Enabling Live Migration of Containerized Applications Across Clouds

Thad Benjaponpitak Meatasit Karakate Kunwadee Sripanidkulchai [*]
Department of Computer Engineering, Chulalongkorn University
Bangkok, Thailand

***Abstract*—Live migration, the process of transferring a running application to a different physical location with minimal downtime, can provide many benefits desired by modern cloud-based systems. Furthermore, live migration between different cloud providers enables a new level of freedom for cloud users to move their workloads around for performance or business objectives without having to be tied down to any single provider. While this vision is not new, to-date, there are few solutions and proof-of-concepts that provide this capability. As containerized applications are gaining popularity, we focus on the design and implementation of live migration of containers across cloud providers. CloudHopper, our proof-of-concept live migration service for containers to hop around between Amazon Web Services, Google Cloud Platform, and Microsoft Azure is evaluated using a common web-based workload. CloudHopper is automated and supports pre-copy optimization, connection holding, traffic redirection, and multiple interdependent container migration. It is applicable to a broad range of application use cases.**

***Index Terms*—containers, live migration, cloud computing**

I. INTRODUCTION

The popularity of deploying containerized applications in the cloud has been increasingly gaining momentum as container technologies are maturing. Recent offerings based on container technologies for Platform-as-a-Service (PaaS) clouds, such as Amazon Elastic Container Service [1] and Google Kubernetes Engine [2] are a testament to this new demand. Containers are becoming the de-facto standard for deployment whether they are deployed inside Infrastructure-as-a-Service (IaaS) virtual machines (VMs) or deployed in cloud container services.

With the rise of containers, we revisit a long-desired capability to freely move around application workloads across different data centers and cloud providers. If containers can be live-migrated with no client-perceived downtime, cloud users would have the flexibility to run their services on the right provider at the right time, enabling a number of useful application management capabilities such as load balancing, resource management, and live maintenance [3]–[7].

Live application migration is the process of transferring the state of a running application to a remote location where it will be restored with minimal downtime. Typically, live migration is performed on a hypervisor-based virtual machine (VM) in order to migrate its running operating system and processes. While it is feasible to migrate VMs across the wide area [8], [9], it may not be practical for cloud users as it requires hypervisor-level functionality that is not readily accessible by cloud users.

Instead of VMs, containers are an alternative virtualization technology commonly used to deploy applications. Containers are a more application-focused solution [7] as abstractions of system calls and resources are provided directly to each process [10], but isolated from the default resource context of the host machine. This isolation is normally implemented using namespaces and control groups [11], [12]. Namespaces are used to confine a process to its own resource group. For example, a container can be assigned a network namespace that only allows it to see a specific set of network interfaces and a PID namespace that limits the processes it can see to the ones that are running inside it. Additionally, control groups are used to limit the amount of resources a certain process can use. For example, the memory control group can be used to limit the amount of memory a process can utilize. These two core concepts make containerization possible.

In this paper, we focus on live migration of applications running within multiple containers. Our contributions are a fully automated live migration solution, CloudHopper that enables containerized applications to *hop* around different clouds with no client-perceived downtime. We have successfully migrated applications between three of the most popular cloud providers (Amazon Web Services, Google Cloud Platform, and Microsoft Azure) across the wide area. For realistic application workloads, using pre-migration and migration optimizations, migrated containers may be down for under half a minute. However, during that period, CloudHopper maintains connectivity such that clients only perceive a delayed response, instead of actual downtime. Cloud users can simply use our automated solution to live-migrate entire applications or parts of applications, hopping out of one cloud into another cloud, or hopping between data centers. The main ideas that distinguish CloudHopper from previous work are *automated container workflow, multi-container support, realistic workloads, real-world conditions using commercial clouds, and zero downtime migration with connection holding and traffic redirection*. We believe this is a step towards achieving cloud interoperability [13] with increased mobility and ease of management [14].

II. BACKGROUND

In this section, we present the background on live migration and support for live migration in containers.

[*]Corresponding author. Email: kunwadee @cp.eng.chula.ac.th.

A. Live Migration

There are two types of live memory migration: (1) process migration, and (2) virtual machine migration. The former focuses on migrating a specific group of processes, while the latter migrates the memory state of the entire system altogether from the operating system up. Process migration is lightweight, but more restrictive due to problems of dependencies of the migrated process being left on the host machine [15], [16]. Virtual machine migration is less restrictive but can create unnecessary performance overhead from migrating the operating system when the goal is solely to migrate an application [17].

Container migration can be classified as a type of process migration, since a running container is simply a set of processes running in an isolated environment. Although containers provide less isolation than traditional virtual machines, from the point of view of the users, a container functions very similarly to a virtual machine [10]. This makes container migration a middle ground between process migration and virtual machine migration.

There are many different implementations of container-based live migration. Mainly, there are kernel-based implementations (BLCR [18], CRAK [19], legacy OpenVZ [20], Zap [21], Linux-CR [4]) and user space ones (DMTCP [22], CRIU [5]). Kernel-based implementations have more capabilities and are able to fully migrate the system state of a process. However, they are highly dependent on loading a kernel module or modifying the kernel itself [4]. As a result, user space approaches were introduced with a common goal of avoiding kernel modification. Many of these implementations require loading libraries or modules in advance, reducing their general ease of use [16]. CRIU is one of the most actively developed implementations, works well with a standard kernel, and requires no changes to application code. Thus, CloudHopper is built on top of CRIU's memory migration capabilities.

Note that to fully support live container migration across clouds, migrating just the memory state is insufficient. We will also need to migrate storage and networks, as described in Section III.

B. Migration Support for Containers

Next, we discuss existing container technologies and their support for live migration. Docker [23] is the most popular implementation of operating system-level virtualization. Compared to traditional virtual machines, Docker focuses more on running applications and less on emulating hardware. Its developer-centric features are also easy to use, making it widely popular [24]. Given its mainstream adoption, we would like to implement our work to support Docker containers. However, according to our initial experiments, Docker's integration with CRIU (since version 3.11) is experimental and not mature enough to function well in practice.

Therefore, we consider a lower-level container technology that is also used by Docker, runC [25] as an alternative. Docker uses runC under containerd, an API backend for accepting action requests from the frontend Docker binary, to manage containers [26]. However, runC is designed to be more generic than Docker and can be used on its own to manage containers.

CloudHopper supports runC containers. One key disadvantage of using runC is that container networking is not supported, so we will need to manually create a networking stack to enable containers to communicate. Details of our container networking approach is further explained in Section IV-A2.

III. System Design

In this section, we discuss the design of CloudHopper to enable live migration of containerized applications across clouds. Our design goals are as follows:

1) Multi-cloud support. Containers can be live migrated between different cloud providers across the wide area.
2) Interdependent container support. Web applications are often deployed as multiple interdependent containers that need to be migrated together.
3) Short migration time. Containers are not responsive during migration, so fast migrations will make the service appear live and responsive to clients.
4) Secure data transfer. Live migration requires transfer of system state possibly containing sensitive client and service information which should be encrypted.
5) No failed client connections. Containers are not responsive during migration, so we need a hold and redirect mechanism to handle new incoming client connections during migration.
6) Automated migration. Live migration requires setting up the right environment and coordinating between source and target sites. Automating these steps is desirable for ease of use.

Next, we discuss each design goal and solution in details.

A. Multi-Cloud Support

CloudHopper needs to work between multiple clouds, so our design should be agnostic to the cloud provider. All required software and components should be able to run on any provider and should not need to use any provider-specific services.

We implement and evaluate our system on commercial clouds by migrating between Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure as they are the top three most popular cloud providers [27]. Our experiments are the first of its kind that demonstrate live migration of realistic workloads between commercial providers across the wide area. This is in contrast to previous work that have experimented with migrating containers between hosts inside a testbed or inside the same enterprise data center [28], [29].

B. Interdependent Container Support

We support migrating a common realistic setup where multiple interdependent containers are deployed together as a fully functional web application. These containers need to work together to service client requests. For example, WordPress, is a commonly used website software that consists of a MySQL and a WordPress container. While live migration of multiple interdependent VMs has been previously studied [8], [30],

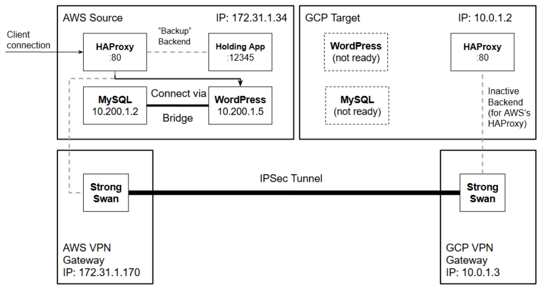

Fig. 1. System architecture with containers running at the source site before migration.

[31], we are the first to design and evaluate the performance of live migration on interdependent containers.

A key issue in designing CloudHopper to support interdependent container migration is network migration. In order for components to easily locate each other after migration, their IP addresses should not change. We implemented basic container networking using network namespaces to emulate ”bridge mode” networking in Docker and solve this problem for runC containers as described in Section IV. In order to keep the same IP addresses, an IPSec VPN as shown in Figure 1 is set up between the source and target hosts across different cloud providers such that all hosts and containers can communicate using private IP addresses in the same subnet. This VPN tunnel has an added benefit as it allows data between the two sites to be transferred securely. We use strongSwan, an open source IPSec implementation [32] for this. An additional host is created on each side to act as a NAT gateway as shown at the bottom of Figure 1.

C. Short Migration Time

Typically, container live storage and memory migration can be broken down into three basic steps:

i Checkpoint: perform a memory checkpoint freezing the container and dumping all of the container’s memory state to a set of files using CRIU,
ii Transfer: transfer the checkpointed memory state and the container’s local persistent storage from the source to the target, and
iii Restore: restore the container at the target.

During all three steps, the container is non-responsive and appears down. The downtime could vary depending on the amount of active memory that is checkpointed, the amount of local persistent storage the container uses, and available bandwidth in the network between source and target.

In order for storage and memory migration to appear live, all three steps need to take a very short amount of time. CloudHopper’s approach is to reduce the amount of memory state and persistent storage that needs to be checkpointed/transferred while the container is down by *pre-copying and transferring only changes during downtime.*

We introduce a set of pre-migration steps that perform storage and memory pre-copy prior to the actual migration, thus the name pre-copy. For storage pre-copy, we first copy the image from the source to the target. Then, during migration, we use rsync to copy only the changes from the last copied version. For memory pre-copy, we use CRIU’s iterative migration capability which can pre-dump a full copy of the memory state, keep track of changes, and iteratively dump changes. The full copy is large as it is the same size as the amount of memory used by the container, whereas iterative dumps are typically smaller. A pre-dump does not freeze the container, but it may have an impact on its performance. With this approach, the pre-dump can be transferred ahead of time during pre-migration while the container is still running and responsive. Then during migration, the iterative dump is transferred using rsync resulting in a shorter downtime.

Putting pre-migration and migration together, we have

i Pre-dump: checkpoint the container’s memory using CRIU’s pre-dump capabilities,
ii Pre-copy: transfer the pre-dumped memory state and the container’s local persistent storage from the source to the target,
iii Checkpoint: perform a memory checkpoint, freezing the container and dumping memory state *changes* to a set of files called a “checkpoint image” using CRIU’s iterative checkpoint,
iv Transfer: transfer the checkpointed memory state *changes* and the container’s local persistent storage *changes* from the source to the target using rsync,
v Restore: restore the container at the target.

The container is down in steps (iii)-(v), which, by design, takes a shorter amount of time than the basic approach.

We further considered reducing the overhead of dumping to local disk in step (iii) and copying from local disk over the network in step (iv) by using Network File System (NFS) to perform these two steps as one single action. We set up an NFS server at the target site and mount it on the source and target as a networked file system. Then, we execute steps (iii) and (iv) together by dumping the memory state directly to a directory on the NFS server, automatically transferring it to the target. Also, for step (iv) we copy the image at the source to a directory on the NFS server which again is automatically transferred to the target. However, we found in our initial evaluation that NFS introduced significant performance overhead and our downtime for this approach was high. Therefore, this optimization is excluded from our evaluation.

We also introduce a few more optimizations that attempt to shorten the transfer time in step (iv) as discussed and evaluated in Section IV-C.

D. Secure Data Transfer

During live migration, persistent storage and memory state that contains sensitive client and service information such as passwords, session keys, and transaction information is transferred. Securely transferring such data over an encrypted channel is required. The VPN tunnel between source and target sites as discussed in Section III-B provides many benefits, including a secure channel that we use for data transfer.

E. No Failed Client Connections During Migration

During the short container downtime in steps (iii)-(v), we must continue to appear *live* to existing clients and any new incoming clients. Therefore, CloudHopper incorporates a hold and redirect mechanism at the source which holds incoming traffic during migration and eventually redirects them to the target when the service is restored and ready.

In our system, we build a web application to hold connections called the *holding application* and use HAProxy [33] to handle redirection. HAProxy is already used to handle incoming connections at the host to support multiple container networking as to be discussed later in Section IV-A2. During normal operations, HAProxy would forward incoming client connections to the web server container. However, during migration, new incoming client connections are instead forwarded from HAProxy to the holding application. The holding application keeps the connections alive until it detects that the target is up. It then redirects all held connections to the target. In addition, HAProxy will periodically run a health check on the target site and redirect all clients to the target site when the site is up. With these two components, it is guaranteed that a web-based application will not drop any connections, given that the server is working under normal load conditions, i.e., not overloaded.

For the HAProxy configuration, source containers, target containers, and the holding application are set up as "backends". Prior to migration, only the source container is enabled in the configuration while both the target and the holding application are disabled. The holding application is also designated as the backup backend, meaning it will only be used when both normal backends (source or target containers) are unavailable. This is intentional because it should only be accessible during migration per its design.

CloudHopper's holding application is a Python-based web application developed with Flask [34]. We chose Flask as it is easy to setup and can perform connection buffering as desired. The application is then deployed with Gunicorn [35], a Python-based web server which can serve the application with good performance. The holding application works by first accepting incoming connections from HAProxy. Then, it periodically polls the target service with HTTP requests until it receives a desired response. Finally, it will return an HTTP 302 response to redirect the client to the target site. In order to ensure that target site polling is not excessive, we use a Memcached [36] instance to store the status of the target site as global state for all holding application threads to check before polling.

During the same time, the HAProxy instance on the source host will also be polling the target backend, so the remaining connections in its own queue will also be automatically dispatched to the (now online) target.

To ensure that the holding application does not become overloaded, HAProxy will send only a few (5-10) connections to the holding application while the remaining client connections will wait in HAProxy's queue. The holding application is not as robust as HAProxy, so we use HAProxy to handle the majority of the load. We also set the HAProxy queue timeout to a value long enough to cover the migration duration.

Note that although HAProxy has a queue, its queue cannot completely replace the holding application's functionality. This is because HAProxy was not designed for this purpose and always requires an available backend to function. Without a holding application as an available "backend" during migration, HAProxy would mark the service as down and immediately return a HTTP 503 Service Unavailable response while the source container is being migrated. We experimented with many HAProxy settings to see if we could avoid using a holding application such as setting the frontend connection limit in HAProxy to zero, effectively preventing connections from being forwarded to the backends regardless of availability. However, several clients experienced connection resets during migration and HTTP 503 responses. As a result, the holding application is *required* to guarantee liveness.

This design, combined with transfer time optimizations, will make a client connection during migration appear as experiencing slow connectivity. Clients do not know that the server is being migrated or is under maintenance. Our holding application solution is designed specifically for web-based applications. We have yet to explore solutions for other types of applications, but plan to explore them as part of future work.

F. Automated Migration

The process of setting up the system architecture and performing live migration is automated using Ansible [37], an open-source software provisioning, configuration management, and application-deployment tool. This allows anyone to easily reproduce the environment and control it during the migration process. This also facilitates evaluation of the system because the migration process can be repeatedly performed in a controlled manner. Since Ansible is already used in many cloud-based application deployments, this will make it easy for end-users to incorporate our work into their existing workflow.

IV. System Implementation

In this section, we describe the details of our system implementation for the containers, automated live migration, and live migration optimizations.

A. Containers

Our approach to container creation and networking to support deployment of applications is discussed next.

1) Container creation: Our containers are created using Docker images. Although our implementation uses runC, we use Docker images as they represent state-of-the-art common real-life applications.

Initially, Docker images are pulled from the image server (DockerHub) to the host. Skopeo [38] is used to convert each image into an Open Containers Initiative (OCI) compliant image, which is then extracted into an OCI bundle with umoci [39]. The bundle, at this stage, is ready for container creation. However, we still need to provide parameters such as port mappings, volume mappings, and resource requests.

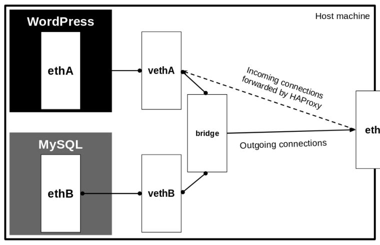

Fig. 2. Container networking setup using namespaces, virtual interfaces, bridge, and HAProxy.

These options will be translated from Docker's format and then specified in the configuration file in the extracted OCI bundle accordingly.

2) Container networking: Most basic use cases for Docker containers involve the use of port mappings and inter-container network links. Networking is essential because containers are usually designed to be self-contained (per the microservices architecture [40]) and thus require links to other containers to function together as a system. However, runC does not directly support networking between containers. We can implement container networking by creating a network namespace for the container.

Figure 2 depicts the networking setup used on both source and target hosts to support container networking in runC. We create a bridge interface between the host and the containers. Additional containers on the same host can be added to the same bridge to enable them to communicate with each other. We then create a pair of virtual Ethernet interfaces per container to connect between the bridge and the container's network namespace.

Finally, the container will need to communicate to external clients. A service on the host which can redirect requests from the host to the appropriate container is needed. We use HAProxy for this purpose. Note that HAProxy is also used for other purposes as discussed in Section III-E.

Initially, we experimented with using Socat [41] for this purpose instead of HAProxy because it is easy to set up Socat to forward incoming connections seamlessly. However, Socat has limited capabilities as it can only directly forward incoming connections. When the container is being migrated, Socat cannot hold incoming connections, leading to perceived application downtime. Connection holding is needed to support live migration. Therefore, we use HAProxy not only to provide networking between the containers and the outside world, but also to avoid perceived application downtime along with the holding application described in Section III-E.

B. Migration

CloudHopper uses Ansible to automate the entire migration solution from setting up the source and target hosts and environment, to running the migration, and tearing down the setup. Currently, our implementation supports migration between three cloud providers (AWS, GCP and Azure).

Assuming that the source host is already running the Wordpress and MySQL containers, our automated workflow provisions target hosts and VPN hosts on both sides. Then, it configures the cloud provider's VPN (i.e., AWS VPC) settings to have the source and target hosts use the respective source and target VPN hosts as gateways. Container images are placed at the target before migration to set the persistent storage's initial state. HAProxy is also set up on both sides to forward traffic to the application. Refer back to Figure 1 for the initial state before a migration takes place.

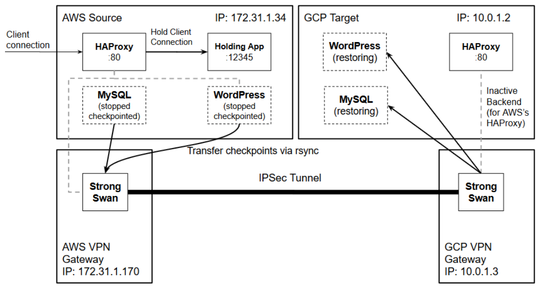

Fig. 3. Restoring containers at the target site in the final stage of migration.

In our implementation, the migration process can be broken down into five steps as discussed in Section III. We coordinate the migration steps with HAProxy and our holding application as follows:

i Pre-dump using CRIU.
ii Pre-copy transfer using rsync over the VPN.
iii Checkpoint using CRIU. At this point the container is down, so HAProxy is instructed to disable the source container in its list of backends, and the holding application is enabled in its place.
iv Checkpoint transfer using rsync over the VPN. Optimizations in this step to reduce transfer time are discussed in Section IV-C.
v Restore after the transfer completes. We instruct the target host to immediately restore the container. When migrating interdependent containers, we must be careful about when we issue restores. We do not want the web container up and running, handling client requests while the database container is still being migrated over, as that would cause application-level errors. We discuss a scheduling optimization in Section IV-C to ensure that containers are restored at the same time to shorten restoration delays.

As we have multiple containers, each container is similarly migrated in parallel. The entire migration process is handled by a Python script called by Ansible to run on both hosts. Subprocesses are created to handle each container. Figure 3 depicts the overall migration process at step (v) when we are restoring the containers at the target.

After the restore, the holding application polls the application at the target site until the target site is ready and all held connections are redirected. The holding application is then disabled from HAProxy's list of backends. Ideally, DNS records should be updated to point to the target site after this step, but in actual production deployments DNS propagation takes time. Some clients may still access the application at the source site. To handle these new incoming connections, we keep HAProxy and the VPN running. HAProxy will

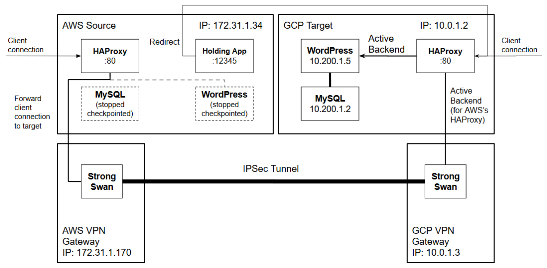

Fig. 4. System after migration.

automatically use the target host as a new "backend" once it has detected that the target is up. The set up at the source host can then be gracefully taken down once client connections no longer arrive at the source. Figure 4 depicts the system after a successful migration. Migrated containers continue to function like regular containers. They can be migrated again using the same workflow.

C. Optimizations

We employ the following optimizations in CloudHopper to increase migration performance.

1) Scheduling: To ensure that all interdependent containers can be restored without causing application-level errors in the checkpoint transfer step (step iv), the transfer of each container can be scheduled relative to each other so that every container finishes its transfer at the same time. This is done by sorting all containers by transfer size. We schedule the largest container first. We schedule transfer of the next container when the previous container has a remaining transfer size that is equal to its transfer size, and so on. In contrast to sequential transfers and parallel transfers without scheduling, scheduling uses network bandwidth more efficiently and enables us to start all containers immediately upon arrival at the target. However, in order to measure the transfer size, an extra step needs to be performed before the actual transfer in which we use rsync to perform a transfer dry run and record the total transfer size from its output.

2) Compression: Using rsync's `-z` command line option, data can be compressed and then transferred to the target host. This reduces the network bandwidth used during migration. We use the default compression level implemented in rsync.

V. Evaluation

In this section, we describe our experimental setup, evaluation methodology, and evaluation metrics.

A. Experimental Setup

The goal of our evaluation is to migrate common web-based workloads running on containers. While there are no standard benchmarks, we use WordPress as the target application as it is a popular choice for creating websites. Thus, we will be live-migrating an application consisting of two containers, a WordPress Apache httpd container and a MySQL container.

In order to test the performance of CloudHopper, we evaluate the impact of client load on migration performance. We use a load generator, Siege [42], and vary the number of concurrent clients issuing POST requests to the WordPress container. Note that this creates a mix of read/write workload not only on the WordPress container, but also consequently on the MySQL container. The set of URLs selected are from the ones available in the WordPress REST API. For all of our experiments, Siege tries to randomly add new content to five parts of a Wordpress blog, namely posts, comments, categories, tags, and users via a POST request to the corresponding API endpoint.

Siege, by default, generates load to a static set of HTTP requests, but WordPress does not allow the same POST request (same content) to be submitted more than once. If the same request shows up, WordPress does not create a new blog element. In order to create new posts throughout our migration experiments, Siege was modified to add string randomization functionality so that every HTTP request is unique. The flag for our custom option is "`-s`" or "`--random-strings`" which will replace every region in the POST body starting with a "`" and ending with a "^" with a random string of the same length (inclusive of the region markers).

TABLE I
Machine specifications.

Host	Provider	Machine type	vCPUs	RAM (GB)	Region
Source	AWS	t3.medium	2	4	ap-northeast-1
Source VPN	AWS	t3.small	2	2	ap-northeast-1
Target	GCP	n1-standard-1	1	3.75	asia-northeast-a
Target VPN	GCP	n1-standard-1	1	3.75	asia-northeast-a
Target	Azure	Standard D1 v2	1	3.5	Japan East
Target VPN	Azure	Standard D1 v2	1	3.5	Japan East
Client	Azure	Standard D2s v3	2	8	Japan East

TABLE II
Experiment scenarios.

Scenario Name	Workload		Optimization		
	Concurrent connections	*Throughput (transaction/s)*	*Parallel*	*Scheduling*	*Compression*
0c	0	0	✓	✓	
1c	1	27.34	✓	✓	
5c	5	68.12	✓	✓	
10c	10	71.53	✓	✓	
50c	50	68.14	✓	✓	
100c	100	65.74	✓	✓	
200c	200	64.04	✓	✓	
400c	400	63.12	✓	✓	
Unscheduled	400	63.12	✓		
Sequential	400	63.12			
Compressed	400	63.12	✓	✓	✓

B. Methodology

The test environment consists of a source host on Amazon Web Services (AWS) and a target host on Google Cloud Platform (GCP) or Microsoft Azure. When migrating between a pair of providers, a client host on the third provider runs Siege to create load on the application containers during migration.

As mentioned earlier, VPN hosts were also set up on both source and target cloud sites. Specifications of virtual machines used as hosts in our experiments are provided in Table I.

Experiments were performed with varying amounts of workload and different optimizations to evaluate their impact on performance. Each experiment is repeated ten times. Details of experiment scenarios are provided in Table II. Note that Siege allows us to control the number of concurrent clients, but not the transaction rate. The reported transaction rates

TABLE III
AVERAGE TIME SPENT DURING PRE-MIGRATION AND MIGRATION FOR ALL EXPERIMENT SCENARIOS.

Scenario	Pre-Migration			Migration					Total
	Pre-Dump (s)	***Pre-copy (s)***	***Total (s)***	***Checkpoint (s)***	***Diff (s)***	***Transfer (s)***	***Restore (s)***	***Total Downtime (s)***	***(s)***
0c (Azure)	0.300	4.243	4.543	0.587	0.337	0.720	5.460	7.104	11.647
0c (GCP)	0.327	4.487	4.814	0.613	0.350	0.717	5.867	7.547	12.361
1c (GCP)	0.310	4.33	4.640	0.633	0.380	3.420	5.470	9.903	14.543
5c (GCP)	0.433	4.953	5.387	0.713	0.357	4.227	5.347	10.643	16.030
10c (GCP)	0.590	6.133	6.723	0.907	0.383	4.673	5.353	11.317	18.040
50c (GCP)	1.170	7.033	8.203	1.917	0.376	12.710	5.630	20.633	28.836
100c (GCP)	1.015	7.222	8.237	3.930	0.455	11.410	5.222	21.017	29.254
200c (GCP)	0.936	7.053	7.989	4.763	0.503	13.370	5.470	24.106	32.095
400c (GCP)	1.430	8.890	10.320	5.948	0.690	12.370	5.448	24.456	34.776
Unscheduled	1.133	8.506	9.639	5.853	0.590	14.016	5.890	26.349	35.988
Sequential	1.370	7.640	9.010	5.550	0.720	15.080	11.320	32.670	41.680
Compressed	0.956	14.596	15.552	6.680	0.600	33.534	5.008	45.822	61.374

are measured from the experiments. The system throughput is already maxed out at around 60-70 transactions/sec when there are 10 concurrent clients. Beyond that, the system is working under heavy load.

C. Metrics

The following metrics are used to evaluate performance and are averaged over the ten runs for each experiment scenario.

1) Pre-Migration and Migration Time: Each step of the migration process is measured separately, providing a detailed perspective of migration performance.

2) Image Size: In the migration process, two sets of files are created: pre-dump image and checkpoint image. The sizes of these two images directly influence the time needed for their transfer and consequently the total time needed to migrate the application. The size of each image is measured separately with the disk usage (`du`) command.

3) Migration Transfer Throughput: To easily compare between different optimization strategies, migration transfer throughput is also calculated from the actual amount of data transferred during the migration (not including the pre-dump step) and the time taken to transfer.

4) Client Response Time: To measure the impact of live migration on the application's performance from the perspective of a client, client-perceived response times during the migration were recorded from Siege's output.

VI. RESULTS

In this section, experiment results are presented and discussed using the metrics defined in the previous section.

A. Migration Time

CloudHopper's migration time is shown in Table III. Results from all experimental scenarios between AWS and GCP are reported and results from one scenario between AWS and Azure is reported as the results are similar to GCP.

Total time includes the time taken in all steps, whereas downtime only starts from the checkpoint step onwards. Total time and downtime increase as the workload increases in all scenarios because more data needs to be transferred. The total downtime is as short as 7.547 seconds when there is no client workload and up to 24.456 seconds with 400 concurrent connections for AWS to GCP migrations. For heavy workloads, the downtime is roughly the same at around 24 seconds regardless of the number of concurrent clients. Note that during this downtime, roughly a GB of data is being migrated. Migrating from AWS to Microsoft Azure has similar but slightly better performance.

TABLE IV
SIZE OF PRE-DUMP AND CHECKPOINT FILES THAT NEED TO BE MIGRATED.

Scenario	MySQL		WordPress	
	Pre-dump (MB)	***Checkpoint (MB)***	***Pre-dump (MB)***	***Checkpoint (MB)***
0c (Azure)	129.000	2.067	88.333	19.667
0c (GCP)	127.272	3.592	71.227	17.882
1c (GCP)	131.333	20.333	89.000	45.000
5c (GCP)	134.333	35.000	108.667	59.000
10c (GCP)	135.000	36.000	176.000	87.333
50c (GCP)	167.000	54.250	347.000	433.800
100c (GCP)	167.000	56.900	320.400	849.000
200c (GCP)	168.200	73.000	385.600	954.200
400c (GCP)	168.250	63.750	388.750	1102.250

Additionally, there appears to be a short delay after the restore step until the containers become responsive. Restore times are roughly the same for all workloads at 5 seconds.

B. Image Size

Image sizes are provided in Table IV. The size of each pre-dump image is dependent on the amount of memory allocated and utilized by the container. The size of each checkpoint image indicates the amount of changes to the memory state. MySQL, as a database application, has larger image sizes when there are more transactions and updates. On the other hand, WordPress, as a mostly stateless web server, uses more memory to serve a larger number of concurrent connections, resulting in larger image sizes.

When idle (no clients), the pre-dump size of MySQL is considerably larger than that of WordPress at 127.272 MB compared to 71.227 MB. This is due to the way MySQL allocates memory to maintain its buffer pool [43]. This can be further observed as the size of the MySQL pre-dump image barely grows with increasing workload, maintaining a consistent size of 167–168 MB when there are 50–400 concurrent connections.

The checkpoint size of MySQL increases due to the amount of modifications made to the content of the website. In idle state, barely any changes are made as the checkpoint size is only 3.592 MB. The checkpoint size significantly increases

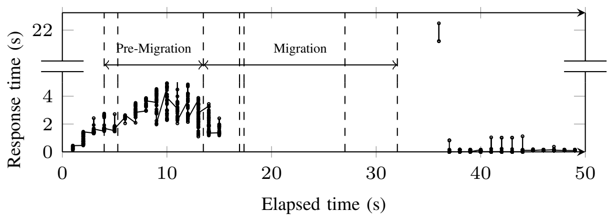

Fig. 5. Response time.

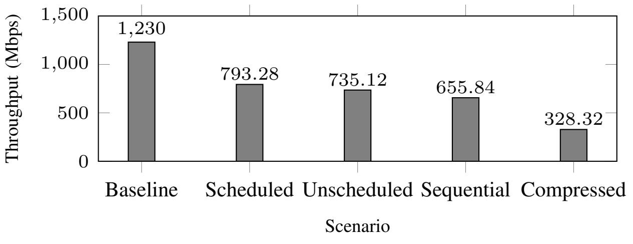

Fig. 6. Transfer throughput comparison between different optimizations.

when workload is introduced, steadily growing up to 73 MB. The size with 400 concurrent clients being smaller could possibly be an anomaly in experimentation, but it might also indicate that fewer changes were able to be made within a similar amount of time because WordPress was overwhelmed by the number of connections.

The pre-dump size of WordPress increases according to the amount of workload. Interestingly, its pre-dump size drops to 320.4 MB with 100 concurrent connections. Again, this might be an error in experimentation, or simply convenient timing when WordPress is able to handle the workload more efficiently due to other dynamic environmental factors. The pre-dump size further increases to 385.6 and 388.75 MB with 200 and 400 connections. The increase in size gets smaller as the amount of workload increases which might indicate that WordPress is reaching its maximum capacity. This is also likely the reason why migrations with 200 and 400 concurrent connections take a similar amount of time.

As WordPress needs to handle a large amount of connections, its checkpoint size also increases indicating a large amount of memory allocation and change. Starting from 17.882 MB in idle state, the size increases up to 1.1 Gigabytes with 400 concurrent connections.

Comparing the two containers, WordPress's image sizes are significantly larger, especially at higher amounts of workload. This indicates that migration time is largely dependent on the sheer size of WordPress's images when workload is introduced.

C. Response Time

Measured response times from the experiment with 100 concurrent connections are shown in Figure 5. Each step in the migration process is illustrated with a vertical dashed line.

During normal operations, the average response time is around 0.25 seconds. When workload is introduced, a significant jump in response time up to 1.44 seconds can be observed. The performance is further reduced in the pre-migration phase as pre-migration competes for available computing resources and bandwidth. After pre-migration, the average response time starts to decrease again as the containers start to stabilize.

In the migration phase, the state of each container is frozen, dumped, and transferred to the target host. No client requests are served during this phase until the containers are completely restored. The first set of responses to arrive at the target after migration completes are the ones initiated before the migration itself. They are held and see very high response times. This shows that CloudHopper successfully holds and redirect connections during and after migration without dropping any clients. After migration, the average response time recovers back to normal levels.

D. Optimizations

1) Parallel Migration and Scheduling: Migration time and throughput comparisons between each optimization strategy are provided in Table III and Figure 6. All experiments shown in Figure 6 are performed with 400 concurrent connections to show the impact of CloudHopper's optimizations under heavy load. We also performed sequential migrations of individual containers where no process is done in parallel for comparison. The baseline throughput is the bandwidth of the network connection between source and target measured with iPerf3 [44]. Note that for scheduling, we need to check the size of the changes using rsync to perform a "diff" calculation. However, for unscheduled or sequential migration, this step is not necessary, but we performed it to simplify measurements.

It is quite evident that sequential migration performs worse than parallel migrations, with an average downtime of 32.67 seconds. The restore time is double for sequential migration as we need to restore the containers one after the other in order of dependency (i.e., MySQL before WordPress).

Scheduling (400c) actually achieves slightly better performance with 12.370 seconds transfer time compared to 14.016 seconds of unscheduled transfer. Both scheduled and unscheduled migration achieve a reasonably high transfer throughput of 793.28 Mbps and 735.12 Mbps between AWS and GCP, respectively.

Note that the time spent in the pre-migration phase for both scheduled and unscheduled migrations theoretically should have the same performance. However, our experiments show that the time varies without any clear trend perhaps due to dynamic conditions in the cloud.

Using compression does not provide any performance benefit in our experiments. Compression takes roughly twice the amount of time to transfer compared to other experiments. This is due to the fact that compression adds more load on the computing resources of the host, which actually becomes a bottleneck compared to the available network bandwidth. If the available network bandwidth is limited, then compression might improve migration performance.

VII. Related Work

In this section, we discuss related work. Table V presents a concise summary of our discussion.

TABLE V
COMPARISON BETWEEN RELATED WORK.

Name	Target	Network Migration	Memory and Storage Migration	Application	Environment
CloudHopper	Multi-container	VPN, connection holding and redirection	pre-copy, scheduling	Web server/database	AWS, GCP, Azure
MIGRATE [45]	Multi-container	Container-level	pre-copy	-	Different datacenter (testbed)
Voyager [28]	Single container	-	post-copy, layered FS	Web server/database	Same datacenter
ElasticDocker [29]	Single container	by Cloud provider	pre-copy	Web server	Same datacenter
CloudNet [8]	Multi-VM	Commercial VPLS/ Layer-2 VPN	pre-copy, DRDB	SPECjbb 2005, Kernel Compile, TPC-W	Different datacenter (testbed)
COMMA [30]	Multi-App, Multi-VM	VPN	pre-copy, controlled pace, scheduling	SPECWeb 2005, RUBis 3-tier web app	AWS, Hybrid-Cloud
Supercloud [31]	Multi-VM	SDN, VXLAN	post-copy, layered storage	Zookeeper, Cassandra	AWS, GCP

A. Container Live Migration

MIGRATE [45] is a container management framework that prevents side-channel attacks by live-migrating containers from one host to another in a different datacenter within the same cloud provider. MIGRATE uses CRIU for memory migration and maintains network connectivity after migration. However, their focus is on hardening container environments more than migration performance, which is not evaluated.

Voyager [28] is a single-container migration platform using CRIU for live memory migration. A union filesystem is used to support lazy copy of files (post-copy) so that restores can happen immediately even before data transfer is completed. Voyager has zero downtime during migration.

ElasticDocker [29] is a container platform which allows automatic vertical scaling of an application. They support live migration as one of their resource allocation mechanisms. When there are insufficient computing resources on a host VM, a container will be live migrated to a new host VM. ElasticDocker uses pre-copy migration and compresses the container into a tarball.

The main ideas that distinguish our work from the previous work in container migration are *multi-container setup, realistic workloads, real-world conditions using commercial clouds, and zero downtime migration with connection holding and traffic redirection*.

B. VM Live Migration

CloudNet [8] supports live WAN migration of multiple Xen virtual machines across datacenters. They use pre-copy for memory migration and Distributed Replicated Block Device (DRBD) to replicate disk state. We share some common migration optimizations. To support network migration, CloudNet needs access to routers in the source and target datacenters to set up a layer-2 VPN. Their solution is suitable for cloud providers who want to move workloads around, but not accessible to the common cloud users like CloudHopper.

COMMA [30] is a coordination system for live WAN migration of multiple KVM virtual machines across datacenters. COMMA's goal is to minimize performance impact while migrating multiple applications and multiple VMs. Pre-copy memory migration is performed for each VM at a controlled pace [9] and a VPN is used for network migration.

Supercloud [31] is a VM live migration system that leverages a geo-replicated image file storage for disk/image migration and Xen's pre-copy live memory migration. For network migration, they use SDN and VXLAN tunnels. They focus on developing an algorithm that automatically determines when and where to move VMs for optimal performance.

The main ideas that distinguish our work from these previous work in multiple-VM live migration are *multi-container setup, real-world conditions using commercial clouds, and zero downtime migration with connection holding and traffic redirection*.

VIII. Summary

In this paper, we designed and implemented CloudHopper, a functional live migration system for containerized applications. Our work supports migration of multiple interdependent containers across commercial cloud providers with active clients during migration. We have tested the system under realistic load constraints. CloudHopper, with scheduled parallel migration optimizations, can live-migrate a heavily loaded WordPress and MySQL container from one cloud provider to another across the wide-area, with an average downtime of under 30 seconds. Note that this downtime is dominated by data transfer between clouds. While the downtime may seem long enough to be noticeable, CloudHopper hides this from clients and keeps the application online by holding and redirecting client connections.

We believe that our approach can be generalized for every Docker image and container publicly available, provided that the container is converted into the OCI format and that its external resources are supported by CRIU. Future work consists of further reducing migration time with other optimization techniques and creating a more generalized live container migration platform that is applicable for non-web applications.

References

[1] Amazon Web Services, Inc., "Amazon ECS - Run containerized applications in production." [Online]. Available: https://aws.amazon.com/ecs/

[2] Google Cloud, "Google Kubernetes Engine | Kubernetes Engine." [Online]. Available: https://cloud.google.com/kubernetes-engine/

[3] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live Migration of Virtual Machines," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 2005, pp. 273–286.

[4] O. Laadan and S. E. Hallyn, "Linux-CR: Transparent Application Checkpoint-Restart in Linux," in *Linux Symposium*, vol. 159, 2010.

[5] S. Pickartz, N. Eiling, S. Lankes, L. Razik, and A. Monti, "Migrating LinuX containers using CRIU," in *International Conference on High Performance Computing*. Springer, 2016, pp. 674–684.

[6] W. Voorsluys, J. Broberg, S. Venugopal, and R. Buyya, “Cost of Virtual Machine Live Migration in Clouds: A Performance Evaluation,” in *IEEE International Conference on Cloud Computing*. Springer, 2009, pp. 254–265.

[7] Y. Chen, “Checkpoint and Restore of Micro-service in Docker Containers,” in *2015 3rd International Conference on Mechatronics and Industrial Informatics (ICMII 2015)*. Atlantis Press, 2015.

[8] T. Wood, K. K. Ramakrishnan, P. Shenoy, and J. van der Merwe, “CloudNet: Dynamic Pooling of Cloud Resources by Live WAN Migration of Virtual Machines,” *SIGPLAN Not.*, vol. 46, no. 7, p. 121132, Mar. 2011.

[9] J. Zheng, T. S. E. Ng, K. Sripanidkulchai, and Z. Liu, “Pacer: A Progress Management System for Live Virtual Machine Migration in Cloud Computing,” *IEEE Transactions on Network and Service Management*, vol. 10, no. 4, pp. 369–382, Dec. 2013.

[10] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. De Rose, “Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments,” in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, 2013, pp. 233–240.

[11] T. Combe, A. Martin, and R. Di Pietro, “To Docker or Not to Docker: A Security Perspective,” *IEEE Cloud Computing*, vol. 3, no. 5, pp. 54–62, Sep. 2016.

[12] A. Martin, S. Raponi, T. Combe, and R. Di Pietro, “Docker ecosystem–Vulnerability Analysis,” *Computer Communications*, vol. 122, pp. 30–43, Jun. 2018.

[13] D. Petcu, “Portability and Interoperability between Clouds: Challenges and Case Study,” in *European Conference on a Service-Based Internet*. Springer, 2011, pp. 62–74.

[14] C. Puliafito, E. Mingozzi, and G. Anastasi, “Fog Computing for the Internet of Mobile Things: Issues and Challenges,” in *2017 IEEE International Conference on Smart Computing (SMARTCOMP)*. IEEE, 2017, pp. 1–6.

[15] D. S. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, “Process Migration,” *ACM Computing Surveys (CSUR)*, vol. 32, no. 3, pp. 241–299, 2000.

[16] E. Roman, “A survey of checkpoint/restart implementations,” Lawrence Berkeley National Laboratory, Tech. Rep., 2002.

[17] A. Mirkin, A. Kuznetsov, and K. Kolyshkin, “Containers checkpointing and live migration,” in *Proceedings of the Linux Symposium*, vol. 2, 2008, pp. 85–90.

[18] P. H. Hargrove and J. C. Duell, “Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters,” vol. 46, pp. 494–499, Sep. 2006.

[19] H. Zhong and J. Nieh, “CRAK: Linux Checkpoint/Restart As a Kernel Module,” Technical Report CUCS-014-01, Department of Computer Science, Columbia University, Tech. Rep., 2001.

[20] K. Kolyshkin, “Virtualization in Linux,” *White paper, OpenVZ*, vol. 3, p. 39, 2006.

[21] S. Osman, D. Subhraveti, G. Su, and J. Nieh, “The Design and Implementation of Zap: A System for Migrating Computing Environments,” *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 361–376, 2002.

[22] J. Ansel, K. Arya, and G. Cooperman, “DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop,” in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*. IEEE Computer Society, 2009, pp. 1–12.

[23] “Container Runtime with Docker Engine.” [Online]. Available: https://www.docker.com/products/container-runtime

[24] J. Fink, “Docker: a Software as a Service, Operating System-Level Virtualization framework,” *Code4Lib Journal*, vol. 25, 2014.

[25] Open Container Initiative, “runC: CLI tool for spawning and running containers according to the OCI specification.” [Online]. Available: https://github.com/opencontainers/runc

[26] “containerd: An open and reliable container runtime.” [Online]. Available: https://github.com/containerd/containerd

[27] “Fourth Quarter Growth in Cloud Services Tops off a Banner Year for Cloud Providers | Synergy Research Group.” [Online]. Available: https://www.srgresearch.com/articles/fourth-quarter-growth-cloud-services-tops-banner-year-cloud-providers

[28] S. Nadgowda, S. Suneja, N. Bila, and C. Isci, “Voyager: Complete Container State Migration,” in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017, pp. 2137–2142.

[29] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, “Autonomic Vertical Elasticity of Docker Containers with ElasticDocker,” in *10th IEEE International Conference on Cloud Computing, IEEE CLOUD 2017*, 2017, pp. 472 – 479.

[30] J. Zheng, T. S. E. Ng, K. Sripanidkulchai, and Z. Liu, “COMMA: Coordinating the Migration of Multi-Tier Applications,” *SIGPLAN Not.*, vol. 49, no. 7, pp. 153–164, Mar. 2014.

[31] Z. Shen, Q. Jia, G.-E. Sela, B. Rainero, W. Song, R. Van Renesse, and H. Weatherspoon, “Follow the Sun through the Clouds: Application Migration for Geographically Shifting Workloads,” in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, 2016, pp. 141–154.

[32] “strongSwan - IPsec VPN for Linux, Android, FreeBSD, Mac OS X, Windows.” [Online]. Available: https://strongswan.org/

[33] “HAProxy - The Reliable, High Performance TCP/HTTP Load Balancer.” [Online]. Available: https://www.haproxy.org/

[34] “Flask.” [Online]. Available: https://palletsprojects.com/p/flask/

[35] “Gunicorn - Python WSGI HTTP Server for UNIX.” [Online]. Available: https://gunicorn.org/

[36] “memcached - a distributed memory object caching system.” [Online]. Available: https://memcached.org/

[37] Red Hat, “Ansible is Simple IT Automation.” [Online]. Available: https://www.ansible.com

[38] Project Atomic, “skopeo: Work with remote images registries - retrieving information, images, signing content.” [Online]. Available: https://github.com/containers/skopeo

[39] “umoci modifies Open Container images.” [Online]. Available: https://github.com/openSUSE/umoci

[40] L. Chen, “Microservices: Architecting for Continuous Delivery and DevOps,” in *2018 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2018, pp. 39–397.

[41] dest-unreach, “socat.” [Online]. Available: http://www.dest-unreach.org/socat/

[42] Joe Dog Software, “Siege Home.” [Online]. Available: https://www.joedog.org/siege-home/

[43] “How MySQL Uses Memory,” 2019. [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/memory-use.html

[44] ESnet/Lawrence Berkeley National Laboratory, “iperf, the tcp, udp, and sctp network bandwidth measurement tool.” [Online]. Available: https://iperf.fr/

[45] M. Azab and M. Eltoweissy, “MIGRATE: Towards a Lightweight Moving-Target Defense Against Cloud Side-Channels,” in *2016 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2016, pp. 96–103.