# Voyager: Complete Container State Migration

Shripad Nadgowda, Sahil Suneja, Nilton Bila and Canturk Isci

*IBM T.J. Watson Research Center*

***Abstract*—Due to the small memory footprint and fast startup times offerred by container virtualization, made ever more popular by the Docker platform, containers are seeing rapid adoption as a foundational capability to build PaaS and SaaS clouds. For such container clouds, which are fundamentally different from VM clouds, various cloud management services need to be revisited. In this paper, we present our Voyager - just-in-time live container migration service, designed in accordance with the Open Container Initiative (OCI) principles. Voyager is a novel filesystem-agnostic and vendor-agnostic migration service that provides consistent full-system migration.Voyager combines CRIU-based memory migration together with the data federation capabilities of union mounts to minimize migration downtime. With a union view of data between the source and target hosts, Voyager containers can resume operation instantly on the target host, while performing disk state transfer lazily in the background.**

## I. Introduction

Container virtualization has existed since two decades in the form of FreeBSD Jails[18], Solaris Zones[26], IBM AIX Workload Partitional WPAR[5], and LXC for Linux, amongst others. But these have recently started gaining acceptance as a lightweight alternative to virtual machines (VMs), owing to technology maturity and popularization by platforms like Docker[14], CoreOS's rocket[12], Cloud Foundry Warden[9]. Containers are being adopted as a foundational vitualization capability in building Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS) cloud solutions, e.g. Amazon Container service[6], Google Container Engine [15] and IBM's Container Cloud[17]. For such container clouds, which are fundamentally different from VM clouds, various cloud management services need to be revisited. In this paper, we target one such service- container migration.

An efficient migration solution becomes essential as containers start running production workloads. Borrowing the scenarios from their VM counterparts, migrations are required during host maintenance, load balancing, server consolidation and movement between availability zones (e.g., a *Silver zone* with HDD storage and 100 IOPS vs. a *Gold zone* with SSD storage with 1000 IOPS).

In this paper, we present our Voyager container migration service, tailored specifically for containers in accordance with OCI principles. Voyager is a novel filesystem-agnostic and vendor-agnostic migration service that provides consistent full-system migration, unlike existing alternatives that either provide memory-only migration, or rely on specific filesystems to migrate persistent storage (Section 2). Voyager performs just-in-time live container migration with minimal downtime, by combining the data federation capabilities of union mounts together with CRIU-based memory migration. With a union view of data between the source and target hosts, Voyager containers can resume operation instantly on the target host, while performing disk state transfer either on-demand (Copy-on-Write) or through lazy replication. Our experiments show Voyager's federation framework imposes *no* overhead during data updates/writes, and $\approx 1\%$ overheads for reads and upto 10% for scans.

We have open-sourced an initial version of our data migration framework[8], although it works specifically with docker containers and does not support live migration. We intend to open-source the enhancements described in this paper as well, which include in-memory state migration, OCI compliance, and support for multiple data storage types (rootfs, local and network attached data volumes).

## II. Background and Related Work

The objective in this paper is not to compare and establish performance advantages between VM and container migration techniques. Containers are fundamentally different than VMs in terms of their system resource requirements, high density and agility. We believe a more optimal migration service can be designed for container clouds than the prevalent ones in VM clouds.

### A. VM Migration

Migration has extensively been studied primarily for VM Clouds[7][25][24][16][22]. Different vendors use different virtual disk (vDisk) formats to encapsulate a VM's persistent state (e.g. vmdk, VHD, qcow2), which are migrated via proprietary hypervisor-assisted services like vMotion[29] and Hyper-V Live Migration[27], third party tools [7], [25], [13], or via explicit vDisk conversions[28][19] across hypervisors. Containers, on the other hand, are (being) standardised by the Open Container Initiative (OCI)[3] which specifies an industry standard for container image format- a $filesystem\ bundle$ or $rootfs$, and multiple *data volumes* (Section III). These being essentially directories on the host filesystem, a generic file-based migration solution can be designed for containers. There also exist vendor-agnostic file-based migration solutions for VMs like

I2Map[22] and racemi[4], but there require extra agents to be installed inside the systems.

Some container clouds[6][15] provision VMs to host containers, primarily to mitigate security and isolation concerns for containers, while others run containers directly on cloud hosts [17]. This paper shows how to migrate the small execution state that is part of a container without resorting to installing and migrating VMs that impose different constraints on their hosts. Installing VMs for the purpose of migration defeats the advantages of using containers for application deployment.

### B. Container Migration

Containers have been popular with the microservice architecture. Although container migration may seem redundant for stateless containerized applications, but it is still pertinent to several stateful microservice applications like databases (e.g. Mysql, cassandra), message brokers (kafka), and state-coordination service (zookeeper), amongst others. This is being acknowledged and supported in standard frameworks like Kubernetes' 'StatefulSet'. Portability of stateful containers is also explored in existing solutions like ClusterHQ's Flocker[10], Virtuozzo[20] and Picocenter[30].

Flocker[10] primarily is a data management solution specifically for docker containers. It supports migration for network-attached storage backends like Amazon EBS, Openstack Cinder, VMware vSphere etc., by re-attaching these network storage for containers. Local attached volume migration is supported only for ZFS filesystem. On the other hand, Voyager is a generic, filesystem-agnostic and vendor-agnostic migration solution.

Virtuozzo is a bare-metal virtualization solution that includes container virtualization. It facilitates Zero-downtime live migration for containers[23][20]. During this migration it first transfer container's filesystem and virtual memory to target host. Once transfer is finished, it freeze all container processes and disable networking. It then dump this memory state to file and copies these dump file to target host. Any changed memory and disk blocks since the last transfer are then migrated to target host and then container is resumed. It has an underlying assumption that amount of memory pages and disk blocks changed (*deltas*) is small, thus outage time imperceptible. For any data intensive application these *deltas* specially persistent data changes could be large. Voyager differs from this technique primarily on two aspects. First, Voyager is a Just-in-Time (jit) Zero-copy migration solution, wherein container is migrated immediately before whole data is copied to the target host. Secondly, Voyager does not require the task of second data transfer performed by Virtuozzo, thus application downtime for Voyager is still smaller than Virtuozzo. Further Voyager provided features like data federation access, dual-band data replication, OCI compliance which are not provided by any existing container migration solution.

Picocenter[30] on the other hand is a system that enables swapping-out in-active containers in cloud to object store (Amazon S3) and swapping-in on-demand. It uses CRIU to capture and migrate memory-state and $btrfs$ filesystem snapshots for persistent state. It also proposes feature of *ActiveSet* for memory in which memory pages are restored for container on-access and lazily. This is again a filesystem specific solution and not optimized for container migration use-case.

## III. Design and Implementation

As per OCI specifications, every container image is stored in a filesystem bundle, which after unpacking becomes just another directory– rootfs– on the host filesystem. On container instantiation, all runtime environment changes (e.g., new package installations) and data changes (e.g., application state and logs) are persisted in rootfs by default. Since runtime state changes actively, hosting rootfs at the local filesystem also complies from a performance viewpoint. Additionally, any directory from the host filesystem can also be *bind mount* inside a container as data volumes. Such a volume can also be a network attached filesystem mounted on the host. Similarly, although rare, any block devices on the host can also be mapped inside container. As for volatile state, containers have their memory share from the host controlled via *cgroups*.

Thus, migrating a container involves discovering all data end-points of a container, and moving their states, in addition to its memory state, consistently from the source to the target host. Migration of in-memory state can be achieved in userspace via CRIU. Also, any network attached storage can be migrated by *un-mounting* it from source host and *mounting* it on target host. In addition, Voyager provides userspace-level filesystem-agnostic migration of locally persistent container state, while ensuring consistency across all these states, as well as minimum application downtime. Fig.1 shows the complete orchestration framework of Voyager. Primarily, we are migrating container state across *three* different data stores, namely **in-memory, local filesystem, and network filesystem**. We discuss each of these migration capabilities below.

### A. In-memory state migration

Running containers hold a lot of in-memory state like open file descriptors, network sockets, application data structures, and cached data. We use open-sourced linux tool Checkpoint/Restore in Userspace (CRIU)[2] to checkpoint and dump these states in files which we can be migrated and restored. During checkpoint, CRIU freezes a running container for consistency, and then dumps it's memory state into a collection of files. Total size of the dump is dominated by the size of the process' pagemap. For example, a MySQL container checkpointed right after it is initialized creates a pagemap dump of $117MB$ and all other resources dump files accounts for $250KB$.

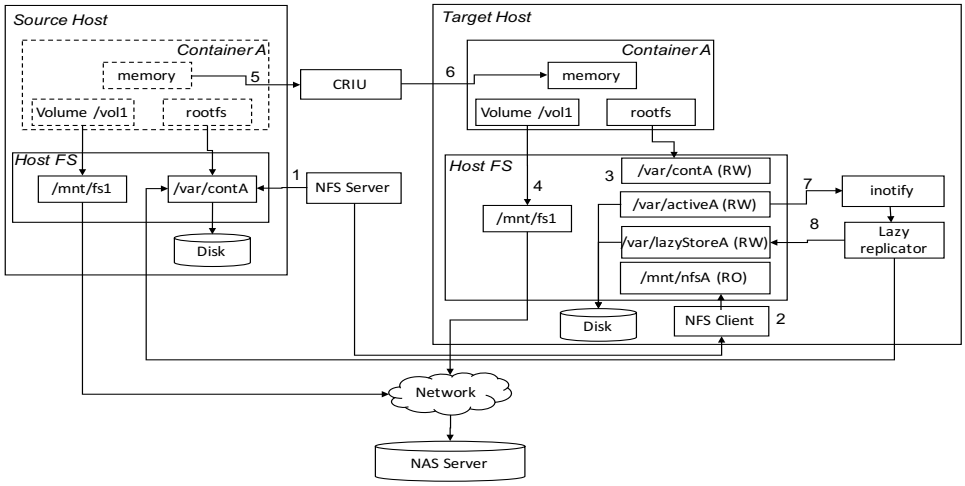



Fig. 1. *Voyager: Orchestration for migrating ContainerA from SourceHost to TargetHost. Numbers indicates order of migration workflow.*

Time to checkpoint memory essentially counts towards application downtime, and thus it is critical to optimize it. In Voyager we use $tmpfs$ to store the dump files to avoid any slower disk transactions. We further use CRIU's page-server model, wherein during checkpointing the pagemap image is directly dumped at the target host, thereby avoiding the source host storage hop. CRIU also provides a construct called *action script*, which allows executing any script after a container is checkpointed and before it is un-frozen. We provide a callback method in *action script* to notify Voyager about the checkpoint status, so that Voyager can perform certain sanity checks, followed by stopping the container at the source and restoring it at the target. Once the container is instantiated, Voyager starts a *lazy migration* process at target, as described next.

### *B. Local filesystem migration*

One of the most significant contributing factor in application downtime during migration is data copy. There exist optimization techniques like transferring incremental filesystem snapshots, and performing the actual fail-over when the incremental change is small enough. For data intensive applications finding such optimal change window could be difficult. Besides, such techniques are more suitable for planned VM migrations wherein the actual migration can happen over time. For containers which in principle are very agile and whose lifetimes are relatively shorter than VMs, Voyager currently targets enabling *just-in-time (jit)* migration. And it is enabled by employing dual-band data transfer between the source and the target- *in-band* transfer through **data federation**, and *out-of-band* transfer via **lazy replication** as described next.

#### Data federation

The goal here is to make the container data at the source accessible on the target host, without actually copying any data first. Such federation logic enables remote-reads and local-writes for the migrated container. Voyager orchestrates data federation using a union mount of filesystems across the source and target hosts. This federation is zero-downtime because it is orchestrated before the memory checkpoint, while application is running at the source host. Once it's memory state is restored at the target, the container has access to its persistent data via the union mounted $rootfs$.

As show in Figure 1, first, the $rootfs$ of a container (say $/var/contA$) is NFS-exported from the source (step 1), and mounted on the target host (step 2) as read-only $/mnt/nfsA$. Next, two new directories are created at the target- $/var/lazyStoreA$ to host the lazy replicated data (described below), and $/var/active$ to host any updated or newly written data. These three directories are accessed in the order as shown in step 3, using the AuFS[1] union mount filesystem at the target's rootfs at $/var/contA$. The federation capability of Voyager is equally applicable with any other union filesystem as well.

Initially, the lowest NFS directory branch and lazyStore branch both are marked read-only, and only the active branch has read-write permissions. Every *new* file created by the migrated container is hosted in the active directory. Any *update* to an existing file first transfers it over through the NFS branch to the active branch, and then updates it locally. Thus, essentially the active branch acts as a copy-on-write (CoW) directory. Any further reads or updates to this file are handled locally. The only file reads that are not satisfied by the active branch or the lazyStore branch (discussed below) are routed over NFS to the source host filesystem. Voyager uses NFS client-side caching at the target host to optimize these reads over the network.

#### Lazy replication

Once the container is resumed with access to its data through federation, Voyager launches a *lazy replicator* to transfer the data in background. The replicator first traverses the $rootfs$ of the container at the source to compute a $joblist$ of all files, and then starts copying them over to the lazyStore. We employ *inotity* (step 7) on the active directory which monitors and notifies the replicator

(step 8) about the files that have already been transferred to the target through CoW, so that the replicator can remove them from the *joblist* to avoid any redundant network copy. The replicator first creates the file in the lazyStore with a temporary name $.filename.part$ to avoid surfacing a mid-transfer incomplete view to the container. Once the copy is finished, the file is is renamed to its actual name. Any subsequent reads to this file by the container is served by the lazyStore, while avoiding the NFS branch. When the replicator finishes copying all files, the lowest NFS branch is removed from federation and the lazyStore branch is marked read-write. At this point Voyager marks the container migration as complete.

**Tuning**

Having a dual-band transfer channel allows a sysadmin to prioritize one flow over the other. Out-of band lazy replication rate can be throttled to avoid any network bottleneck for primary federation flow. Also, if the hosts have multiple network ports, the faster port can be used in federation to mount NFS, while the slower employed for lazy replication.

### C. Network filesystem migration

For any network-attached file storage, Voyager simply performs *un-mounting* and *mounting* of the NAS share, along with any host access authorization and firewall configurations. Similarly, distributed filesystem deployments, by virtue of their design, lend themselves well to migration.

### D. Future enhancements

Various on-going efforts to further minimize application downtime, and incorporate new features are listed below:

(i) In Voyager application downtime is incurred only during memory checkpoint/restore, and data replication is zero-downtime. Thus, we are exploring ways to reduce size of pagemap by perfoming page-flush (sync) or any application specific garbage collection inside container before it is checkpointed. Also, selectively discarding checkpoint of certain redundant resources could be helpful.

(ii) Currently Voyager implements Just-in-Time migration. We have very recently also incorporated CRIU's incremental memory checkpointing capabilities in Voyager, which should lower the application downtime significantly. We plan to commit these changes back to *runc* OCI compliant runtime. At the same time we are also exploring incremental disk checkpointing for planned migration.

(iii) For out-of-band lazy replication we are planning to provide a filter wherein user can specify regular expressions for file patterns. For example $*.log$, $tmp*$. These files will be omitted during replication to avoid redundant network transfer.

## IV. Evaluation

*Experiment Setup:* We conducted our experiments on two Ubuntu 14.04.5 LTS VMs acting as a source and target hosts for containers. Each VM was configured with 4 vCPUs, 4 GB memory and 25GB disk with ext4 filesystem. These VMs were in the same datacenter, and average network bandwidth between them was observed to be 2.5 Gbps. We ran our containers using $docker - runc$ (via docker-1.12.3) which is docker's OCI compliant container runtime. For memory migration we used the latest CRIU release in 2.8. We selected the most popular stateful database from docker hub as our test application- MySQL 5.7.15. Finally, we used the standard Yahoo! Cloud Serving Benchmark(YCSB)[11] benchmarking tool to measure MySQL performance under different workload patterns.

We evaluated Voyager on *three* dimensions: **consistency, application downtime and performance overhead**.

### A. Consistency

Voyager is a complete state migration solution for an application container. Thus, once the container is migrated, it is critical to ensure that the application's runtime state is correctly restored. To measure consistency we devised two data-points:

(i) MySQL server performs a set of initialization tasks when it starts, which includes loading configuration, initializing data directory, system tablespace and related in-memory data structures needed to manage InnoDB tables, amongst others. Thus, as a first datapoint, we inserted some records into a MySQL database, and migrated that container using Voyager. We then verified that the a MySQL client can connect to the migrated server and query existing records without errors. This validated that the application was restored consistently with its in-memory data structures and persistent tablespace.

(ii) For our second datapoint evaluation, we embedded a test client inside the same MySQL container. We first inserted 10K person records into a database with incremental index starting from 1. The client program was initialized with person index 1 and an empty file on the disk. It has a periodic function to connect to the database, query a person record (name, age) with next index, append name to file and revise average age. We started this client program, migrated the container in the middle of the program's progress, and ensured that the client program finishes at the target with accurate summary every time, by comparing it with the expected output obtained in a base no-migration setup.

### B. Application downtime

Minimum application downtime is an important criteria in production clouds, for example to maintain SLAs and Business Continuity. Voyager imposes no application downtime during persistent state transfer by virtue of its data federation and lazy replication between the source and target hosts. The downtime is thus limited to the time required for in-memory state transfer via CRIU. We

measure this with time to checkpoint and restore the in-memory process dump of a container.

**Checkpoint**: During checkpoint, CRIU freezes the container to ensure consistency and dumps the process' in-memory state. In Voyager, we use a remote page-server to dump the process' pagemap directly on the target host's $tmpfs$. Thus, checkpoint time includes the time taken to collect the process tree, freeze it, collect the process' resources (e.g., file descriptors, memory mappings, timers and threads), and write the resources in dump files over the network to a remote page-server. As shown in Figure 2(a), we evaluated the checkpoint time for the MySQL container at stages with different number of records, resulting in an increasing pagemap dump file size from 117MB-250MB. Checkpoint time increases linearly with the size of application's memory state. In a micro-service architecture, we can expect the memory usage of individual containers to be less than 1GB, limiting the checkpoint time to <2 seconds.

**Restore**: During restore, CRIU reads dump files, resolves shared resources, forks the process tree and restores the process' resources. Figure 2(a) shows the restore time for the MySQL container. As we can see, the restore time is $\approx 0.7 - 0.8$ seconds for a <250 MB dump size.

Thus, in Voyager migration, the total expected application downtime is between 2-3 seconds. We have very recently also incorporated CRIU's incremental memory checkpointing capabilities in Voyager, which should lower the application downtime significantly. As future work, we plan to evaluate this optimization, and also instrument CRIU to measure both the checkpoint and restore time at a per resource granularity, so that we can prioritize and optimize for individual resources.

### C. Performance overhead

In Voyager, once a container is resumed at the target, it has immediate access to its persistent data storage through Voyager's data federation layer. This layer incurs performance overhead, which we measure using YCSB for different types of workload profiles including reads, inserts, updates, and scans. For each profile, in YCSB's *load* stage we inserted 1M records to a database table, and in the *run* stage we performed 1M record operations of respective types. The records were accessed using zipfian distribution for popularity-based long tail access pattern. For each workload profile, we measured average application throughput ($operations/sec$) every 10 seconds. Each experiment was performed at two application states- (i) *baseline*: application state at the source host before it is migrated, and (ii) $federation$: application state after it is migrated at the target host, having access to data through the federation layer. For common application read/write workload patterns we observe *zero* to 3% overhead in steady state. Performance impact on the individual workload profiles is discussed below:

**Read:** Fig. 2(c) shows the benchmark results for data reads. Initial low throughout is attributed to cache warming phase, and then in steady state phase we observe relatively stable performance. Every read operation through federation layer makes data access over NFS at the source. As a result, in the $federation$ state, read throughput drops by $\approx$ 20% during cache warming, and $\approx$ 1% in steady state.

**Scan:** Fig. 2(d) shows results for scans. Unlike reads, where a zipfian pattern accesses popular records frequently, this workload accesses records in order, starting at a randomly chosen record key, and generates more unique read requests. Thus, even in stead state we record a performance overhead of $\approx$ 10% for read accesses over NFS.

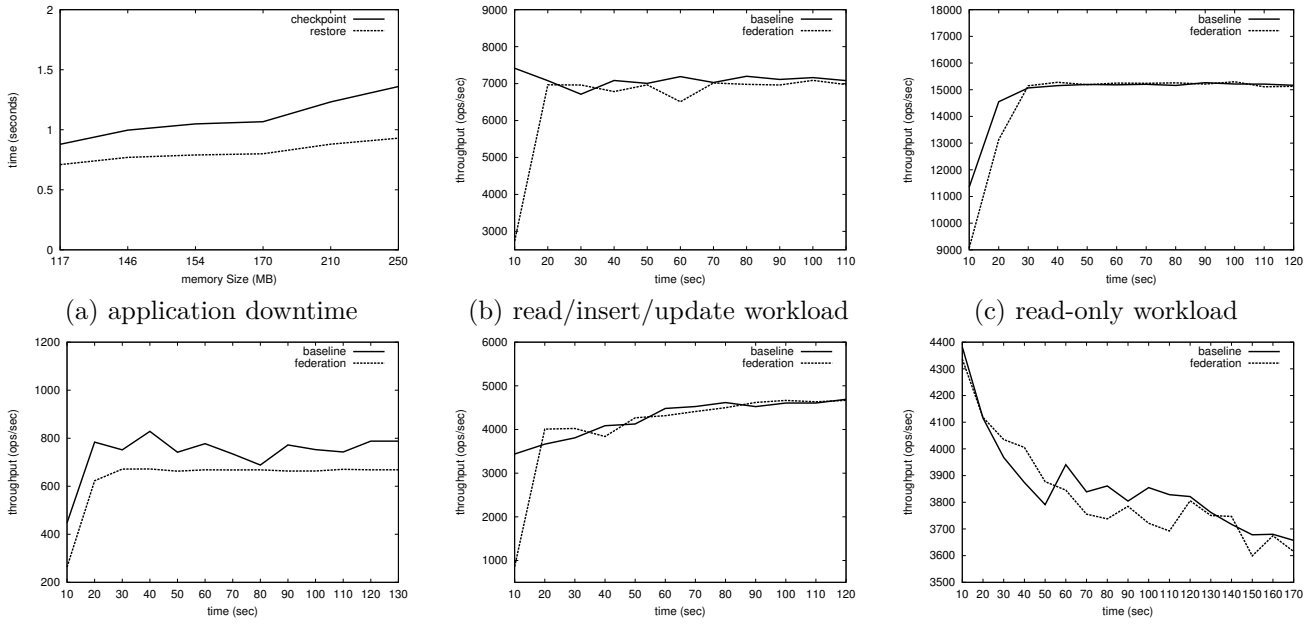
**Updates:** In $federation$ state, an update is essentially a CoW operation i.e. a file is read from source over NFS, copied at target and then updated locally. MySQL stores its InnoDB tables and indexes in separate .ibd data files. Thus, during the $federation$ state when a record is updated, the respective index and tablespace file is CoW'ed at the target host. Then, every subsequent updates to the records are done locally. As a result in Fig. 2(e) we observe almost $\approx$ 75% performance overhead at the start, and in steady state update performance is at par with baseline.

**Inserts:** In $federation$ state, every write operation resulting in creation of new files is performed locally. Thus we observe similar performance for *baseline* and $federation$ state in Fig. 2(f). The size of the table slows down the insertion of indexes by log N, assuming B-tree indexes[21], thus a steady performance drop is observed for both states.

**Read/Update/Insert**: In this profile, we split the IO workload into 60:20:20 for read:update:insert. As seen in Fig. 2(b), we observe $\approx$ 65% performance overhead at the start attributed to file-copy during updates and NFS access for reads, and $\approx$ 3% overhead in steady state.

### D. Application container: startup vs. restore

When contrasted against VMs containers have fast startup times. But, applications inside container also typically have their own startup or initialization time, which ranges from *milliseconds* to few *seconds*. We measured average initialization time to for two application namely MySQL and elasticsearch. The average initialization time for these application was recorded as: $10secs$ for MySQL and $7sec$ for elasticsearch. Then we checkpointed these application containers right after they were initialized and restored them on the same host. Restore time for both of them was less than $500ms$, much faster than their startup time. We are exploring use of Voyager on such platforms where applications are instantiated by restoring their state and $rootfs$ is provisioned through data federation and lazy copy.




(a) application downtime (b) read/insert/update workload (c) read-only workload

(d) scan-only workload (e) update-only workload (f) insert-only workload

Fig. 2. MySQL Application benchmarking with Voyager migration

## V. Conclusion

We presented the design and implementation of Voyager in this work. Voyager employs a data federation across source and target hosts to ensure that application is resumed just-in-time at target host with remote-reads and local-write data access. Performance overhead of such federation framework is evaluated to be within $1-3\%$ for common read/write workloads. We are committed to open-source and further enhanced this work to support incremental migration for containers, optimize memory checkpointing/restore and design policy-driven lazy replication to reduce network overhead.

## References


[1] AUFS: Another Union Filesystem. aufs.sourceforge.net/.

[2] CRIU. www.criu.org/.

[3] Open Container Initiative. https://www.opencontainers.org/.

[4] Racemi. www.racemi.com/.

[5] Chris Almond, Bruno Blanchard, Pedro Coelho, Mary Hazuka, Jerry Petru, Theeraphong Thitayanun, et al. *Introduction to Workload Partition Management in IBM AIX Version 6.1*. IBM Redbooks, 2007.

[6] Amazon. EC2 container service. https://aws.amazon.com/ecs/.

[7] Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, and Harald Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *VEE'07*, pages 169–179.

[8] Cargo. Cargo. https://developer.ibm.com/open/cargo/.

[9] Cloud Foundry. Warden. https://docs.cloudfoundry.org/concepts/architecture/warden.html.

[10] ClusterHQ. Flocker. https://docs.clusterhq.com/en/1.0.3/.

[11] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[12] CoreOS. Rocket. https://coreos.com/blog/rocket/.

[13] Umesh Deshpande, Danny Chan, Steven Chan, Kartik Gopalan, and Nilton Bila. Scatter-gather live migration of virtual machines. *IEEE Transactions on Cloud Computing*, 2015.

[14] Docker. Docker. https://www.docker.com/.

[15] Google Inc. Container engine. https://cloud.google.com/container-engine/.

[16] Michael R Hines and Kartik Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *VEE'09*, pages 51–60. ACM, 2009.

[17] IBM Inc. IBM Container Cloud. https://www.ibm.com/cloud-computing/bluemix/containersâĂŐ.

[18] Poul-Henning Kamp and Robert NM Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, volume 43, page 116, 2000.

[19] Microsoft Inc. System Center Virtual Machine Manager. http://www.microsoft.com/systemcenter.

[20] Andrey Mirkin, Alexey Kuznetsov, and Kir Kolyshkin. Containers checkpointing and live migration. In *Proceedings of the Linux Symposium*, pages 85–92, 2008.

[21] MySQL. Reference manual: Speed of insert statements. http://dev.mysql.com/doc/refman/5.7/en/insert-speed.html.

[22] Shripad Nadgowda, Praveen Jayachandran, and Akshat Verma. 12map: Cloud disaster recovery based on image-instance mapping. In *Middleware 2013*, pages 204–225. Springer, 2013.

[23] Odin. Zero-downtime migration. https://docs.openvz.org/virtuozzo_7_users_guide.webhelp/_zero_downtime_migration.htmlâĂŐ.

[24] Ajay Surie, H Andrés Lagar-Cavilla, Eyal de Lara, and Mahadev Satyanarayanan. Low-bandwidth vm migration via opportunistic replay. In *Proceedings of the 9th workshop on Mobile computing systems and applications*, pages 74–79. ACM, 2008.

[25] Franco Travostino, Paul Daspit, Leon Gommans, Chetan Jog, Cees De Laat, Joe Mambretti, Inder Monga, Bas Van Oudenaarde, Satish Raghunath, and Phil Yonghui Wang. Seamless live migration of virtual machines over the man/wan. *Future Generation Computer Systems*, 22(8):901–907, 2006.

[26] Andrew Tucker and David Comay. Solaris zones: Operating system support for server consolidation. In *Virtual Machine Research and Technology Symposium*, 2004.

[27] Anthony Velte and Toby Velte. *Microsoft virtualization with Hyper-V*. McGraw-Hill, Inc., 2009.

[28] VMware Inc. VConvert. https://mesosphere.github.io/marathon/.

[29] VMWare Inc. VMotion. www.vmware.com/in/products/vsphere/features/vmotion.

[30] Liang Zhang, James Litton, Frank Cangialosi, Theophilus Benson, Dave Levin, and Alan Mislove. Picocenter: Supporting long-lived, mostly-idle applications in cloud environments. In *EuroSys'16*, page 37. ACM, 2016.