TYPE Original Research
PUBLISHED 17 February 2025
DOI 10.3389/fhpcp.2025.1499519

OPEN ACCESS

EDITED BY
Artur Podobas,
Royal Institute of Technology, Sweden

REVIEWED BY
Franco Rino Davoli,
University of Genoa, Italy
Fangming Liu,
Huazhong University of Science and Technology, China

*CORRESPONDENCE
Sean Choi
✉ sean.choi@scu.edu

RECEIVED 21 September 2024
ACCEPTED 20 January 2025
PUBLISHED 17 February 2025

CITATION
Galvankar S and Choi S (2025) AlphaBoot: accelerated container cold start using SmartNICs.
Front. High Perform. Comput. 3:1499519.
doi: 10.3389/fhpcp.2025.1499519

COPYRIGHT
© 2025 Galvankar and Choi. This is an open-access article distributed under the terms of the Creative Commons Attribution License (CC BY). The use, distribution or reproduction in other forums is permitted, provided the original author(s) and the copyright owner(s) are credited and that the original publication in this journal is cited, in accordance with accepted academic practice. No use, distribution or reproduction is permitted which does not comply with these terms.

AlphaBoot: accelerated container cold start using SmartNICs

Shaunak Galvankar and Sean Choi*

Cloud Lab, Department of Computer Science and Engineering, Santa Clara University, Santa Clara, CA, United States

Scalability and flexibility of modern cloud application can be mainly attributed to virtual machines (VMs) and containers, where virtual machines are isolated operating systems that run on a hypervisor while containers are lightweight isolated processes that share the Host OS kernel. To achieve the scalability and flexibility required for modern cloud applications, each bare-metal server in the data center often houses multiple virtual machines, each of which runs multiple containers and multiple containerized applications that often share the same set of libraries and code, often referred to as images. However, while container frameworks are optimized for sharing images within a single VM, sharing images across multiple VMs, even if the VMs are within the same bare-metal server, is nearly non-existent due to the nature of VM isolation, leading to repetitive downloads, causing redundant added network traffic and latency. This work aims to resolve this problem by utilizing SmartNICs, which are specialized network hardware that provide hardware acceleration and offload capabilities for networking tasks, to optimize image retrieval and sharing between containers across multiple VMs on the same server. The method proposed in this work shows promise in cutting down container cold start time by up to 92%, reducing network traffic by 99.9%. Furthermore, the result is even more promising as the performance benefit is directly proportional to the number of VMs in a server that concurrently seek the same image, which guarantees increased efficiency as bare metal machine specifications improve.

KEYWORDS

SmartNICs, cloud computing, containers, data center networks, virtual machines

1 Introduction

Container framework is a suite of software that enables user custom workloads to be packaged as small, self-contained, fine-grained custom programs/executables, commonly referred to as container image, and be deployed efficiently in a dynamic fashion into a running container. Attributed by the recent rise of cloud computing, container frameworks have gained significant traction by enabling programmers to focus solely on their core applications rather dealing with the intricacies of underlying infrastructure, such as software packaging, dynamic auto-scaling, dependency management and much more. To elaborate, one of the major benefits of containers is that they impose little to no limitations on the type and size of the underlying infrastructure needed to run them. Consequently, they can be easily moved from one machine-physical or virtual-to another with minimal or no service interruption. Such aspect is very attractive to cloud providers, since it allows them to dynamically provision, deploy and monitor the infrastructure that is used to run the containers and its associated resources, such as compute, storage, memory, and network, to improve hardware utilization and cost efficiency of their fleet. Therefore, most major cloud providers today, such as Amazon Web Services, Google Cloud and

Microsoft Azure, offer container services to their customers, such as Amazon Elastic Container Service (AWSWhitepapers, 2024; GoogleCloud, 2023) and Azure Container instances (MicrosoftAzure, 2023).

While pros outweigh the cons, there are few inherent drawbacks of container frameworks. One of the major issues with container frameworks is the *cold start* problem, which refers to the substantial time and resources required to retrieve and deploy container images when a container is scheduled to run on a specific machine for the first time. The main reason for the cold start problem is that when a container is deployed on a new machine, it must download the code and libraries required to run the container which are collectively called a container image from a remote online source known as a container registry. However, the process of pulling container images from online registries introduces latency and impacts the overall responsiveness of the containerized service when it is first deployed or dynamically scaled, making it difficult for services with very tight latency requirements to be containerized. In addition, cold start time also depends on multiple factors like choice of programming language, size of the container image deployed, and memory settings. For example, the cold start time for a recursive Fibonacci sequence generation function in JavaScript deployed on Azure cloud functions is around 9.822 s whereas the same function deployed in Java results in a cold start time of around 24.872 s (Manner et al., 2018). Finally, the instance lifetime (the longest time a function instance stays alive) where a tenant's application can maintain state and would suffer from a cold start varies from cloud provider to provider. For example, the median instance lifetime on AWS container service is between 6.2 and 8.3 h, which seems somewhat reasonable to incur the tens of seconds in cold start. However, idle instances undergo recycling to help providers save on compute resources, and it is known that the idle instance recycling time for AWS Lambda is ∼26 min (Wang et al., 2018). This becomes a much bigger problem when the entire fleet spends a significant portion of compute cycles merely launching new containers.

Furthermore, this issue becomes more prominent in services that are built using container frameworks, such as function-as-a-service framework. Function-as-a-Service is a model of software deployment that is widely used today to deploy and dynamically run fine-grained code or functions. To give some context of the popularity, AWS Lambda, a popular function-as-a-service framework, is known to host over a million customers. Function-as-a-service is known to extensively use container frameworks to enable packaging, deployment and dynamic execution of functions. However, services like function-as-a-service are now inherently prone to cold start issues as well and the cold start issue becomes highly critical when these services built using container frameworks suddenly require low-latency responses from their functions.

Computational issues like above are often solved by domain-specific hardware. For instance, graphics processing units (GPUs), with their parallel nature, are used to enable large-scale machine learning training. Another similar example is utilizing network processing units to offload tasks from valuable CPUs. Recently, cloud providers have been deploying programmable network processing units (NPUs) in network interface cards (NICs) called SmartNICs to offload tasks that were originally run by the host (Firestone et al., 2018), in order to save host resources to be used for other complex tasks. Unlike traditional NICs that primarily handle basic network communication and networking-related features, SmartNICs are equipped with programmable NPUs that can run custom programs, enabling them to perform traditional networking tasks such as packet processing, encryption, and load balancing, as well as advanced functions and protocols like virtualization, software-defined networking (SDN), and network function virtualization (NFV).

An even more interesting angle of research in programmable networks is to run customized applications that are not networking-related and thus are traditionally not run on networking devices. For example, a more recent set of research works utilizes programmable network devices to implement and accelerate various computational tasks, such as key-value stores (Jin et al., 2017), consensus protocols (Dang et al., 2015) and function-as-a-service framework (Choi et al., 2020). One of the techniques shared by these works is the use of SmartNICs to provide low-latency computation for various tasks. These works show huge potential in leveraging the capabilities of programmable network devices to solve the cold start problem inherent to container frameworks.

Given the above context, this paper present AlphaBoot, a new method and a software framework for utilizing SmartNICs as a container cache to alleviate the cold start problem. The main thesis of this work is to reduce the time it takes to retrieve a container by combining the idea of a cache, container registry and programmable network units. The main reason why this method solves the cold start problem for cloud providers is due to the SmartNIC's unique placement in the infrastructure. Container frameworks that are hosted on cloud providers often run within virtual machines to enable faster and easier deployment, backup and migration of the machine. These virtual machines are often configured to share a single bare-metal machine, which houses one or more NICs, to improve the fleet utilization, thus this means that SmartNICs are shared by multiple virtual machines (VM) and hence by multiple container frameworks. Such characteristic offers a unique opportunity to offload the image retrieval tasks from the host servers on to the SmartNICs and have the SmartNICs cache widely used and previously retrieved container images, enabling the SmartNICs to expedite the delivery of container images to the container frameworks within a VM that requests such images.

With such optimizations, AlphaBoot not only reduces the impact of cold start issue, but also helps to enhance data center efficiency in three key ways: First, reducing idle times of CPUs by speeding up image retrieval time and significantly speeding up container boot up, ultimately increasing data center utilization. Second, reducing wasted network bandwidth by reducing amount of network traffic used to retrieve same images over and over again. Storing container images on the SmartNIC enables local access to the images within the data center, eliminating the need to transfer them over the network repeatedly freeing up capacity for other critical data center operations. This also saves costs for users as network transport to-and-from outside of the data center is costly. Finally, AlphaBoot unlocks enhanced performance, capabilities and scalability. AlphaBoot allows for efficient scaling of container workloads within the data center by reducing cold start times and

reducing overhead of running containers. Since container images are readily available on the SmartNIC, the framework enables rapid deployment of new containers without waiting for image transfers. This facilitates seamless horizontal scaling, allowing data center operators to efficiently manage workload fluctuations and improve overall scalability. Furthermore, this not only improves the container frameworks themselves but also allows software built on top of these frameworks to be more scalable and opens up more opportunities to run low-latency applications. Given the high-level introduction, the structure of this paper is as follows: We begin the paper by providing the background (Section 2) of current state of container frameworks and SmartNICs within cloud data centers, followed by the motivation and challenges of realizing AlphaBoot (Section 3). Then, we discuss the overview (Section 4) of AlphaBoot followed by evaluation (Section 6) of AlphaBoot performance in multiple aspects. Finally we discuss some related work that motivated AlphaBoot (Section 7) and establish scope for potential future extension of this work (Section 8).

2 Background

We now discuss in depth on the pertinent information in container frameworks and SmartNICs, the two topics that are crucial tenets of AlphaBoot.

2.1 Containers

Container is a lightweight, standalone software process that encapsulates an application along with all its dependencies and configurations (Merkel, 2014), and is mainly developed to be easily created, shared, deployed, and scaled while ensuring isolation between them. Containers are deployed using a software framework, which leverage the features of the Linux kernel, such as cgroups and namespaces to run containers securely and efficiently. Conventionally, a container is started from (one or more layers of) container image, which is a set of software package that contains all of the code and dependency to run the process of interest. These container images are usually stored and distributed using a (either online or offline, public or private) registry, or a centralized repository of container images, such as Docker Hub[1]. Thus, when a container starts, the container framework downloads (or pull) layers of container images from a registry and executes the downloaded images to start a container, and the downloaded image layers are locally cached for future execution of the container.

A container image is built from one or more layers, where each layer consists of a subset of the code and library needed to run the container. This layering system allows multiple subsets of container images to be shared, which reduces redundancy between multiple images that depend on the same set of libraries. All the operations pertaining to the management of containers are controlled by a container framework. Some popular examples of container frameworks are Docker[2], KataContainers[3], Podman[4], gVisor[5], and Singularity[6], with Docker being the most popular and widely used container framework. Since containers are mostly lightweight, it is feasible for container frameworks to manage and execute hundreds of containers in a single bare-metal or virtual machine at once, enabling containers to be widely used in data center settings. Modern data centers use container runtimes nested inside virtual machines to run containerized applications to share the physical resource in a secure and isolated way, enabling better support for multi-tenancy and utilization of the underlying hardware infrastructure. Figure 1 represents a high-level overview of a modern containerized data center architecture.

A well-known problem of containerized environment is the added latency introduced for starting a container from a newly obtained image, as these images must be downloaded from a remote repository. This issue with added latency is also known as a *"cold start problem"* and is incurred when a container needs to be started from a image not present in the local cache due to initial start of a service, reboot of the underlying machine, deployment on a new machine to meet scaling on demand requirements or inactivity. Such issue is exacerbated in the modern data center where containers are hosted within VMs. Due to the isolation introduced by VMs if a container requires the same image used for a different container on a different VM on the same physical server, the physical server would have to download the same container from an online registry multiple times, resulting in multiple cold starts with redundant use of storage and network traffic. Thus, AlphaBoot's main goal is to address such problem of cold start and redundancy by integrating SmartNICs into the container image retrieval workflow to significantly reduce container startup times and save network bandwidth usage. Section 4 describes this process in detail.

2.2 SmartNICs

Network Interface Card (NIC) is a class of hardware devices which help connect a computer to a network. A SmartNIC is a programmable version of a NIC that comes with the capabilities of running custom programs on a NIC, which allows users to offloading network processing tasks, security and storage functions that are originally run on host CPUs, freeing the main processor's cycles which can be utilized by other processes. SmartNICs come in various types majorly classified by the type of the packet processor as either FPGA-based, ASIC-based or SoC-based (Bhalgat, 2014).

1 Docker Hub. Available at: https://hub.docker.com/ (accessed December, 2023).

2 Docker. Available at: https://docs.docker.com/guides/docker-overview/ (accessed December, 2023).

3 KataContainers. Available at: https://github.com/kata-containers/kata-containers/tree/main/docs/design/architecture (accessed December, 2023).

4 Podman. Available at: https://docs.podman.io/en/latest/ (accessed December, 2023).

5 gVisor. Available at: https://github.com/google/gvisor (accessed December, 2023).

6 Singularity. Available at: https://sylabs.io/singularity/ (accessed December, 2023).

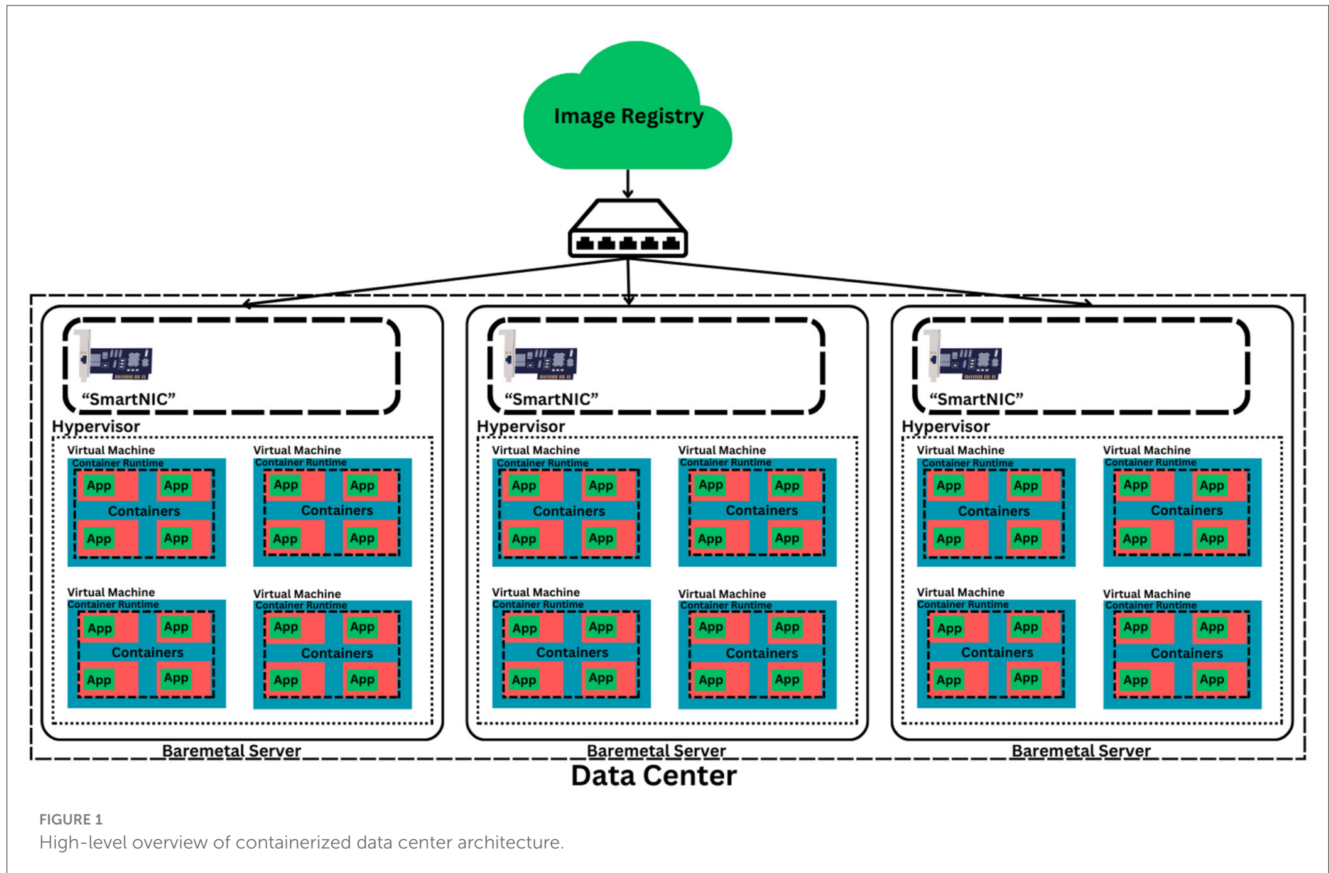

FIGURE 1
High-level overview of containerized data center architecture.

The three types of SmartNICs mainly differ on price-to-performance ratio, ease and capability of programming and number of feature sets. For example, ASIC-based SmartNICs require a dedicated and custom designed/built chip, which makes the NIC less flexible, harder to program, but provide fastest performance and best-in-class price-to-performance ratio. FPGA-based SmartNICs are more configurable, but are more costly and slower. SoC-based SmartNICs are built for ease of programming, thus allows various types of applications to be run with little to no motification, but such flexibility comes at a performance hit and higher cost. Thus, the choice of which type of SmartNIC to use mainly depends on the application of interest.

AlphaBoot's initial goal is to show the feasibility of utilizing SmartNICs to improve container cold start times. Therefore, the main feature of interest in a SmartNIC is the feasibility of integrating with existing container run times that are widely used. SoC-based SmartNICs run ARM-based cores that are capable of running vanilla Linux operating systems with little to no customization. This enables AlphaBoot to easily integrate existing container run times by treating the SmartNIC as a shared container image cache. Section 8 discusses future works on integrating AlphaBoot into other types of SmartNICs. AlphaBoot is implemented using an SoC-based SmartNIC called NVIDIA Bluefield (NVIDIA, 2024), which contains an ARM-based CPU with up to 16 cores, 32GB of onboard RAM and up to 400Gbps ports. The main reason for this choice is due to the large user base and rich support for various accelerators. When shared with multiple virtual machines (VMs), SmartNICs securely operate as shared hardware resources, enabling multiple VMs running on the same bare metal server to interact efficiently with the network infrastructure. Similar to how network cards are shared across multiple VMs, SmartNICs are efficiently isolated using techniques such as network isolation. In addition to VM level isolation, usage of secure protocols like SSL, SCP and NFS, which are widely known for their security and robustness in ensuring data integrity and confidentiality, makes the use of SmartNICs in such environments both feasible and secure.

2.3 Container image storage management

Container image de-duplication and eviction are critical processes in the management of containerized environments, especially in scenarios where resources are limited and are maily used to optimize storage utilization. Users choose similar operating systems and similar applications to run which introduces redundancy in container images for which container image de-duplication is an effective solution. This is particularly useful in environments where multiple containers are based on the same base image or contain overlapping layers. By storing only unique layers and sharing common data among containers, de-duplication significantly decreases the amount of disk space required. In addition to saving storage space de-duplication can also improve the container I/O performance. Few recent innovations in de-duplication of container images are: (1) DupHunter (Zhao

et al., 2024): introduces a novel architecture for Docker registry deduplication, aiming to enhance both storage efficiency and performance. It introduced a 6.9× improvement in terms of storage space required over the current state of the art. (2) Medes (Saxena et al., 2022): a novel serverless framework that addresses the compromise between resource use and performance typically faced in serverless platforms. It introduces a "dedup state" by exploiting memory redundancy in warm sandboxes which leads to significant improvement in end-to-end latencies better than the ones faced during cold start.

On the other hand, container image eviction is the process of removing unused or least recently used images from a system to free up resources for new containers. This is crucial in environments with finite storage capacity where maintaining an optimal set of images is necessary to prevent resource exhaustion. Eviction policies can be configured based on various criteria, such as the age of the image, frequency of use, or specific tags that prioritize certain images over others. A few innovations in the area of container image eviction are as follows: (1) Yang et al. (2023) integrates an ML module with a traditional heuristic caching system. It uses the heuristic algorithm as a filter to reduce the number of predictions required for evictions and the samples needed for training the ML model. This approach aims to minimize computational overhead while maintaining effective eviction decisions. (2) Gummadi (2023) combines a lightweight heuristic baseline eviction rule with a neural reward model trained through preference learning. This approach continuously trains a small neural network with automated feedback to predict the eviction priority of cached items. (3) Mangal et al. (2020) (Deep Eviction Admission and Prefetching) cache framework uses deep learning to optimize pre-fetching, admission, and eviction processes. It models pre-fetching as a sequence prediction task and uses online reinforcement learning to balance eviction strategies based on frequency and recency. As AlphaBoot implements a cache of container image on a SmartNIC for quick retrieval by any virtual machine running on a bare metal server, the limited resources on the SmartNIC would need the user to periodically evict the cached images. The choice of an eviction algorithm would depend on the environment and the infrastructure available as different SmartNICs would have varying onboard storage. The frequency of cache eviction and the choice of candidates to be kept in the cache would vary from case to case. AlphaBoot allows flexible configuration of the cache eviction algorithm of their choice as the performance gain of the framework is independent of this choice. The user could implement any of the available rule based or ML based eviction techniques mentioned. The chosen eviction algorithm essentially runs on top of AlphaBoot and would use the SmartNIC storage as its target. We discuss the planned work on container eviction strategy of AlphaBoot in Section 8.

3 Motivation and challenges

3.1 Motivations

In modern day cloud data centers where containers are utilized extensively, a significant amount of time and compute/network resources are consumed when pulling container images from online registries repeatedly, especially when average container image size spans around 500–700 MB. In Google Kubernetes Engine (GKE), a standard image pull can take ∼24 s to complete for a 500 MB image Google Cloud[7]. However, using image streaming technology, this time can be reduced to about 1.5 s. Google Cloud's image streaming, as described in the documentation, streams parts of the container image that are required for the application to start, allowing the container to begin execution while the rest of the image is still being pulled. This approach reduces initial latency but relies heavily on the assumption that the containerized application can be partitioned effectively to benefit from this streaming technique. Additionally, image streaming is most effective in environments where network bandwidth is abundant and where there are fewer repeated requests for the same image across different VMs on the same physical host. This inefficiency arises when multiple containers across different VMs on the same physical machine require the same image to run, and each time a new container is launched, the same image is pulled from the registry again, just to be saved to the same physical machine. This redundant process leads to a waste of bandwidth and results in unnecessary delays in container deployment. One of AlphaBoot's motivations is to eliminate such inefficiencies making it a better fit especially for high density VM Deployments such as in a data center.

Next motivation of AlphaBoot stems from the recent wave of domain-specific processors (DSP) that are making their way into the cloud data centers. Due to the cost and performance limitation of CPUs on running a large number of parallel tasks, DSPs, such as GPUs, have made their way into the data center to run simple and repetitive tasks, such as large matrix computation and machine learning model computation. Similar to this trend, SmartNICs are making their way into the data centers to offload *networking* tasks that are traditionally run by CPUs, such as firewall, load balancing and encryption. However, given the unique position of SmartNICs in the data center where all networking packets for a host go through a NIC (Figure 1), many believe that SmartNICs possess huge potential to perform non-networking tasks, such as key-value store, machine learning model training and even generic compute tasks like running containers. Thus, AlphaBoot motivation shows that SmartNICs are capable of a new kind of offloading non-networking tasks.

Finally, last motivation of AlphaBoot is to improve cloud data center utilization and cost efficiency, which follows from the first two motivations. Every CPU cycle is very precious in data centers as it directly relates to revenue from the users, thus one major reason for deploying DSPs in cloud data center is to improve CPU utilization by offloading domain specific tasks to DSPs. Similarly, AlphaBoot aims to free server resources by offloading container image caching to the SmartNIC, thereby letting the servers focus on their complex primary processing functions and increasing overall data center efficiency.

7 Google Cloud. *Gke image pull using streaming*. Available at: https://cloud.google.com/kubernetes-engine/docs/how-to/image-streaming (accessed June, 2024).

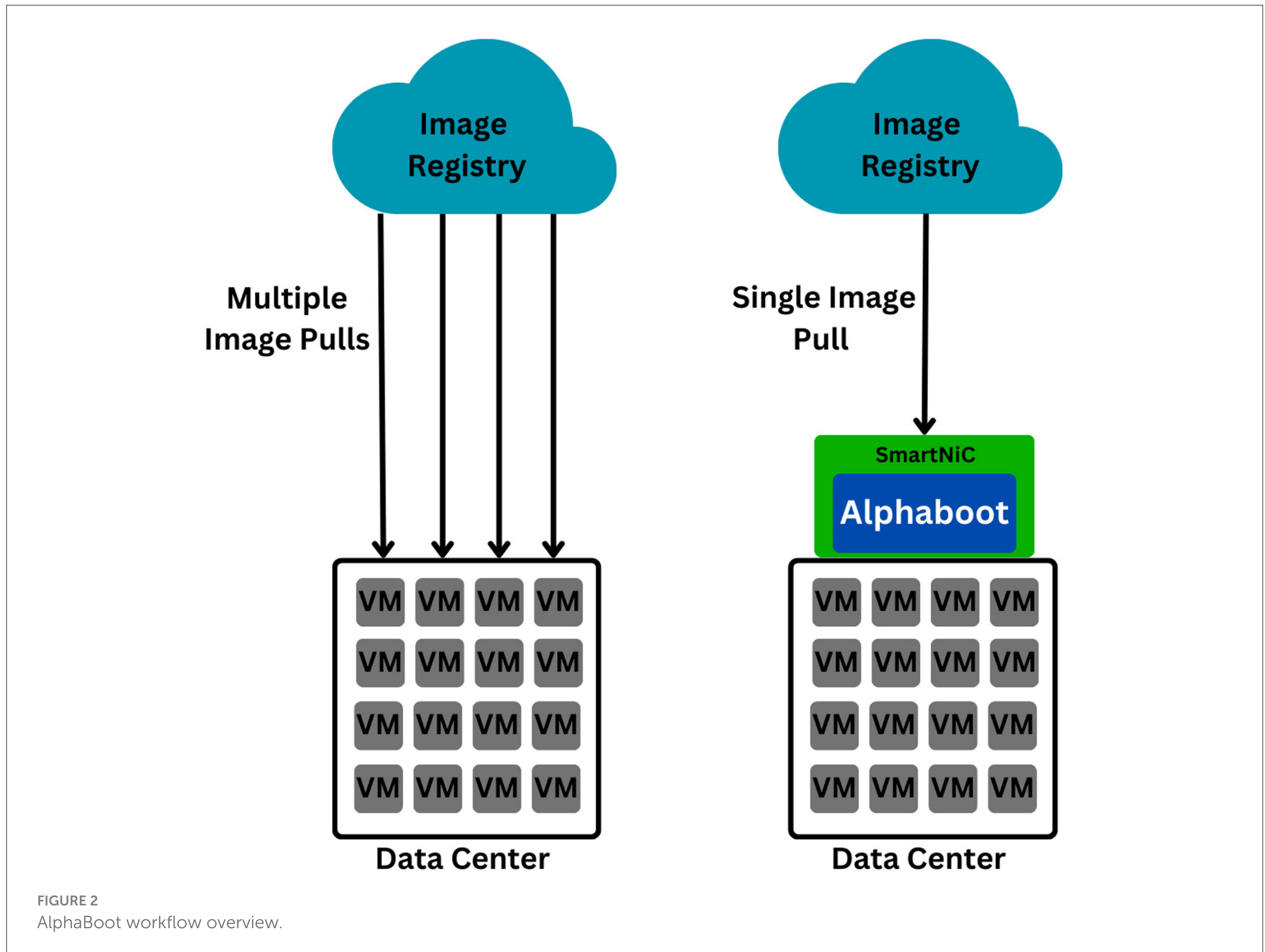

FIGURE 2
AlphaBoot workflow overview.

3.2 Key challenges

Given the listed motivations, here are some key challenges that AlphaBoot addresses:

1. Resource allocation: an effective caching framework should strike a balance between reserving adequate resources, particularly memory, for routine NIC tasks like checksum calculations, data packet encryption, and TCP/IP transfers, while ensuring there is ample space left for caching as many container images as possible. This balance is crucial to ensure both efficient NIC operations and improved container image retrieval speeds.
2. Scalability and performance: SmartNICs are often deployed in data-intensive and high-throughput environments. Therefore, the caching framework must be highly scalable and capable of handling increased network demands without compromising performance. It should seamlessly adapt to changing network conditions and accommodate growing numbers of containerized applications.
3. Monitoring and telemetry: to effectively manage and optimize the caching process, the framework should offer comprehensive monitoring and telemetry capabilities. Real-time insights into cache hit rates, memory usage, and system performance will aid in fine-tuning the configuration and ensuring the system operates optimally.

4 AlphaBoot overview

In this section, we discuss a high-level overview of the components that make up AlphaBoot. As mentioned in Section 2, modern cloud data centers employ numerous servers, each running hundreds of VMs hosting containerized workloads that are stored in container images. A container image consists of multiple stacked layers, typically built upon a base layer containing core functionality and essential dependencies. Consequently, when initializing a new container, it must retrieve the requisite base image from an online registry such as DockerHub. While containerization promotes lightweight and efficient deployment by reusing base images across multiple containers on the *same* VM, this is not the case across different VMs due to strong isolation. Thus, the need to fetch the same image separately by different VMs introduces increased cold start latency and redundancy within the data center. AlphaBoot's main contribution is to allow multiple VMs to share base image layers by caching the images in the SmartNIC. There are two fundamental steps that AlphaBoot follows for efficient container image caching on SmartNICs. First, fetching container images from online registries and storing them on SmartNICs (Section 4.1), and then retrieving the cached container images from SmartNICs into VMs (Section 4.2). These steps are outlined in Figure 2.

4.1 AlphaBoot image caching

Consider a container framework that resides in a VM that needs to pull one or more images from an online registry. Once the request for the images is provided, AlphaBoot first intercepts the request and checks if the image is present within the cache present in the AlphaBoot's cache. If the image is in the cache, the image is retrieved from the cache and sent to the container framework, which, in turn, boots up a container. If it is not present on the cache, AlphaBoot directly pulls the requested image from an online registry, sends it to the container framework and also stores the image in AlphaBoot cache.

4.2 Retrieving cached images

When a container framework in a VM requests a container image that is already cached by AlphaBoot, AlphaBoot efficiently transfers the image to the VM over the network. Once done, the VM proceeds to boots the container from the received image. Similarly, when a subsequent container framework residing in a different VM requests for the same container images, it can be efficiently satisfied from this local cache, avoiding the necessity of repeated and potentially time consuming calls to the online registry as shown in Figure 2.

4.3 Image eviction policy

AlphaBoot caches container images on the SmartNICs in the form of tarball images. However, the number of container images which can be cached is limited by the disk space on the SmartNIC. This calls for the eviction of stale container images which are less likely to result in a cache miss if evicted to free up space and be able to accommodate more images. We explored two eviction policies which can be used to evict such container images from the SmartNiC. First is the rule based Least Recently used (LRU) cache designed to optimize the cache's hit ratio by prioritizing the retention of recently accessed items. In an LRU cache, each container image is assigned a timestamp indicating the last time it was accessed. When the cache reaches its capacity and a new image needs to be accommodated, the LRU policy identifies and evicts the image with the oldest access timestamp, thereby making space for the new image. This approach operates under the assumption that images that have not been accessed for the longest period are less likely to be needed in the immediate future. The second policy leverages a machine learning-based approach to enhance the eviction strategy inspired by Mangal et al. (2020), Gummadi (2023), and Yang et al. (2023). Unlike the rule-based LRU cache, this approach uses predictive analytics to make more informed eviction decisions. The choice of the cache eviction policy ultimately depends on the nature of the workload and infrastructure constraints. AlphaBoot is flexible and allows users to pick a container image cache eviction policy as suitable to their environment.

4.4 Image dedupe/compression

Growing adoption of containers results in an increasing need for efficient storage systems for container images. Images stored in registries face performance challenges because of the inherent redundant nature of container images. Das et al. (2021) proposes a solution using file-level deduplication, which stores only unique files rather than entire layers, thus significantly reducing storage requirements. Primary storage employs file-level deduplication, where each file is hashed and only unique files are stored. Metadata files track the association between images and their files, facilitating image reconstruction during pull requests. To mitigate the increased latency caused by reconstructing images, secondary storage is used to store popular and recently accessed images in their compressed formats. This approach reduces pull time latency by allowing frequently accessed images to be served directly from secondary storage without reconstruction. The results show that combining file-level deduplication with popularity-based and time-based staging storage effectively balances storage optimization and performance, making the registry efficient in terms of both storage and access time. The storage requirement for 64 container images tried went down from 22GB to 14 GB which translates to a deduplication ratio of 1.58.For a sequence of 25 pulls using 10 images with time and popularity the pull latency showed an improvement of 69%.

Current deduplication methods are ineffective for container images stored as gzip-compressed tar files due to low deduplication ratios and increased overhead during image pull. Huang (2018) models deduplication using a Markov Decision Process (MDP) to optimize storage savings while minimizing performance degradation. Simulation demonstrated that file-level deduplication could significantly reduce storage requirements without severely impacting pull performance, processing ∼4.4 layers per second on average using 60 threads. Analyzing the effect of deduplication on performance shows that compression time is the major bottleneck during the pull.

5 Experimental setup

5.1 Experimental testbed setup

Our evaluation test bed consists of a Linux server containing 8-core AMD EPYC 7232P 3.10GHz, 8GB 3200MT/S RAM, 480GB SSD SATA 6Gbps, where the servers are equipped with Nvidia Bluefield 2 P-Series DPU 25GbE Dual-Port SFP56, acting as the cache for our container Images. The Nvidia Bluefield 2 SmartNIC has 8-core Cortex A72 ARM running at 2.75GHz, 1MB L2 cache per 2 cores, 6MB L3 cache, 16 GB of DDR4 RAM and SSD drive. Both the server and the SmartNIC are configured with the 1 Gbps uplink to the internet and are configured with the Ubuntu 20.04.4 operating system.

6 Results

We now compare the performance of AlphaBoot when processing image retrieval requests from the container frameworks

across varying number of VMs. Under such setting, we evaluate the impact of AlphaBoot on network utilization, as well as cold start time measured from requesting an image to successfully starting a container from retrieved images.

- N: Container Image Name
- A: Alphaboot Cache
- R: Container Registry
- C: Resulting Container Image

```
1: procedure PullContainerImage(N, A, R)
2:    if N ∈ A then
3:       C = LoadFromAlphaboot(N, A)
4:    else
5:       C = LoadFromRegistry(N, R)  ▷ Slow image load from the remote registry
6:       if C ≠ −1 then        ▷ Image found in online registry
7:          SaveToAlphaboot(N, A)
8:    return C         ▷ C is the retrieved container
```

Algorithm 1. Alphaboot container image retrieval algorithm.

6.1 AlphaBoot Implementation

AlphaBoot implementation involves modifying existing container framework within each VM to retrieve the container images from the local cache. For this purpose, we used Docker (Merkel, 2014), one of the most popular container frameworks deployed in production, as the baseline container framework. The modification made to Docker consists of these three steps and as demonstrated by Algorithm 1 and also illustrated in Figure 3:

1. Transferring the image retrieval request to AlphaBoot This step requires the Docker installation within each VM to send the image retrieval request directly to AlphaBoot. To enable this, AlphaBoot simply acts as one of the image repositories that Docker framework can utilize, which minimizes the code change required.

2. Retrieving the image and storing into the cache Once the image retrieval request is received, AlphaBoot simply pulls the image from a repository of choice and stores the image into the cache, if the image is not stored. If the image is already in the cache, this step is ignored. For example, the following command is relayed and run by AlphaBoot.

```
docker pull <image_name>
```

Then, the image is saved into AlphaBoot cache as follows.

```
docker save <image_name> > alphaboot_img
```

Now the image is ready to be transferred to each VM of choice.

3. Transferring the image from AlphaBoot to the VMs Finally, the image from the cache is transferred to the VMs. There are multiple methods to do this, but the two methods chosen for simplicity are Secure Copy(SCP) and usage of dedicated network drive. Given that AlphaBoot is on a SmartNIC which runs a Linux, SCP is used to transfer the image file from one host to another, i.e. VMs. However, evaluations show that transfer over SCP results in high cold start time due to handshakes required. Thus, the second method that was chosen is to have the VMs mount a network drive (NFS) located on AlphaBoot. Once the new images is saved into the network drive, the VMs can simply load the image file from the mounted drive, which removes the need for a handshake.

Once the image is transferred to the VMs, the Docker framework in each VM can simply load the image by running the following command:

```
docker load < alphaboot_img
```

6.2 Testing methodology

We evaluate AlphaBoot's performance on four frequently used popular base image layers; namely Ubuntu, Python, Alpine and Nginx (each with more than a billion downloads on DockerHub). Alpine is the lightest minimal Docker image based on Alpine Linux, Python is an open source object oriented programming language, Ubuntu is a popular open source Linux distribution and Nginx is a popular web server. The choice of images was made on the basis of the popularity and the sizes in Megabytes of the images as they vary from Alpine being the smallest (<5MB) in size to Python being the biggest (>300 MB in most functionally rich images) to Ubuntu and Nginx being in between. Figure 4 illustrates the latency comparison of an image pull request between a pull from a web registry and direct load from AlphaBoot cache for the images mentioned above. The popularity was considered due to the frequency of the images being used as base images for other images. In addition, the size of the image has a direct corelation to the time it takes to retrieve an image from its source. The aim is to establish a wide spectrum of container images that AlphaBoot can efficiently cache and save up on boot up time proving the effectiveness of its mechanism.

Given each type of image, there are two variables introduced to the evaluation: number of VMs requesting the base image and type of transfer protocol. First, varying number of VMs allows AlphaBoot to show its scalability and also its increased efficiency as the number of VMs requesting the same base image increases. Second, AlphaBoot implementation requires the transfer of images from the cache to the requested VM, thus two different implementations are evaluated for this work: secure copy protocol (SCP) and a dedicated network drive.

6.3 Running openFaas serverless on Alphabooot

Third, we run an application on top of Alphaboot to evaluate the performance gain and to illustrate its functionality and benefits in the context of a serverless computing framework. We implemented openFaas, a Function as a Service (FaaS) platform that is open-source, on a Minikube Cluster. This cluster represents a scaled-down variant of a Kubernetes Cluster, predominantly

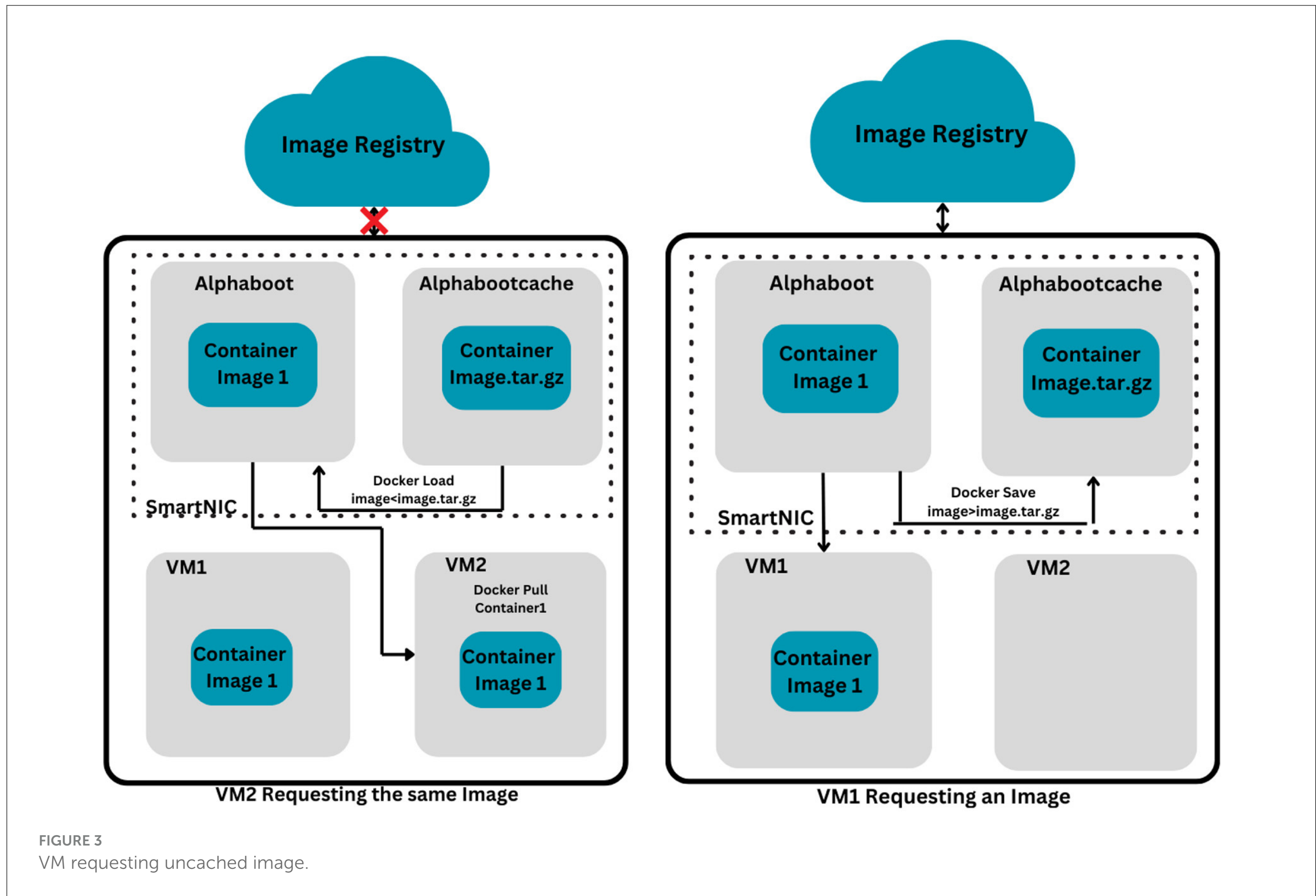

FIGURE 3
VM requesting uncached image.

employed for the local development and testing of applications destined for Kubernetes clusters. Utilizing the faas-cli, a command-line tool designed to facilitate the creation and deployment of serverless functions, we developed a custom function in Python. Following this, we uploaded the function's image to DockerHub and subsequently deployed the function onto a remote server. Within this configuration, the remote server retrieves the custom-created function image from DockerHub and initiates a container using that image, which is then deployed as a function. When executing Openfaas over Alphaboot, on the initial retrieval of an image from an online registry, Alphaboot caches the container image on a SmartNIC. The next occurrence of any VM requesting the same function image running on the same physical infrastructure can load the image from the cache instead of sending a network request to fetch the image from an online registry. Alphaboot is versatile and not limited to particular applications; it supports the execution of various applications that utilize Docker as a container engine. The decision to utilize a serverless function platform for our testing was driven by the susceptibility of serverless functions to the cold start issue, making them an ideal scenario for evaluating a platform such as Alphaboot. A cloud service provider that operates several virtual machines (VMs) on shared physical infrastructure, offering serverless function services to clients, often deploys identical functions across multiple VMs. Alphaboot leverages the computational and storage strengths of SmartNICs to minimize cold start durations and conserve network bandwidth. It does so by uniquely utilizing the redundancy of container images that naturally occurs in cloud data centers, where multiple VMs are hosted on a single server.

6.4 Evaluation results

6.4.1 Latency comparison

We first compare the latencies involved in pulling the chosen container images from an online registry (T_{fetch}) called DockerHub and then we discuss the AlphaBoot's image retrieval times (T_{load}) in contrast to the images pulled directly by the VMs. The calculations for the time taken to retrieve an image and boot up a container for all the performance evaluation in our research considers the time taken to transfer the cached image from AlphaBoot to the host server and then further to the VM requesting it. Figure 5 shows the comparison of image retrieval times for different images. The time required for booting up from a cached image are as low as 93% of the original pull time for the Alpine image, 63% for Ubuntu, 57% for Python and 53% for Nginx.

The time saved can be calculated using the equation

$$\text{Overall Latency Improvement} = N \times (T_{fetch} - T_{load})$$

where,

T_{fetch} =Time taken to fetch an image from the online registry.
T_{load} =Time taken to load an image from the SmartNIC cache.
N =Number of times the same image is requested.

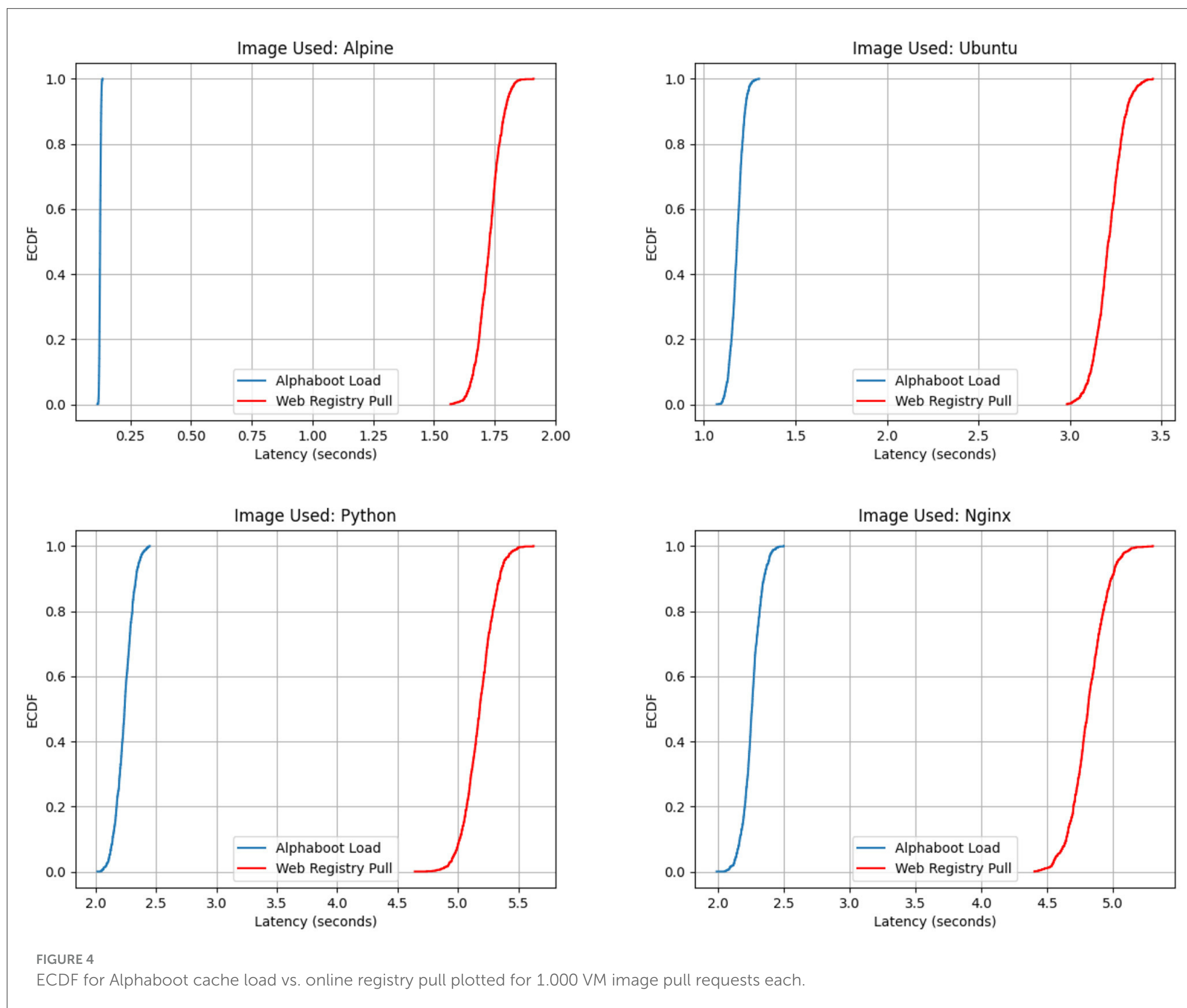

FIGURE 4
ECDF for Alphaboot cache load vs. online registry pull plotted for 1.000 VM image pull requests each.

The time taken by AlphaBoot to pull an image from an online registry and convert the container image into an AlphaBoot image file inside the cache is incurred only once per container image. In contrast, the VMs requesting the cached image to build containers is a repetitive process, which is compared against the repetitive pull requests made by the VMs to an online registry. Thus, as more virtual machines (VMs) request the same cached image, the initial image retrieval time becomes less significant compared to the time saved each time a VM requests the container image. The time spent pulling and saving the image into the cache adds up initially, but its impact diminishes as the number of VMs utilizing the cached image increases. Ultimately, the time saved from serving cached images outweighs the initial overhead, illustrated in Figure 6.

We also compared two ways of transferring the cached image from the AlphaBoot to the VM, SCP and NFS. As shown in Table 1, we can see that AlphaBoot suffers greatly when SCP is used to transfer the images from the cache, even up to 433.87%. However, when used the NFS method of transfer, the performance gain is quite significant, which is up to 92.75% improvement when comparing with retrieving directly from DockerHub. This shows that with more optimizations, AlphaBoot can have a significant improvement over retrieving directly over the internet. Also, we see that the improvement is large for a small image size, whose retrieval times are often limited by initial connection establishment time. Thus, as AlphaBoot is used for more container layers, which are often smaller in size, we expect a much greater performance gain in production.

6.4.2 Network bandwidth usage

As an outcome of caching container images on SmartNICs, we achieve a noteworthy reduction in the network bandwidth needed to retrieve these images from an online container registry. Instead of constantly downloading the same images from an online container registry on the internet, we can now access them directly from the locally cached copies on the SmartNICs. This bandwidth-saving effect becomes particularly prominent when there's an increasing number of virtual machines (VMs) that require the same container image. The significance of this bandwidth-saving measure is further emphasized when we consider the scale of operations. This not only enhances network efficiency but also

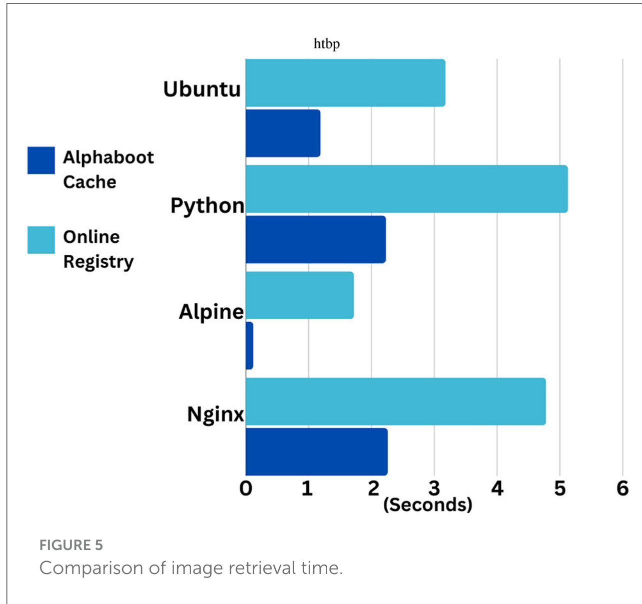

FIGURE 5
Comparison of image retrieval time.

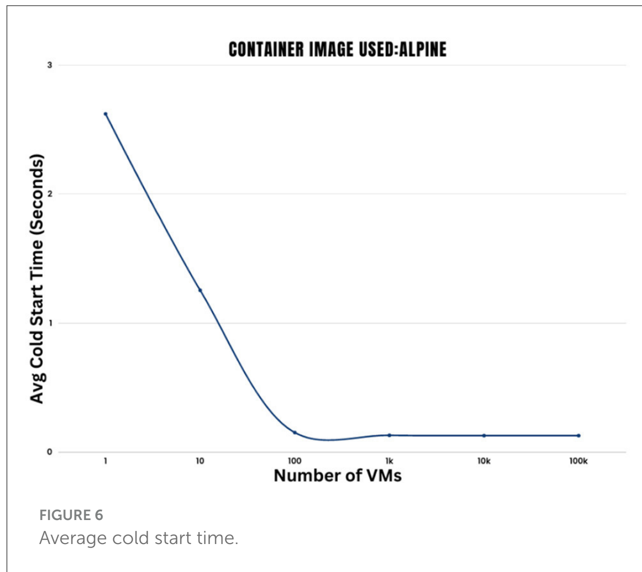

FIGURE 6
Average cold start time.

TABLE 1 Image retrieval time comparison between online registry and AlphaBoot using SCP and NFS.

Image name	Baseline (s)	AlphaBoot (s)	% Improvement
SCP			
Ubuntu	3.185	13.819	−433.87%
Python	5.133	14.316	−278.9%
Alpine	1.724	4.368	−253.36%
Nginx	4.789	5.640	−117.76%
NFS			
Ubuntu	3.185	1.194	62.52%
Python	5.133	2.236	56.44%
Alpine	1.724	0.127	92.64%
Nginx	4.789	2.263	52.75%

has cost-saving implications for the cloud provider. The results in Figure 7 are computed using a specific set of container images. The images and the image sizes that are used for the calculations are: Python (128 MB), Ubuntu (77.8 MB), Nginx (143 MB), and Alpine (7.33 MB). Under this setup, as expected, we can see that the network bandwidth utilization goes down by up to 99.9% as the number of VMs utilizing the same base image grows. Another point to note is that VMs are often restarted from scratch on the same host machine, so rather than assuming the 100k VMs to run at the same time on a single host, it is more realistic to assume the 100k VMs are booted over a period of time on the same host. The network bandwidth saved can be calculated using the equation

$$\text{Bandwidth Saved} = N \times B$$

where,

N = number of times the same image is requested.
B = size of the image in bytes.

6.4.3 SmartNIC memory and disk usage

We analyze the peak memory usage by Alphaboot and the Disk space used by cached container images stored as gzip tarball files to understand the overhead created by running Alphaboot on a SmartNIC and to understand if it is feasible to run alongside regular SmartNIC operations and if it hinders the SmartNIC's ability to do its regular tasks. We run the command: /usr/bin/time -v sudo docker pull python and observed that on an average the Maximum resident set size was (referring to the peak amount of physical memory that the process used during its execution) around 4.98MB for all the four container images under test mentioned in the above sections which happens to be way less as compared to the 15Gb total memory present on the SmartNIC. As far as the disk space occupied by the cached container images is concerned they occupy:Alpine-7.7MB;Nginx-188MB;Python-996MB;Ubuntu-69MB, which is also negligible when compared to the 42GB total disk space available.

6.4.4 Serverless function cold start time

We analyze the latency differences between deploying a function whose container image is pre-cached on a SmartNIC and the deployment time required for a function whose image must be retrieved from an online registry. The calculation of the time taken to deploy a function on a serverless platform consists of the time it takes after a deploy command is issued, following which Alphaboot checks its cache for the necessary image to initiate a container. If found, it retrieves the image, boots up a container, and ensures the function is deployed on a container orchestration platform, making it ready for invocation. The time required to deploy a serverless function that is cached by Alphaboot is 0.08s, in contrast to when the container image was to be fetched from an online registry, which comes out to be 4.7s, showing a gain in the cold start time of upto 98.2% when a simple function that returns a string when invoked is deployed.

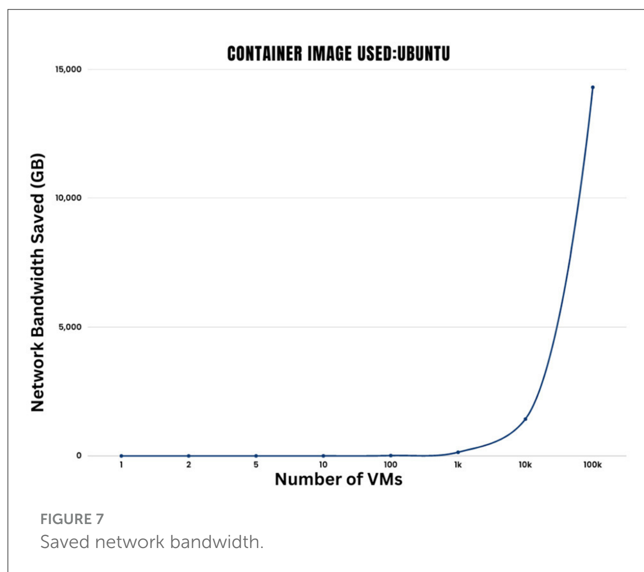

FIGURE 7
Saved network bandwidth.

7 Related work

7.1 SmartNICs in the cloud

The growing adoption of SmartNICs in modern data centers has presented researchers with opportunities to explore novel ways of offloading not only traditional networking tasks, such as checksum computation and encryption, but also other resource-intensive data center operations that were previously handled by bare metal servers. By offloading such operations to SmartNICs, data centers can reduce the burden on server CPUs and achieve more efficient storage management, which translates into cost savings for cloud providers. One notable development is Hyperloop (Kim et al., 2018), which leverages SmartNICs with Remote Direct Memory Access (RDMA) capabilities to accelerate storage replication operations. This means that we can further optimize AlphaBoot by employing RDMA usage to transfer data from the SmartNIC to the host machine and the VMs. Another significant advancement is λ-NIC (Choi et al., 2020), which enables serverless compute on programmable SmartNICs. This framework allows data center operators to dynamically offload compute tasks from server CPUs to SmartNICs, facilitating improved resource utilization and scalability. With λ-NIC, cloud providers can better meet the demands of their customers while optimizing their server infrastructure. Coupled along with AlphaBoot, we believe that λ-NIC can further improve the cold start performance of serverless functions in the cloud. Similar extension to this work is Speedo (Daw et al., 2021) a system that offloads FaaS dispatch and orchestration services to SmartNICs from user space. Finally, major cloud providers like Microsoft have also explored the use of FPGA-based SmartNICs to offload tasks, such as Hypervisor switching (Firestone et al., 2018). This offloading improves the overall performance of virtual environments, resulting in enhanced service quality for cloud customers. A future work for AlphaBoot is to investigate other SmartNIC architecture for added performance.

7.2 Methods for improving cold start

In the realm of tackling the cold start problem, it is observed that container images with shared layers tend to have a huge positive impact on the cold boot time (Beni et al., 2021). In the context of serverless functions cold start times FaaSLight (Liu et al., 2023) achieves the reduction of cold-start latency through application-level optimization, specifically targeting the preparation phase latency, application code loading latency, and total response latency of serverless functions. By optimizing the code loading process and reducing unnecessary code, FaaSLight significantly improves the performance of FaaS applications, leading to a decrease in cold-start latency and overall response time. Many research works aiming to tackle the issue of cold start in containers use scheduling algorithms or machine learning to try to reduce the frequency of cold start occurrences as much as possible. For example, there are a set of works trying to reduce the cold start time by using a scheduling algorithms, such as Least Recently Used warm Container Selection (LCS) by keeping the containers alive for a longer period of time (Sethi et al., 2023). Pan et al. (2022) also deploys a similar approach of keeping the containers warm (running) to serve serverless request on IoT devices. AlphaBoot's approach toward tackling the cold start problem differs in one major way that is it doesn't use the warm start approach but instead actually reduces the latency in the case of occurrence of a cold start. Chen et al. (2024) attempts to reduce the cold start latency by deploying software stack optimizations. Wen et al. (2024) is an advanced resource scaling system designed to optimize container allocation and placement in serverless platforms. ComboFunc employs a heuristic greedy algorithm to efficiently address the NP-hard problem of combining and placing heterogeneous containers, ensuring effective resource utilization and scalability within the Kubernetes environment. There are also machine learning based approaches to reduce cold start frequencies (Agarwal et al., 2021) by looking into server metrics such as CPU utilization. The approaches do show performance improvements and are a valid set of works that can complement AlphaBoot.

8 Future directions

We now discuss the future direction of AlphaBoot to further improve the performance, network and cost benefits.

8.1 Accelerating file transfer with RDMA

AlphaBoot currently utilizes NFS to transfer cached images from the SmartNIC to the VM on a bare metal server. However, NFS introduces added latency overhead due to its software abstraction and multiple network layers. To enhance the transfer process and further improve speed, a future direction is to utilize a lower-level interface, such as the Remote Direct Memory Access (RDMA) over Peripherral Component Interconnect express(PCIe), which can transfer small images from the SmartNIC to the bare metal server and/or even to the VMs without any software abstraction or CPU usage. RDMA allows for direct memory access between the memory of different computers without

involving the operating system, thereby significantly reducing latency and CPU overhead. Thus, we expect that, by utilizing RDMA, container images will be directly loaded into the server's memory, thus expediting the container building process and improving overall system performance. To further enhance this process, future exploration could involve optimizing image files themselves for RDMA transfer. Such advancements would make the container deployment process more efficient and suitable for high-throughput, low-latency environments.

However, this technique was not considered for this paper, since it requires additional considerations in code to ensure security in the host, since RDMA bypasses any software and can introduce the bare-metal servers to malicious images quite easily. Thus, the level of security should be evaluated for the specific use case and the user should determine if the potential performance gains justify the security concerns. Additionally, implementing security measures such as hardware isolation, access controls, and encryption could help mitigate some of the risks associated with sharing a physical interface. Thus, any decision to switch to a RDMA-based approach for file transfer should involve a thorough risk analysis and take into account the specific requirements and security needs of the system.

8.2 Image compression

The AlphaBoot system utilizes gzip compression to perform a lossless compression of container images, but this method is quite generic and is a file-level compression. Therefore, gzip compression does not take the container architecture into consideration, such as compressing and combining layers separately. As a recap, container images are constructed using layers that are stacked on top of each other and each layer only stores information about the differences between them; then, it is possible to compress the images by looking at the diffs that are redundant across multiple layers. Our inspection shows that current Docker's image compression mechanism is insufficient in eliminating such redundancies. Surprisingly, only 3% of the files in these registries are unique, indicating a high level of duplication at the file level (Zhao et al., 2019). This duplication likely extends to the layers themselves, presenting an opportunity to develop mechanisms that can effectively de-duplicate common differences among layers within the same image. Thus, for a follow up work, we are exploring methods to compress a group of containers by evaluating the contents in their layers.

There are some existing works on container image compression. First, DockerSlim is a tool designed to optimize Docker container images by reducing their size significantly without modifying their functionality. Quest (2024) optimizes container images through a combination of dynamic and static analyses. Initially, it runs the container in a special monitoring state to observe its typical operations, identifying which components are actively used during runtime. Concurrently, it performs a static analysis to inspect the container's file system, removing unused files and dependencies that are not necessary for its functionality. This comprehensive analysis allows slim toolkit to create a significantly smaller container image by including only the essential components. The minimization process can reduce the image size by up to 30x, streamlining deployment without sacrificing the container's operational capabilities. Notably, well-known container image registries like Docker Hub offer pre-existing slim versions of these commonly used base container images. Thus, first line of future work is to integrate AlphaBoot with the Slim Toolkit to compress the size of container images cached on SmartNICs. To achieve image compression on AlphaBoot, we will need to utilize the Slim Toolkit specifically for widely used generic base layers, as these tend to be bulky and contain many files unnecessary for every application. However, this slimming process may obfuscate layer information during compression, which may not be ideal for AlphaBoot.

Lastly, for further space optimization, we should consider consolidating multiple RUN commands in DockerFiles to prevent the creation of excessive layers and utilize tools like docker-squash to merge layers and reclaim disk space. Also, we plan to ensure that caches are cleaned up during the build process, which is another effective strategy to keep image sizes minimal in AlphaBoot.

8.3 Image eviction

In the proposed AlphaBoot system, addressing the challenge of limited storage on SmartNICs is most important for effective scaling in data center environments. Currently, AlphaBoot enhances the retrieval and sharing of container images across virtual machines by caching them locally on SmartNICs, thus mitigating cold start issues. However, with thousands of gigabytes worth of container images needing management, the finite storage capacity of SmartNICs necessitates a robust image eviction strategy to maintain efficiency.

To scale AlphaBoot for data center-level distributed workloads, image eviction becomes critical. This involves removing less frequently used or outdated images from the cache to free up space for new ones. The eviction policy could be based on various heuristics, such as least recently used (LRU) or least frequently used (LFU) criteria, which prioritize the removal of images that are least likely to be needed soon. Advanced strategies might incorporate machine learning to predict which images could be evicted with minimal impact on performance, thereby optimizing the use of limited SmartNIC storage.

Further research could explore dynamic adjustment of the caching strategy based on real-time analysis of container usage patterns and network conditions. By integrating these adaptive strategies, AlphaBoot could proactively manage storage and enhance overall data center efficiency, ensuring that the most critical images are always readily available while less pertinent ones are efficiently cycled out. This adaptive approach would not only maintain but also enhance the performance benefits of the AlphaBoot system in larger-scale operations.

8.4 Utilizing other types of SmartNICs

In data centers with varying workload requirements, different types of SmartNICs—such as FPGA-based, ASIC-based, or SoC-based—are employed to optimize performance and flexibility (Kfoury et al., 2024). FPGA-based SmartNICs are highly

programmable, enabling real-time updates and modifications to data processing tasks without necessitating hardware modifications. This feature could be helpful for AlphaBoot for testing and deploying new algorithms for image caching and retrieval. The adaptability of FPGAs allows for the development and refinement of container image handling techniques that can evolve alongside advancements in AlphaBoot's algorithms or emerging container technologies.

Conversely, ASIC-based SmartNICs are tailored for optimized performance in predefined tasks, owing to their fixed-function architecture. While they offer less flexibility than FPGAs, they excel in speed and energy efficiency for tasks with well-defined and stable processing requirements. In the context of AlphaBoot, ASICs could be specifically tuned to manage serverless functions or image handling routines that are consistent and undergo minimal changes, yet require rapid processing in high-traffic data center environments. Thus, they are ideal for enhancing performance during peak operational periods in production settings.

SoC-based SmartNICs provide a compromise, incorporating both processing power and programmability, thereby balancing the trade-offs between FPGA and ASIC technologies. This makes them a versatile option suitable for a variety of data center applications, ranging from development and testing phases to stable, high-volume production environments.

9 Discussion

9.1 SmartNIC adoption

The Deployment of SmartNICs in the data centers of major public cloud providers has resulted in the opportunity for adoption of offloading techniques in cloud environments. Recent initiatives by leading cloud providers highlight the growing trend toward integrating SmartNICs into data center infrastructure. For example, Microsoft has employed SmartNICs in Azure Data Centers to offload network functions Microsoft[8] and Alibaba has also joined this trend, deploying data processing units (DPUs) to enhance their cloud services' offload capabilities, providing additional efficiency, both in compute and energy usage, and security layers (Mann, 2022). These examples illustrate that SmartNICs are already part of the core infrastructure for public cloud operators, effectively making this technology accessible at scale and indicate their viability and readiness for practical use in large-scale data centers. Our approach, as detailed in this paper, is targeted specifically at data centers, public cloud providers and operators who can leverage such advanced infrastructure to optimize service performance, as opposed to individual or smaller scale users with standard hardware configurations. The availability of SmartNICs in modern day cloud data centers makes our proposed method relevant for optimizing container image retrieval tackling the container cold start problem and sharing base container images with multiple virtual environments running on common bare metal servers.

8 Microsoft. *Azure accelerated networking: Smartnics in the public cloud*. Available at: https://www.microsoft.com (accessed June, 2024).

9.2 Security concerns around SmartNIC offload

In addressing security concerns, our framework ensures strong data isolation by caching only the most commonly used layers, which are primarily pulled from trusted sources like official images in online registries such as DockerHub. By exclusively caching and sharing these verified layers, we mitigate the risks associated with unverified or potentially malicious content. If the layer is modified by the processes inside the VMs, they are stored in the VMs directly and will not be pulled from a remote repository again. In addition, in case the user-modified image is added to the remote repository, the hash signature of the image will be different than any other images available, eliminating the need for added security.

AlphaBoot also provides stronger isolation than storing and caching images on the host server. In fact, this is one of the main advantages of AlphaBoot that is different than placing cache on the host machine. First of all, given that SmartNICs operate within a separate security domain with separate credentials for authentication, even if the host server is compromised, SmartNICs will not be compromised. Furthermore, network for exchanging control traffic to the SmartNIC typically is configured to be separated and isolated from production network that VMs utilize. Thus, even if the production network is compromised somehow, control traffic to the SmartNIC will likely be not compromised, reducing another security concern. Some notable works like Choi et al. (2024) and Patel and Choi (2023) utilizes this property to build a secure system for federated machine learning and blockchain.

10 Conclusion

Container Image retrieval speeds can be improved to a great degree especially if the same image is required by a VM multiple times by caching the image on a SmartNIC in a data Center. Especially when there are repeated pulls for the same images it makes a lot of sense to cache them locally. In this paper, we present AlphaBoot a framework to cache container images on SmartNICs and a way to retrieve images to greatly reduce cold start times and network bandwidth usage. We believe that utilizing AlphaBoot will allow cloud providers to save huge cost and energy in serving containerized workloads that power many important workloads today.

Author's note

We release an Open Source Framework that is built based on the popular Docker framework that demonstrates the reduction in the Container cold start time at https://github.com/The-Cloud-Lab/Alphaboot.

Data availability statement

The datasets presented in this study can be found in online repositories. The names of the repository/repositories and accession number(s) can be found in the article/supplementary material.

Author contributions

SG: Data curation, Investigation, Software, Validation, Writing – original draft, Writing – review & editing. SC: Conceptualization, Data curation, Formal analysis, Funding acquisition, Investigation, Methodology, Project administration, Resources, Software, Supervision, Validation, Visualization, Writing – original draft, Writing – review & editing.

Funding

The author(s) declare financial support was received for the research, authorship, and/or publication of this article. This project has been funded by NSF CRII 2245352.

Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's note

All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

References

Agarwal, S., Rodriguez, M. A., and Buyya, R. (2021). "A reinforcement learning approach to reduce serverless function cold start frequency," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)* (Melbourne, VIC: IEEE), 797–803. doi: 10.1109/CCGrid51090.2021.00097

AWSWhitepapers (2024). *Overview of deployment options on AWS updates*. Available at: https://docs.aws.amazon.com/pdfs/whitepapers/latest/overview-deployment-options/overview-deployment-options.pdf#amazon-elastic-container-service

Beni, E. H., Truyen, E., Lagaisse, B., Joosen, W., and Dieltjens, J. (2021). "Reducing cold starts during elastic scaling of containers in Kubernetes," in *Proceedings of the 36th Annual ACM Symposium on Applied Computing, SAC '21* (New York, NY: Association for Computing Machinery), 60–68. doi: 10.1145/3412841.3441887

Bhalgat, A. (2014). *Choosing the best smartnic*. Available at: https://developer.nvidia.com/blog/choosing-the-best-dpu-based-smartnic/

Chen, Q., Qian, J., Che, Y., Lin, Z., Wang, J., Zhou, J., et al. (2024). "Yuanrong: a production general-purpose serverless system for distributed applications in the cloud," in *Proceedings of the ACM SIGCOMM 2024 Conference, ACM SIGCOMM '24* (New York, NY: Association for Computing Machinery), 843–859. doi: 10.1145/3651890.3672216

Choi, S., Patel, D., Zad Tootaghaj, D., Cao, L., Ahmed, F., Sharma, P., et al. (2024). FEDNIC: enhancing privacy-preserving federated learning via homomorphic encryption offload on smartnic. *Front. Comput. Sci.* 6:1465352 doi: 10.3389/fcomp.2024.1465352

Choi, S., Shahbaz, M., Prabhakar, B., and Rosenblum, M. (2020). "λ-NIC: Interactive serverless compute on programmable smartnics," in *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)* (Singapore: IEEE), 67–77. doi: 10.1109/ICDCS47774.2020.00029

Dang, H. T., Sciascia, D., Canini, M., Pedone, F., and Soulé, R. (2015). "Netpaxos: consensus at network speed," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, SOSR '15* (New York, NY: Association for Computing Machinery), 1–7. doi: 10.1145/2774993.2774999

Das, S., Saraf, M., Jagadeesh, V., Amardeep, M., and Phalachandra, H. (2021). "Deduplication of Docker image registry," in *2021 IEEE Madras Section Conference (MASCON)* (Chennai: IEEE), 1–8. doi: 10.1109/MASCON51689.2021.9563465

Daw, N., Bellur, U., and Kulkarni, P. (2021). "Speedo: fast dispatch and orchestration of serverless workflows," in *Proceedings of the ACM Symposium on Cloud Computing, SoCC '21* (New York, NY: Association for Computing Machinery), 585–599. doi: 10.1145/3472883.3486982

Firestone, D., Putnam, A., Mundkur, S., Chiou, D., Dabagh, A., Andrewartha, M., et al. (2018). "Azure accelerated networking:$*SmartNICs*$ in the public cloud," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 51–66. Available at: https://www.usenix.org/conference/nsdi18/presentation/firestone

GoogleCloud (2023). *Google Cloud Run*. Available at: https://cloud.google.com/run/docs (accessed June, 2024).

Gummadi, R. (2023). *Preference learning with automated feedback for cache eviction*.

Huang, B. (2018). *Deduplication in container image storage system: Model and cost-optimization analysis*.

Jin, X., Li, X., Zhang, H., Soulé, R., Lee, J., Foster, N., et al. (2017). "Netcache: balancing key-value stores with fast in-network caching," in *Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17* (New York, NY: Association for Computing Machinery), 121–136. doi: 10.1145/3132747.3132764

Kfoury, E. F., Choueiri, S., Mazloum, A., AlSabeh, A., Gomez, J., and Crichigno, J. (2024). A comprehensive survey on smartnics: architectures, development models, applications, and research directions. *IEEE Access* 12, 107297–107336. doi: 10.1109/ACCESS.2024.3437203

Kim, D., Memaripour, A., Badam, A., Zhu, Y., Liu, H. H., Padhye, J., et al. (2018). "Hyperloop: group-based nic-offloading to accelerate replicated transactions in multi-tenant storage systems," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (New York, NY: ACM), 297–312. doi: 10.1145/3230543.3230572

Liu, X., Wen, J., Chen, Z., Li, D., Chen, J., Liu, Y., et al. (2023). Faaslight: general application-level cold-start latency optimization for function-as-a-service in serverless computing. *ACM Trans. Softw. Eng. Methodol.* 32, 1–29. doi: 10.1145/3585007

Mangal, A., Jain, J., Guliani, K. K., and Bhalerao, O. (2020). Deap cache: deep eviction admission and prefetching for cache. *arXiv* [Preprint]. arXiv:2009.09206. doi: 10.48550/arXiv:2009.09206

Mann, T. (2022). *Gke image pull using streaming*. Available at: https://www.theregister.com/2022/06/14/alibaba_dpu_cloud/ (accessed February, 2024).

Manner, J., Endreß, M., Heckel, T., and Wirtz, G. (2018). "Cold start influencing factors in function as a service," *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)* (Zurich: IEEE), 181–188. doi: 10.1109/UCC-Companion.2018.00054

Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux J.* 2014:2.

MicrosoftAzure (2023). *Azure container instances*. Available at: https://learn.microsoft.com/en-us/azure/container-instances/ (accessed June, 2024).

NVIDIA (2024). *NVDIA Bluefield Networking Platform*. Available at: https://www.nvidia.com/en-us/networking/products/data-processing-unit/ (accessed June, 2024).

Pan, L., Wang, L., Chen, S., and Liu, F. (2022). "Retention-aware container caching for serverless edge computing," in *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications* (London: IEEE), 1069–1078. doi: 10.1109/INFOCOM48880.2022.9796705

Patel, D., and Choi, S. (2023). "Smartnic-powered multi-threaded proof of work," in *2023 Fifth International Conference on Blockchain Computing and Applications (BCCA)* (Kuwait: IEEE), 200–207. doi: 10.1109/BCCA58897.2023.10338942

Quest, K. (2024). *Slim toolkit*. Available at: https://github.com/slimtoolkit/slim (accessed June 3, 2024).

Saxena, D., Ji, T., Singhvi, A., Khalid, J., and Akella, A. (2022). "Memory deduplication for serverless computing with medes," in *Proceedings of the Seventeenth European Conference on Computer Systems, EuroSys '22* (New York, NY: Association for Computing Machinery), 714–729. doi: 10.1145/3492321.3524272

Sethi, B., Addya, S. K., and Ghosh, S. K. (2023). "LCS: alleviating total cold start latency in serverless applications with LRU warm container approach," in *Proceedings of the 24th International Conference on Distributed Computing and Networking, ICDCN '23* (New York, NY: Association for Computing Machinery), 197–206. doi: 10.1145/3571306.3571404

Wang, L., Li, M., Zhang, Y., Ristenpart, T., and Swift, M. (2018). "Peeking behind the curtains of serverless platforms," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA: USENIX Association), 133–146.

Wen, Z., Chen, Q., Deng, Q., Niu, Y., Song, Z., Liu, F., et al. (2024). Combofunc: joint resource combination and container placement for serverless function scaling with heterogeneous container. *IEEE Trans. Parallel Distrib. Syst.* 35, 1989–2005. doi: 10.1109/TPDS.2024.3454071

Yang, D., Berger, D. S., Li, K., and Lloyd, W. (2023). A learned cache eviction framework with minimal overhead. *arXiv* [Preprint]. arXiv:2301.11886. doi: 10.48550/arXiv:2301.11886

Zhao, N., Lin, M., Albahar, H., Paul, A. K., Huang, Z., Abraham, S., et al. (2024). An end-to-end high-performance deduplication scheme for Docker registries and Docker container storage systems. *ACM Trans. Storage* 20:18. doi: 10.1145/3643819

Zhao, N., Tarasov, V., Anwar, A., Rupprecht, L., Skourtis, D., Warke, A., et al. (2019). "Slimmer: weight loss secrets for Docker registries," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)* (Milan: IEEE), 517–519. doi: 10.1109/CLOUD.2019.00096