2016 4th Intl Conf on Applied Computing and Information Technology/3rd Intl Conf on Computational Science/Intelligence and Applied Informatics/1st Intl Conf on Big Data, Cloud Computing, Data Science & Engineering

Behavior-based Memory Resource Management for Container-based Virtualization

Gaku Nakagawa
Department of Computer Science
University of Tsukuba
Tsukuba, Ibaraki, JAPAN
Email: gnakagaw@cs.tsukuba.ac.jp

Shuichi Oikawa
Department of Computer Science
University of Tsukuba
Tsukuba, Ibaraki, JAPAN

***Abstract*—Container-based virtualization is a virtualization technique at operating system level. It realizes a lightweight virtualization whose overhead is much less than hypervisor-based virtualization. In a container, the programs often consume much more memory than the developer or the administrator expected. There are several methods to prevent such memory overuse. They, however, have each shortcoming such as its operation cost, the false-positive problem and so on. In this paper, we propose a new resource management method for container-based virtualization environment based on the resource consumption behavior. The method detects a wrong container that has a sign of memory overuse and makes a limitation to the detected container. A preliminary experiment for a proof-of-concept shows that it is possible to implement the proposed method in Linux kernel and it is effective to attack the problem situation.**

1. Introduction

Container-based virtualization is a virtualization technique at operating system level [1], [2]. It realizes a lightweight virtualization whose overhead is much less than hypervisor-based virtualization. Especially, web application developers and web application provider company focuses on container-based virtualization as a platform for their web applications.

A container hosts various programs. These programs occasionally consume much more memory resource rapidly than the developers or the administrators expected. An example is an unexpected access concentration to the target system. The http server or application server increase its instances (processes or threads) to deal with the many requests. The increasing of the instances consumes additional memory area. If the administrator or developer don't set the limitation correctly, many requests to the service occur memory shortage on the host system. Unintended memory consumption by software is another example. Programs often have the bugs related to memory allocation, such as memory leak. The failures cause rapid memory consumption and degrade the system availability. In addition, the malicious users exploit the failures to perform denial-of-service attacks.

The straightforward measure to such memory consumptions is memory utilization limitation. The limitation is effective to prevent memory overuse. However, it is required to estimate the memory usage of each container. The measure increase the operating cost. It is another measure in a memory shortage situation to terminate processes that overuse memory. In the situation, the operating system cannot always select a suitable process. The OS often terminate a innocent process. The tactics terminating process has a false-positive problem, as mentiond above.

In this paper, we propose a new memory management method for container-based virtualization environment based on the memory consumption behavior. The method detects a wrong container that has a sign of memory overuse and makes a limit to the detected container. A preliminary experiment for a proof-of-concept shows that it is possible to implement the proposed method in Linux kernel and it is effective to attack the problem situation.

The paper is organized as follows. In Section 2, we discuss about the problems of the existing memory management method for container-based virtualization environment. In Section 3, we discuss about the proposed method: a method based on the memory comsumption behavior. Section 3 also explain the preliminay implementation of the proposed method. We executed a preliminary proof-of-concept experiment to evaluate the proposed method. Section 4 explain about the experiment and the result. Finally, in Seciton 5, we conclude the paper.

2. Background

In this section, we discuss the problem of the existing memory management method to prevent memory shortages. There are two existing methods to prevent memory shortage for container-based virtualization. One is memory resource limitation for each container. The other is termination of victim process.

The straightforward method is memory resource limit. We estimate the maximum amount of memory that each container consumes, and set the values as the limitation value of the container. The limitation prevents the overuse memory even if the containers go out of control. The method has two problems. One is high operation cost. The method

978-1-5090-4871-7/16 $31.00 © 2016 IEEE
DOI 10.1109/ACIT-CSII-BCD.2016.48

requires the precise memory usage estimation for each container. If the estimated value is less than the actual needs, it occurs the performance degradation or the termination of the processes in the container. The other problem is the decreasing in the number of containers that a host system can manage because we must keep the overcommit rate low. Processes always do not use the maximum memory. It is possible to concentrate container in a host system by memory overcommit. However, high overcommit rate makes the effect of the method weak.

Termination of victim process is another measure. The method terminates the selected process when the amount of free memory is under the threshold. The operating system kernel chooses the victim process based on the memory usage of each process, the importance of process and so on. The method has two problems. One is a false-positive problem. The method often terminate an innocent process which is not the actual cause. The victim process is not necessarily alone because the operating system kernel continues to terminate process until they obtain a certain amount of memory. The other problem is the execution timing. As mentioned above, the method has a significant risk to terminate an inappropriate process. We can only use the method in a impending memory shortcoming situation. Thus, the timing may be too late to respond.

3. Memory Management based on Process Behavior

As mentioned in Section 2, the two existing memory management methods have the problems to address the unintended rapid memory consumption. The both of methods adopt the usage memory as the main index to make a decision. In this section, we proposed a memory management method that adopts the memory consumption behavior of containers as the decision index.

Memory consumption behavior is a tendency to consume memory. Each container has its memory consumption behavior; the amount of consumed memory, the speed of memory consuming. They are different between containers. Even if two containers have the same contents, the memory consumption behavior may be different when they have different processing target data each other. For example, assuming that two containers provide the same web service, the memory consumption behavior is different depends on the requests to each container. In the proposed method, we regard the memory consuming per unit time as memory consumption behavior.

We can detect an abnormal memory consumption by memory consumption behavior. A problem about memory consumption in a container-based virtualization is rapid memory consuming. For example, when many access to a web application container, the container consume memory rapidly to process the requests. For another example, when a faulty program often consumes much memory rapidly. Of course, there is a case that a program consumes much memory slowly. We can attack the case with periodic restarts containers.

Here, we propose a memory management method to prevent abnormal memory consuming based on its memory consumption behavior. As mentioned above, we can regard a rapid memory consuming as a sign of abnormal memory consuming. The proposed method watches the amount of memory consuming by each container. When it detects a rapid memory consuming, it makes a memory utilization limitation for the detected container. The proposed method reconsider the limitation periodically because the detected container may be innocent.

The remains of the section describe the detailed design of the proposed method. In addition, they describe the preliminary implementation of the proposed method. The target of the implementation is Linux 3.14.0.

3.1. Monitoring memory consumption behavior

The proposed method focuses on memory consuming behavior for each container. The memory consuming behavior is the increase of the amount of memory consumption per unit time. When a container reduces the amount of memory consuming, the behavior is a negative value. The proposed method calcs the behavior when the operating system kernel (OS kernel) allocates a new memory page to a process. The proposed method doesn't hook all memory allocation because OS kernel executes the memory allocation process frequently. The proposed method samples a memory allocation process per certain times. In a calculation process, the proposed method use the difference of memory consumption between two calculation timing.

In the preliminary implementation, we add a memory consumption calculation process to the page fault process. The added processing records the amount of memory consuming and the kernel internal time to the kernel data structure that hold the information of each container. Also, the processing calcs memory consumption behavior based on the recorded one before memory consuming information and the current one. We use a kernel monotonic time in nanoseconds acquired via ktime_get_ns() kernel function.

3.2. Making a memory limitation

The proposed method compares the memory consumption behavior with the threshold value at the same time as a calculation of the behavior. When the behavior exceeds the threshold value, the proposed method make a memory limitation for the detected container. We call the threshold value as LIMIT_THRESHOLD. The memory limit value is MEMORY_LIMIT. The proposed method records the information about the detected container; the target container, the memory consumption behavior at the detected timing, the amount of memory consumption at the detected timing.

In the preliminary implementation, the proposed method doesn't execute a making limitation process at the same time as the behavior calculation and detection of a limitation target container because the making limitation process is not suitable to run in an interrupt context. When the method

detects a limitation target, it makes a kernel thread to make a limitation. The OS kernel runs the thread later.

3.3. Release from a limitation in memory release

When the method detects a resource-limited container that consumes memory normally, it releases the detected container from the memory limitation. The method compares the memory consumption behavior at the memory release and the behavior at the making the limitation. The two conditions for the release are as follows, 1) The memory consumption behavior at the memory release timing is less than the behavior at the making limitation timing. 2) The amount of memory utilization of the container at the memory release timing is less than MEMORY_LIMIT.

In the preliminary implementation, we added a check process mentioned above to the munmap() system call and mremap() system call. The added process examine whether the system call source process is in the resource-restricted container or not. If not, the OS kernel continues to execute normal system call processing. If the process is the resource -restricted container, the OS kernel calc the memory consumption behavior at the timing and determine whether the kernel should release the container or not.

3.4. Periodic release from a limitaiton

As mentioned above, the proposed method monitors the memory release processings and it makes a decision of whether they should continue the memory limitation or not. In addition, the proposed method examines the resource-restricted containers periodically to the process that releases memory. When the OS kernel detected a container whose memory consumption behavior is normal, the OS kernel releases the detected container from the memory limitation. The cycle of the examination is defined as RELEASE_CHECK_CYCLE. The conditions to release are the same as the examination in memory release.

In the preliminary implementation, we add a examine process to the kernel. A kernel thread implements the added process. The OS kernel invokes the thread periodically.

4. Experiment

This section describes an evaluation experiments. The aims of the proposed method is a detection of illegal containers and prevention from performance degradation. For a proof of concept, we executed an experiment to detect illegal containers and limit the resource utilization for them.

4.1. The Experimental Setup

In the experiment, we construct a container host environment that manages three containers. We apply the proposed method to the environment to evaluate the effect of the proposed method. The host OS is Linux kernel 3.14.0. We adopt Docker as a manager of the container virtualization environment. Docker mainly handles the management of container images and dispatching of them. Table 1 shows the specification of the experiment environment.

TABLE 1. THE SPECIFICATION OF THE EXPERIMENT ENVIRONMENT

	Container Host	Load Generator
CPU	Intel Core i7-2600 (4core)	virtual (2core)
RAM	2.0 GiB	2.0 GiB
Swap	4.0 GiB	5.0 GiB
OS	Linux kernel 3.14.0	Linux kernel 3.10.0

The container host manages three containers. We call them as Container A, Container B, and Container C. Container A provides an RDBMS system environment. The container hosts MySQL environments. Container B hosts an online contents management system. The software is WordPress. Container B depends on Container A to store its contents. Container C is an illegal container. In the container, a program to occur much memory consumption. We call the program as memory eater. Memory eater generates 20 child processes. The parent and children write dummy data in its 100 MiB memory space. The memory eater occurs a rapid memory shortage situation in the container host. We apply the proposed method to the memory shortage situation to evaluate whether the method prevents a performance degradation by memory eater.

The index of the evaluation is request processing performance of the web application provided by Container B (i.e. WordPress). We set up a load generator in the same network of the container host. We make requests to WordPress in Container B continually and measure the request processing performance of it.

4.2. Result: Throughput

Figure 1 describes the request processing performance. The x-axis describes the number of threads to send the requests. The y-axis describes the throughput of the target web application. Condition A (green line) describes the request processing performance when only Container A and B are running (i.e. Container C, the illegal container, didn't run in this condition). Condition B (blue line) describes the request processing performance when Container C occurs a memory shortage situation without the proposed method. Condition C (red line) describes the performance in the same situation but with the proposed method.

The comparison between Condition A (green line) and Condition B (blue line) shows that memory eater in Container C occurs a performance degradation phenomena. The maximum throughput in Condition A is 66.8 requests/seconds. The average one is 59.1 requests/seconds. The performance degraded in Condition B. The maximum throughput in Condition B is 37.0 requests/seconds. The average one is 30.23 requests/seconds.

In contrast, the comparison between Condition A (green line) and Condition C (red line) shows that the proposed method reduce the performance degradation by Container C. The maximum throughput in Condition C is 56.6 re-

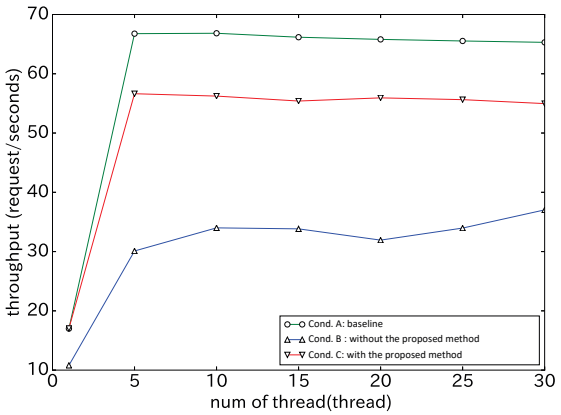

Figure 1. Request Processing Throughput

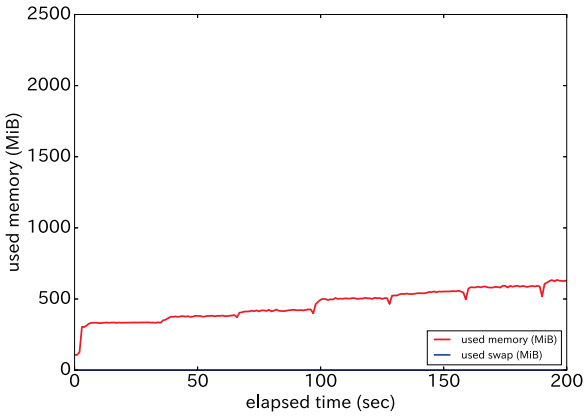

Figure 2. Memory Consumption (Baseline)

quests/seconds. The average one is 50.3 requests/seconds. The performance degradation rate between Condition A (baseline) and Condition B is 48.8 %. In contrast, that between Condition A and Condition C is 14.9 %. Therefore, the results show that the proposed method is effective to prevent the performance degradation by rapid much memory demand on a container virtualization environment.

4.3. Result: Memory Utilization

The results in Section 4.2 show that the proposed method can reduce the performance degradation of legitimate containers. However, the reason is unclear because we have not inspected it based on the point of view of the container host yet. We measured the change of memory utilization during the experiments for each experimental condition (i.e. Condition A, B and C). Figure 2, 3, 4 describe the results. The red line in the each graph describes the amount of used main memory. The blue line describes the amount of used swap area. Figure 2 describes the change of the amount of used main memory and used swap area in Condition A. The maximum memory usage is only 634.1 MiB.

Figure 3 describes the results in Condition B. Condition B is a situation that we add a Container C (mem_eater) to the case in Condition A. The graph shows that a memory shortage occurred. It also shows that the operating system uses much swap area. The average amount of used main memory is 1883.5 MiB: that is more than 90% of the main memory in the target system. The mem_eater program consumed much memory, and the operating system swapped out other containers. The memory pressure by the mem_eater is the cause of the 48.7% performance degradation in Condition B, as mentioned in Section 4.2.

Figure 4 describes the results in Condition C. Condition C is the situation that we apply the proposed method to the situation in Condition B. The graph shows that the operating system used more swap area than in Condition B. The average amount of the used swap area was 1818.5 MiB. In contrast, the maximum amount of the used main

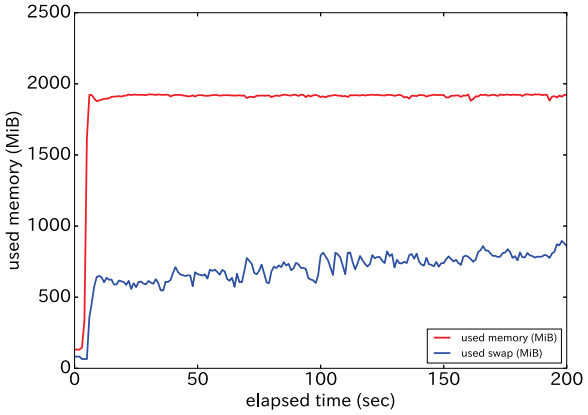

Figure 3. Memory Consumption (Disabled)

memory was 1126.8 MiB: that is 55.0% of the amount of memory area of the container host (2048 MiB). The reason is that the proposed method limits the use of main memory for the detected container, and the operating system assigns the swap area for it. As mentioned in Section 4.2, the request processing performance of the legitimate service (i.e. WordPress and MySQL) did not degrade while we add Container C (mem_eater). It is clear that the proposed method detected Container C as the target of the memory limitation, and prevent the overuse of main memory. Therefore, the proposed method can prevent the overuse of memory based on the behavior of the containers.

In this experiment, for the proof-of-concept, we constructed a container host system for web application hosting, and evaluated the effect of the proposed method. As a result, the proposed method can prevent the degradation of the request processing performance of the legitimate service.

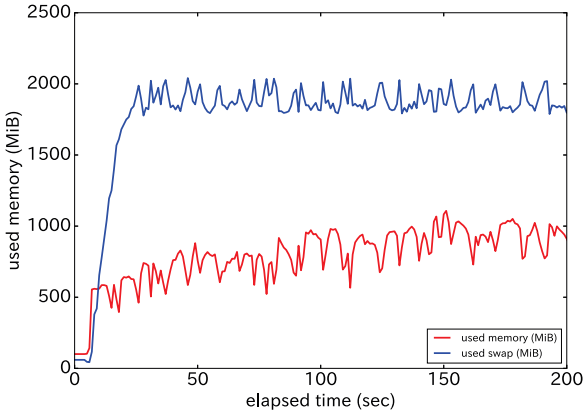

Figure 4. Memory Consumption (Enabled)

5. Conclusion

In this paper, we proposed a new memory management method for container-based virtualization systems. With the proposed method, the operating systems detect a container that overuses memory space based on the memory consumption behavior, and limit the use of memory for the detected container. The result of a proof-of-concept experiment shows that the proposed method can prevent a memory shortage in the situation that an illegal container overuses main memory.

The key parameter of the proposed method is the threshold value to detect memory overuses. In this paper, we use a provisional value as the parameter. The optimal parameter depends on the target system. It is necessary to establish a method to determine the parameter by evaluating the method on various systems. Auto configuration for the parameter is also one of the important challenges.

References

[1] G. Banga, P. Druschel, and J. C. Mogul, "Resource Containers: A New Facility for Resource Management in Server Systems," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, ser. OSDI '99. Berkeley, CA, USA: USENIX Association, 1999, pp. 45–58. [Online]. Available: http://dl.acm.org/citation.cfm?id=296806.296810

[2] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 275–287, 2007. [Online]. Available: http://doi.acm.org/10.1145/1272998.1273025