IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received December 28, 2018, accepted January 29, 2019, date of publication February 21, 2019, date of current version March 25, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2900451

PINE: Optimizing Performance Isolation in Container Environments

YOUHUIZI LI[1,2,3], (Member, IEEE), JIANCHENG ZHANG[1,2], CONGFENG JIANG[1,2], (Member, IEEE), JIAN WAN[1,4], AND ZUJIE REN[1,5]

[1] 1Key Laboratory of Complex Systems Modeling and Simulation, Ministry of Education, Hangzhou Dianzi University, Hangzhou 310018, China
[2] School of Computer Science and Technology, Hangzhou Dianzi University, Hangzhou 310018, China
[3] Xi'an Key Laboratory of Mobile Edge Computing and Security, Xidian University, Xi'an 710071, China
[4] School of Information and Electronic Engineering, Zhejiang University of Science and Technology, Hangzhou 310023, China
[5] Zhejiang Lab, Hangzhou 311121, China

Corresponding author: Jian Wan (wanjian@zust.edu.cn)

This work was supported in part by the Key Research and Development Program of Zhejiang Province under Grant 2018C01098, in part by the Natural Science Foundation of Zhejiang Province under Grant LY18F020014, Grant LQ18F020003, in part by the Natural Science Foundation of China under Grant 61802093, Grant 61472109, and Grant 61572163, and in part by the Xi'an Key Laboratory of Mobile Edge Computing and Security under Grant 201805052-ZD3CG36.

ABSTRACT With the development of virtualization technologies, containers are widely used to provide a light-weight isolated runtime environment. Compared with virtual machines, containers can achieve high-resource utilization and provide a more convenient way of sharing, but there are significant security challenges due to potential resource contention among services. When the regular services that share the storage system with the key services are over-used, a lot of resources may be preempted, which breakdowns the whole system and delays other services. This paper addressed the performance isolation in fierce resource competition situations, that is, when different services (latency-sensitive services and throughput-first services) are deployed on a host machine using container technology. We proposed PINE, a performance isolation optimization solution in container environments, which can adaptively allocate the storage resource for each service according to their performance behaviors through dynamical resource management and I/O concurrency configuration. The experimental results show that PINE can effectively optimize the performance of the running services and achieve the optimal result in a relatively short time.

INDEX TERMS Container, performance isolation, shared storage.

I. INTRODUCTION

The development of virtualization technologies, like VMs and container, has greatly improved the resource utilization of physical machines. An increasing number of systems are deployed in container environments to save costs and facilitate data sharing between different applications [1]. As an emerging virtualization technology, the container is more lightweight and has a higher resource utilization than virtual machine [7], [8], [26], which makes it perfect for resource limited devices, like mobile devices and gateways.

However, deploying different services in a cluster or even a single physical machine can also cause significant security problems [31], [32]. Rigorous security methods are often imposed on critical services to ensure common operations, while only a few attentions were put to non-critical services. When critical services and non-critical services are deployed together on a single machine through virtualization technologies like containers, all security methods may face major challenges. Non-critical services may be maliciously used or misused by individuals or groups because they have less protection. When non-critical services are overloaded, the native isolation mechanism of container, for example, the Linux kernel feature Cgroups, cannot guarantee the reasonable allocation of resources, the non-critical service may occupy all the resources which should be shared with the critical services. Moreover, in some situations, the resource isolation mechanisms lead to an uneven resources allocation, part or even all services fail to meet their requirements [3]. As a result, the performance of critical services and the entire system will be greatly influenced.

To address the issue of misallocation of resources in a shared storage system, we propose that isolation optimization methods can be used to mitigate resource contentions and guarantee the resources of critical services will not be maliciously occupied.

The associate editor coordinating the review of this manuscript and approving it for publication was Gaurav Somani.

Isolation mechanisms are commonly applied in performance optimization field. Different services have different performance requirements. Tail latency, like 99.9th percentile latency, is one of the most important performance requirements for latency-sensitive applications. A request from nowadays applications can be divided into many parallel sub-requests [11], and the response time for the request is decided by the slowest sub-request, so 99.9th percentile latency becomes important for latency-sensitive applications like web service or Database Service. Another important performance requirement is the throughput, which is preferred in batch-processing applications like Hadoop clusters [2]. This type of services generate large-volume data operations and do not care about the tail latency.

The existing optimization methods mainly focus on the situations that shared storage services have the same type of performance requirements. For the 99.9th percentile latency requirement, the commonly applied optimization approaches are redundancy and rate limiting technologies [3]–[5], [10], [12], [13]. For the throughput requirement, the optimization methods mainly include space reservation and IO throttling [2], [15], [16], [19], [21], [23]. The above approaches aim at situations where services have the single performance requirement (99.9th percentile latency or throughput) and do not differentiate services based on their own specific performance requirements. The single-dimensional solutions cannot work for other types except the objective service type. Another optimization approach [9] can handle the services that have both 99.9th percentile latency and throughput requirements. It also treats all services as the same type, but the services are constrained by two-dimensional performance indicator. However, tail latency and throughput of a service is a conflict when the resource is limited. Higher throughput needs larger concurrency, while a larger concurrency value may result in a longer tail latency. In some situations, the optimization approach will completely abandon one indicator [9]. Hence, the services that contain a two-dimensional indicator (both tail latency and throughput) are not considered in our system. We focus on the hybrid deployment of multiple services, and the requirement for one service is either latency or throughput.

In this paper, we proposed a performance isolation optimization strategy PINE for services which share a storage system. When a service is overloaded and a large number of resources are consumed, performance optimization approaches can be quickly applied to ensure that each service in the system can run normally as usual. The main idea behind PINE is to distinguish services according to their performance requirements and to optimize them accordingly. For the throughput-first services, PINE ensures throughput by allocating disk resources; and for 99.9th percentile latency-sensitive services, PINE will adjust their IO concurrency levels [27] to catch the latency deadline. To the best of our knowledge, PINE is the first solution that differentiates services and performs corresponding optimization approaches to achieve services' own performance requirements and maximize the system's disk resource utilization.

The rest of this paper is organized as follows. Section II describes the motivations and reviews the related work in the performance optimization field. Section III introduces the design of PINE, from scenario illustration, available optimization mechanism evaluation to detailed module implementation. Section IV presents the evaluation of the proposed system and shows comparison results of PINE and previous strategies. Section V summarizes the paper and describes the future work.

II. MOTIVATION AND RELATED WORK

This section first presents a series of experimental observation results which illustrate the disadvantages of the current container isolation mechanism and show the necessity and significance of this work. The results shed light on the proposed optimization ideas. Then, some existing work related to performance optimization are introduced.

A. MOTIVATION

Enterprises normally deploy multiple services on a single physical machine using container technology to improve the resource utilization. Currently, hybrid deployment of latency-sensitive service and throughput-first service is a popular deployment strategy. By leveraging the complementary workload characteristics of the two types of services, cloud providers can greatly improve infrastructure utilization and increase profit. These services which share storage resources may experience resource contention, especially when the storage resources are tight. As a result, some services (even all of them) may not meet their performance requirements.

In order to illustrate how the performance of each service will be affected when storage resource contention is intense, we set up the observation experiments in which two types of services were deployed on a single physical machine. For latency-sensitive service, users expect to immediately get the feedback after making a request. The typical services include social network applications, online map service, search engine service, real-time translation service, and so on. These latency-sensitive services are all related to database services, and the performance of database service will become one of the serious bottlenecks in high concurrency situations. Therefore, we directly select the database application, MySQL, to represent a typical 99.9th percentile latency-sensitive service in the paper. For throughput-first service, a Hadoop MapReduce application is used. The main purpose of hybrid deployment is to make full utilization of the idle time and potential resources when the workload of latency-sensitive services is low. So the throughput-first services should have good interrupt handling capability. Tasks can be postponed when the resources are intense. If the resources are sufficient, the previous tasks can also be correctly continued. MapReduce service can appropriately

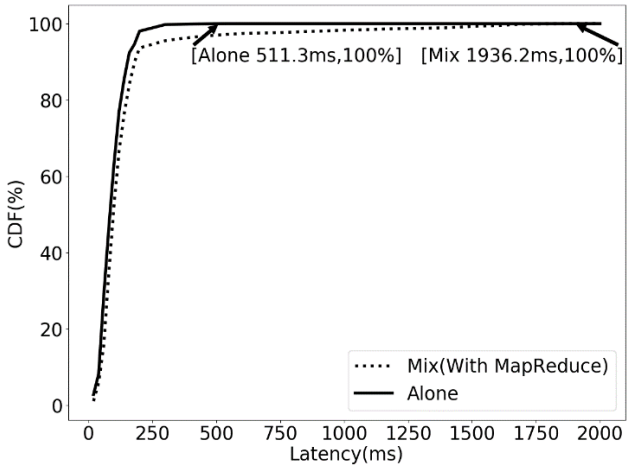


FIGURE 1. Comparison of the MySQL latency distributions.

represent the throughput-first service, other services situations should be similar.

A comparison of the MySQL latency distributions of the two cases (with and without MapReduce) is demonstrated in Figure 1. MapReduce's throughput is about 100MB/s and the server's disk IO occupancy rate is close to 100%, which indicates that the competition of disk resource is fierce between MySQL and MapReduce. The experimental results show that MySQL's tail latency is stretched with MapReduce. MapReduce service competes against the MySQL service for the storage resources when they are running at the same time, hence, the MySQL tail latency rises.

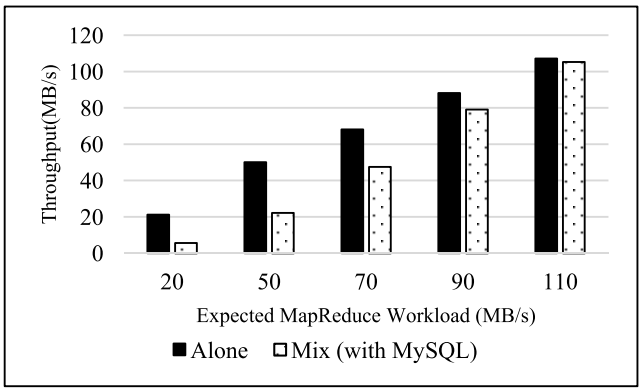


FIGURE 2. Comparison of MapReduce throughput.

Figure 2 presents the throughput comparison in the two cases. With the increasing load of MapReduce (from left to right), the throughput difference in the two cases gets smaller and smaller (the load of MySQL is constant). The reason is that when the MapReduce load is heavy, it has an advantage in the competition of disk resources, and even the MySQL will be rolled. Due to the uneven resource allocation, there was a very high tail latency in the MySQL service.

The two experiments show that the services which share the storage resources have a great potential interfere with each other. Therefore, it is necessary to apply isolation mechanisms to optimize the performance of services when they are deployed together.

The most common isolation method is to reserving disk resources (bandwidth) for different services. It is helpful to handle the latency increase due to the service request staying caused by insufficient storage resources. To evaluate the disk reserving method, we ran the MySQL and MapReduce together, and controlled the bandwidth of MySQL by Cgroups blkio. The Expected Latency is set to the latency of MySQL running alone. Figure 3 illustrates the distribution of the 99.9th percentile latency of MySQL under different disk bandwidth.

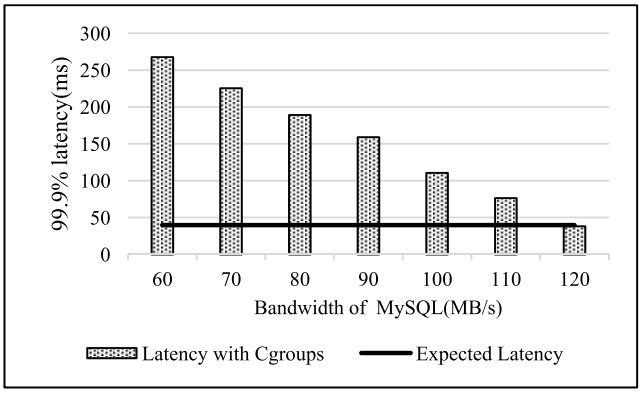


FIGURE 3. MySQL latency distribution under different disk bandwidth.

With the increase of disk resources, the 99.9th percentile latency of MySQL is decreasing. When the disk resources of MySQL reaches 120MB/s, the latency is close to the expected value. However, the total disk resources of the machine used in this experiment is also about 120MB/s, which means that we allocated almost all disk resources to MySQL to guarantee its latency. It will undoubtedly destroy the operation of the MapReduce in the same machine. In addition, this method may fail when the expected latency specified by users is more strict, that is, even if all disk resources are allocated to latency-sensitive service, the requirement may still not meet. Moreover, this method is inefficient and slow, because it only focuses on resource allocation without considering the service characteristics.

Therefore, it is necessary to design a novel efficient optimization approach which not only ensures the latency-sensitive services to reach their requirements but also guarantees the performance of throughput-first services.

B. RELATED WORK

Based on the objectives, the existing optimization strategies can be classified as the following three types: for 99.9th percentile latency, for throughput, and for both 99.9th percentile latency and throughput. Noted that we classify services into the 99.9th percentile latency type and the throughput type based on whether the service requires the real-time response.

Regarding 99.9th percentile latency, there are mainly three optimization strategies. The first one modifies the Linux kernel settings. Li *et al.* [6] from the University of Washington discussed the reasons for the long tail latency and proposed setting the priority of the applications to eliminate the influences of processes interference. Besides, they concluded that FIFO (First In First Out) was the best or the most suitable scheduling strategy for 99.9th percentile latency, but it led to a higher median delay. The second strategy is sending redundant requests [5], [20], [22]. Dean and Barroso [5] from Google investigated the long tail latency when multiple loads were deployed on a single physical machine. They believed that the long tail latency was caused by the sudden simultaneous emerging of a large number of requests, and thus the competition for resources became fierce. They proposed using redundant requests to optimize the long tail latency. Repeated requests were sent to different nodes, and they would take the fastest arrived response as the final result, thereby shortening the 99.9th percentile latency. This method can achieve the latency goal but the system resources are abused. It is generally used in web services, like C3 [10], Bobtail [14] as well as predictive parallelization [12]. The third strategy optimizes the 99.9th percentile latency by leveraging multi-resource scheduling. Zhu *et al.* [13] designed a detailed 99.9th percentile latency model for a cluster system to optimize the latency. Wang *et al.* [3] designed a multi-layer Cake framework that spans HBase and HDFS to coordinate various resource. Zhang*et al.* [4], [24] designed an interposed 2-Level IO scheduling framework for performance virtualization.

For throughput optimization, researches focus on achieving the throughput requirement of each service. The mClock is a throughput optimization strategy proposed by Gulati *et al.* [2]. It defines the upper and lower limits of the disk resources available to each service based on their QoS. The allocated disk resources for each service is calculated as the ratio of the QoS required by all services. The disk allocation for a service will be added to the lower limit if it is less than the preset lower limit. Similarly, it will be restricted if it is greater than the preset upper limit. Thus the mClock can bring the services up to their throughput requirement and services will not illegally occupy disk resources of other services. Jin *et al.* [16] from Duke University propose an approach to proportional share resource control between services by interposed request scheduling. This approach allocates disk resources reasonably by controlling the number of requests entering the service from the perspective of network traffic. And they propose three proportional share scheduling algorithms for this approach. Normally, the fairness of disk resource allocation can be guaranteed by these algorithms. However, they are unable to provide appropriate emergency measures if there is a sudden demand for disk resource from a high priority service. Another solution is Request Windows which is a share-based reservation scheme. It configures network traffic limits for each service based on the service weight. Zhang *et. al.* proposed the SARC [24], a new 2-level framework, to ensure the performance requirements of services. It can allocate the disk resource reasonably through the IO throttling technology. Many strategies [15], [19], [21] have similar ideas and allocate disk resources based on the ratio of the performance requirements.

Asides from optimizing one type of performance requirements, there are also strategies that take both latency and throughput into consideration. The PSLO framework proposed by Li *et. al*. integrates an X^{th} percentile latency and throughput of each application into the shared storage system [9], which optimizes the services' performance through I/O rate control and I/O concurrency management. The strategy emphasizes to consider both 99.9th percentile latency and throughput of a service. While there is a trade-off between the two performance indicators, in some situations, PSLO will completely abandon one requirement and only focuses on another. It is not an appropriate optimization strategy in the fierce resources competition scenario, we should guarantee the requirements are satisfied as much as possible.

We have introduced three types of performance optimization schemes. The first two only focus on one performance indicator. The assumption is that the services in the system have the same performance indicator, which is too ideal for the real environment. The third scheme extends the indicator of service and considers that the services in the system have both performance indicators of throughput and latency. In essence, it still only works for single type of service which contains two type of performance requirements. In other words, it still cannot handle multiple types of services. Comparing with previous solutions, we propose PINE, an adaptive 99.9th percentile latency and throughput optimization strategy. PINE believes that each service is different and has its unique performance requirement, such as 99.9th percentile latency or throughput, and it can apply different optimizations to different types of services. When system resources are limited, PINE still can provide appropriate disk resources to each service according to its performance requirements. Specifically, PINE improves the 99.9th percentile latency of each service by modifying the IO concurrency level and restricting the number of outstanding requests.

III. DESIGN AND IMPLEMENTATION

In this section, we first illustrate the targeted optimization scenario, then validate the effectiveness of the adopted control methods, finally, the overall design and implementation of PINE are introduced.

A. EFFECTIVENESS OF THE OPTIMIZATION METHODS

A series of experiments were conducted to validate the effectiveness of the performance control methods. As mentioned earlier, the services are classified into 99.9th percentile latency-sensitive services and throughput-first services according to their performance requirements. Hence, PINE uses the CRUD of MySQL to simulate a typical 99.9th percentile latency-sensitive service and MapReduce with a Hadoop cluster to simulate a typical throughput-first service.

The benchmarks we used is the sysbench [17] for MySQL and TestDFSIO [33] for MapReduce. Both are classic benchmark tools, so the results can represent the real production environment data. Since Docker configures a virtual disk device block for each container and one service run in one container, we can get the throughput of each service by using the IOSTAT tool which collects performance data at virtual disk device block level. The 99.9th percentile latency was calculated based on the 99.9th percentile data of the sorting list which is composed of all requests' latency within a fixed period.

The server is x86_64 CPU(s) architecture, 16 AMD Opteron^{tm} Processor 6136 is the host for the application services, and Docker container is used as the virtualization technology. The Docker version is 1.17-ce and the MySQL version is 5.7. The Hadoop cluster is composed of one master node and two slave nodes.

1) SCENARIO DESIGN

To ensure the effectiveness of the performance optimization methods, this section defines the basic scenario and requirements. The services run in the cluster contained 99.9th percentile latency-sensitive services and throughput-first services. The most important purpose of sharing the same storage system with different services is to improve storage utilization.

In a real production environment, single service tends to focus on just one type of performance data. Take the Hadoop MapReduce for example, it pays more attention to throughput. For web service applications, the only concern is whether the 99.9th percentile latency of the request is satisfied because it is directly related to the user experience. Hence, we chose to deploy MySQL and Hadoop's MapReduce in the system, where MySQL focuses on the 99.9th percentile latency and MapReduce focuses on the throughput. Due to the contradiction between the two performance requirements (latency and throughput), the optimization strategies are more conservative when balancing them. As a result, the storage utilization improving space is limited, even no way to improve in the case of fierce storage resource competition. It is contrary to the original goal of the hybrid deployment. Hence, we only consider one-dimension performance requirements for one service, and there can be multiple services running in the system. Performance optimization strategies are provided separately depending on the type of performance requirements with the aiming of improving the system storage utilization as much as possible.

2) EFFECTIVENESS ON 99.9% LATENCY

PINE optimizes the 99.9th percentile latency by configuring the I/O concurrency level of the service. It can change the size of the tail delay more flexible than using disk allocation in the Cake, and the optimal results also can be achieved faster.

Since performance requirements need to be set according to specific scenarios, and different users may have different requirements for the same service, we take each service running alone on the host machine as the baseline case in the subsequent experiments.

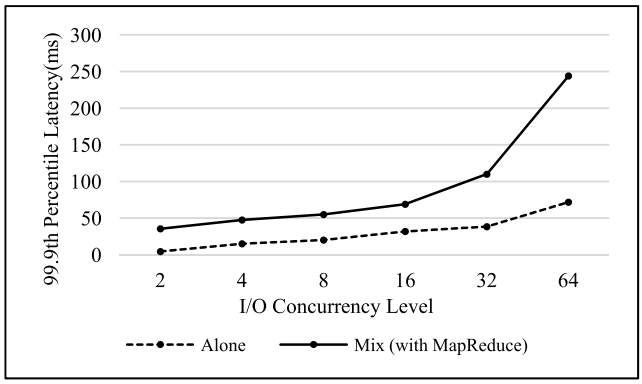


FIGURE 4. The distribution of 99.9th percentile latency at different I/O concurrency levels.

We analyzed the 99.9th percentile latency of MySQL under different I/O concurrency level with constant MapReduce load. Figure 4 shows that in the mix case (with MapReduce), the overall 99.9th percentile latency of MySQL was higher than running alone. The hybrid deployment indeed may cause performance downgrade if no optimization methods were applied. Besides, The 99.9th percentile latency increased 31.07ms when the IO concurrency level was 2, and the latency increased 172.17ms when the level was 64. The higher the IO concurrency level, the greater the magnitude of the increment in MySQL's 99.9th percentile latency. Hence, modifying the I/O concurrency level is an effective way to control the 99.9th percentile latency.

3) EFFECTIVENESS ON THROUGHPUT

Throughput is a more intuitive performance requirement compared to the 99.9th percentile latency. The host creates a virtual disk for each service, so the real-time throughput of individual service can be captured accurately. Based on the performance requirements of the service, PINE controls the throughput by enforcing allocation of disk resources through Cgroups.

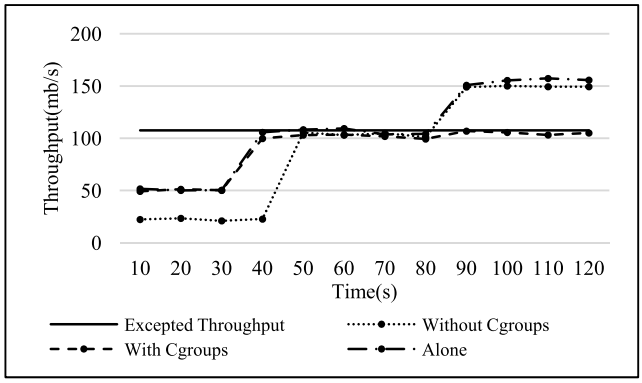


FIGURE 5. MapReduce's throughput distribution with varies load of MySQL.

In Figure 5, the MapReduce load was kept constant and the MySQL load was increased gradually. We compared

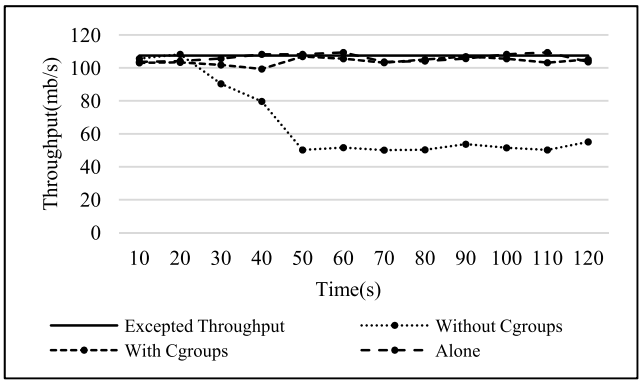


FIGURE 6. The throughput of MapReduce under different loads.

the MapReduce throughput variation in the with and without Cgroups cases. With the load increasing of MySQL, the storage resources is occupied by MySQL, which leads to a low throughput of MapReduce in the Without Cgroups case. On the contrary, in the With Cgroups case, MapReduce throughput was not affected and the data was around the expected value. In Figure 6, the roles of MySQL and MapReduce were interchanged, the MapReduce load was increased gradually and the MySQL load was kept constant. With the increasing of MapReduce load, its throughput also increased even after reaching the expected throughput value. However, when we used Cgroups to restrict the storage resource of MapReduce, the throughput remained close to the requirement and the heavy load of MapReduce did not arbitrarily preempt other services.

Hence, although different services will influence each other when competing the shared storage resources, Cgroups can ensure the throughput-first services to reach their own performance requirements.

B. THE ARCHITECTURE OF PINE

PINE is designed for the scenario that different services are running on a single physical machine. In order to ensure scalability and portability, PINE treats the entire scenario as a black box and shields the underlying complex storage performance analysis.

The idea of PINE is to implement different optimization methods for each service. Based on the performance requirements and current compliance status, the optimization approaches can be divided into two categories: throughput optimization and 99.9th percentile latency optimization. The specific algorithm can be described as follows.

The two branches in the loop provide different optimization approaches for services based on their requirements, and the two-layer optimization, throughput optimization and the 99.9th percentile latency optimization, are organized in a hierarchical way.

According to the functionality and performance requirements of PINE, we designed three modules. The first

Algorithm 1 Optimization Algorithm

Input: real-time performance data: ***99.9Lat*** and ***Thr***
99.9Lat is latency-sensitive service's 99.9th percentile Latency
Thr is throughput-first service's Throughput
Output: IO concurrency and bandwidth allocation for next cycle

```
For each service S_i in S
While get 99.9Lat and Thr
  If Thr<ThrSLO then        //Throughput not met
    Give (ThrSLO-Thr) of Storage resource to Throughput-first service
    else if 99.9Lat>99.9LatSLO //99.9%Latency not met
  //Adjust Latency-sensitive service's IO Concurrency level
  AdjustIOConLevel()
  End if
End for
Function AdjustIOConLevel()
  Target←min(IOCLevel * (99.9LatSLO/99.9Lat),IOCLevel-1)
  Result←max(Target,1)
  Adjust Latency-sensitive service's IO Concurrency level to Result
```

module is the performance monitoring module, which monitors the performance of each service periodically and stores the data in Redis database that will be used by the optimization decision module. Besides, the module also records the optimization parameters in each cycle. The second module is the optimization decision module which leverages the algorithms (e.g. AdjustIOConLevel) and the collected real-time performance data to calculate the optimization parameters that will be used in the implementation module in the next cycle, it also ensures that the performance of each service is close to its requirements. The third module is the optimization implementation module, which contains the throughput optimization layer and the 99.9th percentile latency optimization layer. The throughput optimization layer is implemented on the operating system level, while the 99.9th percentile latency optimization layer is deployed on each service separately so that they can meet their latency requirements.

The overall framework is shown in Figure 7. After obtaining the data of the monitoring module, PINE will make two decisions. First, it determines whether the throughput reaches the required value. If not, the throughput optimization process will be executed. Then PINE checks whether the 99.9% latency meets the standard. If not, the 99.9% latency optimization process will be executed. The whole process is repeated for each service to make sure all of them are normal.

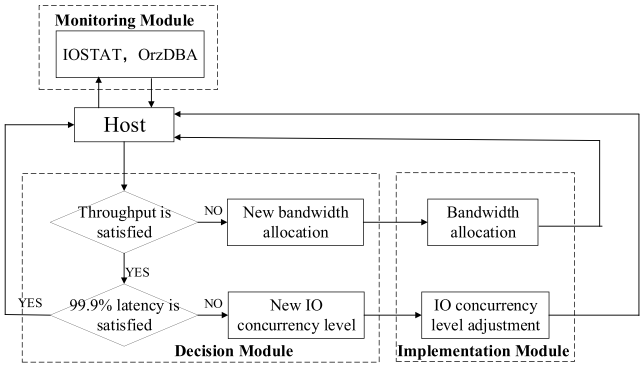


FIGURE 7. The overall flow of PINE.

C. IMPLEMENTATION

IN this subsection, we introduced the implementation details of the three modules of PINE.

1) MONITOR MODULE

Gathering the real-time performance data of each service is a prerequisite for the entire optimization process because the overall idea of PINE is to optimize different services based on their requirements and apply a negative feedback modification. The performance monitoring module periodically collects the real-time performance data, mainly including 99.9th percentile latency data and throughput data, in multiple ways. The collection cycle was set to 10s since negative feedback adjustment needs some time to take effect.

The data collection for throughput is relatively straightforward, Docker allocates a disk device block for each service respectively. PINE uses IOSTAT to monitor the IOPS of each disk device block, and then calculates the throughput of each service by IOPS. The collection cycle for throughput was also set to 10s to keep the consistency with the 99.9th percentile latency.

The 99.9th percentile latency indicates the maximum latency tolerance rate, it aims to restrict the request response time and ensure stable performance. The performance monitoring module needs to collect the data of the 99.9th percentile latency-sensitive services periodically, and it records the latency of all requests within one second. After sorting the latency data in an ascending order, we take the data in the 99.9th percentile position as the latency result for the service.

2) DECISION MODULE

The decision module is used to provide optimization parameters for the optimization implementation module in the next cycle, so the performance of each service can be guaranteed to reach or approximate the specified requirements. With the collected throughput or 99.9th percentile latency data, the module can determine if the service's performance is up to par by comparing with the corresponding requirements. In the case of a performance data violation, the associated algorithm is executed to determine the optimization parameters in the next cycle according to the type of the service's performance requirements.

Throughput can be directly controlled by disk IO bandwidth allocation. In the case of limited storage resources, the total throughput of all services remains stable and the decrease in the throughput of one service will be preempted by another service immediately. Therefore, the bandwidth allocation $\mathrm{Th}_{\mathrm{new}}(\mathrm{k}, \mathrm{t}+1)$ of service k in the next cycle can be obtained based on the following equation:

$$Th_{new}(k, t+1) = \begin{cases} ThS(k)\ Th(k,t) < ThS(k) \\ Th(k,t)\ Th(k,t) \geq ThS(k) \end{cases} \quad (1)$$

The Th (k, t) is the real-time throughput of the service k (k is a throughput-first service), and ThS (k) is the throughput requirement of the service k.

The bandwidth allocated to the throughput-first service is not only the lower limit of its available bandwidth value but also the upper limit of its bandwidth acquisition. In this way, PINE can prevent it from occupying excessive bandwidth and guarantee the storage resources of other services.

According to the experiment in Section III, adjusting the I/O concurrency level can effectively control the 99.9th percentile latency of the service. However, unlike the throughput, there is no obvious mathematical relationship between 99.9th percentile latency and the I/O concurrency level. The 99.9th percentile latency is obtained using statistical method, the module cannot directly calculate the optimal I/O concurrency level with the latency requirement. For the portability of PINE, decision module iteratively calls the AdjustIOConLevel algorithm to obtain a stable optimal I/O concurrency data after several cycles. When the 99.9th percentile latency is higher than the expected value, the I/O concurrency level of the service is modified, and the magnitude of adjustment (i.e. the iterative step size) is set dynamically as the ratio of 99.9th percentile latency observed in real time to the expected 99.9th percentile latency. Then the IO concurrency level $\mathrm{P}_{\mathrm{new}}$ (l, t + 1) for the next cycle can be calculated. The specific equations are as follows:

$$P(l, t+1) = Min(P(l,t) * \frac{TlS(l)}{Tl(l,t)},\quad P(l,t) - 1) \quad (2)$$

$$P_{new}(l, t+1) = \begin{cases} Max(P(l, t+1), 1) & Tl(l,t) > TlS(l) \\ P(l,t) & Tl(l,t) \leq TlS(l) \end{cases} \quad (3)$$

The TlS(l) is the 99.9th percentile latency requirement of the service l, and the Tl (l, t) is the real-time 99.9th percentile latency of the service l(l is a 99.9th percentile latency-sensitive service).

From the previous experimental results, we know that the 99.9th percentile latency of the latency-sensitive service would be decreased by gradually reducing its I/O concurrency level. Since the magnitude of adjustment is set dynamically based on the ratio value, the optimal I/O concurrency level can be found more quickly. As a result, PINE can achieve a lower 99.9th percentile latency in a short period of time.

3) IMPLEMENTATION MODULE

This module is mainly responsible for implementing the specific optimization approaches supported by PINE. According to the calculated optimization parameters, the implementation module adjusts the I/O concurrency level of the 99.9th percentile latency-sensitive service and modifies the disk allocation of the throughput-first services.

The optimization implementation module uses the two-tier structure shown in Figure 8. The first tier implements throughput optimization at the operating system level, and the second tier implements 99.9% latency optimization at the container level.

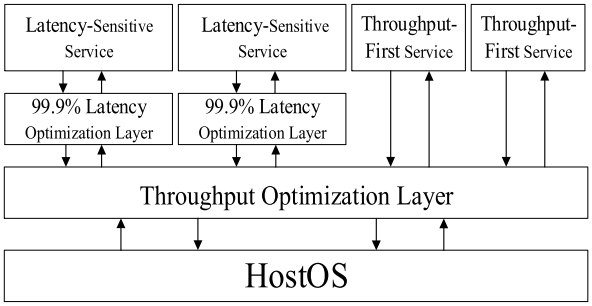


FIGURE 8. Architecture of the optimization implementation module.

We deployed two percentile latency-sensitive services and two throughput-first services on a single physical machine by Docker. A disk device block with a unique disk device number for each container was created respectively. The throughput optimization layer can obtain the performance parameters of each throughput-first service from the Redis database. PINE controls the disk allocation through Cgroups. First, it needs to get the disk device number of each throughput-first service, and then it sets the bandwidth allocation of each disk device block according to the performance parameters that calculated by the decision module.

The 99.9th percentile latency optimization implementation is more difficult than the throughput optimization implementation. The 99.9th percentile latency is a statistical data, so the optimization implementation module needs to get the performance parameters from the database periodically Besides, the 99.9th percentile latency optimization implementation module needs to be deployed in each latency-sensitive service. PINE sets the I/O queue length for each service according to the expected I/O concurrency level which is calculated by the decision module. If the number of I/O requests in the current queue is less than the calculated value, then an I/O request is allowed to enter the request queue, otherwise, it cannot enter the I/O queue.

Through the two-level optimization procedure, PINE can effectively support multiple services running coordinately, even in fierce resource competition situations.

IV. EVALUATION

After analyzing the influence of monitoring cycle, we comprehensively evaluated the PINE from the aspect of functionality and optimization performance, as well as compared its optimization effects with previous solutions.

TABLE 1. Experimental setup.

Item	Version
CPU	16 AMD Opteron^{tm} Processor 6136
Memory	32GB
Storage	5,400 RPM 120GB SATA disks
Operating System	CentOS7
Linux Kernel	3.10.5-3.el6.x86_64
Docker	1.17-ce

The experimental setup is presented in Table 1. As described in the previous section, we select MySQL to represent a typical 99.9th percentile latency-sensitive service, and a Hadoop MapReduce application to stand for a typical throughput-first service.

A. MONITORING CYCLE

Based on the monitoring data, PINE dynamically configures the storage allocation and I/O concurrency to achieve optimization. Monitoring cycle directly influences the performance and overhead of PINE. So we evaluate different monitoring cycles in this experiment. MySQL and Hadoop MapReduce are deployed together on a physical server. Specifically, MySQL enables 64 threads to read and write to 10 tables together. Hadoop does the MapReduce operation for 100 files. The optimization effects of PINE under different monitoring cycle are shown in Figure 9.

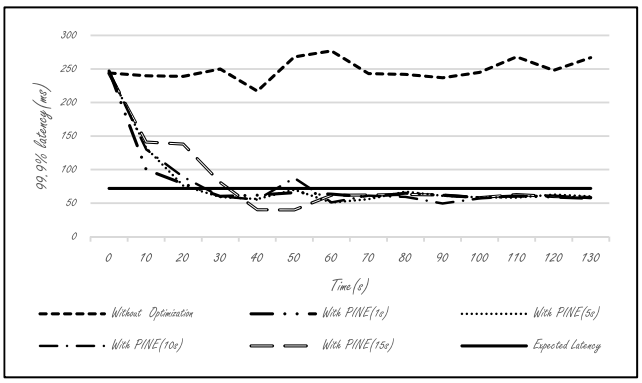


FIGURE 9. Optimization effect of different monitoring cycle.

With the monitoring cycle increases, PINE took a longer time to converge to the optimal results. After running the system for 10 seconds, the 99.9th percentile latency of MySQL has been reduced to about 100ms in the 1s monitoring cycle case, while the latency is about 150ms in the 15s monitoring cycle case. Hence, for a long monitoring cycle, the optimization will be slow, and even not be able to respond to the latency violation in time. From the overhead perspective, we analyzed the optimization calculation under different cases. In the 1s monitoring cycle case, PINE repeated

optimizted the similar data in the first eight cycles. The performance data collected in these periods are very close to each other, so the repeated calculation was done. The phenomenon appeared regularly in the follow-up cycles. In the 5s monitoring cycle case, there is also duplication of configurations for multiple cycles. However, no repeated results in the 10 s and 15 s case. Therefore, we know that PINE will perform repeated calculation/optimization if the monitoring cycle is too short. The reason is that PINE applies feedback strategy to dynamically calculate settings for next cycle. When the new parameters are set, it will take some time to see the effect. So, if the sampling frequency is high, the logged data actually belongs to the results of the previous configuration, which leads to the repeated calculation.

As Figure 9 presents, the optimization speed of the 1s, 5s and 10s monitoring cycle cases are similar. Considering the resource usage in the optimization calculation, the monitoring cycle of PINE is set to 10s. It not only ensure the reasonable speed of optimization but also reduce the overhead.

B. FUNCTIONALITY EVALUATION

1) CONSTANT LOAD OPTIMIZATION

PINE is designed to guarantee each service to achieve its own performance requirements in the hybrid deployment situation. First, we analyzed the constant load case, the workload was configured similarly: MySQL enables 64 threads to read and write to 10 tables together (i.e., the initial IO concurrency level is 64); Hadoop does the MapReduce operation for 100 files. The disk utilization is close to 100% according to IOSTAT when the two services were running together, which means that the disk resource competition is fierce. The 99.9th percentile latency of MySQL running separately is considered as the latency requirement for MySQL. Similarly, the throughput of MapReduce running alone is considered as the throughput requirement for MapReduce. The requirements were set strictly to reflect the effectiveness of PINE.

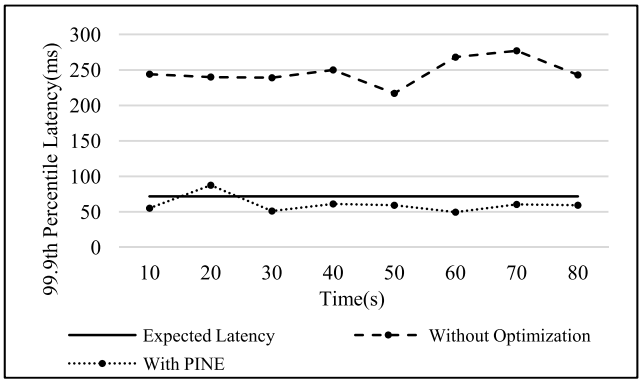


FIGURE 10. Comparison of the effect of 99.9th percentile latency optimization under constant load.

As shown in Figure 10, the 99.9th percentile latency of MySQL (244ms in the first cycle as an example) greatly exceeded the performance requirement (71.83ms) without PINE. In contrast, after using PINE, the 99.9th percentile latency of MySQL was 65.2ms, which was 73.3% lower than that without optimization.

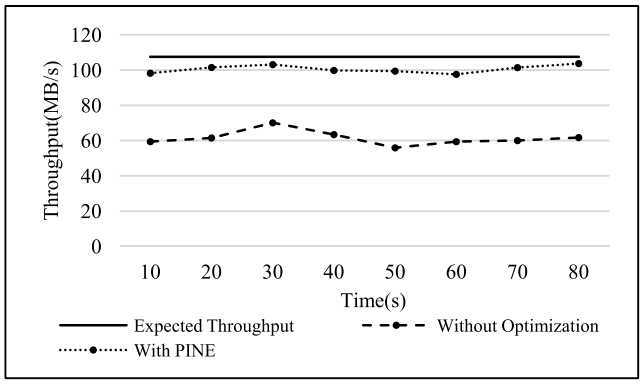


FIGURE 11. Comparison of the throughput optimization under constant load.

As shown in Figure 11, for throughput-first services, the MapReduce throughput is nearly half of the required value when running without optimization, while its real-time throughput is close to the set SLO value (105.4 MB/s in this scenario) when using PINE.

2) DYNAMIC LOAD OPTIMIZATION

The average server utilization of most data centers is low, ranging between 10% and 50% [29], because the load of latency-sensitive services can change dramatically over time. Hence, data centers launch the throughput-first services on the same physical servers to exploit the underutilized sources [30]. In the following experiments, the dynamic load was simulated to evaluate the optimization effect of PINE in the real produc- tion environment.

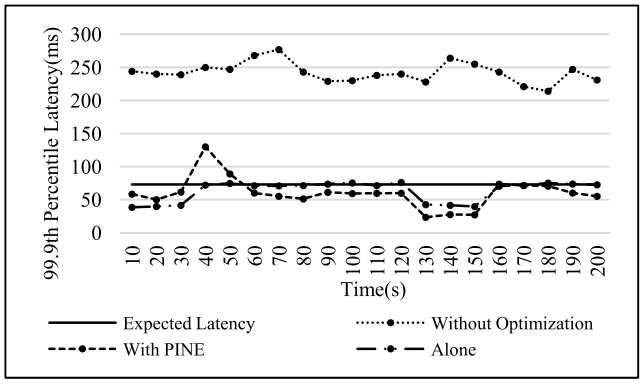


FIGURE 12. Comparison of the 99.9th percentile latency optimization under dynamic load.

First, we evaluated the dynamic load of MySQL with constant MapReduce load. Figure 12 shows the latency comparison of MySQL in the with and without PINE cases. Due to the fierce competition for storage resources between MySQL and MapReduce, the 99.9th percentile latency of MySQL was always around 250ms no matter how the workload changed in the without PINE case. MapReduce occupied

the rest disk resources when the MySQL load was low, which leads to a high 99.9th percentile latency of MySQL. After applying PINE, disk resources were allocated first, so the disk occupied by MapReduce was limited and the 99.9th percentile latency of MySQL can be guaranteed in a reasonable range. When the MySQL load increased, the latency would rise and exceed the target value configured by the users, then PINE would start the second-level optimization and reduce the 99.9th percentile latency by adjusting the IO concurrency level to guarantee the normal operations of MySQL. As shown in Figure 12, the 99.9th percentile latency of MySQL was maintained around the expected latency while the load of MySQL fluctuated in the subsequent cycles.

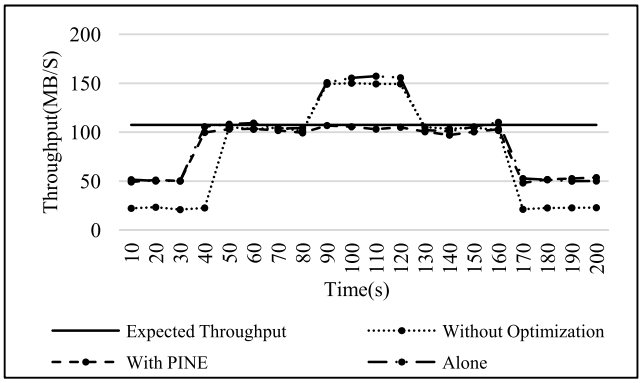


FIGURE 13. Comparison of the effect of throughput optimization under dynamic load.

For throughput-first services, we evaluated the dynamic load of MapReduce with constant MySQL load. Figure 13 shows the throughput comparison of the MapReduce in the with and without PINE cases. When the MapReduce load was low, the throughput of MapReduce was impacted by MySQL. As the load of MapReduce increased, the throughput of MapReduce also increased, so did the latency of MySQL. At this time, the 99.9th percentile latency of MySQL had exceeded the reasonable range. After deploying PINE, the throughput of MapReduce in the low workload situation was similar to the results that MapReduce runs separately; in the high workload situation, PINE restricted the disk occupied by MapReduce to ensure the reasonable disk resources for MySQL.

The results show that PINE can guarantee the performance requirements of services in a hybrid deployment environment.

C. OPTIMIZATION PERFORMANCE FOR DIFFERENT LOAD TYPES

The disk operations mainly contain read and write, different operation combinations may also influence the optimization effect of PINE. For example, compared with writing a hundred of strings to the disk, the latency improvement would be different in the case of reading the strings. In order to evaluate this scenario, we designed four types of read and write combinations which are shown in Table 2. By changing

TABLE 2. Four types of read and write combinations.

	MySQL		MapReduce	
	read	write	read	write
Combination1		√		√
Combination2		√	√	
Combination3	√			√
Combination4	√		√	

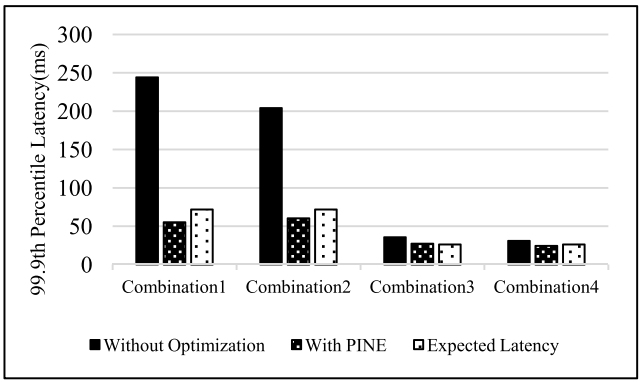


FIGURE 14. Comparison of the optimization effect of 99.9th percentile latency under different load combinations.

the load type of MySQL and MapReduce, the optimization effect in different situations could be observed.

As can be seen from Figure 14, after optimization, the 99.9th percentile latency of MySQL in the four scenarios decreased by 77.3%, 70.4%, 23.8%, and 21.4% respectively, and all reached their performance requirements. When the MySQL load type was a write operation, PINE showed an excellent optimization results, because the write operation was more sensitive to the storage resources. The latency will increase sharply if the storage is used by other workloads, so there is a large optimization space. However, the read operation was not as sensitive as write operation in terms of the storage resource, the optimization space is limited. The experimental results show that PINE still can improve the performance to some extent and help the service meet the expectations.

Similarly, the throughput of MapReduce in these four scenarios (as Figure 15 shows) increased by 65.4%, 33.9%, 41.1% and 10.7% respectively, and all reached the requirements. When the MapReduce load type was write operation, the throughput would decrease more significantly if it is interfered by other workloads. Write operation is more sensitive to the storage resources. Hence, it showed an obvious improvement after using PINE. In the case of reading, although the optimization space was not that large, PINE still could guarantee that the throughput of MapReduce was maintained near the expected value.

These results show that the different load types directly influence the optimization space. PINE can ensure that the performance of each service reaches their corresponding

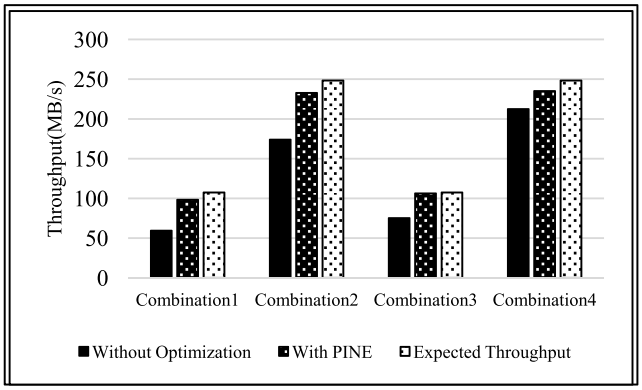


FIGURE 15. Comparison of the optimization effect of throughput under different load combinations.

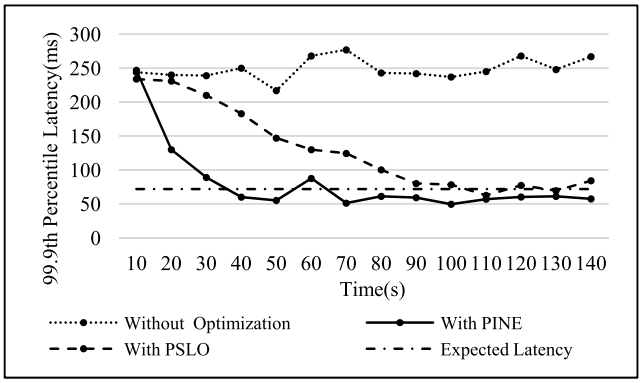


FIGURE 16. Comparison of the recovery speed of 99.9th percentile latency with PSLO.

requirements. In addition, PINE can significantly reduce the 99.9th percentile latency and improve the throughput in the write scenarios.

D. OPTIMIZATION COMPARISON

To evaluate the recovery capability of PINE when the load changes significantly in a short time, and to verify the rationality of using the ratio of real-time 99.9th percentile latency to 99.9th percentile latency requirement as the step size in latency optimization layer, we first ran MySQL and MapReduce together under the low load, then increased their load at the first second. We compared the distribution of MySQL's 99.9th percentile latency in PINE and PSLO [9] in Figure 16. When the load increased, the 99.9th percentile latency of MySQL was maintained at about 239ms without optimization, which was a serious violation of MySQL's requirement. In the case of using PSLO, the 99.9th percentile latency of MySQL dropped to about 78ms after ten cycles (set to 10s for one cycle). When using PINE, the 99.9th percentile latency of MySQL dropped to 65.2ms after three cycles. The results show that PINE has a faster convergence speed and better optimization effect for 99.9th percentile latency. This is because the step length for the IO concurrency level is 1 in PSLO, which is too conservative in the sudden load change situation. PINE can adaptively configure the step length as the ratio of the real-time 99.9th percentile latency to the 99.9th percentile latency requirement, so PINE takes less time to achieve a lower 99.9th percentile latency than the PSLO.

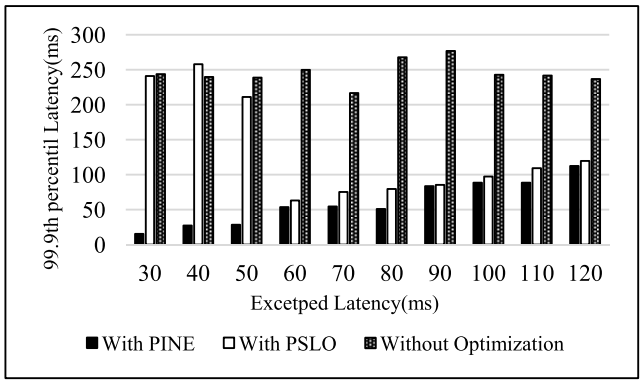


FIGURE 17. Comparison of the optimization effect of 99.9th percentile latency with PSLO.

To further explore the advantages of PINE, we designed a set of experiments to compare the 99.9th percentile latency optimization effect of PINE and PSLO under the different performance requirements. As shown in Figure 17, when the 99.9th percentile latency requirement was set to 30ms, 40ms and 50ms, pre-computation mechanism in PSLO thought that the requirements could not be optimized and ignored it, hence, the 99.9th percentile latency value was similar to the without optimization case. After relaxing the requirements, PSLO began to optimize the latency. As can be seen from Figure 17, optimization with PINE has a lower 99.9th percentile latency than optimization with PSLO.

V. CONCLUSION

This paper aims to enhance the security of all services that share system resources through optimizing performance isolation. Specifically, we target the situation that services with different performance requirements are deployed in a single physical machine at the same time. To guarantee the performance requirements of each service, we proposed PINE, a performance isolation optimization solution, which dynamically manages the disk resource allocation and the I/O concurrency level for each service according to the real-time performance data and pre-defined requirements. In this way, PINE ensures the safety and stability of the system, and reaches the performance requirements of all the running services, even in the fierce storage competition situations.

We implemented PINE on Linux system and evaluated its effectiveness and robustness in complex load scenarios which are used to simulate the real production environment. The results show that PINE can effectively optimize the performance of the running services. Compared to the without optimization case, the 99.9th percentile latency of latency-sensitive services drops by 76.9% and the throughput of throughput-first services increases by 68.3% after

using PINE. Besides, the optimation is efficient, the optimal results can be achieved in 3 cycles.

In the future, we will improve PINE in the following aspects: considering prioritizing services with the same performance requirements; implementing fine-grained performance control methods; designing priority systems and evaluating their effectiveness in more complex scenarios. Such efforts will enhance the versatility of our optimization system.

REFERENCES

[1] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Mar. 2015, vol. 54, no. 1, pp. 171–172.

[2] A. Gulati, A. Merchant, and P. J. Varman, "mClock: Handling throughput variability for hypervisor IO scheduling," in *Proc. 9th USENIX Conf. Oper. Syst. Design Implementat.*, Oct. 2010, pp. 437–450.

[3] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica, "Cake: Enabling high-level SLOs on shared storage systems," in *Proc. 3rd ACM Symp. Cloud Comput.*, Oct. 2012, pp. 1–14.

[4] J. Zhang, A. Sivasubramaniam, A. Riska, Q. Wang, and E. Riedel, "An interposed 2-Level I/O scheduling framework for performance virtualization," in *Proc. ACM SIGMETRICS Perform. Eval. Rev.*, Jun. 2005, vol. 33, no. 1, pp. 406–407.

[5] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM.*, vol. 56, no. 2, pp. 74–80, Feb. 2013.

[6] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble, "Tales of the tail: Hardware, OS, and application-level sources of tail latency," in *Proc. ACM Symp. Cloud Comput.*, Nov. 2014, pp. 1–14.

[7] R. Peinl, F. Holzschuher, and F. Pfitzer, "Docker cluster management for the cloud - survey results and own solution," *J. Grid Comput.*, vol. 14, no. 2, pp. 265–282, Jun. 2016.

[8] D. Bernstein, "Containers and cloud: From LXC to docker to kubernetes," *IEEE Cloud Comput.*, vol. 1, no. 3, pp. 81–84, Sep. 2014.

[9] N. Li, H. Jiang, D. Feng, and Z. Shi, "PSLO: Enforcing the X^{th} percentile latency and throughput SLOs for consolidated VM storage," in *Proc. 11th Eur. Conf. Comput. Syst.*, Apr. 2016, pp. 1–14.

[10] L. Suresh, M. Canini, S. Schmid, and A. Feldmann "C3: Cutting tail latency in cloud data stores via adaptive replica selection," in *Proc. 12th USENIX Conf. Netw. Syst. Design Implement.*, May 2015, pp. 513–527.

[11] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker, "Low latency via redundancy," in *Proc. 9th ACM Conf. Emerg. Netw. Exp. Technol.*, Dec. 2013, pp. 283–294.

[12] M. Jeon *et al.*, "Predictive parallelization: Taming tail latencies in Web search," in *Proc. 37th Int. ACM SIGIR Conf. Res. Develop. Inf. Retr.*, Jul. 2014, pp. 253–262.

[13] T. Zhu, A. Tumanov, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger, "PriorityMeister: Tail latency QoS for shared networked storage," in *Proc. ACM Symp. Cloud Comput.*, Nov. 2014, pp. 1–14.

[14] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey, "Bobtail: Voiding long tails in the cloud," in *Proc. 10th USENIX Conf. Netw. Syst. Design Implement.*, Apr. 2013, pp. 329–342.

[15] A. Gulati, I. Ahmad, and C. A. Waldspurger, "PARDA : Proportional allocation of resources for distributed storage access," in *Proc. 7th USENIX Conf. File Storage Technol.*, Feb. 2009, pp. 85–98.

[16] W. Jin, J. S. Chase, and J. Kaur, "Interposed proportional sharing for a storage service utility," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 32, no. 1, pp. 37–48, Jun. 2004.

[17] A. K. Coskun, T. S. Rosing, K. A. Whisnant, and K. C. Gross, "Temperature-aware MPSoC scheduling for reducing hot spots and gradients," in *Proc. Asia South Pacific Design Autom. Conf.*, Mar. 2008, pp. 49–54.

[18] D. Shankar, X. Lu, and D. K. Panda, "Boldio: A hybrid and resilient burst-buffer over lustre for accelerating big data I/O," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2016, pp. 404–409.

[19] A. Gulati, G. Shanmuganathan, X. Zhang, and P. Varman, "Demand based hierarchical QoS using storage resource pools," in *Proc. USENIX Conf. Annu. Tech. Conf.*, Jun. 2012, pp. 1–13.

[20] Z. Qiu, J. F. Pérez, R. Birke, L. Chen, and P. G. Harrison, "Cutting latency tail: Analyzing and validating replication without canceling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 11, pp. 3128–3141, Nov. 2017.

[21] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz, "Disk scheduling with quality of service guarantees," in *Proc. IEEE Int. Conf. Multimedia Comput. Syst.*, Jun. 1999, pp. 400–405.

[22] M. Jeon, Y. He, H. Kim, S. Elnikety, S. Rixner, and A. L. Cox, "TPC: Target-driven parallelism combining prediction and correction to reduce tail latency in interactive services," in *Proc. ACM SIGPLAN*, Apr. 2016, pp. 129–141.

[23] N. Li, H. Jiang, D. Feng, and Z. Shi "Customizable SLO and its near-precise enforcement for storage bandwidth," *ACM Trans. Storage*, vol. 13, no. 1, pp. 1–25, Mar. 2017.

[24] J. Zhang, A. Riska, A. Sivasubramaniam, Q. Wang, and E. Riedel "Storage performance virtualization via throughput and latency control," *ACM Trans. Storage*, vol. 2, no. 3, pp. 283–308, Aug. 2006.

[25] M. G. Xavier, I. C. De Oliveira, F. D. Rossi, R. D. Dos Passos, K. J. Matteussi, and C. A. F. De Rose, "A performance isolation analysis of disk-intensive workloads on container-based clouds," in *Proc. 23rd Euromicro Int. Conf. Parallel, Distrib., Netw.-Based Process.*, Mar. 2015, pp. 253–260.

[26] S. Mcdaniel, S. Herbein, and M. Taufer, "A two-tiered approach to I/O quality of service in docker containers," in *Proc. IEEE Int. Conf. Cluster Comput.*, Sep. 2015, pp. 490–491.

[27] P. J. Denning and J. P. Buzen, "The operational analysis of queueing network models," *Comput. Surv.*, vol. 10, no. 3, pp. 225–261, Sep. 1978.

[28] M. Karlsson, C. Karamanolis, and X. Zhu, "Triage: Performance differentiation for storage systems using adaptive control," in *Proc. Conf. ACM Trans. Storage*, Nov. 2005, vol. 1, no. 4, pp. 457–480.

[29] McKinsey & Company, "Revolutionizing data center efficiency," in *Proc. Uptime Inst. Symp.*, 2008, pp. 1–15.

[30] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proc. 42nd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2015, vol. 43, no. 3, pp. 450–460.

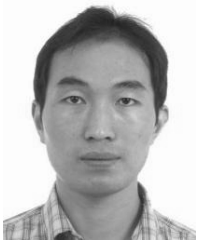
[31] H. Takabi, J. B. D. Joshi , and G.-J. Ahn, "SecureCloud: Towards a comprehensive security framework for cloud computing environments," in *Proc. IEEE 34th Annu. Comput. Softw. Appl. Conf. Workshops*, Jul. 2010, pp. 393–398.

[32] Y. Zhang and J. Joshi, "Access control and trust management for emerging multidomain environments," in *Annals of Emerging Research in Information Assurance, Security and Privacy Services*. Bingley, U.K.: Emerald Group Publishing, 2009.

[33] S. Huang, J. Huang, J. Dai, T. Xie, and B.Huang, "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis," in *Proc. IEEE 26th Int. Conf. Data Eng. Workshops (ICDEW)*, Mar. 2010, pp. 41–51.

YOUHUIZI LI (M'17) received the B.S. degree in computer science from Xidian University, in 2010, and the Ph.D. degree from Wayne State University, in 2016. She is currently an Assistant Professor with the Key Laboratory of Complex Systems Modeling and Simulation, Ministry of Education, Hangzhou. She is also with the School of Computer Science and Technology, Hangzhou Dianzi University, China. Her research interests include energy efficiency, edge computing, and mobile systems. She is a member of the CCF.

JIANCHENG ZHANG is currently pursuing the M.S. degree with the School of Computer Science and Technology, Hangzhou Dianzi University, China. His research interests include the optimization of IO isolation, container, and virtualization.

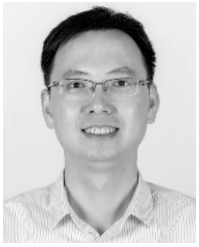


CONGFENG JIANG received the B.E. degree and in hydro-electrical engineering from the North China University of Water Resources and Electric Power, in 2002, and the Ph.D. degree from the Huazhong University of Science and Technology, in 2007.

Since 2007, he has been an Assistant Professor with the School of Compute Science and Technology, Hangzhou Dianzi University, China. He is currently an Associate Professor with the Key Laboratory of Complex Systems Modeling and Simulation, Ministry of Education, Hangzhou. He is also with the School of Compute Science and Technology, Hangzhou Dianzi University. He has published over 50 articles in grid computing, cloud computing, virtualization, and big data systems. His research interests include system optimization and performance evaluation, and distributed system benchmarking.

Dr. Jiang is a member of the ACM and CCF.

JIAN WAN received the B.S. degree in mechanical engineering, the M.S. degree in mathematics, and the Ph.D. degree in computer science, from Zhejiang University, in 1990, in 1993, and 1996, respectively. He is currently a Professor with the School of Information Engineering, Zhejiang University of Science and Technology, Hangzhou, China, where he is currently the Vice Presidenis the Vice Director of the Key Laboratory of Complex Systems Modeling and Simulation, Ministry of Education, Hangzhou. From 2000 to 2015, he was the Dean of the School of Computer Science and Technology, Hangzhou Dianzi University. He has published over 150 articles in grid and services computing, cloud computing, virtualization, and distributed systems. His research interests include distributed systems, computer networks, and big data analytics.



ZUJIE REN received the B.Sc. degree in mathematics from the Nanjing University of Information Science and Technology, in 2005, and the Ph.D. degree in computer engineering from Zhejiang University, in 2010. He is currently the Director of the Research Center for Cross-Media Co-Processing, Zhejiang Lab, China. His research interests include big data systems, cloud computing, and data center technologies.