ImageHarbour

Kiran Hombal, Jiyu Hu, Shreesha G. Bhat
University of Illinois at Urbana-Champaign
{khombal2,jiyuhu2,sgbhat3}@illinois.edu

Abstract

In this paper, we present ImageHarbour, a novel image caching mechanism that leverages stranded memory [5] from multiple hosts to create a large, unified memory pool in data centers [1]. By utilizing one-sided RDMA, which offers latencies in the sub-microsecond range [3] compared to 5-10 milliseconds for disk access [6] and seconds to minutes for Docker pull over the internet [2], ImageHarbour efficiently accesses memory on remote nodes without interrupting the CPU, enabling effective utilization of underutilized resources. This approach can improve latency by up to 190X compared to traditional methods.

1 Introduction

In modern data centers, the efficient utilization of resources is crucial for optimal performance and cost-effectiveness. One of the challenges faced by these systems is the presence of stranded memory, which refers to the allocated but unused memory on remote nodes as seen in Figure 1, there is over 30% of the memory is stranded. This stranded memory often coexists with overUtilized CPUs, leading to an imbalance in resource utilization. To address this issue, we propose ImageHarbour, a novel image caching mechanism that aggregates stranded memory from multiple hosts into a single, large memory pool.

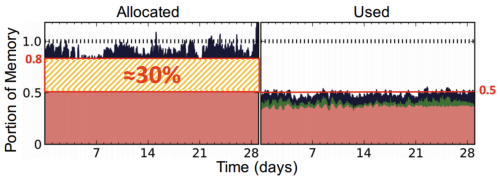

Figure 1. stranded memory

ImageHarbour draws inspiration from two key concepts: stranded memory and efficient memory disaggregation. The notion of stranded memory is derived from the study by Reiss et al. [5], which highlights the presence of allocated but unused memory in Google's data centers. Their analysis reveals that while some nodes have high CPU utilization, their memory remains underutilized, resulting in stranded memory. On the other hand, the idea of efficient memory disaggregation is borrowed from the work of Gu et al. [1], which proposes Infiniswap, a remote memory paging system that leverages one-sided Remote Direct Memory Access (RDMA) operations to access memory on remote nodes without interrupting the CPU.

RDMA is a high-performance networking technology that allows direct memory access from the memory of one computer to that of another without involving either computer's operating system [3]. This enables low-latency and high-bandwidth communication between nodes in a data center. By exploiting one-sided RDMA operations, ImageHarbour can efficiently access memory on remote nodes with minimal overhead, achieving latencies in the sub-microsecond range. In contrast, traditional disk-based storage systems have latencies around 5-10 milliseconds for a 4KB read operation [6], while retrieving Docker images over the internet can take seconds to minutes, depending on the image size and network conditions [2].

ImageHarbour leverages these concepts to create a distributed image caching system that can significantly improve the performance of image-based workloads in data centers. By pooling together stranded memory from multiple hosts, ImageHarbour creates a large, unified memory pool that can be used to cache frequently accessed Docker images. The system employs a control plane that intelligently manages the cache, deciding which images to bring in, which images to evict, and serving metadata to clients regarding the location of images within the memory pool.

Clients accessing the cached images through ImageHarbour experience significantly reduced latencies compared to traditional image retrieval methods. By serving images directly from memory using one-sided RDMA operations, ImageHarbour can achieve up to 83X lower latency compared to storing the image on the disk and upto 190X lower latency compared to downloading images over the internet.

2 System Design

ImageHarbour is designed as a distributed image caching system that leverages stranded memory resources across multiple hosts in a data center. The system architecture consists of three main components: the control plane, the memory pool, and the client nodes. Figure **??** provides an overview of the ImageHarbour system design.

2.1 Control Plane

The control plane is responsible for managing the overall operation of ImageHarbour. It makes intelligent caching decisions, coordinates the allocation and deallocation of memory resources, and maintains metadata about the cached images. The control plane consists of the following subcomponents:

2.1.1 Image Metadata Store

The image metadata store maintains information about the cached images, including their unique identifiers, sizes, access frequencies, and locations within the memory pool.

2.1.2 Caching Policy Engine

The caching policy engine determines which images should be cached in the memory pool and which images should be evicted when the cache reaches its capacity. It can employ algorithms, such as Least Recently Used (LRU) or Adaptive Replacement Cache (ARC) [4], to make caching decisions based on factors like image popularity and access patterns.

2.1.3 Memory Allocator

The memory allocator manages the allocation and deallocation of memory resources within the memory pool. It keeps track of the available memory on each host and assigns memory regions to cached images based on their sizes and access frequencies.

2.2 Memory Pool

The memory pool is a distributed cache that aggregates the stranded memory resources from multiple hosts in the data center. It serves as a high-performance storage layer for cached Docker images, allowing fast access to frequently used images.

Each host in the memory pool registers with the control plane. Control plane distriutes which images to cache and performs one-sided RDMA write to the available region. It then stores meta data of that operation. When a client node requests an image, the control plane provides the location of the image in the memory pool, and the client node directly reads the image data using one-sided RDMA operations.

2.3 Client Nodes

Client nodes are the consumers of the cached Docker images. They interact with ImageHarbour to retrieve images from the memory pool instead of fetching them from disk or over the network. When a client node requests an image, it first consults the control plane to obtain the location of the image in the memory pool. If the image is found in the cache, the client node establishes an RDMA connection with the appropriate host in the memory pool and directly reads the image data using one-sided RDMA operations.

If the requested image is not present in the cache, the client node falls back to the traditional method of retrieving the image from disk or over the network. In such cases, the control plane may decide to cache the image in the memory pool for future requests, based on the caching policy engine's decisions.

2.4 Scalability

ImageHarbour is designed to scale horizontally by adding more hosts to the memory pool as the demand for cached images grows. The control plane dynamically manages the distribution of images across the memory pool hosts to achieve load balancing.

3 Experimental Setup and Evaluation

To evaluate the performance of ImageHarbour, we conducted experiments comparing it with three other image retrieval methods: local disk access, private registry access, and Docker Hub access. The experiments were designed to measure the latency and throughput of image retrieval under different scenarios and image sizes.

3.1 Experimental Setup

The experimental setup consisted of a cluster of machines connected via a high-speed network. Each machine was equipped with an Intel Xeon processor, 128 GB of RAM, and a 1 TB SSD. The machines were running Ubuntu 22.04 LTS and had Docker installed.

We set up the following four systems for comparison:

1. **Local Disk**: In this setup, the required Docker image was already present on the local disk of the host machine.
2. **Private Registry**: We deployed a private Docker registry within the same cluster as the client machines. The required Docker images were hosted on this private registry.
3. **Docker Hub**: The client machines fetched the required Docker images directly from Docker Hub, the public Docker image registry.
4. **ImageHarbour**: Our proposed ImageHarbour system was deployed on the cluster, with the control plane, memory pool, and client nodes set up according to the architecture described in Section 2.

To evaluate the performance of each system under different image sizes, we selected three representative Docker images:

1. **Hello-World**: A lightweight Docker image with a size of less than 1 MB. This image represents small-sized images commonly used for testing and simple applications.
2. **Alpine**: A popular lightweight Linux distribution image with a size of 3-4 MB. Alpine is widely used as a base image for containerized applications due to its small footprint.
3. **Debian**: A larger Docker image based on the Debian Linux distribution, with a size exceeding 100 MB. This image represents more substantial application images that include a full-fledged operating system and additional dependencies.

In the following subsections, we present the results of our experiments and discuss the performance of ImageHarbour compared to the other image retrieval methods.

3.2 Results

3.2.1 Hello-World Image

Table 1 presents the image retrieval times for the Hello-World image across different systems.

System	Time (us)
Disk	130350.038
Local Registry	150416.970
Docker Hub	1226197.943
ImageHarbor	16608.400

Table 1. Image retrieval times for Hello-World image

Figure **??** shows a graph comparing the image retrieval times for the Hello-World image.

3.2.2 Debian Image

Table 2 presents the image retrieval times for the Debian image across different systems.

System	Time (us)
Disk	1387899.586
Local Registry	1387751.932
Docker Hub	3167976.727
ImageHarbor	16608.000

Table 2. Image retrieval times for Debian image

Figure **??** shows a graph comparing the image retrieval times for the Debian image.

4 Conclusion

This project is awesome.

5 Metadata

The presentation of the project can be found at:

https://zoom/cloud/link/

The code/data of the project can be found at:

https://github.com/you/repo

References

[1] Gu, J., Hu, Y., Zhang, T., Guo, Z., Cheng, M., Jin, X., Li, J., Dadiomov, N., Williams, K., and Ng, K. Efficient memory disaggregation with infiniswap. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)* (2017), pp. 649–667.

[2] Harter, T., Salmon, B., Liu, R., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Slacker: Fast distribution with lazy docker containers. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (2016), pp. 181–195.

[3] Kalia, A., Kaminsky, M., and Andersen, D. G. Design guidelines for high performance rdma systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (2016), pp. 437–450.

[4] Megiddo, N., and Modha, D. S. Arc: A self-tuning, low overhead replacement cache. In *Fast'03: 2nd USENIX Conference on File and Storage Technologies* (2003), pp. 115–130.

[5] Reiss, C., Wilkes, J., and Hellerstein, J. L. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing* (2012), pp. 1–13.

[6] Yang, J., Plasson, N., Gillis, G., Talagala, N., and Sundararaman, S. Don't stack your log on my log. In *2nd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads (INFLOW)* (2014), vol. 71.