with high interference tolerance ability can be deployed along with more BE jobs. Based on Servpods, we present `Rhythm`, a co-location controller that maximizes the resource utilization while guaranteeing LC service's tail latency requirement. It quantifies the interference tolerance ability of each servpod through the analysis of *tail-latency contribution*. We evaluate `Rhythm` using LC services in forms of containerized processes and microservices, and find that it can improve the system throughput by 31.7%, CPU utilization by 26.2%, and memory bandwidth utilization by 34% while guaranteeing the SLA (service level agreement).

ACM Reference Format:
Laiping Zhao, Yanan Yang, Kaixuan Zhang, Xiaobo Zhou, Tie Qiu, Keqiu Li, and Yungang Bao. 2020. Rhythm: Component-distinguishable Workload Deployment in Datacenters. In *Fifteenth European Conference on Computer Systems (EuroSys '20), April 27–30, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3342195.3387534

1 Introduction

The multitenant sharing nature of cloud computing aggravates contention for shared resources, such as cores, memory, cache, memory bandwidth, and networks in datacenters. The resulting disorder of a system, in which the full dimensions of resource consumption at the instruction-cycle-level (known as states) become more intractable, causes a long tail latency for services [13]. For example, the fluctuation range of tail latency in Google's latency-critical (LC) service is between 0 and 500ms, and the highest variation difference can even exceed 600× [38].

To mitigate the disorder caused by the contention, prior work seeks to enhance control of the resource management system through two approaches: *hardware methods* and *software methods*. The *hardware methods*, such as the Intel RDT [56] and PARD [43], open control interfaces for differentiating services at the hardware level for resourcing-on-demand. Although they are effective in performance isolation, their adoption requires the new hardware support. The *software methods* commonly rely on resource overprovisioning to reduce the interference. However, wasteful overprovisioning results in low resource utilization, thereby increasing the cost of cloud services. For example, resource reservations by Twitter could reach up to 80% of total capacity, while their production cluster's CPU utilization is constantly below 20% [16]. Similarly, traces from both Google [62] and Aliyun [40]

EuroSys '20, April 27–30, 2020, Heraklion, Greece

ACM ISBN 978-1-4503-6882-7/20/04…$15.00
https://doi.org/10.1145/3342195.3387534

showed that they merely achieve aggregate CPU utilization of 25-35% and aggregate memory utilization of 40%.

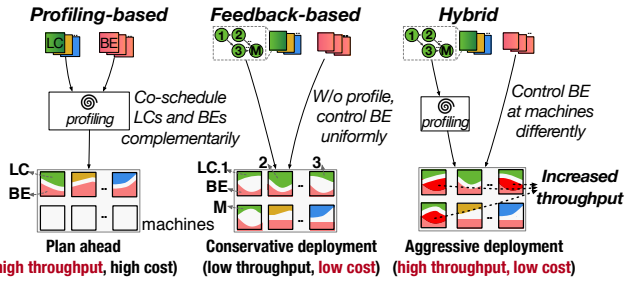


Figure 1. Schematic overview of the workload deployment method.

Improving the resources utilization of cloud systems while guaranteeing the user's experience needs more control from the management system. We identified the two prevalent approaches shown in Figure 1. *Profiling-based* strategy consolidates noncontending applications according to their resource usage profiles [12, 15, 16, 48, 69, 73, 75, 80]. *Feedback-based* strategy deploys workloads directly in machines, but continuously responses to service level agreement (SLA) violation monitoring using actions, such as system reconfiguration [45] or resource reallocation [8, 41, 51]. However, profiling-based strategy generates high throughput at the high cost of profiling. This does not make it easy to deploy them in large-scale cloud systems widely, due to the large number of continuously emerging applications and possible consolidations in cloud today. *Without profiling, feedback-based strategy has to control workloads at the granularity of whole LC or best-effort batch (BE) jobs uniformly, ignoring the differences among service parts when the LC service is distributively deployed. BE jobs are even treated as "second-class citizens": they can only be allocated with limited resources and might be rescheduled at any time for avoiding SLA violations [26]. This coarse-grained and conservative strategy leads to low throughput.*

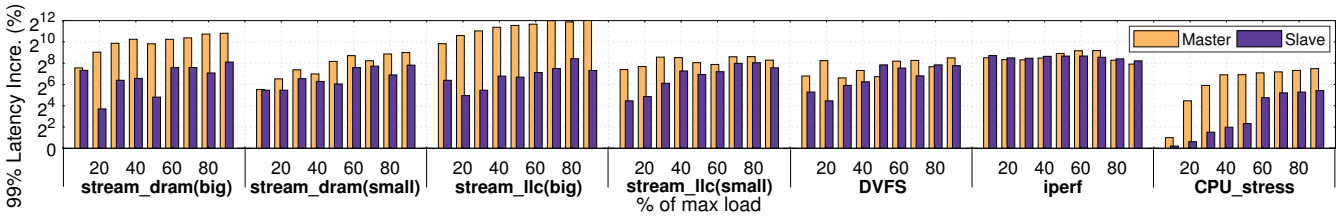
We suggest to launch BE jobs more aggressively in feedback way while capturing a little more profile information about the solo LC workload (i.e., **the hybrid strategy**). Leveraging an aggressive policy would highly risk SLA violations; hence, where and how to launch more BE jobs should be very carefully decided. With the virtualization technique evolving from VM to container and further to the state-of-the-art serverless functions [33], the controllable objects on the cloud providers' side are becoming more varied and lightweight, and the control method is becoming more flexible. Therefore, control at a finer level of granularity is a promising technique of the next cloud management system. To this end, we design a fine-grained controller built around the hybrid strategy. It differentiates the BE throughput launched with each LC *Servpod*, a new abstraction which is a collection of a LC service parts that are deployed on the same physical machine together. A *Servpod* is a service-deployment unit, denoting the mappings between LC structure and physical machines.

To guide the launch of BE jobs, we analyze the *tail-latency contribution* of each Servpod under the solo-run of LC workload, to measure their weights in overall tail latency.

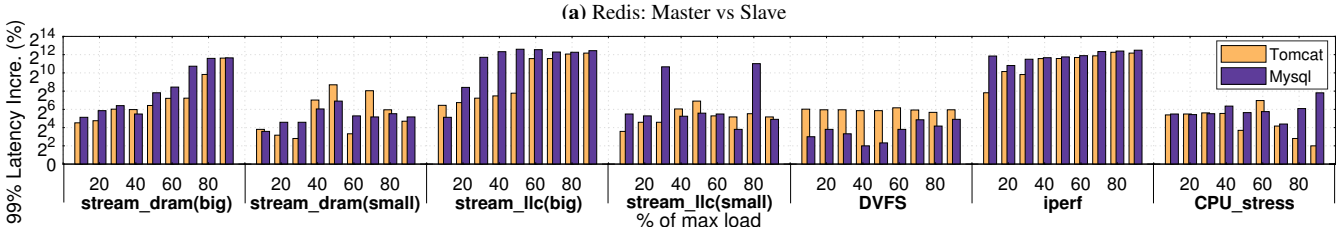
The analysis depends on the request sojourn time in each Servpod. There are several challenges towards this problem. **(1) How can we track a service request and extract its sojourn time at each Servpod?** An LC workload likely consists of many Servpods, and user requests may be processed by different paths of the service call. While program instrumentation can provide accurate measurement on the sojourn time of each request in each Servpod, it requires a high development cost. Thus, we choose the non-intrusive method, that is, 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 control decisions? Following the call path of an LC request, every Servpod performs different contributions over the end-to-end tail latency. We define the *contribution* for each Servpod using the sojourn time mean, variance and correlation coefficient, to direct the deployment of BE jobs: Servpod with small contribution to the tail latency can be deployed along with more BE jobs.
(3) How can we implement the controller by making use of the different contributions of Servpods? We derive two thresholds *loadlimit* and *slacklimit* using the contributions of Servpods. By monitoring the realtime workload and slack between current tail latency and SLA target, the top controller in each Servpod continuously makes decisions about adding or releasing resources of BE jobs and the subcontrollers implement the decisions accordingly.

To address these challenges, we present `Rhythm`, a cloud controller that maximizes the resource utilization while guaranteeing LC service's tail latency requirement. `Rhythm` supports to track the requests in all Servpods, and derives the contribution of each Servpod to the end-to-end tail latency. It also implements a runtime agent at each Servpod, enabling the aggressive launch of BE jobs at Servpods with less contributions. It 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 the inconsistent interference-tolerance abilities of different components of LC workload. (§ 2)
- A new abstraction, *Servpod*, and the analysis of *tail-latency contributions* of Servpods that allows distinguishable workloads co-location in cloud system (§ 3)



(a) Redis: Master vs Slave



(b) E-commerce website: Tomcat vs MySQL

Figure 2. Impact of interference on the 99*th* 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 99*th* percentile latency normalized to the solo-run performance (presented in logarithmic scale).

- A prototype system, `Rhythm`, designed following the hybrid strategy: "profiling LC once, feedback control BE", which has a high extensibility and flexibility in traditional processes and microservice scenarios. (§ 3 and § 4)
- A detailed comparative evaluation of `Rhythm` and non-component-distinguishable system, shows significant improvement in the system throughput without SLA violations. (§ 5)

2 Inconsistent Interference Tolerance Ability

In this section, we show the inconsistent interference-tolerance feature of components of an LC service, utilizing two typical applications: the multitier *E-commerce website* [50] consists of four components, namely, *HAProxy*, *Tomcat*, *Amoeba* and *MySQL*, and the fan-out *Redis* [61] consists of two components, *Master* and *Slave*. We generate interference using five microbenchmarks as BE jobs, namely, *CPU-stress* [46], *stream-llc* [14], *stream-dram* [14], *DVFS* and *iperf* [71], which can put the 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 99*th* 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` [54] 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 coschedule 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 coschedule 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) and iperf, the difference between Tomcat and MySQL reaches more than 35× and 10×, respectively (The difference in iperf in Figure 2b is not obvious because the Y-axis is presented in logarithmic scale). Tomcat is more sensitive to interference generated by 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 shows significant inconsistency. While resource contention at the high-sensitive components easily causes SLA violation, the other less-sensitive components still have room for more BE deployment. This makes the existing work that control the co-location of LC workload and BE jobs uniformly less-efficient due to their neglect of the different interference-tolerance abilities of components (aka "**Law of the Minimum**" [24]).

3 Rhythm Design

Given the observation that different components of an LC service have inconsistent interference-tolerance abilities, it is possible to implement a distinguishable fine-grained workloads co-location controller to improve the system throughput in datacenter. In this section, we present the design of Rhythm and show where and how to launch BE jobs making use of the features of LC service components.

3.1 The Servpod Abstraction

We introduce *Servpod*, a collection of service components from one LC workload that are deployed together on the same physical machine. If an LC workload can be represented by a directed acyclic graph (DAG), where the set of vertices represents LC components and edges denotes the precedence relation among them, then a Servpod could include multiple LC components if they are scheduled on the same physical machine. Hence, Servpod reveals the mappings between physical machines and its service structure. The number of the Servpods equals the number of deployed physical machines.

A Servpod could be one or multiple processes, containers or microservices [21]. 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.

3.2 System Overview

The insight of Rhythm is that, while Servpods with larger *contributions* to the tail latency have to be controlled conservatively, we shall allow the others to launch BE jobs aggressively. Quantifying *contribuitons* can be done through an offline profiling of Servpods or an online analysis of real-time monitoring. We choose the offline profiling way for two

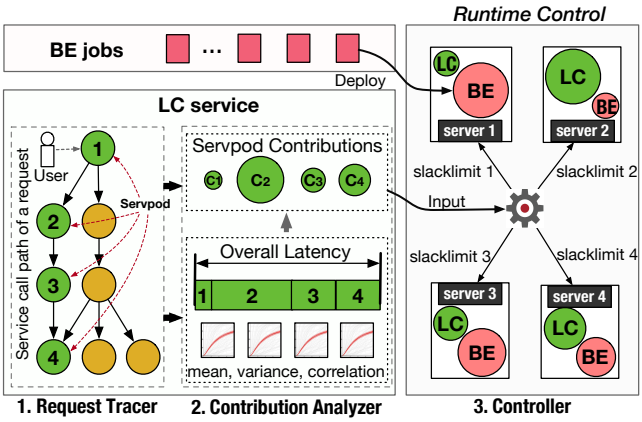


Figure 3. Design of Rhythm.

reasons: (1) As the contribuiton 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 SLA 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. As shown in Figure 3, 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 on the overall tail-latency includes two ways: *Directed way* requires to collect the sojourn times of requests in each Servpod, while the contribution can be derived through statistical modeling and analysis on sojourn times. *Indirected way*, like "bubble pressure" [48], measures the performance degradation of LC service (e.g., increasing of tail-latency, IPC) by putting a tunable amount of pressure on individual Servpod, then the contribution can be defined as the "bubble size" it can tolerate while guaranteeing SLA. However, as shown in Figure 2, a Servpod performs different sensitivities under the interferences from different bubbles [57]. In addition, variation of the workload is also an non-negligible factor affecting the LC service's performance. Using the "bubble pressure" that generates only one-dimensional interference to measure the contribution of Servpod is surely insufficient. For example. a CPU-intensive Servpod that contributes a lot to the overall tail latency could tolerate strong interferences generated by an I/O-intensive bubble. It is also impossible to design a single bubble suite that is general enough to represent all kinds of BE jobs. Hence, we choose the *directed way* to characterize the contributions of Servpods.

In particular, the *request tracer* identifies the service call paths of requests and records their sojourn time at each Servpod when LC workload runs solely. Then, the *contribution analyzer* derives the *contribution* of each Servpod to the tail latency using the mean, variance and the Pearson correlation coefficient of sojourn times. This characterization relies solely

on the LC service itself, and its cost increases linearly over the number of Servpods. Hence, compared with the *profiling-based* approaches that measure the interference of $M \times N$ combinations by M LC services and N BE jobs, `Rhythm` reduces the cost to M. The *controller* adopts a *contribution*-based thresholding methodology to control the resource allocation for BE jobs at every machine. It first derives the control thresholds for different Servpods using a thresholding algorithm, then treats BEs uniformly using a trial-and-error approach for avoiding SLA violations.

For a newly deployed LC service, we activate both the *request tracer* and *contribution analyzer* only once for characterizing its Servpods' contributions. While each *controller* continues running independently for controlling BE jobs along with each Servpod, its thresholding algorithm (i.e., the only step that needs the 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.

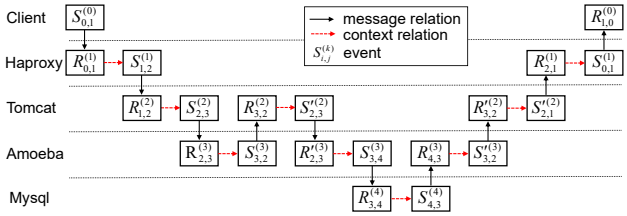


Figure 4. The CPG constructed by a request to e-commerce.

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 4 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.

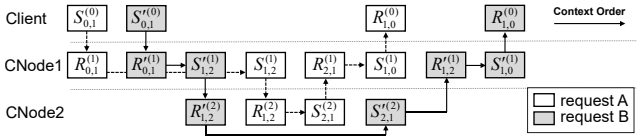


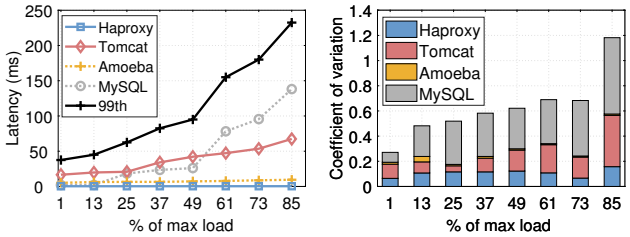
Figure 5. IntraServpod causality: request B is issued earlier by Servpod1, but returned later than request A.

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 5). Since they may share the same *context identifier*, the mappings of RECV and SEND of them would be incorrect, resulting in an incorrect sojourn time at the Servpod. We avoid this problem by analyzing the mean sojourn time of all requests in the design of the *contribution analyzer* (Equations 1-3), instead of using each sojourn time directly. For example, the mean sojourn time of request A and B at Servpod1 in Figure 5 is not affected by the mismatching, because we have $(S_{1,2}^{(1)} - R_{0,1}^{(1)}) + (S_{1,0}^{(1)} - R_{2,1}^{(1)}) + (S_{1,2}^{\prime(1)} - R_{0,1}^{\prime(1)}) + (S_{1,0}^{\prime(1)} - R_{2,1}^{\prime(1)}) = (S_{1,2}^{(1)} - R_{0,1}^{\prime(1)}) + (S_{1,0}^{(1)} - R_{2,1}^{(1)}) + (S_{1,2}^{\prime(1)} - R_{0,1}^{(1)}) + (S_{1,0}^{\prime(1)} - R_{2,1}^{\prime(1)})$.

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 (i.e., timestamp) could also lead to mismatching. Similar to the identification of intraServpod causality, we also adopt the sojourn time mean of communications of all requests in the design of the *contribution analyzer*.

3.4 Contribution Analyzer



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

Figure 6. The average sojourn time of Servpods in E-commerce website and their normalized coefficient of variations collected in solo-run.

Figure 6a shows the average sojourn time of the four Servpods of E-commerce and the overall 99*th* percentile latency under different request loads. Figure 6b 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 maxload, *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 6a.

(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 99*th* percentile latency increases significantly due to the high variance of *MySQL* (Figure 6b).

(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.

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

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 99*th*, 99.9*th* percentile, etc.). Let T_{tail}^j be the *tail* latency under load j, then, we have,

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

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

$$V_i = \frac{1}{\overline{T}_i} \sqrt{\frac{1}{m(m-1)} \sum_{j=1}^{m} (T_i^j - \overline{T}_i)^2} \tag{3}$$

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

$$C_i = f(\rho_{T_i,T_{tail}}, P_i, V_i) = \rho_{T_i,T_{tail}} P_i V_i \tag{4}$$

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:

$$C_i = \alpha_i \rho_{T_i,T_{tail}} P_i V_i \tag{5}$$

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

Note that Equation 5 may not be the only way to define the contribution. We validate its rationality though a comparative analysis between Servpod sensitivity and contribution. Figure 7 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

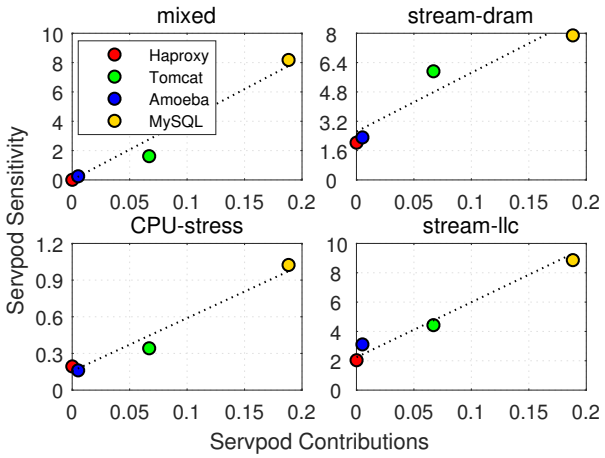


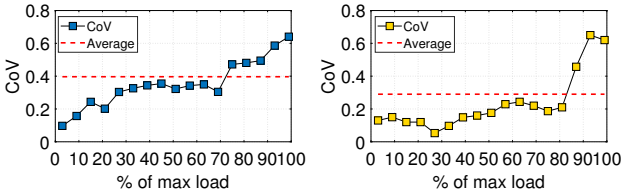
Figure 7. Servpod sensitivity vs contribuitions: the increase in the 99*th* percentile latency of E-commerce when a single Servpod is interfered by different BEs: (1) mixed BEs of wordcount, imageClassify, lstm, CPU-stress, stream-dram and stream-llc, (2) Stream-dram, (3) CPU-stress, (4) Stream-llc.

99*th* 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 control in algorithm 1 based on this contribution, and the experimental results show it works well. (See Figure 9-11).

3.5 Controller

Given the *contributions* of LC Servpods, we next present how the controller operates to control the start/stop of BE workloads. We design a controller running as an agent at every machine holding the LC Servpod. It adopts a 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 thresholding mechanism. Then, each controller runs independently to control BE jobs according to thresholds. 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.



(a) MySQL **(b)** Tomcat

Figure 8. The *CoV* of Servpod sojourn times increase over request loads. Decide 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 8a 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 8b).

Slacklimit: Let *slack* be the gap between the current tail latency and latency target in SLA. 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 SLA 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 SLA 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

Algorithm 1: *findSlacklimit(C_i)*

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

$stepSize = 1 - C_i / \sum_{i=1}^{m} C_i$;
$slacklimit = curLimit = 1.0$; `// Initialization` ;
$SLA\_violation = false$;
while $curLimit > 0$ **do**
    $curLimit = curLimit - stepSize$;
    run_system ($curLimit$) ;
    `// Running for 10 minutes` ;
    $SLA\_violation$=SLA_evaluation() ;
    if $SLA\_violation = true$ **then**
        $slacklimit = Record$.pop() ;
        break;
    else
        $Record$.push($curLimit$) ;

Algorithm 2: Decision making by top controller

Estimate *slack* based on monitored latency and SLA;
while *True* **do**
    $slack = (T_{tail}^{SLA} - T_{tail})/T_{tail}^{SLA}$;
    if $slack < 0$ **then**
        StopBE();
    else if $workload > loadLimit$ **then**
        SuspendBE();
    else if $0 < slack < slackLimit/2$ **then**
        CutBE();
    else if $slackLimit/2 < slack < slackLimit$ **then**
        DisallowBEGrowth();
    else
        AllowBEGrowth();
    sleep(2 seconds);

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. The *controller* in each Servpod manages the launch of BE jobs and their resource allocations through the collaboration of one top-controller and four subcontrollers.

Top controller: It compares the real-time request load and slack with the *loadlimit* and *slacklimit* and manages the running of BE jobs with five actions: *StopBE*, *CutBE*, *DisallowBEGrowth*, *AllowBEGrowth*, and *SuspendBE*. In particular,

1. *StopBE* immediately kills all the running BE jobs and releases all their resources.
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 reduces part of their allocated resources.
4. *DisallowBEGrowth* does not allow the number of BE jobs to 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}^{SLA} as the tail latency requirement stated in SLA. 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. While frequent monitoring and adjustment are effective to detect the load burst and protect the SLA 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 15).

1. CPU/LLC subcontroller: We adopt the same control as in Heracles [41] 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 and 10% LLC. Both *CutBE()* and *AllowBEGrowth()* adjust the cores and LLC of BE jobs at the granularity of one core 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 SLA), 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.

4 Implementation

We implemented a prototype for `Rhythm` in about 5.3KLOC of C, Java and Linux Shell. It runs on 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* [30]-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

Table 1. LC workloads and BE jobs.

LC Workloads						BE Jobs		
Workload	Domain	Servpods	MaxLoad	SLA	Containers	Workload	Domain	-intensive
E-commerce [50]	TPC-W website	Haproxy,Tomcat, Amoeba,MySQL	1300 QPS	250 ms	16	**CPU-stress** [46]	CPU stress testing tool	CPU
						Stream-llc [14]	LLC-benchmark in iBench	LLC
Redis [61]	Key-value store	Master,Slave	86K QPS	1.15 ms	18	**Stream-dram**[14]	DRAM-benchmark in iBench	DRAM
Solr [67]	Search	Apache+Solr, Zookeeper	400 QPS	350 ms	15	**iperf** [71]	Network stress testing tool	Network
Elasticsearch [18]	Index Engine	Index,Kibana	750 QPS	200 ms	12	**Wordcount** [23]	Big data analytics	mixed
Elgg [19]	Social Network	Nginx+PHP-FPM, Memcached,MySQL	200 QPS	320 ms	8	**ImageClassify**	Image classification on CycleGAN [85]	mixed
SNMS [22]	Microservice	UserService,frontend, MediaService	1500 QPS	380 ms	30	**LSTM**	Deep learning on Tensorflow [1]	mixed

APIs on latency and system status monitoring, contribution analysis, parameters exchanging, BE deploying and resource allocation updating in each Servpod agent. The interactions with operating system are mainly implemented through JDK runtime library and Linux shell interface.

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 isolation*: `Rhythm` uses Intel CAT (cache allocation technology) to partition the LLC into two parts: one is for the LC workload and the other is 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 firstly decides the schedule of its components. Co-located components form servpods, which are profiled against the load generator for deriving loadlimit and slacklimit. Then, the top controller in each physical machine decides if it is allowable to deploy BEs locally. It notifies the scheduler whether it accepts BEs or not. If yes, the scheduler checks the waiting queue of BE jobs and dispatches them to physical machines with sufficient resources. After receiving BEs, the subcontrollers at local physical machine decide the specific resource allocation for BEs.

System integration: `Rhythm` can be integrated into container management framework like Kubernetes [39] or serverless cloud system. To make `Rhythm` work in Kubernetes, we need to (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 controller agent for providing feedback to scheduling algorithms.

5 Evaluation

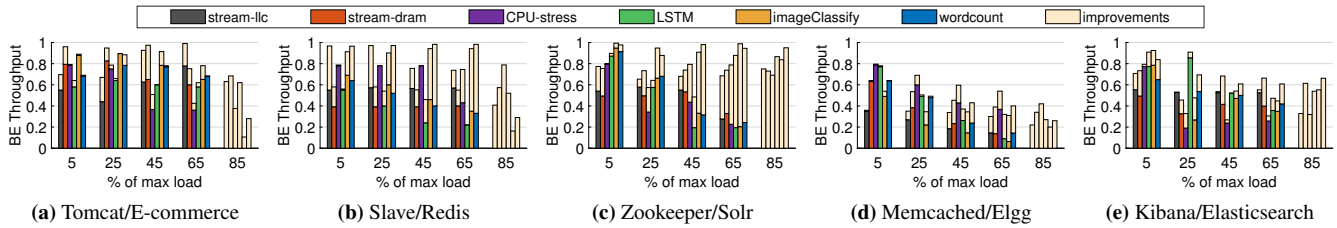
5.1 Methodology

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., *MaxLoad*) is measured when the arrival speed approaches to the maximum processing speed. Their SLAs 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 SLA*. We use query per second (QPS) to indicate the number of concurrent queries processed per second. We also deploy seven BE workloads: Four of them are synthetic for putting strong pressure on a specific resource (i.e., CPU, LLC, DRAM, and network), and the other three are real workloads, putting pressure on multiple resources.

Metrics: We colocate 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* (effective machine utilization) 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.

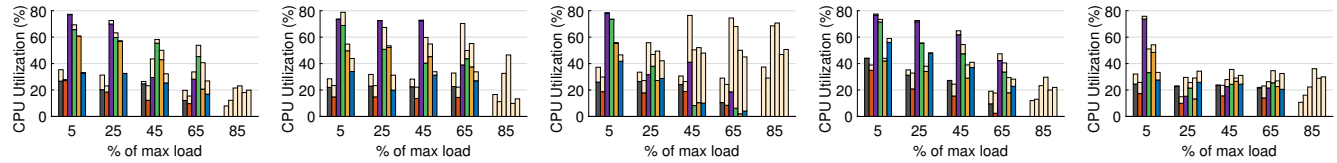
All the experimental results are compared against *Heracles* [41], which is a feedback-based method, but does not distinguish between Servpods: (1) It disables BE jobs at all machines whenever the load exceeds 85%. (2) It disallows the growth of BE jobs whenever the *slack* between the current tail latency and SLA target is less than 10%. We observe that the measured SLA on our testbed is larger than those in *Heracles* because the different software configurations and hardware environments we used. To make a fair comparison, our implementation of *Heracles* [41] also conducts its control using the same SLA as in Table 1.

Testbed: LC service and BE jobs are deployed on a cluster with four physical machines, with each configured with 40



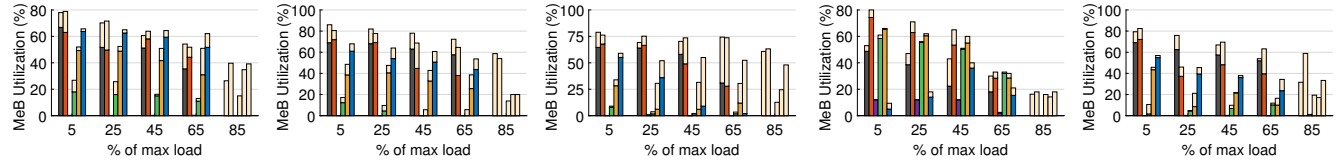
(a) Tomcat/E-commerce **(b)** Slave/Redis **(c)** Zookeeper/Solr **(d)** Memcached/Elgg **(e)** Kibana/Elasticsearch

Figure 9. The BE throughput at Servpods under different loads.



(a) Tomcat/E-commerce **(b)** Slave/Redis **(c)** Zookeeper/Solr **(d)** Memcached/Elgg **(e)** Kibana/Elasticsearch

Figure 10. The CPU utilization at Servpods under different loads.



(a) Tomcat/E-commerce **(b)** Slave/Redis **(c)** Zookeeper/Solr **(d)** Memcached/Elgg **(e)** Kibana/Elasticsearch

Figure 11. The memory bandwidth utilization at Servpods under different loads.

cores of a quad-socket Intel Xeon E7-4820 v4 @ 2.0 GHz and 64 GB of DRAM per socket. Each core has 32 KB of L1 cache and 256 KB of L2 cache, and each socket shares 20 MB of L3 cache. 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.

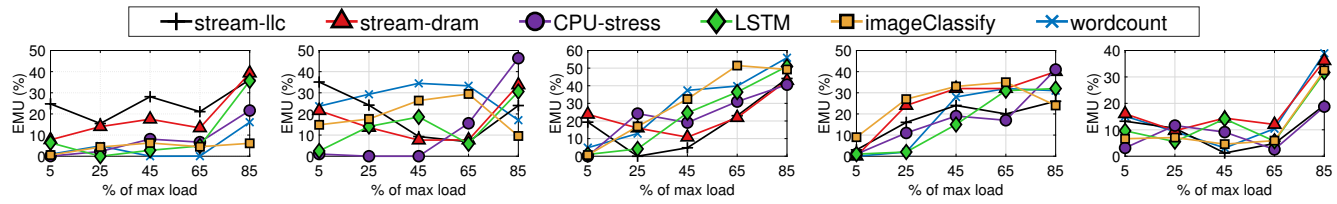
Overhead: After deploying `Rhythm` in the system, we measure its overhead and find that the request tracer only consumes approximately 6% of the CPU and 3 MB of memory, and each controller agent runs every 2 seconds only consumes 3.6% of the CPU and less than 50 MB of memory. `Rhythm` collects request sojourn time in each Servpod by solo-run LC service only once, the off-line profiling takes negligible overhead while the runtime control only occupies a small amount of resources.

5.2 Constant Load

We first evaluate the throughput and resource utilization at each Servpod. Then, we present the overall performance evaluation under constant load.

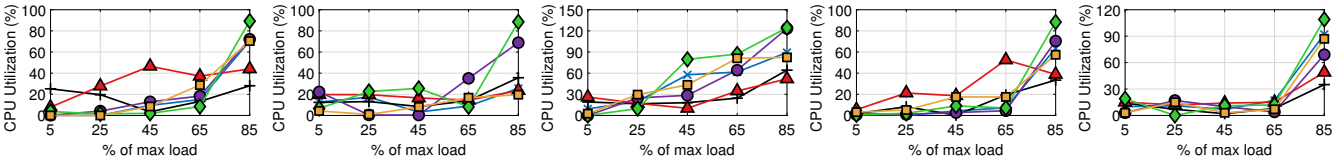
5.2.1 Servpod Analysis. Due to space limitations, we only show one Servpod for each service: Tomcat/E-commerce, Slave/Redis, Zookeeper/Solr, Memcached/Elgg, and Kibana /Elasticsearch. Figures 9-11 show the BE throughput, CPU utilization and memory bandwidth utilization at these Servpods. We see that `Rhythm` 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 colocation 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` 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.

In Figure 9, we see that `Rhythm` 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 `Rhythm` *controller*. For the CPU utilization in Figure 10, when we coschedule 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 utilizes CPU resources at more than 70%, because the training phase of LSTM heavily consumes CPU resources. When they are colocated with other BE jobs, although it cannot achieve the same CPU utilization, `Rhythm` still improves CPU utilization by an average of 7.98%, 11.44%, 27.59%, 8.4%, and 10.44%. For the memory bandwidth utilization in Figure 11, we see that `Rhythm`



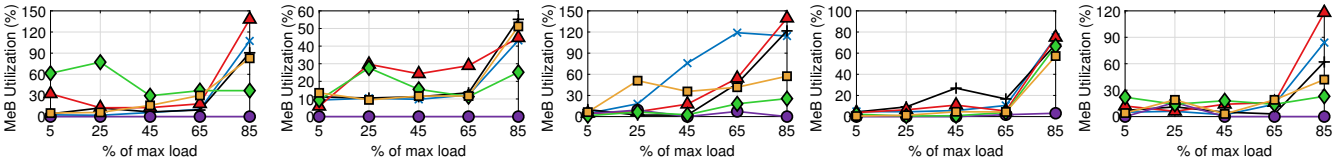
(a) E-commerce **(b)** Redis **(c)** Solr **(d)** Elgg **(e)** Elasticsearch

Figure 12. EMU improvements $((EMU_{Rhythm} - EMU_{Heracles})/EMU_{Heracles})$ under different loads.



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

Figure 13. CPU utilization improvements $((CPU_{Rhythm} - CPU_{Heracles})/CPU_{Heracles})$ under different loads.



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

Figure 14. Memory bandwidth utilization improvements $((MeB_{Rhythm} - MeB_{Heracles})/MeB_{Heracles})$ under different loads.

can drive the utilization up to 82% when coscheduling stream-llc and stream-dram with LC Servpods. CPU-stress does not require much memory bandwidth, so the utilization is quite low. Generally, Rhythm 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 improves the throughput and resource utilization not only when LC is coscheduled 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 on CPU utilization are 25.54% and 29.53%, and improvements on memory bandwidth are 21.03% and 39.13%, respectively.

5.2.2 Overall Performance under Constant Load. We next show the overall improvements in EMU and resource utilization by Rhythm. Figures 12-14 show that Rhythm generates a much higher EMU and resource utilization than *Heracles* in all interference groups. Since both Rhythm and *Heracles* can deploy BE jobs at low load, the improvements generally increase over the load, indicating that Rhythm is more effective when the load towards the LC service is intensive.

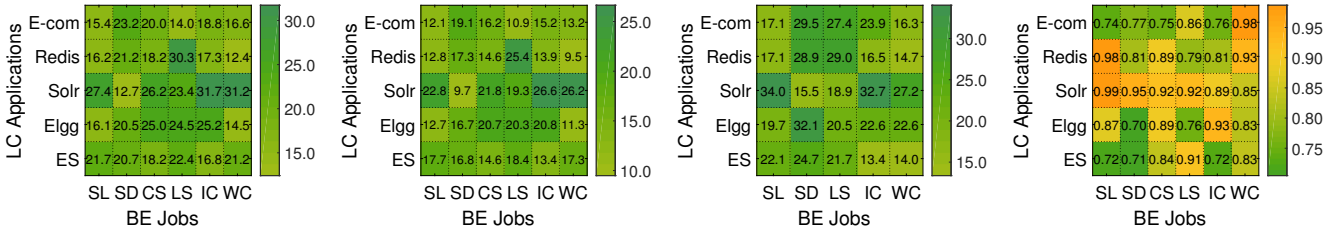
In Figure 12, we see that Rhythm generates 11.6%, 18.4%, 24.6%, 14%, and 12.7% more EMU on average than *Heracles* in E-commerce, Redis, Solr, Elgg, and Elasticsearch, respectively. In particular, when Solr is colocated with ImageClassify and Wordcount, improvements of up to 57% can be achieved because of the significant improvements in Zookeeper. Figure 13 shows the CPU utilization improvements for the five LC services. Rhythm can improve the CPU utilization by 22.2%, 19.1%, 35.3%, 20.6%, and 23% on average compared with that of *Heracles*. Colocating 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 14 shows that Rhythm can improve memory bandwidth utilization by 28.1%, 16.8%, 33.4%, 28.9%, and 19.5% on average compared with that of *Heracles*. Colocating stream-dram (or Wordcount) 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 colocating stream-dram with Elasticsearch.

5.3 Production Load

We also evaluate Rhythm using a production request load from ClarkNet [11] 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.

5.3.1 Overall Performance under Production Load. Figures 15a-15c show the average performance improvements



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

Figure 15. The average performance improvements by Rhythm 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 SLA. (E-com: E-commerce, ES: Elasticsearch, SL: Stream-llc, SD: Stream-dram, CS: CPU-stress, LS: LSTM, IC: ImageClassify, WC: Wordcount.)

compared to those of *Heracles* on EMU, CPU utilization and memory bandwidth utilization under the production load, respectively. We see that Rhythm 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 can achieve an improvement of 26.2% in the Solr-Wordcount group. For memory bandwidth utilization, Rhythm can achieve an improvement of 34% in the Solr-Wordcount group. Generally, while Rhythm 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 15d presents the worst 99*th* percentile latency normalized to the SLA latency of Rhythm 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 can strictly guarantee the SLA in all cases (the worst case is 0.99× SLA). The result shows the effectiveness of Servpod-level control in Rhythm, which can improve throughput without hurting the SLA.

5.3.2 Running with Microservice. Rhythm can be effective in managing processes, containers or microservices. In this section, we evaluate its efficiency using the cloud microservice benchmark: SNMS (social network in microservice), an LC service in DeathStarBench [22]. SNMS contains 30 unique microservices that use RPC for communication. We divide them into three Servpods: *mediaservice* includes 13 microservices related with media data processing, *frontend* includes 3 microservices (nginx-thrift, media-frontend and jaeger) and *userservice* includes 14 microservices related with user operations. We allocate 20 CPU cores and 64 GB memory for each Servpod and deploy them distributedly. SNMS does not need Rhythm's request tracer, because it has a built-in jaeger [31], which is a distributed tracing system like Dapper [65] and can record the sojourn time of each request at each microservice.

Figure 16 shows the overall performance evaluation results. Since the contributions of the three Servpods are 0.295, 0.14

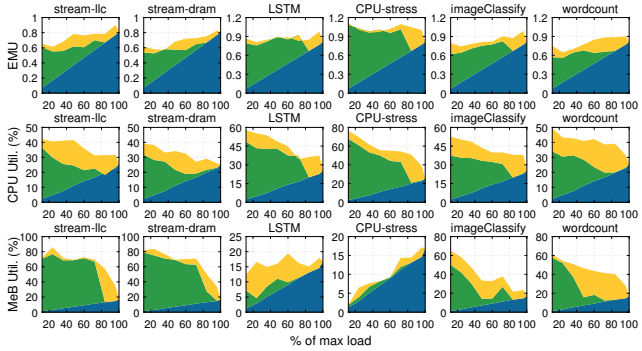


Figure 16. Performance comparison when running with microservice. Improvements are color-coded as follows: ■ represents the EMU or resource utilization of LC itself; ■ represents the improvements achieved by Heracles; ■ represents the further improvements achieved by Rhythm.

and 0.565, respectively, and their *slackLimits* are 0.189, 0.054 and 0.381, respectively, the improvements mainly benefit from the mediaservice 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 Wordcount group because Wordcount performs many computations and IO operations, which affects the tail latency significantly. The CPU-stress group also shows the best improvements in CPU utilization, but the least in memory bandwidth utilization for the same reason as in previous experiments.

5.4 Example of Running Process

5.4.1 Timeline. Figure 17 shows the timeline of Rhythm's running process on two Servpods (Tomcat and MySQL) when they are colocated with Wordcount under the production load. At the beginning, Rhythm allows the growth of the BE workload due to the sufficient slack between the actual latency and SLA target. Thus, BE throughput, BE instances, BE cores, BE LLC and CPU utilization are all continuing to rise. At time 3.3, Rhythm calls the *SuspendBE()* operation because the request load exceeds the *loadlimit*. In this case, although all

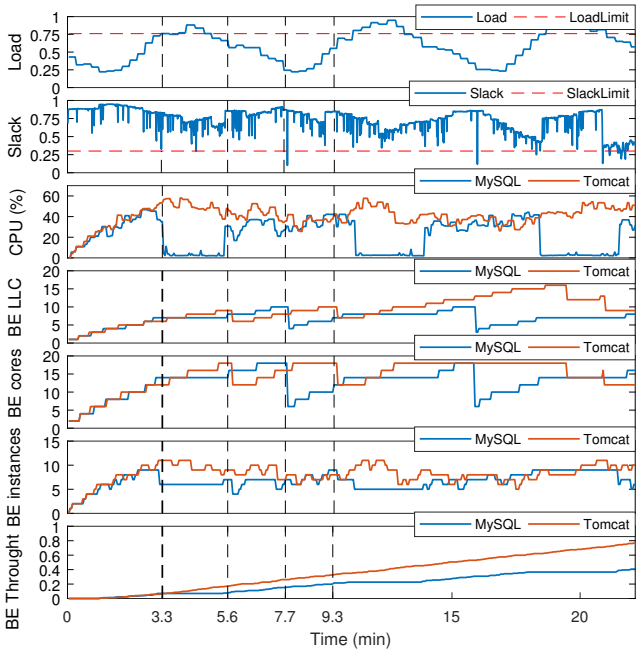


Figure 17. The timeline of Rhythm's running process.

BE jobs still keep their allocated resources, the CPU utilization drops quickly, and the BE throughput does not increase anymore. At time 5.6, since the request load declines to <*loadlimit*, BE jobs return to growth until time 7.7. Then, due to the sudden drop on the slack, `Rhythm` launches the *CutBE()* operation. While the number of BE instances does not change, their LLC and core allocations are cut. In the period of 7.7-9.3, the CPU utilization does not change much because BE jobs are actually not using those cut cores and LLC resources. At time 9.3, `Rhythm` calls the *SuspendBE()* operation again, and the whole process is repeated.

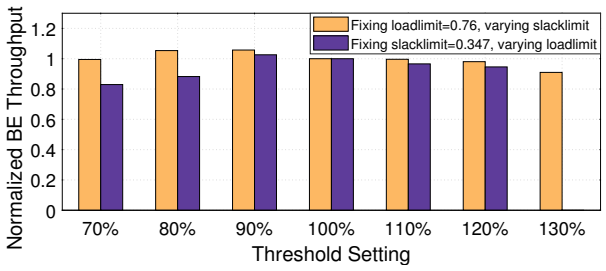


Figure 18. Trade-off between loadlimit (slacklimit) and BE throughput.

5.4.2 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 18 shows how the BE throughput varies over the *loadlimit* and *slacklimit*. We see that the BE througput peaks when the *loadlimit* is at the 90% level (i.e., 90% of the actual derived values.). In case of "fixing loadlimit, varying slacklimit", the BE throughput at level 80% and 90% are both higher than the 100% level. However, Table 2 shows that 90% of *slacklimit* also causes 13 SLA violations and kills 3 BE jobs in the period. For *loadlimit*, the number of SLA violations and BE kills at 90% level are the same as those at 100%, indicating that 90% level is a better choice. Hence, while our derivation of *slacklimit* and *loadlimit* works well in experiments, they can be further improved in the future.

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

	Fixed loadlimit = 0.76			Fixed Slacklimit=0.347		
Level	Slacklimit	SLA Violation	BE kills	Loadlimit	SLA 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	-	-	-

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* [2, 5, 10, 20, 34, 63, 65, 70, 74, 86] 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* [3, 4, 7, 9] 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* [5, 20, 34, 65, 70, 86] or the *non-intrusive method* [2, 10, 30, 42, 63, 74]. As the *intrusive method* causes high instrumentation cost, we simply use the easy-to-use systemTap [30] 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 [6, 28, 64], including hardware heterogeneity [55], virtualization [72], or the contention on various resources [35, 59, 66]. In particular, contention on cache [12, 25, 47, 68] and I/O [66] are two main sources for performance interference. These contentions are not only from the same core [78, 80] but also possibly from cross-cores [82, 83]. However, these works mainly focus on the evaluation of the overall performance of an application, e.g., the latency of a web application [59], a multimedia service [6], or the execution time of a bigdata analysis job [17]. 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 [49] and Bubble-Flux [77] predict the impact of interference from potential corunners through the instantaneous pressure generated by a dynamic bubble. DeepDive [53] infers performance loss due to interference by clustering low-level metrics. Dirigent [84] and Wrangler [76] supports to control QoS based on the prediction of execution time. Stay-away [60] throttles the batch jobs to avoid contention by predicting any progression towards a QoS violation at runtime. SMiTe [81] achieves precise interference prediction on real-system architectures. Quasar [16] and Paragon [15] use classification techniques to quickly estimate the impact of interference, and improve resource utilization while guaranteeing QoS. Pythia [75] predicts the combined contention of multiple applications using a simple linear regression model to improve utilization.

Other related work like Ubik [37], CQoS [29], CPI2 [79], iAware [73] and [32] present how to guarantee the QoS with cache or memory bandwidth isolation mechanisms. For core isolation, PerfIso [27] colocates batch jobs with production latency-sensitive services using CPU blind isolation to protect SLA from burst workloads. Retro [44] present 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 SLA violation is detected. ICE [45] 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 [51] uses online feedback to capture interference, and tunes resource allocations to mitigate performance interference effects. Heracles [41] enables the safe colocation of BE jobs and LC service through a conservative thresholding method. PARTIES [8] and CLITE [58] further increases the number of co-scheduled LC services per server to improve the throughput. CoPart [57] analyzes the characteristics of workloads and allocates the LLC and memory bandwidth for BE jobs to improve fairness. Twig [52] employs a deep reinforcement learning model for improving the energy-efficiency of colocated latency-critical services. 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 [27].

GrandSLAm [36] 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 [36]: (1) GrandSLAm co-locates multiple LCs, while Rhythm co-locates a 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 SLA, while Rhythm considers the tail latency, which is a statistical result over all latencies. Hence, GrandSLAm is orthogonal to Rhythm.

7 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. 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 the 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

Acknowledgment

We thank our shepherd Christos Kozyrakis and anonymous Eurosys reviewers for their insightful feedback. This work is supported by the National Key Research and Development Program of China No. 2016YFB1000205; the National Natural Science Foundation of China under grant 61872265, 61702480, 61672499; the new Generation of Artificial Intelligence Science and Technology Major Project of Tianjin under grant 18ZXZNGX00190, 19ZXZNGX00010; and YIPA of CAS (2013073).

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry

Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martion Wicke, Yuan Yu, and Zheng Xiaoqiang. 2016. Tensorflow: a system for large-scale machine learning.. In *OSDI*, Vol. 16. 265–283.

[2] 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. https://doi.org/10.1109/DSN.2007.38

[3] 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 (SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 74–89.

[4] 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.

[5] 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.

[6] 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 (MMSys '10)*. ACM, New York, NY, USA, 35–46.

[7] 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. https://doi.org/10.1109/TSC.2016.2607739

[8] 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 (ASPLOS '19)*. ACM, New York, NY, USA, 107–120.

[9] 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 (OSDI'08)*. USENIX Association, USA, 117–130.

[10] 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. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chow

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

[12] 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.

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

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

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

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

[17] Christina Delimitrou and Christos Kozyrakis. 2016. HCloud: Resource-Efficient Provisioning in Shared Cloud Systems. *SIGPLAN Not.* 51, 4 (March 2016), 473–488.

[18] Elasticsearch. 2019. Elasticsearch: a search engine based on the Lucene library. https://lucene.apache.org/solr/.

[19] 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.

[20] 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. https://www.usenix.org/conference/nsdi-07/x-trace-pervasive-network-tracing-framework

[21] Yu Gan and Christina Delimitrou. 2018. The Architectural Implications of Cloud Microservices. *IEEE Computer Architecture Letters* 17, 2 (July 2018), 155–158.

[22] 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 (ASPLOS '19)*. ACM, New York, NY, USA, 3–18.

[23] Wanling Gao, Lei Wang, Jianfeng Zhan, Chunjie Luo, Daoyi Zheng, Zhen Jia, Biwei Xie, Chen Zheng, Qiang Yang, and Haibin Wang. 2017. A Dwarf-based Scalable Big Data Benchmarking Methodology. *CoRR* abs/1711.03229 (2017).

[24] 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.

[25] 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 (SOCC '11)*. ACM, New York, NY, USA, 22:1–22:14.

[26] 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 (IWQoS '19)*. ACM, New York, NY, USA, Article 39, 10 pages.

[27] 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.

[28] 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.

[29] Ravi R. Iyer. 2004. CQoS: a framework for enabling QoS in shared caches of CMP platforms. In *International Conference on Supercomputing*. 257–266.

[30] 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).
[31] jaeger. 2019. https://www.jaegertracing.io/.
[32] 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.
[33] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *CoRR* abs/1902.03383 (2019).
[34] 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.
[35] 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.
[36] 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 (EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article Article 34, 16 pages.
[37] 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.
[38] 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.
[39] Kubernetes. 2019. https://kubernetes.io/.
[40] 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 (SoCC '18)*. ACM, New York, NY, USA, 347–360.
[41] 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.
[42] LTTng. 2019. https://lttng.org/.
[43] 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.
[44] 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.
[45] 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.
[46] 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.
[47] 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.
[48] 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.
[49] 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 (MICRO-44)*. ACM, New York, NY, USA, 248–259.
[50] D. A. Menasce. 2002. TPC-W: A Benchmark for E-Commerce. *IEEE Internet Computing* 6 (05 2002), 83–87.
[51] 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.
[52] 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.
[53] 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 (USENIX ATC'13)*. USENIX Association, Berkeley, CA, USA, 219–230.
[54] Numactl. 2019. https://github.com/numactl/numactl.
[55] 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 (HotCloud'12)*. USENIX Association, Berkeley, CA, USA, 4–4.
[56] 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.
[57] 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 (EuroSys '19)*. ACM, New York, NY, USA, Article 10, 16 pages.
[58] 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.
[59] 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.
[60] 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 (Middleware '14)*. ACM, New York, NY, USA, 301–312.
[61] Redis. 2019. Redis: an open source, in-memory data structure store. https://redis.io.
[62] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. 2012. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *Proceedings of the Third*

ACM Symposium on Cloud Computing (SoCC '12). ACM, New York, NY, USA, 7:1–7:13.

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] Solr. 2019. Solr is the popular, blazing-fast, open source enterprise search platform built on Apache Lucene. https://www.elastic.co.

[68] 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.

[69] 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.

[70] 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. https://doi.org/10.1145/1140103.1140280

[71] 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).

[72] 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 (INFOCOM'10)*. IEEE Press, Piscataway, NJ, USA, 1163–1171.

[73] 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.

[74] 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. https://doi.org/10.1109/ICAC.2016.38

[75] 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 (Middleware '18)*. ACM, New York, NY, USA, 146–160.

[76] 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.

[77] 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.

[78] 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.

[79] 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.

[80] 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.

[81] 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.

[82] 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.

[83] 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 (PACT '13)*. IEEE Press, Piscataway, NJ, USA, 201–212.

[84] 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.

[85] 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.

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