Offloading Computation from Cloud Virtual Containers to the Student's Host Machine

Brandon Bae, Elaine Guo
Duke University, April 21, 2023

I. INTRODUCTION

Over the years, we have seen a dramatic increase in virtualization technology, from the workplace to academic settings. One of the main examples of such are software containers, a unit of software that contains all of the code and dependencies an application needs to run between computing environments while sharing the host OS kernel. With the advancement of cloud technology, containers are now being utilized to provide fully functional and portable computing environments. Companies such as Amazon Web Services, Microsoft Azure, Google Cloud Platform and IBM Cloud have all adopted containerization technology, understanding its extremely beneficial use for cloud computing. Containers are incredibly lightweight and are fast to modify and iterate on and container technology has a thriving ecosystem (What[13]).

Despite the many advantages that containerization brings, a container will experience slow performance due to inadequately allocated resources (Aleksic[12]). When containers do not have access to enough resources, their performance suffers. Depending on the average image size of a project and the number of containers a host is running, the host and network needs to be able to support the workload. This phenomenon can be seen in Duke OIT's management and partitioning of virtual software containers due to the drag in container performance whenever multiple students attempt to use the same container simultaneously.

This work proposes a kind of offloading where the virtual software container offloads intense computing processes to the host computer in order to increase efficiency and reduce processing stall time. Since students will be running containers allocated by Duke OIT on their own machine, their own machine should be able to handle the processing load passed from the container in the event that the container's performance is low enough to deem it necessary. Based on the results of this research, this work may lead to new software technology that promotes the offloading of work from a container to the host machine when the container is being run on a separate machine, creating a new paradigm for container use.

II. PROPOSED SOLUTION

This work explores an application that offloads cloud based containers to local edge devices such as laptops or desktops in order to reduce perceived latency for clients and increase performance for the cloud. In smaller computing clusters, it is feasible for a non-substantial amount of containers to lead to resource contention which can lead to increased latency for clients. In order to address this situation, we will explore the potential of offloading these cloud based containers to local devices. Figure 1 presents the overall design of the system. The computing cluster hosting the containers will run a resource monitoring application in order to predict when resource contention will occur. A possible implementation of this application could be to set a threshold for RAM or CPU usage which when crossed, triggers an alert. When resource contention is triggered, the application then alerts clients of potential latency increases and asks if they would like to offload their container to their local device. If the client chooses to offload, the application closes the client's container on the cloud and redeploys the container on the client's device with a shared file system.

With this design, we predict that handling failures will be straightforward. For example, we predict that we will not need to implement any specific protocols to handle failures of locally deployed containers as any change of state created by the client's work should be reflected and saved in the shared file system. In the case of the shared file system failing, our application will simply close the locally deployed container to prevent desynchronization between the local and cloud container states and allow the client to redeploy their container on the cloud.

Finally, our design seeks to benefit Duke systems by improving latency issues seen on the Duke Container Manager service. The Duke Container service provides students with a remote desktop container with specific applications and software preinstalled for use in conjunction with classes. However, there can be considerable slowdown in these services when many students use the same type of container at once such as during recitations or before homework deadlines. This is a unique situation as a smaller computing cluster is hosting many different computationally small containers. This makes it ideal to offload the containers to the client's local device as the work done on the containers in most instances can run on current generation laptops or computers. Therefore, by offloading containers to local client devices, we free up the underlying compute cluster to handle existing cloud deployed containers or service new clients as well.

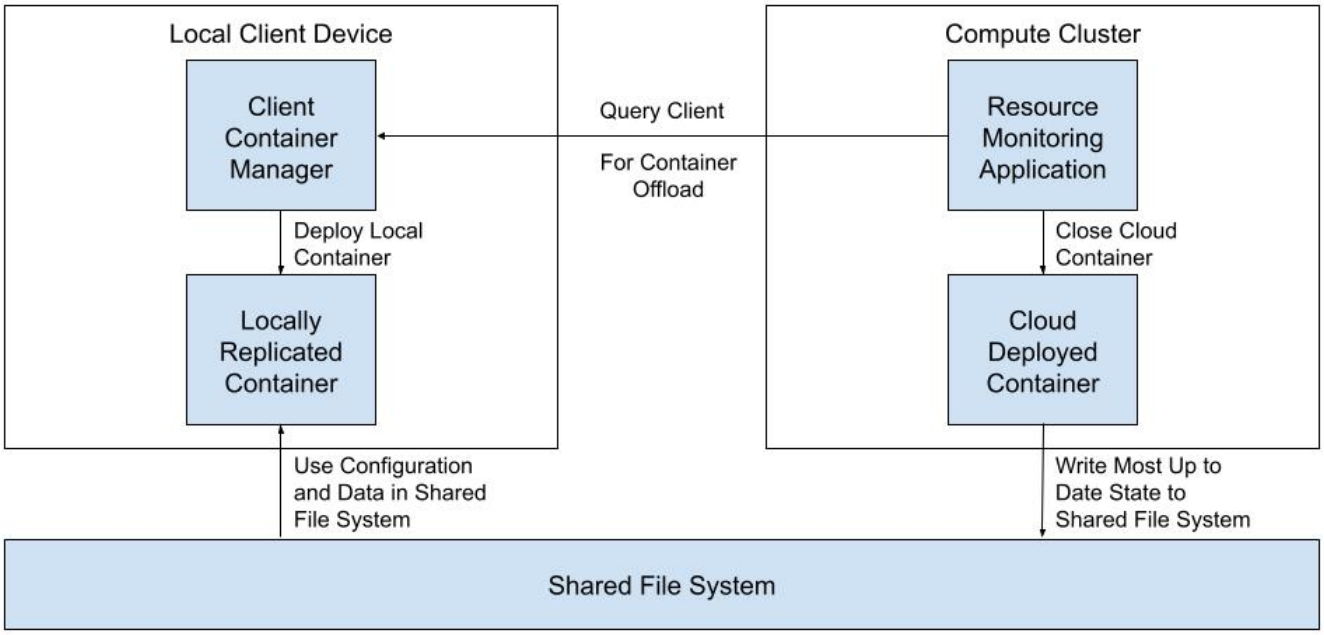

Fig. 1. Offloading Solution Design Overview

III. RELATED WORK

The use of virtual machines for offloading intensive processing tasks is a widely researched topic in order to relieve the computational pressure of host devices. Guo[1] proposes a task offloading strategy with real-time VM repair in an edge computing system, and establishes a repairable queueing model with multiple servers and VM-dependent failure rates. The use of virtual machines as resources to offload computational tasks has been increasingly explored in practice and in search. Cheng[3] introduces engineering and research trends of achieving efficient VM management in edge computing and proposes hBalance, a system designed to offload functionality from SMP virtual machines to the hypervisor, and found that hBalance significantly improves the SMP virtual machines' IO performance. Furthermore, there already exists a precedent for offloading computation from larger servers to edge devices to reduce latency, Wang[6] investigates key concepts related to mobile offloading and draws an overall "big picture" on the existing efforts and research directions.

As such, the techniques used to migrate states and data from one machine to another is also a greatly discussed topic as Mishra[5] presents details of virtual machine migration techniques and components required to use virtual machine migration for dynamic resource management in the virtualized cloud environment. Wu[9] discusses important metrics to consider when live migrating VMs, specifically resource availability and uses performance models for live migration to predict a VM's migration behavior. Goudarzi[10] looks to create multiple copies of VMs on different servers and distribute requests among these copies to reduce resource requirements for each copy.

Modeling and benchmarking of virtualization technology is also important to the functionality of virtual machines; Tickoo[2] examines the challenges of modeling virtual machine performance and uses benchmarking tools to outline key issues that VM modeling faces. Tao[4] outlines virtualization frameworks and virtualization techniques for edge computing.

Other than virtual machines, there have been other methods utilized to offload processing to other devices. Nygren[7] explains how content delivery networks (CDN) also try to reduce latency between cloud and end users by running on servers that are close to end users, specifically offloading load from origin servers. Hylson[8] discusses how Kubernetes, a container management system, conducts state machine replication and explores the idea of restoring state within containers after failing.

Our work is different from previous research because previous research focuses on offloading computation from a local edge device to cloud/cloudlets, which makes the assumption that the cloud/cloudlets are very large and have nearly infinite resources while the local edge device and very limited and constrained in the computing power it has. In contrast, our work focuses on offloading computation from a computing cluster (similar to a cloud) to a local edge device, making an inverse assumption that

the computing cluster is relatively small and limited in resources while the local edge device is powerful enough to run intensive computing tasks.

IV. ARCHITECTURE OVERVIEW

At a high level we developed two separate applications called the Resource Monitoring Application and the Client Container Manager which run on the cloud compute cluster and local client device respectively. These applications both have the responsibility of starting, maintaining, and stopping containers for our system while also communicating with each other using a TCP connection in order to relay information such as container status, container ID's, and resource thresholds. In order to deploy and maintain containers on both the cloud and local device, we utilize Docker which allows us to easily do mass deployment of containers. Finally, we utilize a shared file system that both the cloud and local devices will be connected to in order to maintain container state during the offloading process. This allows user containers on both the cloud and their local devices to have the same file system.

A. RESOURCE MONITORING APPLICATION

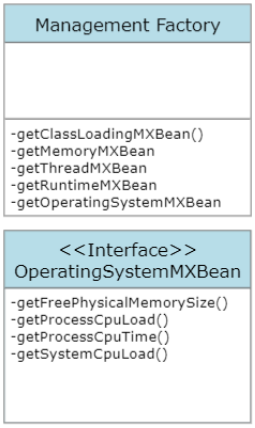

Fig. 2. UML Card of Resource Monitoring Application

In order for our system to determine when to offload computation, a resource monitoring application is deployed on the cloud compute cluster. The main purpose of this component is to monitor for resource contention by keeping track of key resources (total memory and CPU) and checking to see if their usage passes a certain threshold. If the application detects that this has occurred, it then queries the user, letting them know that the cluster's resources are limited and gives them the option to consequently offload whatever computation is being done on the cluster to their local edge device.

To implement the resource monitoring application, we utilized the Java Management Factory class, which is able to return specific MXBean interfaces based on what is requested. We specifically used the OperatingSystemMXBean, which is able to poll the operating system to obtain metrics on CPU and Memory Usage. The UML card of the class and interface used can be seen in Figure 2. We call the Operations part of the OperatingSystemMXBean interface at a regular interval and check if the results cross our set thresholds.

B. CLIENT CONTAINER MANAGER

Fig. 3. UML Card of Client Container Manager Application

Our system also deploys a client container manager application on every client's local device in order to maintain communication between the local device and the cloud. Each local container does this by maintaining a TCP connection with the resource monitoring application on the cloud compute cluster. This TCP connection is used to trigger operations between the two modules which can query users, launch containers, and close containers. The operations that the client container manager can run are seen in Figure 3 where we show the application's UML card.

The client container manager also has the responsibility of starting and maintaining any local containers when offloading occurs. When the client container manager receives an offload request, it will assume the cloud container has been closed by the resource monitoring application and start a container with the same image as the client's cloud container using the same name space in the shared file system.

It is important to note that this application will be the main interface users will use to interact with our system. In our current implementation the user can send commands through the command line to initiate operations such as starting the cloud container or closing the cloud container instance.

C. SHARED FILE SYSTEM

In order to maintain state between the local and cloud containers through the offloading process, we utilize a shared file system that both the local and cloud containers utilize. At a high level the shared file system allows the local and cloud container to share and persist the same file system even after one or both containers has been disconnected. We take advantage of the persisting nature of files in the shared file system to maintain container state. For example, consider a situation where a user was

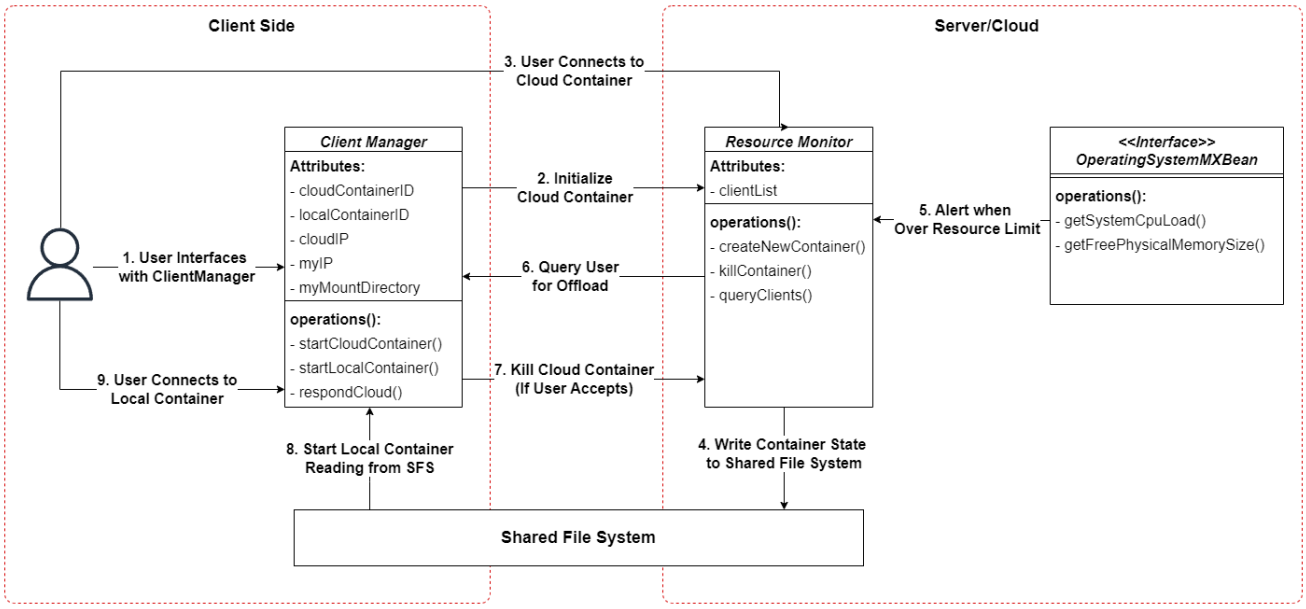

Fig. 4. System Workflow Diagram

writing to a text file when the offloading workflow begins. As long as the user saves any progress they have before the offloading occurs, they will be able to see the file along with any changes they made in the local container as well. Without this module, the local and cloud containers would be separate instances of the same container image with no method of saving work or container state during the offloading process.

We decided to use NFS (Network File Share), a network file sharing protocol, to implement that shared file system between the cloud and local container deployments; While initially we were considering SSHFS as our shared file system protocol due to its ease of use, we decided against utilizing this in our system. This is motivated by research which points to SSHFS being extremely network intensive due to the need to encrypt communication. Not only this, but NFS was built around supporting concurrent accesses through the implementation of built in locking which can prevent the corruption of the shared file system.

It is important to note the limitations of using the shared file system as a way to maintain state. Given that the only module shared between the local and cloud container is the shared file system, anything not on the file system will be lost during the offloading process. Most notably, anything in memory or any running applications at the time of the offload will not be reflected in the local container.

V. SYSTEM WORKFLOW

Here we provide the general workflow for our system (as seen in Figure 4) starting from the initialization of the client's cloud container to offloading to the client's local device:

1) User Interfaces with ClientManager: As mentioned earlier the user will use the startCloudContainer() operation to begin the workflow to create their cloud container.
2) Initialize Cloud Container: The client container manager requests the resource monitoring application to use its createNewContainer() operation over the TCP connection to create the client's cloud container. The client container manager then returns the cloud container's ID to the user.
3) User Connects to Cloud Container: The user utilizes the cloud container's ID to ssh to the compute cluster and connect to the appropriate container.
4) Write Container State to SFS: As the user works on the cloud container, any changes they wish to save are written to the shared file system.
5) Alert when Over Resource Limit: Periodically the resource monitor polls the OSMXBean module to check current compute cluster resource usage. When the resource limit is crossed the resource monitor starts the offloading process.
6) Query User for Offload: The resource monitor queries the user for confirmation on whether to start the offloading process (due to the potential of losing current running applications). This is facilitated through the client manager.
7) Kill Cloud Container: If the user accepts the offload query, the client container manager calls the resource monitor's killContainer() operation to kill the client's cloud container.
8) Start Local Container: As the cloud container is being killed, the client container manager starts a new instance of the user's container on the client's

local device. The shared file system is also mounted to maintain state between the containers. This operation also returns the local container's ID to the user.

9) User Connects to Local Container: The user uses the returned container ID from the previous step to connect to the local container and resume their work.

VI. CURRENT LIMITATIONS/ASSUMPTIONS

Due to time and complexity constraints there are some limitations and assumptions our system currently has. As mentioned previously the most notable limitation that we have with the current system is the inability to support live migrations of containers where a container's memory, disk, and running applications all maintain state after the offloading process. In our current system we only maintain state through the shared file system and only maintain the state of files within the file system during offload. One potential work around could be to write a temporary file to the shared file system describing the state of memory and running applications before offloading and then reading the temporary file to set the state of memory and running applications on the local container. Additionally, both the client and cloud devices need to be running a Linux operating system. This is mainly due to limitations introduced by Docker which requires Linux to run. Since we use Docker as our main method of maintaining containers, our system also must abide by this limitation as well.

Finally, we currently do not have an automatic method of mounting the shared file system to the user's local file system. Therefore, we currently rely on users to manually mount the shared file system using NFS to the same directory as our program to properly run the system.

VII. EVALUATION SETUP

In order to test our hypothesis, we used sysbench, an open-source scriptable multi-threaded benchmarking program for Linux systems, to measure the resources and tasks the container is able to run. The testing environment is made of a Duke hosted Linux VM, which simulates a compute cluster. This VM deployed a varying number of docker containers that were running the benchmark testing program in parallel in order to simulate multiple people running their own container from the computing cluster.

Given that most research on offloading has been done on offloading from edge devices to higher level compute clusters, we were unable to find any suitable solutions to compare our results with. In lieu of this, we will compare our solution with a normal cloud deployed container. Thus, the performance of one container running with the benchmarking program was compared to the performance of multiple containers running the benchmarking program. This helps us identify any overhead that our solution may incur on both the client and the compute cluster while also providing us with a baseline solution to compare any performance increases with.

For our evaluation, we focused on three main benchmarking tests for our system: CPU, Memory, and File IO.

A. CPU TESTS

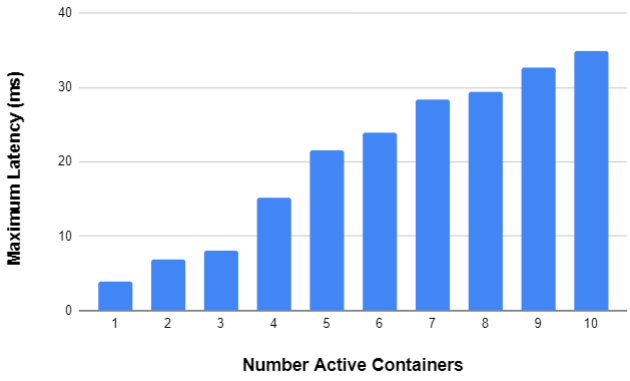

Fig. 5. Maximum Latency vs. Number of Active Containers

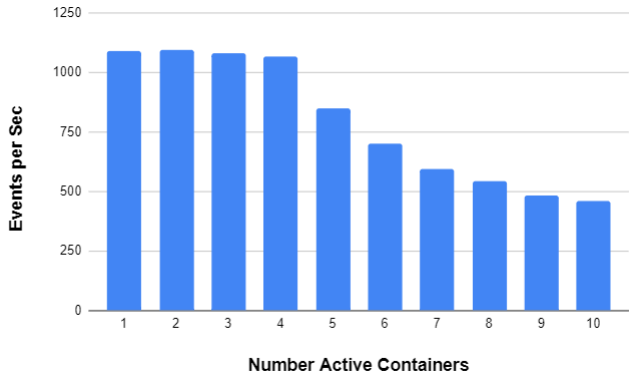

Fig. 6. CPU Events Per Sec vs. Number of Active Containers

In order to test CPU performance of a particular container, the test measured how the container performed while running a computationally intensive program; in this case, the test generated one thread that ran a program to find the first 10,000 prime numbers.

The main metrics we focused on were the number of CPU events per second, which is the rate of the container's CPU events being processed, and the maximum latency between CPU events. The test results can be seen in Figures 5 and 6.

Figure 5 shows that as the number of active containers increases, so does the maximum latency between CPU events being processed in a linear manner. This can be explained by the simultaneous increasing amount of CPU events to schedule, meaning that a specific event for a container would have to wait in the queue longer to be run. Figure 6 shows that as the number of active containers increases, the number of CPU events per second decreases in an exponential manner. It is interesting to

note that the number of CPU events per second is stable for up to four containers because the VM being used has four processors, meaning that four threads that each processor has can handle the extra load of CPU events that a new container generates.

B. MEMORY TESTS

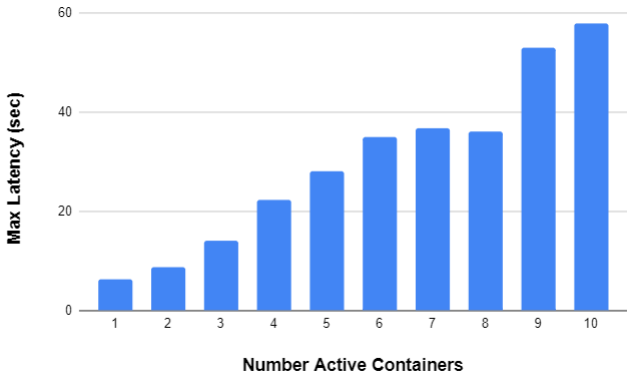

Fig. 7. Maximum Latency vs. Number of Active Containers

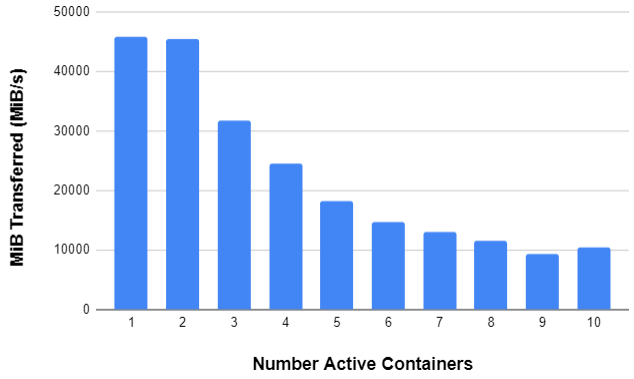

Fig. 8. MiB Transferred vs. Number of Active Containers

In order to test the memory performance of a particular container, the test shows how effective a container is at reading and writing to memory; specifically, the test first allocates a memory buffer and then reads or writes from it until the total buffer size has been read from or written to.

The main metrics we focused on was the total amount of mebibytes (MiB) transferred, or the throughput, and the maximum latency between each read and write. The test results can be seen in figure 7 and 8.

Figure 7 shows how the maximum latency between reads and writes increases in a linear-like manner as the number of active containers also increases. The results also indicate that as the number of active containers increases, the maximum latency will be stable for a certain range of active containers and then increase significantly once another container is added to the workload. Figure 8 shows how the throughput of each container decreases as the number of active containers decreases in an exponential manner. This is to be expected since the addition of more containers results in a decrease of time and resources each container has to read and write to memory effectively, but the impact of adding more containers lessens as the number of containers increases.

C. FILE IO TESTS

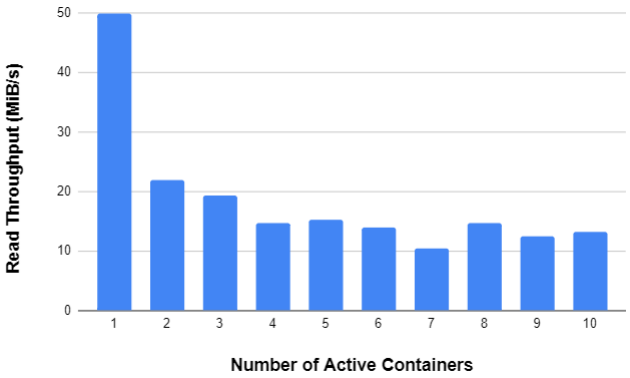

Fig. 9. Read Throughput vs Number of Active Containers

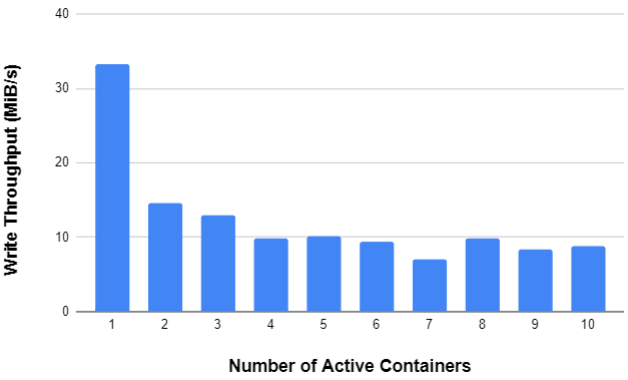

Fig. 10. Write Throughput vs Number of Active Containers

In order to test the file IO scaling of our system, we used the sysbench default file IO test. This test creates 128 files each 16 MiB which are then accessed randomly by all active containers in a parallel manner. In this evaluation we measured the average read and write throughput in MiB/s experienced by the file system. It is important to note that all the test files were created on and accessed through the shared file system which all active containers were connected to in order to simulate our actual system.

As seen through Figure 9 and Figure 10, both the write and read throughputs experience exponential decay with a sharp decline seen after going from one to two active containers for both throughputs. This falls largely in line with expectations as the more containers there are, the higher the chance of containers competing over shared IO resources such as a lock on a file. However, an interesting observation is that both the read and write

throughput stabilize at around 13 MiB/s and 8 MiB/s respectively. While we are not completely sure why this phenomenon occurs, we did observe a pattern in test completion between containers. We observed that when we started the tests on all active containers in parallel, the containers finished the test in a staggered manner with around 2 to 4 containers finishing at a time. A potential explanation of this could be that the file system IO might have a minimum threshold for its write and read throughput and maintains this by only addressing the IO needs of a certain number of containers at a time.

Overall, these results show that file IO will be a major bottleneck for our system. Unlike other parts of our system, the file IO cannot be offloaded to the local client device as all local client devices will have to be connected to the same file system in order to maintain state. This means that the exponential decrease in file IO throughput is something that cannot be avoided. However, our observation of stabilizing read and write throughput does show that there is a lower bound on file IO throughput which means that it cannot completely stall our system when there is a scaling amount of containers.

VIII. DISCUSSION

Our work provides a novel method of offloading cloud containers onto local devices in order to prevent resource contention on a cloud compute cluster. The primary objective of this study was to evaluate the feasibility and potential performance improvements of offloading containers to local devices.

We made the hypothesis that cloud container performance slowdowns were a result of resource contention amongst too many containers and that offloading these containers to the client's local device during periods of high latency would lead to perceived performance improvements for the client. As seen by our results in the previous evaluation sections we proved that increasing the number of containers with running workloads on the same compute cluster leads to significant decreases to CPU, memory, and disk IO performance. Additionally, our results also show that having one container workload running on a device has the best results for performance. Thus we prove that offloading container workloads not only improve performance benchmarks for CPU and memory but also that running a singular workload on a local device would provide better performance as well. Unfortunately the performance increases described above would only be seen for CPU and memory benchmarks as the disk is a shared resource between all local and cloud container instances due to the shared file system.

It is important to identify the appropriate environment in which our offloading solution would provide the biggest performance gains. Most traditional big name cloud service providers have vast amounts of compute resources with extensive fault tolerance and replicas. Due to this, most users can work with these cloud resources with the assumption that their cloud compute cluster will never run out of resources which often leads to extremely intensive workloads. These types of environments are not suitable for our system as the intensive workloads would most likely be unable to run on just the user's local device, while the extensive amount of cloud resources would minimize the need for offloading from the cloud. However, for smaller cloud deployments as seen in places such as universities or smaller companies, we cannot make the same assumptions. In these environments cloud compute clusters have significantly less resources to partition to users. At the same time, the workload these clusters handle are often smaller as well. This is the perfect environment to utilize our system as the workloads are often able to be run on a client's local device while offloaded containers would lead to more noticeable performance increases in the cloud compute cluster.

One potential application that we believe our system would be well suited for is the Duke OIT Virtual Container Service. Duke's container service fits the parameters of the smaller cloud deployment environment described earlier and thus experiences the shortcomings of such an environment. For example, the CS 250 class has a Linux container with preinstalled programs to provide students a standard environment to write and test code for the class. However, during peak usage hours such as before assignment due dates, these containers experience extreme performance degradation which points to resource contention from too many concurrent container workloads. Not only this, Duke containers are often resource restricted to be well within or below the specifications of modern day laptops thus the workload run on these containers would be able to run on most students' laptops as well. Due to the combination of prevalent resource contention issues as well as small workloads, our offloading system would be perfect at increasing performance for Duke's Virtual Container Service.

IX. FUTURE IMPROVEMENTS/EXTENSIONS

While we have developed a basic architecture for our offloading system, there are a variety of future improvements and extensions that can be made. One key weakness of our system is the lack of true live container offloading where running applications and memory is not lost. In order to address this issue we can potentially look into VM live migration techniques [9]. For example, one potential solution could be to implement suspend-copy-resume based migration where the container's entire state is dumped to persistent storage to be read by the new replica.

Additionally, we can look into adding functionality to other OS's outside of Linux in order to make our system more flexible. As the main blocker preventing us from using our system is the limitations of Docker we can look at providing a wrapper to our Docker module to allow it to function in other OS's. For example, we can look at running the Docker containers in Linux VM's so

that Docker can function properly no matter the host machine's OS.

Finally, there is the issue of security which we have not considered in this paper. As our communication between the client and the cloud compute cluster is maintained through TCP, it is possible for malicious actors to intercept communication and find out information such as the user's container ID. This could be prevented by implementing a more secure communication protocol such as TLS which adds an additional layer of security on top of the TCP/IP protocol. Not only this, but it would be important to abstract out the need for users to ssh into the compute cluster and connect to the container themselves to prevent clients from having access to the compute cluster directly.

X. CONCLUSION

In conclusion, we proposed and tested a solution to the problem of over allocation of the resources of a cloud/computing cluster, which leads to an increase in latency and a decrease in performance for reserved containers. This issue is commonly experienced by Duke students using Duke OIT-provided virtual containers to do class work; when a particular container is being heavily used, each user faces unsatisfactory performance. Thus, our solution involves offloading the computation from a cloud-allocated container to one on the host's local machine.

Through conducting a suite of benchmarking tests, we evaluated the performance of our proposed solution using a simulation of a computing cluster. Overall, we find that our solution drastically improves the performance of a container that has been offloaded versus when it is still using heavily contested resources from the computing cluster.

This provides a promising solution to help alleviate resource contention for heavily trafficked cloud or computing clusters. While there are still some future improvements and work that needs to be done for our product, our design has the potential to be extremely useful towards developing the efficiency of how clouds and computing clusters deploy and allocate containers.

REFERENCES

[1] Guo, X., Du, Z. Jin, S. Nash equilibrium and social optimization of a task offloading strategy with real-time virtual machine repair in an edge computing system. Cluster Comput 25, 3785–3797 (2022). https://doi.org/10.1007/s10586-022-03603-5

[2] Omesh Tickoo, Ravi Iyer, Ramesh Illikkal, and Don Newell. 2010. Modeling virtual machine performance: challenges and approaches. SIGMETRICS Perform. Eval. Rev. 37, 3 (December 2009), 55–60. https://doi.org/10.1145/1710115.1710126

[3] L. Cheng and F. C. M. Lau, "Offloading Interrupt Load Balancing from SMP Virtual Machines to the Hypervisor," in IEEE Transactions on Parallel and Distributed Systems, vol. 27, no. 11, pp. 3298-3310, 1 Nov. 2016, doi: 10.1109/TPDS.2016.2537804.

[4] Z. Tao et al., "A Survey of Virtual Machine Management in Edge Computing," in Proceedings of the IEEE, vol. 107, no. 8, pp. 1482-1499, Aug. 2019, doi: 10.1109/JPROC.2019.2927919.

[5] M. Mishra, A. Das, P. Kulkarni and A. Sahoo, "Dynamic resource management using virtual machine migrations," in IEEE Communications Magazine, vol. 50, no. 9, pp. 34-40, September 2012, doi: 10.1109/MCOM.2012.6295709.

[6] Jianyu Wang, Jianli Pan, Flavio Esposito, Prasad Calyam, Zhicheng Yang, and Prasant Mohapatra. 2019. Edge Cloud Offloading Algorithms: Issues, Methods, and Perspectives. ACM Comput. Surv. 52, 1, Article 2 (January 2020), 23 pages. https://doi.org/10.1145/3284387

[7] Erik Nygren, Ramesh K. Sitaraman, and Jennifer Sun. 2010. The Akamai network: a platform for high-performance internet applications. SIGOPS Oper. Syst. Rev. 44, 3 (July 2010), 2–19. https://doi.org/10.1145/1842733.1842736

[8] Hylson V. Netto, Lau Cheuk Lung, Miguel Correia, Aldelir Fernando Luiz, Luciana Moreira Sá de Souza, State machine replication in containers managed by Kubernetes, Journal of Systems Architecture, Volume 73, 2017, Pages 53-59, ISSN 1383-7621, https://doi.org/10.1016/j.sysarc.2016.12.007.

[9] Y. Wu and M. Zhao, "Performance Modeling of Virtual Machine Live Migration," 2011 IEEE 4th International Conference on Cloud Computing, Washington, DC, USA, 2011, pp. 492-499, doi: 10.1109/CLOUD.2011.109.

[10] H. Goudarzi and M. Pedram, "Energy-Efficient Virtual Machine Replication and Placement in a Cloud Computing System," 2012 IEEE Fifth International Conference on Cloud Computing, Honolulu, HI, USA, 2012, pp. 750-757, doi: 10.1109/CLOUD.2012.107.

[11] Buchanan, Ian. "Containers vs Virtual Machines." Atlassian, Atlassian, https://www.atlassian.com/microservices/cloud-computing/containers-vs-vms.

[12] Aleksic, Marko. "How to Optimize Docker Performance." How to Optimize Docker Performance, PhoenixNAP, 7 Dec. 2022, https://phoenixnap.com/kb/docker-optimize-performance.

[13] What Are Containers?, IBM, https://www.ibm.com/topics/containers.