Aridac: Adaptive Resource Isolation of Non-volatile Devices Under Heterogeneous Containerized Environment Project Interim Report

Xiang Yue
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
Email: xiangyue@andrew.cmu.edu

Xuan Peng
Information Networking Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
Email: xuanpeng@andrew.cmu.edu

Zeyu Wang
Information Networking Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
Email: zeyuwang@cmu.edu

***Abstract*—Container technology makes cloud computing possible by offering resource isolation and scalability, however, resource sharing brings the problem where heavy containers consume most of the resources and break the fairness. To achieve fairness, we plan to propose an adaptive resource isolation policy, Aridac, for adjusting the disk resource quota of containers.**

1. Introduction

Cloud computing platforms, as a typical instance of infrastructure-as-a-service, nowadays have become the first choice for internet application developers to deploy their services. The main reason behind the popularity is - cloud computing offers reliability, convenience, isolation, and scalability, along with the pay-as-you-go pricing model. To achieve those features, the quality of service must be ensured with the achievement of resources isolation technologies.

Container, as the core part of isolation, creates an illusion for customers as owning the entire system. This lightweight virtualization notion has been widely leveraged by cloud service providers. Based on Linux Cgroup, which offers the ability to isolate resource (e.g. namespace, network I/O, disk I/O), container technologies and applications raised, like Linux Container (LXC) [1] and Docker [2]. Those form the basic units of cloud platforms and are still a great domain for cloud providers to dive deep into.

However, behind the great idea of containerization, there come new problems with resources isolation in practice. One of the glaring issues to resolve is resource allocation fairness, which means avoiding the resources allocated to containers interfering with each other. For instance, when there are several containers within a node and one of them is running heavy workloads with intense I/O operations, the resources of other nodes can be consumed, greatly delaying the delivery of tasks, like the completion of file reading, the transmission of RPC requests, etc. In addition, Xavier et al. [3] has also analyzed and confirmed the interference of heavy disk workload without limits on resource allocation can degrade the overall performance of the system. Therefore, to avoid sudden delay of container processes, a limitation of resource quota is a good way to control.

Although there are already some tools (cgroup, trickle) for adjusting resource quota, the problem is, those tools only offer fixed limitations of resource quota, whereas, in industry, cloud providers can never know how to adjust quota in advance or in real-time - the application in containers and containers themselves are dynamically created, killed and changed frequently. In addition, one of our group members met with the same problem with an intolerant latency of services when resources are consumed mostly by a disk-intensive container. Some works [4] manages to alleviate the issue. Since there are still few efforts working on this topic in both academy and industry, our group planned to look into a dynamic way of isolation adjustment.

We plan to propose the Aridac, an Adaptive Resource Isolation algorithm of Non-volatile Devices under heterogeneous containerized environment. The basic idea is to monitor, collect and analyze the resource usage habit of containers as time goes on, and allocate their resources to ensure fairness under certain rules. The quota of disk I/O can be allocated by, for instance, the priority of the application, or by containers' history maximum usage. And the objective is to achieve fairness. Besides the algorithm, we plan to design the testing workloads and design experiments for comparing fairness between different policies with specific metrics. Also, we will provide the corresponding benchmark data for further research. To narrow down the scope, we focus on disk isolation, which can be extended to the network I/O or other resources allocation scenario. The basic tools of system design and experiments include cgroup, bash scripts.

The rest of the proposal is organized as follows: Related Work section shows the relevant works of disk I/O isolation; Methodology section shows how we will define the effectiveness of Aridac; Goals includes our promising results of research; Final paper plan illustrates how we will write the final paper.

2. Related Works

Currently, there are not so many efforts on research of container isolation, especially on disk I/O. However, some researchers interested in disk I/O isolation has some works for us to refer. Merchant et al. [5] focused on the fairness and performance of disk I/O allocation for different applications. He hypothesized there are different performance targets (throughput or I/O directed) for each application, and developed a policy to allocate the resources to them to reach the best overall performance. Arunagiri et al. [6] developed the FAIRIO algorithm to schedule prorated I/O resources for fair allocation, which is a more general solution. Li et al. [7] provided an automatic framework to monitor storage contention and optimize bandwidth utilization. In addtion, for the different scenario on solid state disks, Jo et al. [8] discussed the specific solution to it.

The above ideas give us a good inspiration for container disk I/O isolation. A naive idea is to treat containers as applications and apply the above solutions. However, different from applications, containers' behaviors are more complicated. In this paper, we believe each container has a dynamic usage pattern and is variable from period to period. Therefore, adjusting the policy to allocate resources cannot be avoided.

3. Methodology

3.1. Evaluation

To evaluate the effectiveness of Aridac, we will carry out benchmark experiments under different kinds of workloads. Most experiment settings will be invariants, including the host machine (physical machine), container software, container version, testing programs, etc. The only variant is the isolation policy. One set will use Aridac, while the other one simply does not use any.

We will evaluate Aridac using a variety of workloads. In the final report, we will first describe the workloads and then outline our evaluation methodology. The sketch of our evaluation process includes two parts: evaluating the functionality of Aridac, evaluating the robustness of Aridac to different types of workloads. If we have opportunities, we will also like to test Aridac with real-world workloads.

We will use a variety of workloads generated by our fuzzy IO generator and measure the IOPS and Throughput Performance of the disk IO for each container. And the fuzzy workload generator is a simple disk IO generator implemented with FIO Volume Performance Tests. The test workload will include light, increasing, and bursty load traces for random read, random read/write, and sequential read. So far we are not sure what will be the specific size of the disk area we will access during each workload, but we are considering the possibility that we might have tens of Gigabytes of reads/writes in our test cases to distinguish the functionality and robustness of Aridac from running in the normal container environment.

The functionality evaluation will validate the basic functionality of our implementation of Aridac. We want to test that the Aridac achieves the desired performance differentiation, and correctly differentiates disk IO performance based on the desired disk IO level for different containers. The evaluation of the robustness will be implemented with test cases of increasing workload and bursty workloads from the random read/write.

3.2. System Measurement Metrics

In general, we want the metrics to be correct, precise, and able to take the semantics of different I/O operations into consideration. For example, between the extremely heavy disk write operations introduced by image pulling during deployment in one container, and moderately heavy disk write operation from a critical web service in the other container, we definitely want Aridac to catch difference among those operations, whether through some OS observability methods or manually injected configurations. It's fair to say that this metrics will determine the effectiveness of Aridac.

Since the behavior patterns of containers are changing from time to time, we keep monitoring their patterns and divided each one at any moment into two types of container - throughput directed or I/O directed, and allocate resources to them based on the type. Their types are decided by sampling the resource usage by a fixed interval.

For the metric, whatever type the container is, we manage to satisfy their required target. So their performance metric y_i for i-th container c_i in period t is formulated as follows:

$$y_i(t) = \begin{cases} \frac{\text{throughput}_i(t)}{\text{throughput target}_i} & c_i \text{ is throughput directed} \\ \frac{\text{latency target}_i}{\text{latency}_i(t)} & c_i \text{ is I/O directed} \end{cases}$$

Using this performance metrics, we unify the resources requirements of containers and the extent that they are satisfied. Then we can derive our overall performance **y** at

time t (assume there are N containers):

$$\mathbf{y}(t) = \frac{\sum_i^N y_i(t)}{N}$$

The overall performance measures both resource satisfaction and fairness of resource allocation.

4. Re-iteration of Goals

The overall goal is to finish implementing Aridac, testing it under various kinds of workload scenarios, measuring how much improvement we achieve, and finally, analyzing the overhead brought by Aridac and discussing the trade-offs.

To evaluate Aridac, we could start from the following perspectives:

- The completeness of implementation. i.e., whether the isolator can be applied to the resource contention scenario that we mentioned before.
- To what extent does Aridac help. i.e., how much peak & average system metrics improvement can we gain with Aridac.
- What's the price we need to pay? i.e., how many extra machine resource will be consumed by Aridac.

Here, we set 3 concrete goals in terms of the final progress, which represent 75%, 100%, and 125% degree of completion, respectively.

For the 75% completion goal, our aim is to cover the following aspects:

- Finish implementing Aridac, the adaptive disk resource isolator.
- Make sure that Aridac could run successfully without crashing or bugs.
- Aridac achieve a better system metrics than the non-optimized version.

For the 100% completion goal, we will strive to finely tune the isolation policy in order to achieve an optimal resource sharing among containers, which will be gauged by overall system metrics of the system. We will use the following steps to check whether Aridac has achieved this goal:

- A set of tests carrying different workload will be designed to simulate typical resource sharing situations in real-world datacenter containerized environment.
- Aridac must excel in all test cases that we designed.
- Concrete benchmark data will be given, specifying by how much Aridac is leading in terms of system metrics.

Beyond the 100% goal, if everything goes well, we will further develop a fuzzy disk I/O workload generator, so that all possible real-life cases will be covered, incluing assorted corner cases and rare cases. Our anticipation for this 125% goal is that Aridac could survive through this fuzzy test. We will also carry out benchmarking, and do some statistical work to show how well Aridac could be in a totally random environment.

So far, we've been striving to accomplish the 75% goal (surely there is no way for us to bypass a 75% goal and reach a 100% directly). That being said, much effort has been put in the exploration of the feasibility and concrete logic of the implementation.

There are several major issues concerning the implementation, which we must consider thoroughly in order to get the implementation delivered. In the following subsections we will discuss our progress on those issues accordingly.

4.1. Existing Isolation Techniques

It's unrealistic to implement disk I/O isolator from scratch within a limited time, which requires tons of OS domain knowledge. Luckily, the topic of resource isolation has been cahtching people's attention since a long time ago. Both the open source community and the academia achieved a lot so far. Specifically, linux kernel introduced cgroups (abbreviated from control groups) in its early versions, which serves as a feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes. And just a couple of years ago, the linux kernel upgrade the cgroups to v2 with a more refined management hierarchichy and other features. Further, its important to know that most high-level container runtimes (Containerd, Docker, Podman, and Kubernetes) are now capable of fully supporting cgroups v2.

With cgroups, we can assign those processes that are running in containers into different resource groups and then adjust the resource quota of each group dynamically through the OS native cgroups operations.

4.2. Disk I/O Simulator

Another major challenge we have encountered during the exloration is how to correctly simulate the different levels of Disk I/O that satisfies what we need. Originally, our plan is to manually write program in C language that are directly built upon system calls like open(), read(), write(), and lseek(), etc. However, after several trials, we found it very difficult to achieve what we want. It's complex in that we need to know both the domain knowledge of those system calls and the machine's hardware metrics very well.

To ease the burden, we investigated existing disk I/O generators, and we found FIO (Flexible IO) as a simple yet comprehensive I/O generator. And it satisfies all the demands that we have had so far. FIO can generate various kinds of I/O patterns, including IOPS intensive, throughput intensive, and latency intensive. It also provides both read and write generation. Overall we think it shall be very useful if we combine customized load generation logic with it.

4.3. System Observability

On thing to note here is: we need to monitor the hosted processes as the containers keep running, so that we can gather the machine-level metrics in order to adjust the isolation quota & policy accordingly. There is a list of machine-level metrics we may need to care for every process, including the IOPS, throughput, and latency.

So far, we use Atop as the tool to get the runtime machine metrics. We are still not 100% sure if it can live up to our expectation, since our requirement may possibly change as our work progresses. We will keep the update posted in the section if anything changes.

We experimented the above-mentioned existing techniques & tools in practice, and it's proven to be a viable way to implement Aridac. We created our dev & test environment on an Ubuntu 20.04 instance of Microsoft Azure cloud service, on which cgroup 2.0 has been adopted. We used FIO to generate several different levels of I/O workload as the test samples. We launched the workloads in containers and we observed the machine metrics through the Atop tool running on the host machine.

As a next step, we will seek to automate the whole process by collecting the machine metrics through program, and calculate the system metrics dynamically. After having the system metrics component integrated into Aridac, we can then exploit cgroup technique to adjust the quota according the metrics we calculated.

5. Final Paper Plan

In our final paper, we will first introduce the resource isolation problem of the container environment and discuss the background and related work. Next, we will briefly outline the system design and implementation of Aridac. Last, we will describe our experimental setup, including what machines and environments were used, what workload we developed to test our implementation. The most crucial part of the paper will be concerned with describing and evaluating our methods for container resource isolation. We will provide performance analyses, and conclude with a discussion about potential optimization schemes, and if possible, a set of benchmarks for more detailed performance evaluation.

The early stage of our project will be analyzing existing issues caused by weak isolation of the container environment, researching existing relative works on different levels of virtualization technologies and container resource isolation, and finding out a doable logic for performing a dynamic resource isolation scheme in the container environment. Our project will then focus on developing a dynamic resource isolation policy for the container environment, to achieve the goal of balancing the resources among multiple containers running on the same physical machine. We will implement a program working on dynamic resource allocation and isolation for containers sharing the same underly resources, and may try out different policies for resource allocation of network bandwidth and disk I/O of the physical machine to learn their performance characteristics.

We will design and develop a set of workloads to simulate some most common applications running in the industry. Then we will let a few containers run that workload and monitor their resource and performance, to see if any rapid resource obtaining will happen and broke the resource allocation balancing. If we can reproduce the scenario of container resource exhausting caused by weak isolation, then we may follow up by testing whether it strengthens the resource isolation among containers and result in better overall performance.

To measure the success of our project, we will focus on completeness rather than performance. But we will examine if there are any possible optimization for the implementation and leave it as potential future works. Other possible topics include comparisons between static isolation approaches and our dynamic isolation approach, discussing the pros and cons of using different levels of encapsulations and virtualization technologies, and even recursive virtualization for more stable overall performance. We will also try to figure out the weight of different resources in the container environment, discussing trade-offs and complexities. Hopefully, the results and discussions will help us know the effectiveness and generality of the dynamic isolation approach for various working scenarios.

Here are some questions that our experiments will answer:

What kinds of scenarios will require strong resource isolation for applications running in the container environment?

Is dynamic resource allocation and isolation beneficial for the overall performance of applications running in the container environment?

What are the appropriate resource allocation approaches for applications with different characteristics?

How strong is the isolation provided by the dynamic isolation and allocation approach?

How stabilized has the performance of the container environment been achieved after adopting stronger isolation?

What is the overall overhead of our approach?

5.1. Further Schedule

By the date this interim report was written, we have made some achievement in the aspects that are mentioned in section 4.1-4.3. Below is a tentative schedule for the future work.

- By 15th April, we plan to achieve the 75% goal. We will finish the 3 essential components of Aridac, which are the load generation module, the runtime machine metrics monitor module, and the dynamic resource isolation module. And we will make sure that Aridac achieves a better performance in a several given typical tests.
- By 30th April, we plan to achieve the 100% goal, which seeks to build a comprehensive collection of typical workloads under containerized environment. And Aridac will excel in all of those tests, with concrete benchmark given in the report.

References

[1] "Linux containers." [Online]. Available: http://lxc.sourceforge.net

[2] "Docker." [Online]. Available: https://www.docker.com/

[3] M. G. Xavier, I. C. D. Oliveira, F. D. Rossi, R. D. D. Passos, K. J. Matteussi, and C. A. F. D. Rose, "A performance isolation analysis of disk-intensive workloads on container-based clouds," *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 253–260, 2015.

[4] S. Ahn, K. La, and J. Kim, "Improving I/O resource sharing of linux cgroup for NVMe SSDs on multi-core systems," in *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[5] A. Merchant, M. Uysal, P. Padala, X. Zhu, S. Singhal, and K. G. Shin, "Maestro: quality-of-service in large disk arrays," in *ICAC '11*, 2011.

[6] S. Arunagiri, Y. Kwok, P. J. Teller, R. Portillo, and S. R. Seelam, "Fairio: An algorithm for differentiated i/o performance," *2011 23rd International Symposium on Computer Architecture and High Performance Computing*, pp. 88–95, 2011.

[7] Y. Li, X. Lu, E. L. Miller, and D. D. E. Long, "Ascar: Automating contention management for high-performance storage systems," *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–16, 2015.

[8] M. H. Jo and W. W. Ro, "Dynamic load balancing of dispatch scheduling for solid state disks," *IEEE Transactions on Computers*, vol. 66, pp. 1034–1047, 2017.