I/O Performance Isolation Analysis and Optimization on Linux Containers

Li Zhou[1], Yifan Zhang[1], Youhuizi Li[1], Na Yun[1], Lifeng Yu[2]
[1]School of Computer Science, Hangzhou Dianzi University, Hangzhou, China
[2]Hithink RoyalFlush Information Network Co., Ltd.
huizi@hdu.edu.cn

***Abstract*—Container enables a new way to run applications by containerizing the application, which provides kinds of services to make them portable, extensible, and easy to be transferred between private data centers and public clouds. Comparing with virtual machines, containers have several advantages in terms of simplicity, low-overhead and light-weight. However, as the OS kernel and resources are shared by all the hosted containers, performance isolation becomes a challenging issue for guaranteeing their SLA.**

This paper discusses I/O performance isolation issue in container-based clusters. First, we analyze the characteristics of I/O performance isolation from the perspective of the SLA. Then we conduct the observation experiments using multiple containers to obtain the variation trend of the I/O performance parameters and observe the impact of the I/O overload container on the I/O performance isolation of the system. Finally, we propose two algorithms, SLAE and UTE, to improve the I/O performance isolation in container-based systems. These algorithms contribute to decrease the interference caused by the overloaded containers. Experimental results show the feasibility and effectiveness of our proposed algorithms.

Keywords-Container-based System; I/O Performance Isolation; SLA; Docker;

1. INTRODUCTION

Cloud service provides resources to users through the virtual machine [1]. Traditional virtual machine technology is based on the host hardware to build a virtualization management hypervisor, which allocates resources (CPU, memory, etc.) to each virtual machine [9], and each virtual machine has its own OS kernel. But this approach adds overhead when translates machine instructions from guest to host OS [5][14].

As the supporting platform that provides the microservice architecture for software applications, the container has attracted plenty of research attentions in the fields of cloud computing and virtualization. Containers, as Docker standardizes applications and services, opens another door for the operating environment of the application services. Compared with the traditional virtual machine, container isolates resources and permissions into a sandbox, which called a container, and each container is a separate resource usage space. Container isolation technology comes from LXC [7], it achieves environmental resources isolation by the namespaces mechanism and restricts the use of resources by the Cgroups mechanism. Namespaces mechanism and Cgroups mechanism are provided by the Linux kernel. However, because all containers at the same host share one OS core and multiple containers running simultaneously, there will be inevitably disturbances between containers. To improve the performance of container, isolation and safety are the key issues that need to be studied and solved.

The rest of this paper is organized as follows. We introduce the background information and analyze the characteristics of I/O performance isolation in Chapter 2. Chapter 3 shows the variation trends of I/O performance parameters and argues the influence of the overload container on the I/O performance isolation. Chapter 4 describes a prototype system that can dynamically maintains I/O performance isolation of the system and illustrates two different maintenance algorithms, then verifies the feasibility of algorithms. Chapter 5 concludes the paper and discusses future works.

2. BACKGROUND

2.1 Containers

The container technology isolates resources and permissions into containers. Docker makes isolation for PID, UTS, IPC, Network and other environmental resources through the NameSpaces mechanism provided by Linux kernel, besides, it limits the use of CPU, Memory, I/O and other shared resources by leveraging the Cgroups mechanism. These constraints can allow containers follow certain rules at runtime.

Docker provides a few commands which use the Blkio in Cgroups mechanism to restrict containers' I/O usage. There are three groups of I/O configurations available in Docker so far. The first group is: --blkio-weight, which can set the I/O weight for a container. The second group is: --device-write/read-bps, which can limit the read/write rate (bytes per second) from/to a device. The last group is: --device write/read-iops, which can limit read/write rate (I/O per second) from/to a device for a container.

Although Docker provides several ways to restrict the use of the container's I/O. But it requires the admin estimate the I/O usage of the container first, and then admin use the above methods to restrict the I/O of the container. This kind of method is obvious hysteresis and uncertainty, especially when I/O competition becomes intense and complex. Current container management software such as the popular Kubernetes, a tool for scheduling and managing containers [7].

DOI reference number: 10.18293/SEKE2017-201

The smallest unit for managing of Kubernetes is a Pod, which is a set of containers, Kubernetes is responsible for completing their management. The main purpose of the container-based cloud management software is to cluster the deployment and management of containers. However, there is no specific guarantee for the isolation between containers on the host OS.

Sean McDaniel et al. [3] proposed a two-tiered approach to guarantee I/O quality of service in Docker containers. They thought that combining the node-level QOS and the cluster-level QOS together will have better load balancing and higher resource utilization. Miguel Xavier et al. [2] found the container-based system is not yet mature to ensure performance isolation among disk-intensive workloads. To address this problem, they [2] proposed to combine different types of loads on containers to reduce the interaction. Their results suggest the combination of CPU intensive and I/O intensive to alleviate the performance impacts, rather than a combination of disk-intensive and memory-intensive.

2.2 *Isolation*

2.2.1 *I/O Performance Isolation based on SLA*

In general, cloud providers are responsible for guarantee the SLA for their users. They will allocate the number of containers and the resource cap in order to ensure that the SLA requirement of all the containers on the host can be satisfied at the same time. But if some users or applications produce excessive load, it is possible that the balance is destroyed and other containers' performance is influenced.

Based on the SLA, we can classify the I/O performance isolation of container-based system into two categories as follows:

Poor I/O Performance Isolation: When the multi-container running on the host, the I/O interference between containers is very strong, and the I/O state of the container is easy to change. Especially when some containers produce overload I/O operations, it will cause other containers' I/O performance fail to reach the SLA requirement, such as I/O wait time and IOPS. In this case we believe that the I/O performance isolation of the container-based system is poor.

Excellent I/O performance isolation: When the multi-container running on the host, the I/O interference between containers is very weak, and the I/O state of the container is very stable. Even if some containers produce overload I/O operations, because the system has a good I/O performance isolation, it will still guarantee other containers' I/O performance reach the SLA standard. In this case we believe that the I/O performance isolation of the container-based system is excellent.

2.2.2 *WorkLoad Model*

We can treat containers as processes that produce I/O, since they share one OS kernel, we can assume the I/O queue is a shared service queue. So the system can be viewed as the M/M/1 model that have single-queue and single-server.

In the M/M/1 model of single-queue and single-server, we know W = S/(1-U) according to Litte's Law and Utilization Law [4]. Where W is the job wait time, U is the utilization of the system, and S is the actual service time of the job. Based on the equation, we can observe that the higher current disk device I/O utilization is, the greater the I/O delay will be, and when the utilization exceeds a certain threshold or so, the I/O wait time will appear rapid growth.

3. Performance Trends And Impact Analysis

3.1 *Experimental Setup*

3.1.1 *Experimental Environment*

We use a node server as a host, the node has 16 AMD Opteron(TM) Processor 6136 CPU and 32GB of Memory capacity, and it equipped with CentOS7 and Docker 1.12.5.

3.1.2 *Experimental Content*

We execute the script file in the container to generate the I/O request to write the file. The rate of script writing is relatively small, so we use the Linux DD command to simulate the overload I/O case. It is worth noting that the I/O should avoid Buffer for the accuracy of the observation.

The observation is mainly divided into two parts. The first part we open a number of containers on the host while running the script file to generate write I/O, the variable is the number of growing containers, meanwhile observing the trend of the I/O indicators. The second part of the experiment, we run a number of normal containers coupled with a container generate high I/O load to observe the effect of overload container on the I/O performance for other normal containers. Since each container has a specific disk device number, so we observe the I/O of the corresponding container through the disk device.

3.2 *Results And Analysis*

3.2.1 *IOPS, write rates show a decreasing trend of power function*

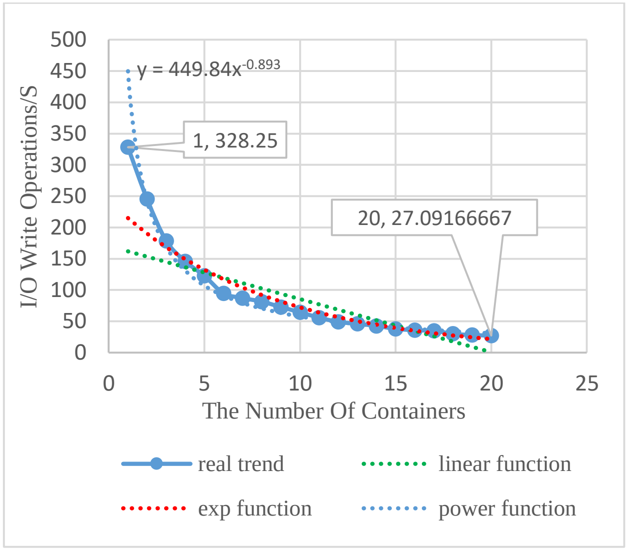

Figure 1. Variation trend of I/O write operations/s

We allow each container to achieve the same I/O load. As we can see from Figure 1, with the number of containers increased from 1 to 20, the average I/O write operations is reduced from 328 times per second to 27 times per second. The real data trend is a solid line, the whole process fits closely the decline of the power function, rather than fits other functions.

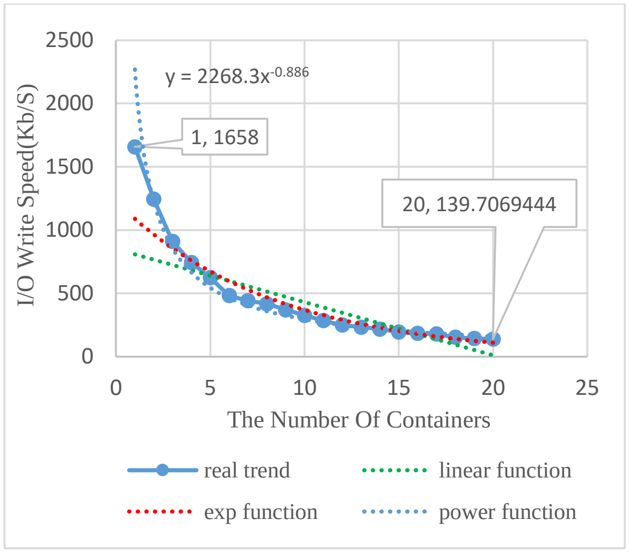

Figure 2. Variation trend of I/O write speed

Similarly, in the above case, we can see from Figure 2, the average write speed of each container also shows a similar exponential decline trend. On our machine, the write speed decreased from 1658Kb/s to 139Kb/s. The real data trend is the solid line and the fitting curve is represented by the dotted line.

3.2.2 I/O utilization, the length of wait queue fit logarithmic function growth trend

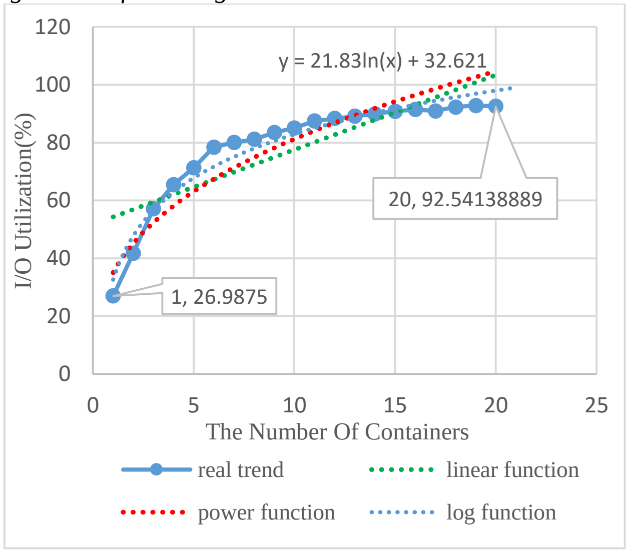

Figure 3. Variation trend of I/O utilization

From Figure 3 we can see that with the number of containers increased from 1 to 20, the average I/O utilization of disk device increased from 26.98% to 92%. The variation trend of real data is the solid line, we fit the whole process very close to the growth of the logarithmic function, rather than other functions.

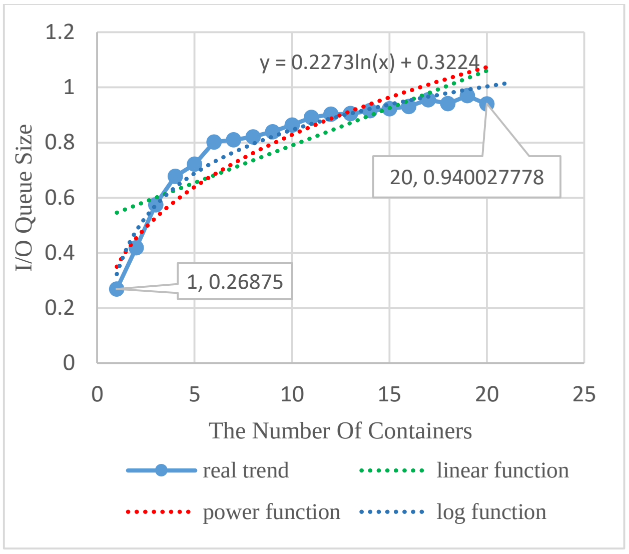

Figure 4. Variation trend of I/O queue size

Similarly, in the above case, we can see from Figure 4, the waiting queue length of containers is also growth as the logarithmic function. On our machine, the length of the I/O queue increased from 0.268 to 0.94. The real variation trend of the data is the solid line and the fitting curve in the graph with the dotted line.

3.2.3 The I/O utilization exceeds a certain threshold, then wait time soared

We increase I/O utilization of the system by increasing the number of containers, and observe the relationship between the I/O utilization and I/O wait time.

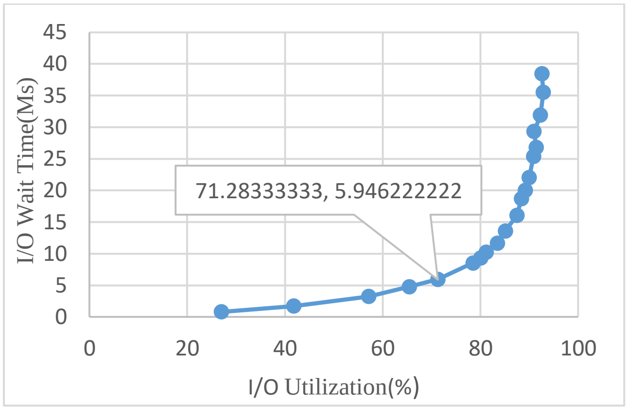

Figure 5. Variation trend of I/O utilization and wait time

As Figure 5 shows, with the increase of containers, the I/O utilization increases from 27% to 92%. And the wait time is close to a linear growth before the I/O utilization reaches 70%-80%. When the I/O Utilization is 71%, the average I/O wait time is about 5.9Ms. However, when the I/O utilization exceeds 70%-80%, the wait time grows rapidly near the exponential function. Soon the wait time is up to 30Ms or more, which is far exceeded the upper limit of SLA for a normal disk I/O.

3.2.4 *Overloaded container will cause damage to the I/O performance isolation of the system*

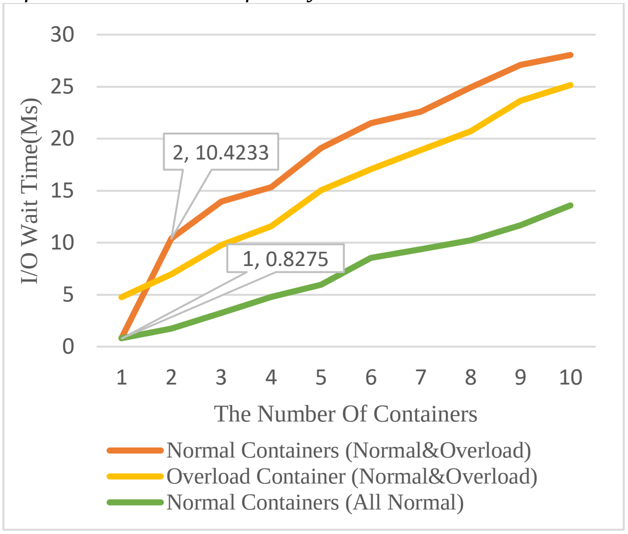

Figure 6. Variation trend of I/O wait time when all normal containers and one overloaded container & some normal containers

Figure 6 describes the impact of a container with a high I/O utilization on the performance of other normal containers. If the I/O loads of all hosted containers are normal, with the increase of containers, we can see the variation trend of I/O wait time fits a slow growth of a linear. When running 1-9 normal containers with an overloaded container, the result shows that the overload container resulted in a significant increase in the I/O wait time of the normal containers, while the change in the overload container is small.

Here we define ΔW = W_New – W_Pre, which represents the change in the I/O wait time of one container. W_New indicates the I/O wait time of the container before the overload container appears, W_Pre indicates the I/O wait time of the container after the overload container appears, it can reflect the impact of container's I/O performance. The large ΔW shows big impact, the small ΔW shows small impact. And we use ΔSW to indicate the size of I/O performance isolation of the system that has m containers when the I/O strength of the container n grows. The more containers the system has, the more susceptible the system's I/O isolation is. The smaller ΔSW is, the more excellent the system isolation is.

$$\Delta SW = \sum_{\substack{0 \le i \le m \\ i != n}} \Delta Wi$$

When there is only one normal container, the I/O wait time is approximately 0.82Ms. After an overloaded container appears, the I/O wait time of the normal containers rises immediately to 10.4Ms, at this time ΔM = 10.94-0.82 = 10.1Ms, while the I/O wait time of overload container rises from 4.7Ms (running alone) to 6.9Ms, the ΔM is only 2.2Ms. With the increase of containers, the ΔM of each container changes only a little, while the ΔSM is growing fast because of the increase in the number of containers.

3.3 *Summary*

In order to gain a deeper understanding of the trend in the I/O performance and the intrinsic relationship between the I/O parameters in the container system, we summarized some of the above meaningful conclusions as follow. We found that as the number of containers grows, IOPS, write rates show a decreasing trend of the power function. The reason why they are not a linear is due to the presence of I/O kernel scheduling. Meanwhile, we found that I/O utilization and the length of wait queue present logarithmic function growth trend. Then we run normal containers and add an overloaded container at the same time. The observation parameter is the I/O wait time which is universality, we find that the emergence of overload container led to significantly waiting time increasing of other containers, the I/O performance isolation of system is damaged. As the number of normal containers increases, ΔW itself does not change much, but the value of ΔSW is rising. At the same time, we found that the ΔW value of the overload container is smaller relative to the ΔW value of the normal container. We propose that the I/O strength of the overload container should be reduced to maintain I/O performance isolation of container-based system. And the container with the highest I/O utilization and read/write speed should be regulated first for both fairness and effect priority.

4. System And Algorithms Design

4.1 *System Design*

According to the observations, we found that the I/O performance isolation of the container-based system is not guaranteed, especially when overload containers exist, it may cause strong interference to other containers. So we propose a system which can dynamically regulates containers' I/O to provide I/O performance isolation and I/O load balancing for the container-based system.

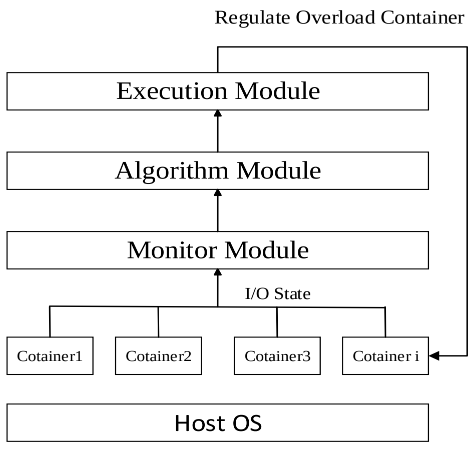

Figure 7. I/O dynamic adjustment system framework

Figure 7 demonstrates the architecture of the system. It is built on the host OS and composed of 3 major modules: Monitor Module, Algorithm Module, and Execution Module.

Monitor Module: The module is responsible for collecting real-time I/O status of all containers running on the host currently, including IOPS, read/write rate, I/O wait time, the average length of the request queue, I/O utilization etc. It will pass this information to the Algorithm Module.

Algorithm Module: The module is responsible for policy analysis based on the information collected by Monitor

Module. Adjustment algorithms determine whether the I/O performance isolation of the system will be destroyed. Based on the algorithm, this module decides if it is necessary to regulate the containers' I/O, and the corresponding results are sent to the execution module.

Execution Module: If the result of the algorithm module is that the I/O performance isolation of the current system needs to be maintained, then the execution module will perform the specific operation accordingly and use Cgroups to restrict the I/O intensity of the overload container appropriately so that the I/O performance isolation of the container-based system is maintained.

4.2 *Algorithms Design*

According to the results of analysis and observations, we designed the following two dynamic adjustment algorithms for judging the I/O performance isolation.

4.2.1 *SLAE(Service-Level Agreement Ensure) dynamic adjustment algorithm*

In the previous section, we discussed the I/O performance isolation of container-based system based on the SLA. And we have experimentally observed that when an overloaded container generates excessive I/O requests, it may cause I/O state of other containers fail to meet SLA requirements.

In the SLAE dynamic adjustment algorithm, we use the SLA requirements of the I/O service as the parameter to evaluate the isolated damage. Take I/O wait time as an example. If the overloaded container has caused the I/O Wait Time of other normal containers exceed the SLA requirements, we believe that the I/O performance isolation of the current container-based system has been disrupted. Hence, we decrease the overall I/O utilization of the system by reducing the I/O strength of the overload container so that the I/O wait Time, as well as other parameters of other containers, can be guaranteed, that is, the I/O performance isolation of the system is guaranteed.

4.2.2 *UTE(Utilization Threshold Ensure) dynamic adjustment algorithm*

Since the multi-container case can be seen as a model of single-queue and single-server. And experiments verify that when I/O utilization of the system exceeds a threshold, the average I/O wait time of the container will be a sharp increase, then I/O state will be very unstable, which means I/O performance isolation is poor.

So judging I/O performance isolation of the system can also depend on the I/O utilization of the system. We can set a threshold to the I/O utilization of the system. And the threshold is about 0.7-0.8 according to the observation results. We know that when the I/O utilization exceeds the threshold, the I/O wait time of all containers becomes high and the rate of increase is accelerating, hence the I/O performance isolation of system is threatened. At this time, the execution module chooses the container with the highest I/O utilization in the current running containers and reduces its I/O intensity, so that the I/O utilization of the system is reduced, the I/O performance isolation and load balancing of the system are maintained.

4.2.3 *Algorithms comparison*

When the service provider and the user have signed the SLA, the SLAE algorithm should be adopted. If users bought the service, then the enterprise should ensure the service quality. For users, if the I/O metrics reach the SLA requirements, then the I/O service performance is good. Therefore, if there is a clear SLA requirement, it is recommended to use the SLAE algorithm.

In the absence of SLA situation, for example, the container-based system is used to be the operating environment of applications. The operating conditions of application are complex and unstable. To ensure I/O load balancing, the UTE algorithm should be used. Our experiments show that when the I/O utilization exceeds a certain threshold, the I/O performance of the applications in containers will be unstable and drops rapidly. The UTE algorithm can guarantee I/O performance and load balancing for container applications in this situation.

4.3 *Regulation of Overload Container*

Linux's Cgroups mechanism can restrict the I/O strength of a particular process on a specific device. Each container has a main process at the host, all application processes within the container are child processes of the main process. Docker has assigned a separate disk device for each container, each of disk devices has its own disk device number. So we can set the Cgroups group by the specific device number and process ID, reduce the corresponding container process for their own disk device I/O strength to achieve the corresponding restriction of the container's I/O.

We run three normal I/O containers and a strong overload I/O container simultaneously. Then we restricted the I/O of the overload container and study the changes of the I/O wait time of the three normal containers.

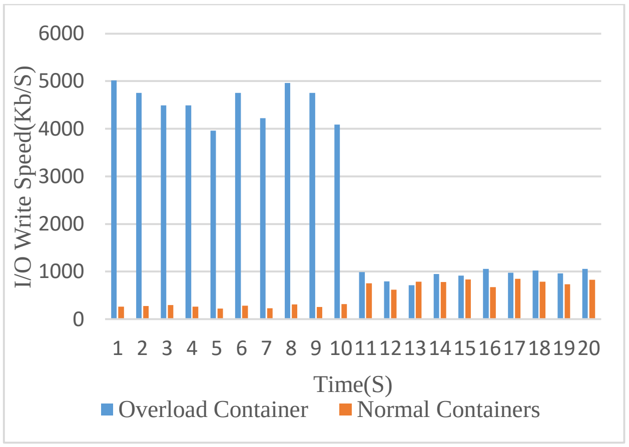

Figure 8. The I/O write speed's variation of containers

Figure 8 depicts the variation in I/O write speed of the overload container and the normal containers. We can see that in the first 10 seconds, the I/O write speed of the overload container is 4.5Mb/S or so, then it immediately reduced to about 1 MB/s due to our restriction, and the I/O write speed of the normal container raised simultaneously.

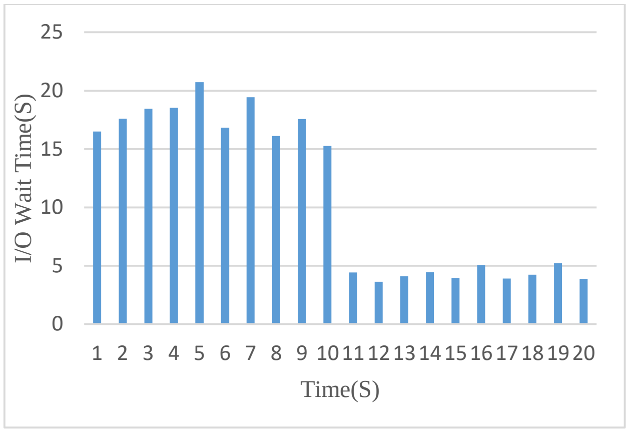

Figure 9. Variation of normal containers' I/O wait time

Figure 9 describes the variation of the average I/O waiting time. In the first 10 seconds, the average I/O wait time of normal containers reached 17.7Ms, and the maximum has exceeded 20Ms due to the high I/O write speed and high I/O utilization of the overload container. After 10 seconds we restrict the I/O write speed of the overload container, the I/O utilization of the system is reduced, we can see the average I/O wait time of normal containers drops immediately below 5Ms which is an excellent level.

4.4 Summary

In order to improve the I/O performance isolation of the container-based system, we propose a system which can dynamically maintain the I/O performance isolation. It consists of three modules, Monitor Module, Algorithm Module and Execution Module. We design two judgment algorithms, the SLAE algorithm and the UTE algorithm, and we discuss their details as well as usage scenarios. In order to verify the feasibility of the algorithms, we did the following experiments. We run an overloaded container and three normal containers on one host at the same time. We observe that the overload container greatly influenced on the I/O performance of other normal containers. We use the Cgroups mechanism to restrict the I/O intensity of the overload container. The results show that, we restrict the overload container's I/O write rate from 4.5Mb/s to 1Mb/s, so the average I/O wait time of the normal containers is reduced from 17.7Ms to below 5Ms. The I/O performance isolation of the container-based system is guaranteed, and the feasibility of the algorithms is also proved.

5. Conclusion And Future Works

The performance isolation is an important issue for the container-based system. In this paper, we analyzed the characteristics of I/O performance isolation from the perspective of the SLA. We observed the variation trend of containers' I/O performance parameters under the condition that multiple containers running simultaneously, and discussed the impact of the I/O overload container on the I/O performance isolation of the container-based system. In order to improve the I/O performance isolation of container system, we proposed a system that can dynamically maintain the I/O performance isolation. Besides, we designed two I/O isolation maintenance algorithms and discussed their usage scenarios. Moreover, we use Cgroups mechanism to restrict the I/O strength of the overload container in order to reduce the impact on other normal containers, so the I/O performance isolation of the container-based system can be guaranteed, and the feasibility of the algorithms is also proved.

Our future research is to achieve dynamic maintenance of I/O isolation systems, and design more algorithms to further improve the I/O isolation performance. Besides, in order to comprehensively enhance the performance isolation of the container-based system, we plan to extend the above ideas to other shared resources such as CPU and Memory etc.

Acknowledgment

This work is supported by the National Key Technology R&D Program under Grant (No.2015BAH17F02), the NSF of China (No. 61572163) and the NSF of China (No. 61602137).

References

[1] R. Krebs, C. Momm, and S. Kounev, "Metrics and techniques for quantifying performance isolation in cloud environments," Science of Computer Programming, 2012.

[2] M. G. Xavier, I. C. Olivera, F. D. Rossi, and R. D. D. Passos, "A performance isolation analysis of disk-intensive workloads on container-based clouds," Euromicro International Conference on Parallel, 2015, pp. 253-260.

[3] S. Mcdaniel, S. Herbein, and M. Taufer, "A two-tiered approach to I/O quality of service in docker containers," IEEE International Conference on Cluster Computing, 2015, pp. 490-491.

[4] P. J. Denning, and J. P. Buzen, "The operational analysis of queueing network models," Computing Surveys, vol. 10, no. 3, pp. 225-261, 1978.

[5] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," Lecture Notes in Computer Science, 2014, pp. 438-453.

[6] R. Peinl, F. Holzschuher, and F. Pfitzer. "Docker cluster management for the cloud - survey results and own solution." Journal of Grid Computing, vol. 14, no. 2, pp. 1-18, 2016.

[7] D. Bernstein, "Containers and cloud: from LXC to docker to kubernetes," IEEE Cloud Computing, vol. 1, no. 3, pp. 81-84, 2014.

[8] C. Boettiger, "An introduction to docker for reproducible research," Acm Sigops Operating Systems Review, vol. 49, no. 1, pp. 71-79, 2015.

[9] A. M. Joy, "Performance comparison between linux containers and virtual machines," Computer Engineering and Applications IEEE, 2015, pp. 342-346.

[10] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," Linux Journal, vol. 2014, no. 239, 2014.

[11] K. T. Seo, H. S. Hwang, I. Y. Moon, O. Y. Kwon, and B. J. Kim, "Performance comparison analysis of linux container and virtual machine for building cloud," Networking and Communication, 2014, pp. 105-111.

[12] J. K. Patel, S. Akhtar, V. K. Agrawal, K. N. Bala, S. Murthy, and A. R. Manu, "Docker container security via heuristics-based multilateral security-conceptual and pragmatic study," IEEE–Iccpct, 2016.

[13] A. Celesti, D. Mulfari, M. Fazio, M. Villari, and A. Puliafito, "Exploring container virtualization in IoT clouds," IEEE International Conference on Smart Computing, 2016, pp. 1-6.

[14] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization vs containerization to support paas," IEEE International Conference on Cloud Engineering, 2014, pp. 610-614.

[15] N. Kratzke, "About microservices, containers and their underestimated impact on network performance," Cloud Computing, 2015, pp. 165-169.