Highlights

Practicable Live Container Migrations in High Performance Computing Clouds: Diskless, Iterative, and Connection-persistent

Jordi Guitart

- We carry out fully-featured seamless live migrations of runC containers.
- We minimize the downtime and disk utilization through diskless, iterative migrations.
- We migrate containers with established TCP connections transparently to the clients.
- We characterize the facets of live migrations in representative HPC benchmarks.

Practicable Live Container Migrations in High Performance Computing Clouds: Diskless, Iterative, and Connection-persistent

Jordi Guitart[a,b]

[a] *Universitat Politècnica de Catalunya, Jordi Girona 1-3, Barcelona, 08034, Spain*
[b] *Barcelona Supercomputing Center, Pl. Eusebi Güell, 1-3, Barcelona, 08034, Spain*

Abstract

Checkpoint/Restore techniques had been thoroughly used by the High Performance Computing (HPC) community in the context of failure recovery. Given the current trend in HPC to use containerization to obtain fast, customized, portable, flexible, and reproducible deployments of their workloads, as well as efficient and reliable sharing and management of HPC Cloud infrastructures, there is a need to integrate Checkpoint/Restore with containerization in such a way that the freeze time of the application is minimal and live migrations are practicable. Whereas current Checkpoint/Restore tools (such as CRIU) support several options to accomplish this, most of them are rarely exploited in HPC Clouds and, consequently, their potential impact on the performance is barely known. Therefore, this paper explores the use of CRIU's advanced features to implement diskless, iterative (pre-copy and post-copy) migrations of containers with external network namespaces and established TCP connections, so that memory-intensive and connection-persistent HPC applications can live-migrate. Our extensive experiments to characterize the performance impact of those features demonstrate that properly-configured live migrations incur low application downtime and memory/disk usage and are indeed feasible in containerized HPC Clouds.

Keywords: checkpoint/restore, live migration, diskless migration, iterative migration, networking migration, CRIU, runC, containerization, HPC Cloud

1. Introduction

Containerization technology offers an appealing alternative for encapsulating and operating applications (and all their dependencies) without being constrained by the performance penalties of using Virtual Machines and, as a result, has got the interest of the High Performance Computing (HPC) community to obtain fast, customized, portable, flexible, and reproducible deployments of their workloads [2].

Containerization is a key technology for Cloud Computing, which has been also attracting HPC users by promising the access to unlimited resources on a pay-per-use basis and the option to scale-up and down the resources allocated to an application on-demand. This convergence between HPC and Cloud Computing has resulted in a new computation model, so-called HPC Cloud [13], which considers three flavors [15], namely, (i) executing HPC applications in Cloud platforms, (ii) complementing HPC infrastructures with Cloud resources to handle peak demands, and (iii) exposing HPC resources using on-demand Cloud abstractions.

Containers support efficient and reliable resource sharing and management in HPC Clouds. In addition to their workload isolation capabilities, containers allow balancing the load of the various physical hosts depending on changing resource requirements of the workloads, consolidating workloads that run on under-utilized hosts and suspend idle hosts to lower energy consumption, resuming workloads execution upon a host failure, or proactively moving workloads from hosts anticipating hardware problems to healthy spare hosts.

Many of these features rely on application checkpointing or live migration techniques. By dumping the state of an application and restoring it at a later time (possibly in another physical host), it can resume from the exact point it was dumped at. This procedure constitutes a live migration when it can be done without interrupting the service, that is, transparently to the application and their users (if any). Such live migration would be also seamless if the time before the restoration is small enough, so that the user will not perceive any downtime (or minimal). Acceptable downtimes for seamless live migrations have been established on few seconds (between 1 and 2) by major vendors featuring live migrations, such as Google[1] and

[1] https://cloud.google.com/compute/docs/instances/live-migration-process

VMware[2].

Originally developed for the HPC domain, checkpointing was used to save the intermediate state of long-running jobs. In the event of an unexpected failure, the job could be restarted from the last stored checkpoint rather than from the beginning, which would lead to lose several hours or days worth of work [7].

Checkpoint/Restore in Userspace[3] (CRIU) is an open-source software tool to dump and restore running applications transparently to the user. It does so entirely from userspace, by strongly leveraging interfaces exposed by the Linux kernel. CRIU has particularly targeted its integration with container engines and runtimes. In fact, nowadays most of those offering checkpoint/restore functionalities such as runC[4], crun[5], Docker[6], Podman[7], OpenVZ[8], or LXC[9], rely on CRIU at a lower level.

CRIU supports multiple options to optimize the checkpoint/restore procedure, particularly to reduce the freeze time of the application and enable (really) live migrations. However, many of those options are rarely exploited in HPC Clouds, either because the container engine does not offer them (for instance, Docker's support for CRIU is quite minimal), their correct use and proper configuration is not well understood, or their potential impact on the performance has not been well analyzed yet.

Consequently, the main goal of this work is to carry out practicable seamless live migrations of running runC containers in the context of memory-intensive and connection-persistent HPC applications. runC is our container runtime of choice because of its advanced support for container live migration, better than other container engines and runtimes, such as Docker, which only offers diskful (i.e., without page-server) live migrations, and Podman/crun, which only support diskful iterative pre-copy live migrations.

We exploit runC/CRIU checkpoint and restore advanced features to implement diskless, iterative (pre-copy and post-copy) migrations of containers

[2]https://www.vmware.com/products/vsphere/vmotion.html
[3]https://criu.org/
[4]https://github.com/opencontainers/runc
[5]https://github.com/containers/crun
[6]https://docs.docker.com/engine/
[7]https://podman.io/
[8]https://openvz.org/
[9]https://linuxcontainers.org/

with external network namespaces (which include MACVLAN devices) and established TCP connections, and we perform extensive experiments to characterize the impact of those features on the application downtime (for how long a migrating application is not running), the overall application performance, and the migration overhead in terms of allocated (and duplicated) memory and disk usage.

Our main contributions can be summarized as follows:

- We carry out fully-featured seamless live migrations of runC containers running HPC applications.
- We implement diskless, iterative migrations to minimize the downtime and resource utilization when migrating memory-intensive containers.
- We support live migrations of containers running in external network namespaces with established TCP connections transparently to their clients.
- We characterize the impact of the several facets of live migrations in representative benchmarks of the HPC domain.

2. Background

2.1. Container Terminology

Containers are isolated executable units of software in which application code is packaged along with its libraries and dependencies. They are running instances of a container image, which is a (set of) files that are used locally as a mount point. To enhance portability and vendor interoperability, images are stored using a standardized format by the Open Container Initiative (OCI)[10], an open governance structure for container-related standards. A container engine turns the image into a running container and acts as the interaction point with the user.

Current engines do not usually instantiate the containers themselves, but rather rely on a container runtime. The runtime is the lower-level component that interacts with the kernel. Its specification [14] is also maintained and developed by the OCI. runC is its reference implementation, and our tool of choice to implement live migration. We have chosen to skip the engine

[10]https://opencontainers.org/

layer and interact directly with the runtime as support for advanced CRIU features is lacking in higher-level tools.

2.2. runC: the Reference Runtime Implementation

Originally developed at Docker, runC is a lightweight container runtime aimed to provide low-level interaction with containers. In 2015 [10], Docker open-sourced the component and transferred the ownership to the OCI, who has since then led the project in a fashion similar to that of the Linux Foundation. Several container engines such as containerd[11] and CRI-O[12] have made runC their runtime of choice. Podman also supports runC, although it prefers to use crun if present. The OCI releases specifications for container runtimes, engines, images, and image distribution. runC is an OCI-compliant container runtime (it is, in fact, the reference implementation). Users are encouraged to interact with containers through container engines, but runC itself provides an interface to create, run, and manage containers natively. Integration with CRIU has to be done on a per-project basis, and runC has the most advanced and stable integration. Therefore, we decided to use it to manage our containers. Running a container with runC is slightly different than doing it in, let's say, Docker, as the user's interaction with the underlying system is more direct. In particular, in runC there is no notion of images. To run a container, a user must provide a specification file (*config.json*) and a root filesystem in a directory (*rootfs*). Several low-level options such as namespaces, control groups, and capabilities can be configured through the specification file. The tandem *config.json* and *rootfs* is referred to as OCI bundle.

2.3. CNI: Container Networking

Configuring the container's network is challenging due to the large number of networking technologies available. To address this challenge, the Cloud Native Computing Foundation[13] proposed a driver-based network model, where the container engine can offload the networking configuration to specific drivers. They released the Container Network Interface (CNI)[14], which

[11] https://containerd.io/
[12] https://cri-o.io/
[13] https://cncf.io
[14] https://www.cni.dev/

consists of a specification and libraries for writing plugins to configure network interfaces in Linux containers. The specification [5] defines a JSON schema that defines the inputs and outputs expected of a CNI plugin and provides a clear separation of concerns for the container engine and the CNI plugin. They also released CNI plugins[15] for a variety of basic networks such as *bridge*, *ipvlan*, or *macvlan*.

Many container engines such as Podman and CRI-O use CNI to configure their container's networking. A network namespace is created and the CNI plugin populates it with the network devices and interfaces as defined in the JSON file. After that, the underlying container runtime (e.g., runC) is instructed to use that existing network namespace for the container.

As we are using runC directly to manage our containers, we must perform part of this procedure by ourselves. We decided to create a MACVLAN network so that our experimental setup can be easily ported to a cluster of nodes interconnected through a LAN network. First, we define a *macvlan* CNI profile, which can created by means of a JSON configuration file (*my-macvlan-net.conflist*) as follows:

```
{
  "cniVersion": "1.0.0",
  "name": "my-macvlan-net",
  "plugins": [ {
  "type": "macvlan",
  "master": "enp0s8",
  "mode": "bridge",
  "ipam": {
    "type": "host-local",
    "ranges": [ [ {
      "subnet": "192.168.1.0/24",
      "rangeStart": "192.168.1.17",
      "rangeEnd": "192.168.1.31",
      "gateway": "192.168.1.1" } ] ],
    "routes": [ {
      "dst": "0.0.0.0/0",
      "gw": "192.168.1.1" } ]
  }
```

[15] https://github.com/containernetworking/plugins

```
  } ]
}
```

Then, we create a network namespace (`sudo ip netns add <NETNS_NAME>`) and populate it with the MACVLAN device by using the *cnitool* as follows:

```
sudo CNI_PATH=/opt/cni/bin/ cnitool \
     add my-macvlan-net /var/run/netns/<NETNS_NAME>
```

Finally, we instruct runC to use that network namespace to run the container through the *config.json* specification file as follows:

```
"namespaces": [
    {
        "type": "network",
        "path": "/var/run/netns/<NETNS_NAME>"
    }
]
```

2.4. Checkpoint/Restore (C/R)

As defined by Schulz [21], checkpointing refers to the ability of storing the state of a computation such that it can be continued later at that same state without covering the preceding ones. The saved state is called a checkpoint and the resumed process a restart or restore. C/R tools snapshot an application's state regardless of the software running and without requiring, in general, any additional work from the application's developer. During the checkpointing, and in order to save the process' state, all the essential information such as the program's memory, file descriptors, sockets, pipes, etc. are dumped. With distributed computations, additional logic is required to coordinate the checkpointing across all processes [17].

Even though C/R tools originated in the HPC environment to provide fault tolerance and fast rollback times, they are also useful for debugging, skipping long initialization times and, as in this work, live process migration. In that respect, although Checkpoint/Restore had already been thoroughly studied in the context of failure recovery strategies [7], it became popular in the context of the migration of Virtual Machines (VM) [4]. Checkpointing of VMs is easier when compared to arbitrary process checkpointing, as processes running within a VM are already isolated.

2.5. Live Migration

As mentioned before, a prominent application of Checkpoint/Restore is live migration, which allows moving a running process(es) or container from a physical host to another transparently to the end user. A live migration is seamless when the downtime is minimal. It is clear that (seamless) live migration is a desirable feature for resource providers as it drastically increases their load-balancing capabilities with minimal impact to the perceived quality of service.

A naive migration approach (a.k.a. cold migration) would freeze the container to ensure that it no longer modifies the state, dump the whole state, transfer it through the network while the container is stopped, and resume the container at the destination host when all the state is available. Unfortunately, this approach incurs in very high downtimes. There are other approaches to minimize this downtime, aiming for really live migrations. For this purpose, we highlight two forms of iterative migration, namely pre-copy [4] and post-copy migration [9].

Pre-copy migration. It copies all the state from source to destination while the container is still running on the source, and data which has changed during this time ('dirty') is transferred in a successive round. This procedure is iteratively repeated until the information to transfer is minimal (smaller than a threshold) or after a given number of iterations. Only at this point the container is stopped (minimizing the downtime), the remaining dirty bits are sent over the wire, and the container is resumed in the other host.

With pre-copy migration, there is no need to fetch pages from the source host after the restoration of the container, and an up-to-date state is retained at the source host during migration, thereby if the destination host fails in the meantime, the container can be still recovered. However, it is possible that the migration never ends if source host dirties pages faster than they can be transferred. Moreover, with pre-copy migration the same page can be transferred multiple times if the page is dirtied repeatedly at the source host during migration.

Post-copy migration. Initially, it suspends the container and transfers the minimal execution state for the container to be able to resume on the destination host. Concurrently, the source actively pushes the remaining memory pages of the VM to the target, and any access in this host to a page that

has not yet been transferred (a.k.a. remote page fault) is resolved over the network by fetching the page from the source.

With post-copy migration, the live migration is guaranteed to end in a finite time, as each page is sent exactly once over the network. However, as the container state is distributed over both source and destination hosts, it cannot be recovered if the destination host fails during migration. Moreover, post-copy migration can incur some performance degradation in the application because of the latency of fetching pages from the source host.

3. Related Work

3.1. Checkpoint/Restore of HPC Applications

Apart from CRIU, another popular open-source tool for checkpoint/restore with a special focus on HPC is Distributed Multi-Threaded Checkpointing (DMTCP) [1]. Unlike CRIU, any application to be checkpointed with DMTCP must be prepared, that is, dynamically linked with the DMTCP library and executed with wrappers on a few system calls so that they can be intercepted. Moreover, as DMTCP does not support namespaces, it cannot checkpoint/restore containers. On the plus side, its main advantage over CRIU is its built-in support for checkpointing distributed computations, particularly, MPI [8].

Focusing on CRIU-based distributed checkpointing with containers for HPC, Berg and Brattlof [3] have assessed the successful completion of checkpoint/restore of a NPB benchmark running on Docker depending on the sequence order of the CRIU checkpoint and restore operations (e.g., checkpoint the MPI process launcher first/last and restore it first/last). Not all the sequences were successful, confirming that additional logic is required to coordinate the checkpoint across all the processes.

Sindi and Williams [23] also deal with the migration of (OpenVZ) containers running MPI workloads using CRIU to provide resilience. Their checkpointing is actually not distributed, as they only migrate one of the containers part of the MPI job. This container is suspended during the migration, which causes the entire MPI job to freeze temporarily.

None of the works presented in this subsection has considered advanced C/R features such as diskless and iterative migrations, memory deduplication, or migration of established TCP connections.

3.2. Container Migration with non-HPC applications

Adrian Reber, one of the contributors to the CRIU project, particularly in the integration with runC containers, has some blog posts about live migrations with runC and CRIU. In [18], Reber shows how to migrate a running runC container (executing an httpd webserver) from one system to another. The author employs a basic setup, using the hosts' network interfaces directly (no network namespaces) and a shared NFS filesystem between the hosts. The work does not consider either advanced features such as iterative migrations, a page-server, memory deduplication, or migration of established TCP connections.

In [19], Reber migrates the server part of the Xonotic game around several nodes in the world and compares some basic versions of migration optimizations to reduce the downtime, namely pre-copy migration (with a single pre-dump) and post-copy migration (using a page-server). Other features such as network namespaces, memory deduplication, or migration of established TCP connections are not explored (although the author uses keepalived to move the IP address from one system to another, the disconnection during the migration is perceived by the client).

In his master thesis, Terneborg [25] has implemented a variant of CRIU's iterative pre-copy migration to be used for runC containers failover (automatically move a container to another host upon failure), which is evaluated using the Redis database. This work does not consider advanced features such as a page-server, memory deduplication, or networking migration (including network namespaces and established TCP connections).

Puliafito et al. [16] have carried out a comprehensive performance evaluation of runC container migration techniques based on CRIU over a real fog computing testbed running a client-server Java application. They compare cold, pre-copy (with a single pre-dump), post-copy (with a page-server), and hybrid (pre-copy+post-copy) migrations. Their study does not include other advanced features such as network namespaces, memory deduplication, or migration of established TCP connections.

Nadgowda et al. [12] have presented Voyager, an OCI-compliant live container migration service that combines CRIU-based memory migration together with the data federation capabilities of union mounts to minimize migration downtime. As in our work, Voyager takes advantage of diskless migration, i.e., *tmpfs* to store the dump files and CRIU's page-server, which is used to implement post-copy migration. They use MySQL as test application. Voyager does not consider other advanced features such as network

namespaces, memory deduplication, or migration of established TCP connections.

4. Building Blocks for Live Migration

In this section, we describe how to accomplish live migrations with runC containers. Live migration attempts to provide a seamless transfer of service between physical hosts without impacting client processes or applications. This migrating to the destination host the container's storage, its memory, its network identity, and its established connections.

4.1. Storage Migration

The migration of the container's *rootfs* filesystem can be avoided by using a shared filesystem (such as NFS) that can be accessed both from the source and the destination hosts. Alternatively, a container engine that supports OCI-compliant images could take advantage of the layered structure of the images so that the underlying immutable image layers can be copied between the hosts in advance and only the top writable layer must be transferred during the actual migration [11]. However, as runC does not implement the concept of container images, let alone layered images, we opted for a conceptually similar approach in which we configured the container's filesystem as read-only (using the option `'"readonly": true'` in its *config.json*) so that the filesystem does not need to be migrated, only copied once before running the container. For containers that need to write to the filesystem, we created a *tmpfs* mount in the container, and configured the application to write in there. As this mount is stored in memory, it will be migrated when we migrate the container's memory. This mount can be configured in the *config.json* file as follows:

```
"mounts": [
    {
        "destination": "/tmp",
        "type": "tmpfs",
        "source": "tmpfs",
        "options": [
            "nosuid",
            "strictatime",
            "mode=755",
```

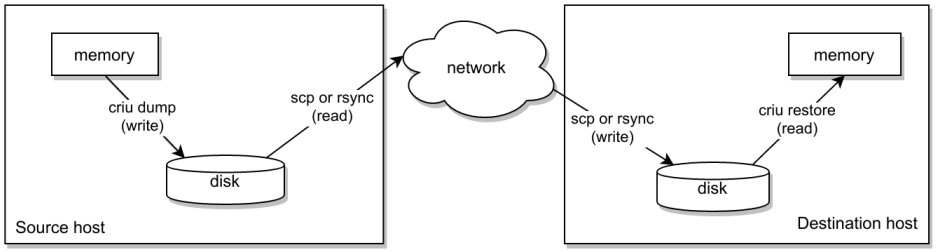


Figure 1: Diskful cold memory migration without page-server (image from `https://criu.org`).

```
            "size=1048576k"
        ]
    }
]
```

A similar rationale applies to ensure that the dump files resulting from the container's memory (see Section 4.2) are available on the remote host upon restore. For this case, we opted to use the *rsync*[16] utility to copy those files from one host to another.

4.2. Memory Migration

Figure 1 shows a simple (but typical) approach to migrate the container's memory, which we introduced before as cold migration. It consists of i) stopping the container and dumping its memory pages to files on disk, ii) reading the files and sending them over the network to the destination host, iii) receiving the files and writing them on disk, and iv) restoring the container's memory by loading the pages from files into memory.

This procedure is supported by runC through two operations, namely, *checkpoint* and *restore*, which can be used to perform a cold migration in the following way:

```
sudo runc checkpoint \
     --image-path ${IMAGES_DIR} \
     --work-path ${LOG_DIR} \
```

[16]https://rsync.samba.org/

```
    ${CONTAINER_NAME}

sudo runc restore \
    --image-path ${IMAGES_DIR} \
    --work-path ${LOG_DIR} \
    -d ${CONTAINER_NAME}
```

As mentioned before, we use the *rsync* utility to transfer the files containing the memory pages. We try to get advantage of the `--sparse` option to save some disk space on the destination host.

Given the notable downtime incurred by cold migrations, in the next subsections we are digging into diskless and iterative migrations to reduce it. The former allows checkpoint/restore without writing to disk. The latter allows for incremental pre-dumps without suspending the container.

4.2.1. Diskless Migration

CRIU stores the image files with the applications' memory on the filesystem location provided by the user. As a consequence, it relies heavily on the performance of the underlying storage backend, which is usually a disk. If the images are too big, this will result in big delays. It is of no surprise then, that reading and writing from and to disk can quickly become the bottleneck in the performance of live migration. Things get even worse due to the need to copy the data to disk several times, i.e., we write once to disk to dump the memory pages, and a second time to transfer the image files to the disk of the destination host. CRIU can perform diskless migration without putting images on disk to address this. As shown in Figure 2, a diskless migration involves two complementary aspects, namely using a *tmpfs* filesystem and a page-server process.

tmpfs. A way to mitigate the disk I/O overhead is to use *tmpfs*, a filesystem that uses RAM as storage and was first presented by Sun Microsystems in 2007 [24]. According to the Linux manual page[17], a *tmpfs* filesystem only consumes as much memory as required to store the contents of its files, which are discarded once the filesystem is unmounted. Since the files actually reside in memory, the user benefits from memory-like read/write performance. A *tmpfs* mount to store the memory image files can be simply created by running: `mount -t tmpfs none ${IMAGES_DIR}`

[17] https://man7.org/linux/man-pages/man5/tmpfs.5.html

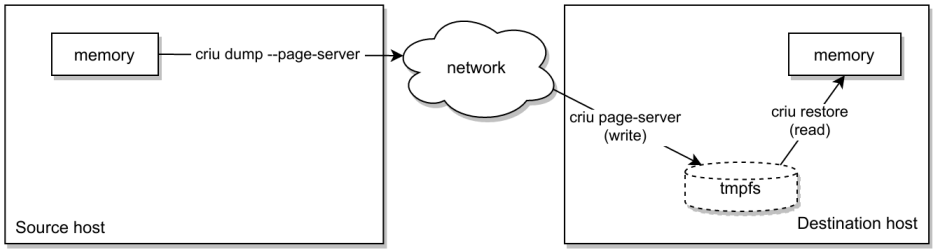


Figure 2: Diskless memory migration with page-server (image from `https://criu.org`).

Page-server. To eliminate the double read/write of the container's memory, CRIU offers a page-server[18]. It allows to send memory dumps (as they are, without encryption nor compression) directly through the network (via TCP), saving disk read/writes on the source host, writing them once they reach the destination host.

The page-server is only used to migrate memory files, which tend to be the largest ones, whereas other image files still need to be transferred (via *rsync*) to accomplish the migration. Nevertheless, these images are typically very small and will not incur significant transfer delay.

The page-server must be started on the destination node by interacting directly with CRIU as follows. Note that this is a one-shot command, hence if we perform multiple dumps (or predumps as described in next subsection), a page-server must be started for each one of them.

```
sudo criu page-server \
    --port ${PORT} \
    --work-dir ${LOG_DIR} \
    --images-dir ${IMAGES_DIR}
```

Additionally, the page-server is supported by runC as a parameter for the *checkpoint* operation as follows:

```
sudo runc checkpoint \
     --page-server ${DST_IP}:${PORT} \
     --image-path ${IMAGES_DIR} \
```

[18]https://criu.org/Page_server

```
--work-path ${LOG_DIR} \
${CONTAINER_NAME}
```

4.2.2. Iterative Migration

Another issue with cold migrations is that the container remains frozen while the memory is dumped and copied to the destination host. If the container uses a high amount of memory, this will translate into a big downtime. We can reduce this time by performing an iterative migration, which is available in two different flavors, namely pre-copy and post-copy memory migration.

Pre-copy Migration. Pre-copy memory migration, which is the more common form of iterative migration, is based on the Linux kernel's ability to track memory changes. It features an iterative pre-copy phase while the container is still running on the source to push its memory to the destination host. In the first round, all the memory pages are copied. In successive rounds, the memory pages that changed (became 'dirty') during the previous round will be copied again. This process is repeated until the number of dirty pages becomes small enough or after a maximum number of rounds. At this point, the container will be paused on the source host, the remaining dirty pages will be transferred, and the container will be resumed at the destination host.

This is based on the hypothesis that all the heavy work for the checkpoint (i.e., dumping the memory pages and transferring them) will have already been done in previous iterations, hence minimizing the application downtime. This requires that each subsequent dump is smaller than the previous one, so that the procedure converges and we are not sending the same pages repeatedly. Memory tracking supports this by marking as dirty the memory pages written between dumps, which will be the only ones included in the following transfer. Of course, this will only work if the memory dirtying rate of the application is lower than the bandwidth available for the transfers.

This procedure for pre-copy iterative migration is supported by runC as follows. First, it offers a new procedure (a.k.a. pre-dump) to snapshot the memory of the container without stopping it (tasks will remain running after pre-dump, unlike a regular dump). This is configured as a parameter (`--pre-dump`) for the *checkpoint* operation as follows. Note that this parameter automatically configures CRIU with the memory tracking option (`--track-mem`) to keep track of the memory changes.

```
sudo runc checkpoint \
```

```
    --pre-dump \
    --image-path ${IMAGES_DIR}/1/ \
    --work-path ${LOG_DIR} \
    ${CONTAINER_NAME}
```

Note that, as the iterative migration can include several dumps, each of them must refer to its own directory for images (in option `--image-path`). Moreover, subsequent dumps must be linked together so that they can be correctly re-interpreted when restoring the container. runC adds this information as a parameter (`--parent-path`) for the *checkpoint* operation as follows. Note that this path is relative to the `--image-path`.

```
sudo runc checkpoint \
    --parent-path ../1/ \
    --image-path ${IMAGES_DIR}/2/ \
    --work-path ${LOG_DIR} \
    ${CONTAINER_NAME}
```

The container can be restored (using the same command described before) from the latest directory with dumped images.

At this point, it is worth mentioning that the result of an iterative checkpoint is a layered stack of memory images in which some data is duplicated (i.e., the same memory page is present in multiple images). RunC provides support for deduplicating such data (by providing the `--auto-dedup` option to the *checkpoint* command) by punching holes in parent images (using `fallocate()` syscall with `FALLOC_FL_PUNCH_HOLE` flag)[19] where the child image is replacing an existing page, effectively freeing used disk space.

Finally, note that iterative and diskless migrations can be used together. For that purpose, runC also supports the `--page-server` parameter when pre-dumping containers. Similarly, starting the page-server also supports the `--auto-dedup` option.

Post-copy Migration. Post-copy memory migration (a.k.a. lazy memory migration) aims to minimize the downtime by transferring only the minimal execution state needed to resume the container in the destination host. Initially, the memory pages are kept in the source host and will be transferred

[19] https://criu.org/Memory_images_deduplication

in the background without stopping the restored container. If the container accesses a not yet received page during its execution, the page is faulted in from the source host over the network and injected into the running task address space (i.e., remote page fault).

This procedure for post-copy iterative migration is supported by runC as follows. First, the `--lazy-pages` parameter must be added to the *checkpoint* operation so that it dumps the container without writing its memory pages into the image files. In addition, the `--page-server` option must be used to start a page-server that will be serving the page requests from the lazy-pages daemon in the destination host. The `--status-fd` parameter can be used to know when the page-server is ready, as CRIU will write '\0' to that fd.

```
sudo runc checkpoint \
     --lazy-pages \
     --status-fd ${FD} \
     --page-server ${SRC_IP}:${PORT} \
     --work-path ${LOG_DIR} \
     --image-path ${IMAGES_DIR} \
     ${CONTAINER_NAME}
```

In the destination host, we need a lazy-pages daemon that handles the page faults in the container and injects the corresponding memory pages into its address space. The `--page-server` option instructs the daemon to connect to the page-server in the source host (as indicated by the `--address` and `--port` parameters) to fetch the missing pages.

```
sudo criu lazy-pages \
    --page-server \
    --address ${SRC_IP} \
    --port ${PORT} \
    --work-dir ${LOG_DIR} \
    --images-dir ${IMAGES_DIR}
```

Finally, the *restore* operation in the destination host must be run with `--lazy-pages` option, so that the container is restored without filling out the memory pages and ready to fill them either on demand when they are accessed or in the background by connecting to the lazy-pages daemon.

```
sudo runc restore \
    --lazy-pages \
    --image-path ${IMAGES_DIR} \
    --work-path ${LOG_DIR} \
    -d ${CONTAINER_NAME}
```

4.3. Networking Migration

The migration of the container's network identity requires that the IP addresses used by the container in the source host are also allocated to the container in the destination host. This should be managed by the container engine by means of the CNI plugin. In our case, as runC does not deal with the container's networking, we interact with the plugin by ourselves to configure the container's networking in the destination host, by following the procedure described before in Section 2.3. In particular, we create a network namespace with the same name, we use the CNI plugin to populate it with the MACVLAN network device and interface (with the same IP address) and we instruct runC to use that network namespace to restore the container. Other container's networking resources (such as interfaces, routing tables, etc.) could be migrated similarly.

To checkpoint successfully a container running in a network namespace, we must also indicate to CRIU to treat it as an external namespace, that is, to leave it out of the dumped state assuming that a third-party tool will take care of restoring it in the destination host. The information about external namespaces can be passed to CRIU with parameter `--external net[<inode>]:<key>`. Luckily, runC manages this automatically when it is asked to checkpoint a container running in an external namespace.

Restoration of a container into an existing network namespace works similarly. CRIU expects the information about the namespace via the parameter `--inherit-fd fd[<fd>]:<key>`. Again, runC configures this automatically when it finds an external namespace to be restored in the checkpoint image files.

The ability to migrate established TCP connections transparently to their clients builds mainly on the use of the TCP_REPAIR socket option [6], which was included in the Linux kernel version 3.5. This option sets an active socket into a special 'repair' mode, in which it can be manipulated without any communication from/to the remote end. It also allows CRIU to gather all of the information about the current state of the corresponding network connection which should move with the socket when it is migrated. This includes,

for instance, the contents of the send and receive queues, the send and receive sequence numbers, the maximum segment size, the window scale factor, the selective acknowledgments flag, and the timestamps flag. To restore the connection, CRIU will create a new socket in the destination host, put it in 'repair' mode, populate the connection state with the information from the origin socket, and reestablish the connection (but without any communication from the remote end). At this point, the networking protocol (i.e., TCP) can restart the traffic and resurrect the data sequence. Note that the connection-oriented nature of TCP will also take care of the in-flight segments that were lost during the migration, by means of the corresponding retransmissions (transparently to the application).

We can enable this behavior in runC by specifying the command-line option `--tcp-established` for both the *checkpoint* and the *restore* operations.

5. Experiments

5.1. Testbed

The experiments in this paper have been executed in a computer with an Intel Core i7-8565U processor at 1.80GHz with 4 cores (with 2 hyperthreads on each one), 16 GB RAM, 512 GB SSD hard disk, and 1-Gigabit Ethernet network.

VirtualBox 7.0.10 has been used to run three virtual machines with 4 VCPUs, 4 GB RAM, 20 GB disk with ext4 filesystem, and a paravirtualized (through virtio-net driver) bridged network adapter. All the VMs run the same software stack, namely LUbuntu 22.04.1 (64 bits) Linux distribution with kernel 5.19.0-32-generic, runC version 1.1.8, CRIU version 3.18[20] (built from source), and *rsync* version 3.2.7.

As computation- and memory-intensive HPC benchmarks, we use the three pseudo-applications part of the NAS Parallel Benchmark[21], namely BT (Block Tri-diagonal matrix solver), LU (Lower Upper Gauss-Seidel solver), and SP (Scalar Penta-Diagonal solver). The OpenMP version 3.4.2 has been used for the reported experiments. Each benchmark runs 4 parallel processes and we use the class sizes A and B. As memory-intensive benchmark, we use

[20]This version is required at least as former ones can result in lower performance of the application after restoring: https://github.com/checkpoint-restore/criu/issues/1171

[21]https://www.nas.nasa.gov/software/npb.html

the Redis in-memory database[22]. We use version 7.0.12 of the redis-server, as well as the redis-benchmark to simulate client load. As network-intensive benchmark, we make use of iPerf3[23] tool (version 3.9), which measures the maximum achievable bandwidth on IP networks.

5.2. Diskless Migration

In this section, we evaluate the performance of diskless migration by comparing four different scenarios:

- Diskful without page-server
- Diskful with page-server
- Diskless without page-server
- Diskless with page-server

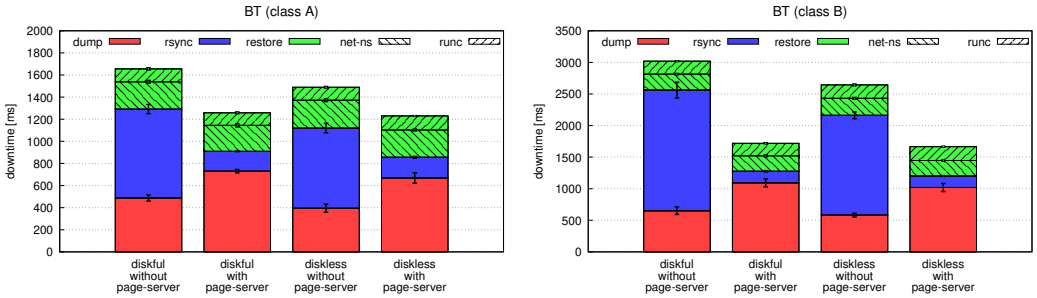
'Diskful' scenarios use disk-backed directories to store the images, whereas 'Diskless' scenarios use *tmpfs* directories.

For each scenario, we measure the time needed i) to dump the container ('dump'), ii) transfer the image files to the destination host using *rsync* ('rsync'), and iii) restore the container ('restore'). For the latter, we differentiate the time needed to restore the network namespace ('net-ns') from the restoration of the actual runC container within that namespace ('runc').

The comparison includes classes A and B from the BT, SP, and LU pseudo-applications from the NPB benchmark, as well as the Redis in-memory database, which has been pre-loaded with 1000000 and 2000000 keys. We present the resulting average and standard deviation values of 5 executions (after filtering outliers) in Figures 3, 4, 5, and 6, respectively. Numeric results are displayed in detail (including also 95% confidence intervals) in Tables A.2 and A.3 in the appendix.

As shown in the figures, diskless is always equal or better than diskful, especially for the dump of the memory pages and, particularly, when migrating without a page-server. This was to be expected, as *tmpfs* gives better raw read/write performance. Because of that, the benefit increases with the size of the memory to be migrated. Therefore, diskless migration is

[22]https://redis.io/
[23]https://iperf.fr/



(a) BT (class A) (b) BT (class B)

Figure 3: Downtime when migrating a runC container running BT comparing settings diskful/diskless and with/without a page-server.

recommended, unless there are constraints in the amount of available RAM memory and/or the container to be migrated uses a lot of memory.

Migration with a page-server clearly outweighs the one without. Although the dump phase takes longer, because it now encompasses the transfer of the memory pages, this is compensated with a notable reduction of the *rsync* time, which now must only take care of the remaining dump files, which are considerably smaller. Migration with a page-server has the added overhead to start the server itself, but this is quickly balanced out as the memory used by the container increases.

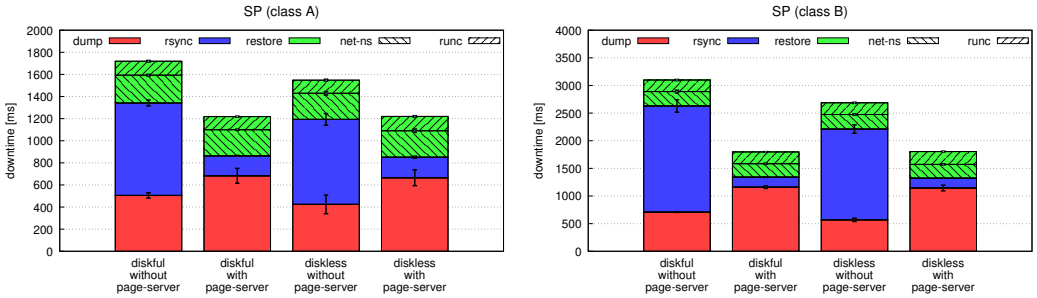
The restore phase is not affected much by the diskful/disless and with/without page-server settings. Restoring the network namespace always needs the same amount of time, and whereas restoring the runC container needs more time as the size of the memory files get bigger, this is not significant in comparison with the dump and transfer phases.

Summarizing, when migrating memory-intensive applications in a distributed environment, using a page-server is more determinant than using a diskless approach based on *tmpfs*, although a combination of both yields the best performance.

5.3. Iterative Migration

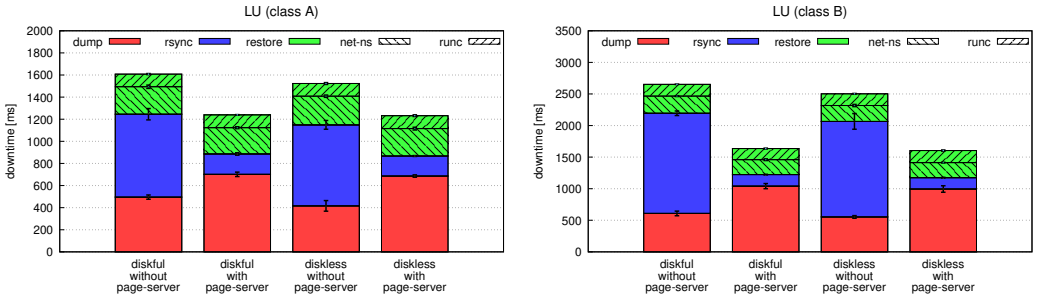
In this section, we evaluate the performance of the two flavors of iterative migration, namely pre-copy and post-copy memory migration.

Pre-copy Migration. We evaluate this migration flavor by comparing four different scenarios:



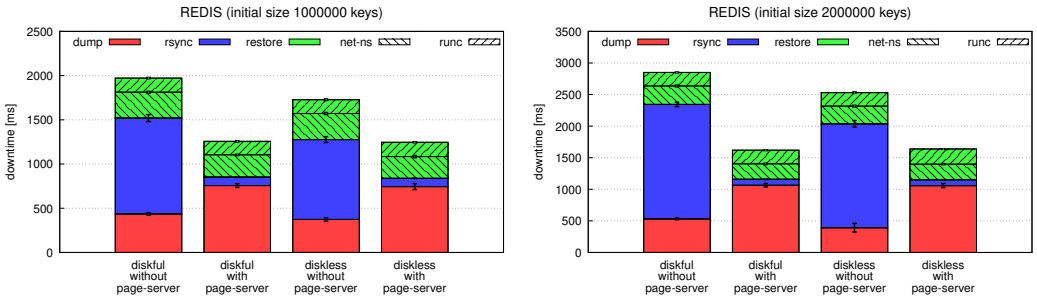
(a) SP (class A) (b) SP (class B)

Figure 4: Downtime when migrating a runC container running SP comparing settings diskful/diskless and with/without a page-server.



(a) LU (class A) (b) LU (class B)

Figure 5: Downtime when migrating a runC container running LU comparing settings diskful/diskless and with/without a page-server.



(a) Redis (1000000 keys) (b) Redis (2000000 keys)

Figure 6: Downtime when migrating a runC container running Redis database comparing settings diskful/diskless and with/without a page-server.

- Without page-server and without auto-deduplication
- With page-server and without auto-deduplication
- Without page-server and with auto-deduplication
- With page-server and with auto-deduplication

For each scenario, we perform three pre-dumps ('pre1', 'pre2', and 'pre3'), and the final dump ('dump'), waiting one second between each of them. Again, the comparison includes classes A and B from the BT, SP, and LU pseudo-applications from the NPB benchmark, as well as the Redis in-memory database, which has been pre-loaded with 1000000 and 2000000 keys. For the latter, we include a static scenario in which Redis does not update the memory during successive dumps, as well as a dynamic one in which, between each dump, we run a client benchmark that issues as many queries as the 1% of the initial size of the Redis database using random keys over a keyspace 20% higher than the initial one. This means that, on average, the client will update 80% of the initial keys, while adding 20% of new keys. Again, we present the resulting average and standard deviation values of 5 executions in two sets of figures.

One set displays the size of each memory dump, in particular, the overall apparent size (in solid color) and real size on disk (in diamond pattern) of the *image-*.img* files (which contain the dumps of the container's memory), both at the source ('src') and the destination ('dst') hosts. See Figures 7, 8, 9, 10, and 11, respectively.

Another set displays the downtime corresponding to the final memory dump, differentiating the time needed i) to dump the container ('dump'), ii) transfer the image files to the destination host using *rsync* ('rsync'), and iii) restore the container ('restore'). For the latter, we differentiate the time needed to restore the network namespace ('net-ns') from the restoration of the actual runC container within that namespace ('runc'). See Figures 12, 13, 14, 15, and 16, respectively. The downtime numeric results are displayed in detail (including also 95% confidence intervals) in Tables A.5 and A.6 in the appendix.

As shown in the figures, pre-dumping the memory pages allows to reduce the size of the final dump, which determines the downtime of the application, as this is the only one that needs it to be stopped. The size reduction depends of the memory usage pattern by the application. For instance, as the NAS

NPB applications are memory-intensive, a noticeable amount of memory pages becomes dirty between each pre-dump and, consequently, the size of successive pre-dumps is still significant. On the other side, we observe that, as expected, if we make no changes to the container's memory after the first dump, as occurs with Redis when there are no clients, the amount of information to be re-transferred is negligible, barely 300 KB.

The impact of the memory deduplication is very noticeable. Thanks to this optimization, the intermediate pre-dumps do not use any disk space, neither in the source nor in the destination hosts. In fact, it guarantees that each page is only stored once on each host. In some cases (see for instance Figure 10), some disk space can be saved in the destination host without deduplicating the memory if the `'--sparse'` option of *rsync* is used to transfer the files.

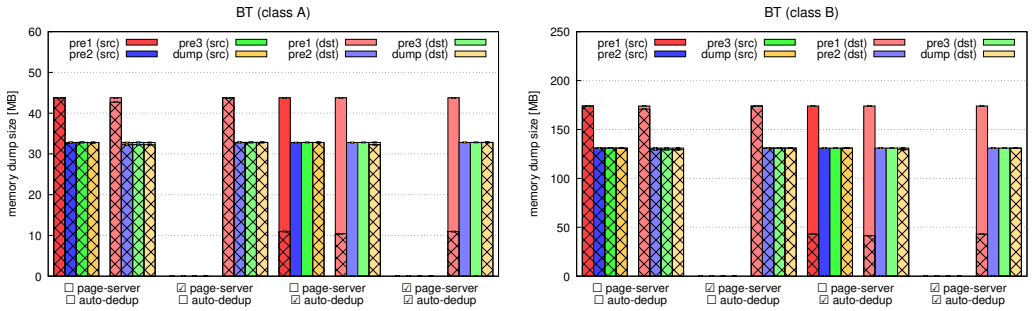
However, memory deduplication has an important impact of the downtime, which increases notably. This occurs when subsequent dumps replace existing pages as these pages, which were already transferred after a previous predump, must be also deduplicated at the destination host, thus involving additional transfers.

The impact of the page-server is also obvious in the figures, as no files are dumped in the source host. Although there is no difference in the size of the memory dump files with or without the page-server, we demonstrated in Section 5.2 that using a page-server for the migration decreases the downtime.

Summarizing, we conclude that iterative migration by means of memory tracking can minimize the downtime with computation-intensive applications, and alleviate it even with memory-intensive ones. Likewise, memory deduplication and using a page-server minimize the amount of disk space used on both the source and the destination hosts, but at the cost of increasing the downtime, especially for memory-intensive applications.

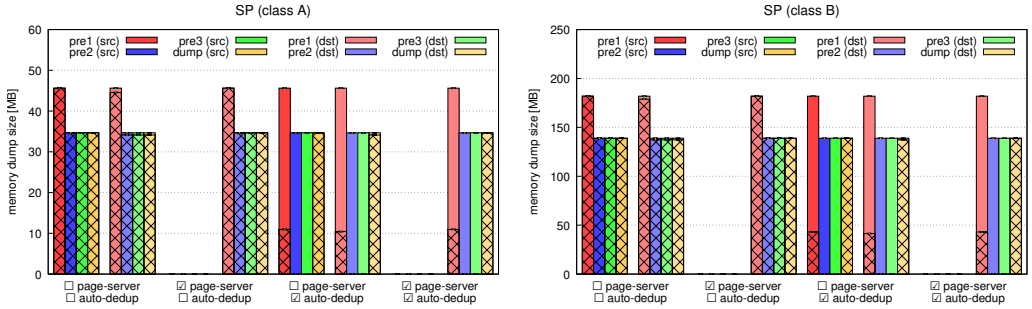
Post-copy Migration. We evaluate this migration flavor by measuring the downtime, particularly, the time needed i) to dump the container ('dump'), ii) transfer the image files to the destination host using *rsync* ('rsync'), and iii) restore the container ('restore'). For the latter, we differentiate the time needed to restore the network namespace ('net-ns') from the restoration of the actual runC container within that namespace ('runc').

Again, the comparison includes classes A and B from the BT, SP, and LU pseudo-applications from the NPB benchmark, as well as the Redis in-memory database, which has been pre-loaded with 1000000 and 2000000



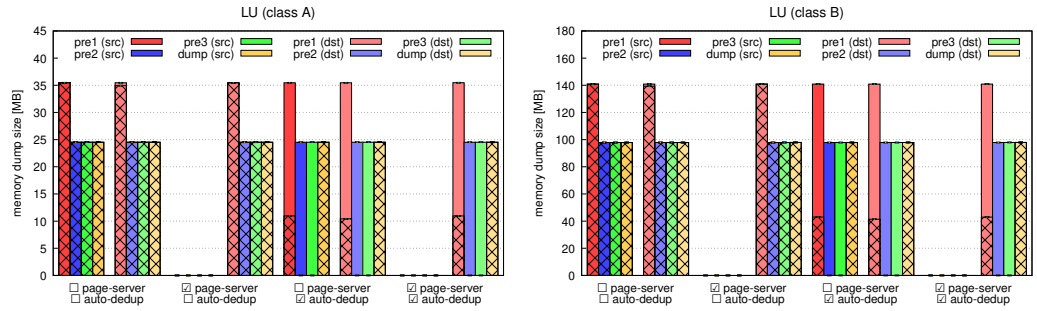
(a) BT (class A) (b) BT (class B)

Figure 7: Size of the memory dump when migrating iteratively a runC container running BT comparing settings with/without auto-dedup and with/without a page-server.



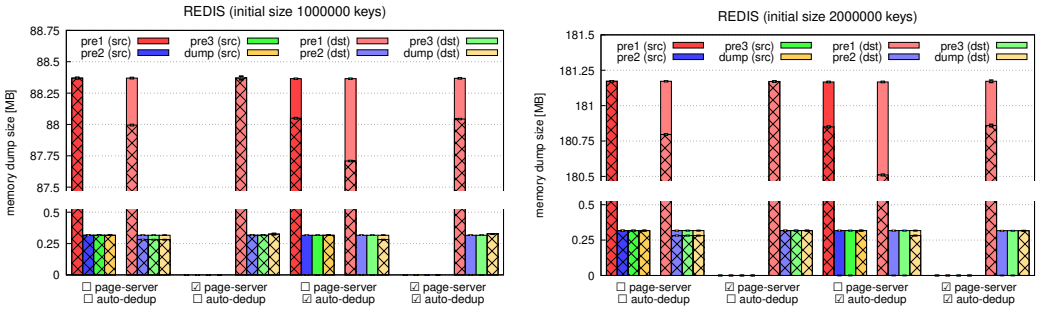
(a) SP (class A) (b) SP (class B)

Figure 8: Size of the memory dump when migrating iteratively a runC container running SP comparing settings with/without auto-dedup and with/without a page-server.



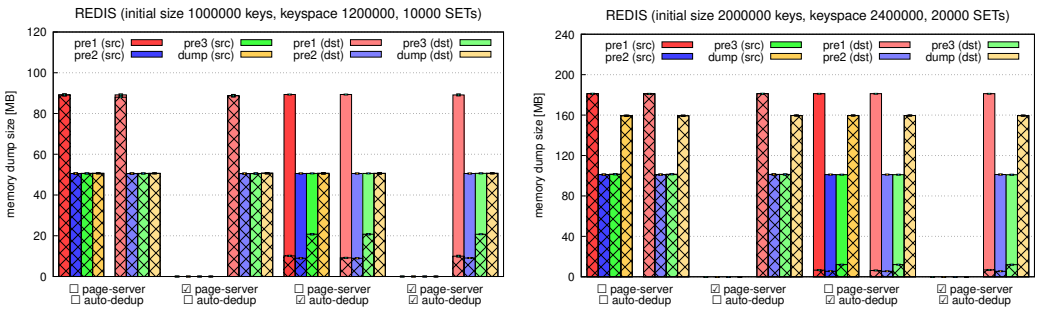
(a) LU (class A) (b) LU (class B)

Figure 9: Size of the memory dump when migrating iteratively a runC container running LU comparing settings with/without auto-dedup and with/without a page-server.



(a) Redis (1000000 keys) (b) Redis (2000000 keys)

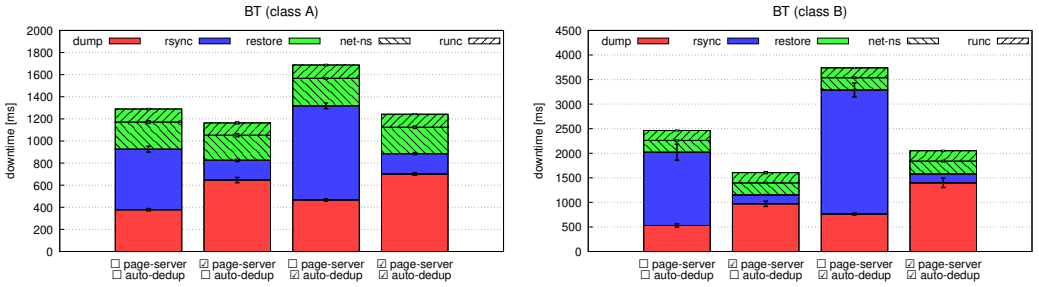
Figure 10: Size of the memory dump when migrating iteratively a runC container running Redis database (without clients) comparing settings with/without auto-dedup and with/without a page-server.



(a) Redis (1000000 keys)

(b) Redis (2000000 keys)

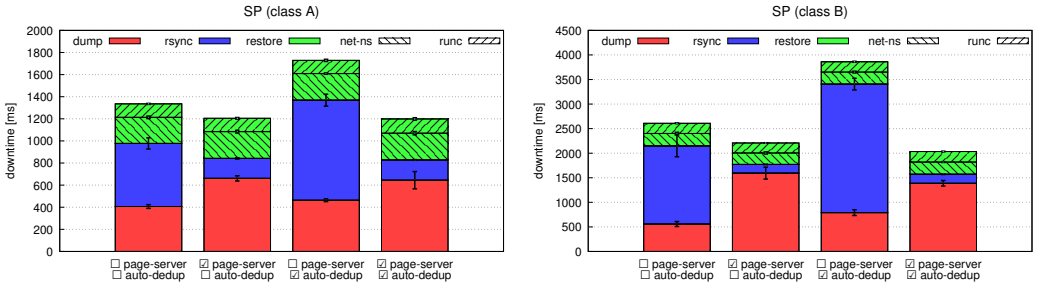
Figure 11: Size of the memory dump when migrating iteratively a runC container running Redis database (with clients) comparing settings with/without auto-dedup and with/without a page-server.



(a) BT (class A)

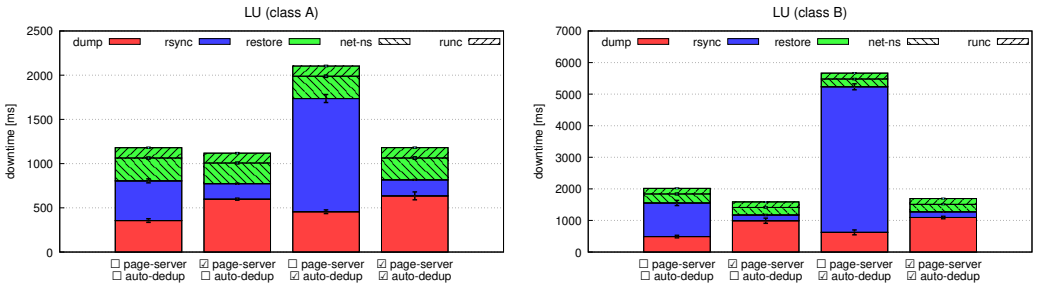
(b) BT (class B)

Figure 12: Downtime when migrating iteratively a runC container running BT comparing settings with/without auto-dedup and with/without a page-server.



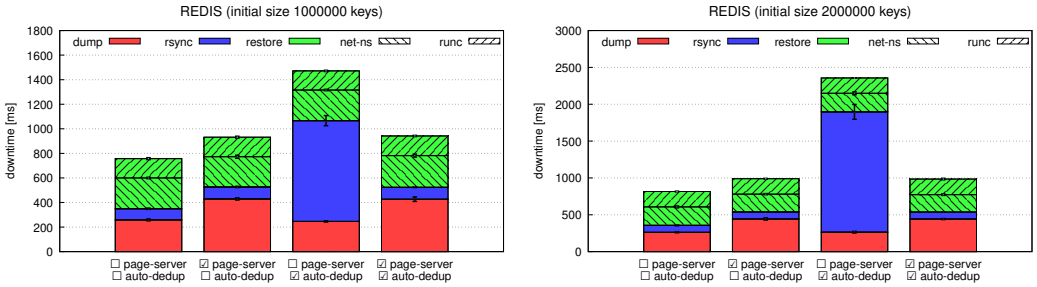
(a) SP (class A) (b) SP (class B)

Figure 13: Downtime when migrating iteratively a runC container running SP comparing settings with/without auto-dedup and with/without a page-server.



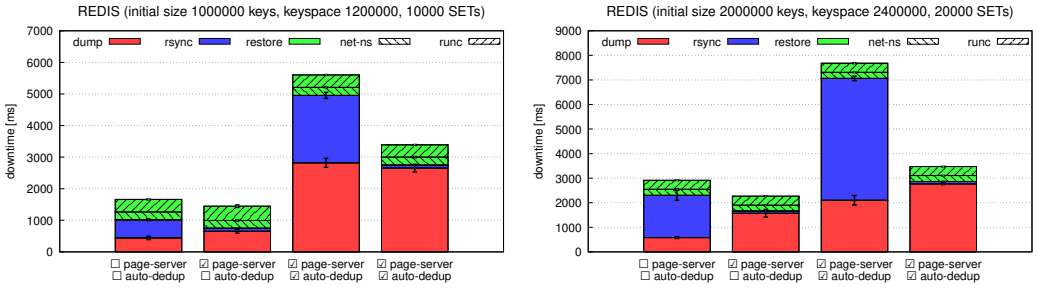
(a) LU (class A) (b) LU (class B)

Figure 14: Downtime when migrating iteratively a runC container running LU comparing settings with/without auto-dedup and with/without a page-server.



(a) Redis (1000000 keys) (b) Redis (2000000 keys)

Figure 15: Downtime when migrating iteratively a runC container running Redis (without clients) comparing settings with/without auto-dedup and with/without a page-server.



(a) Redis (1000000 keys)

(b) Redis (2000000 keys)

Figure 16: Downtime when migrating iteratively a runC container running Redis (with clients) comparing settings with/without auto-dedup and with/without a page-server.

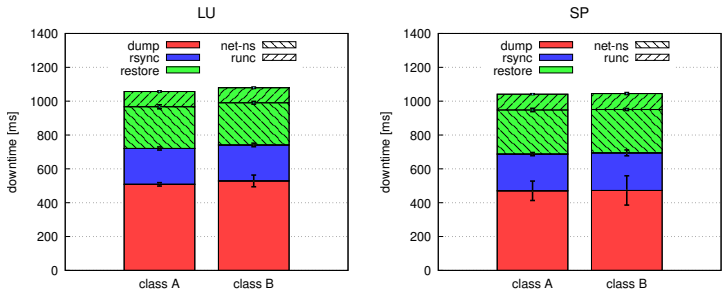
keys. We present the resulting average and standard deviation values of 5 executions (after filtering outliers) in Figures 17 and 18, respectively. Numeric results are displayed in detail (including also 95% confidence intervals) in Table A.4 in the appendix.

As shown in the figures, the times corresponding to the 'dump' and 'rsync' phases are now proportionally lower in comparison with the 'restore', as only the minimal execution state needed to resume the container in the destination host is dumped and transferred at this point. Consequently, this approach shows the lowest overall downtime. However, note that this downtime does not account for the impact on the performance of the transfer of the memory pages from the source host when there is a remote page fault. This will be evaluated in Section 5.5. Finally, it is worth mentioning that the downtime does not show relevant differences depending on the migrated application, because their execution states have similar sizes, unlike the size of the memory dumps, which can differ heavily among applications.

5.4. Migration of Established TCP Connections

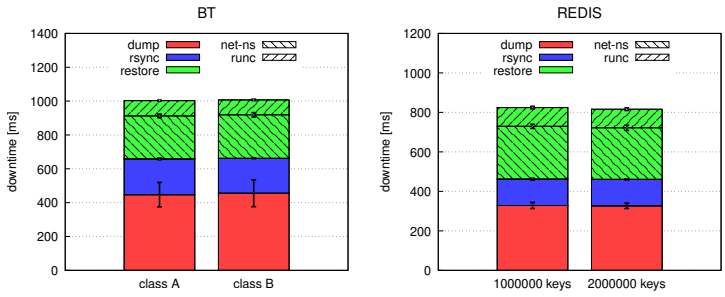
In this section, we evaluate the performance of the capability to migrate established TCP connections.

We start an iPerf3 server within a container, and we run the iPerf3 client against this server. After 10 seconds, we dump the server's container and we restore it immediately on the same host (and using the same network namespace). Then, 10 seconds later, we dump the server again, but now, we restore it in another host after transferring the dump files. All of this is done transparently to the client, whose connection is never closed.



(a) LU (b) SP

Figure 17: Downtime when migrating lazily a runC container running LU and SP.



(a) BT (b) Redis

Figure 18: Downtime when migrating lazily a runC container running BT and Redis database (without clients).

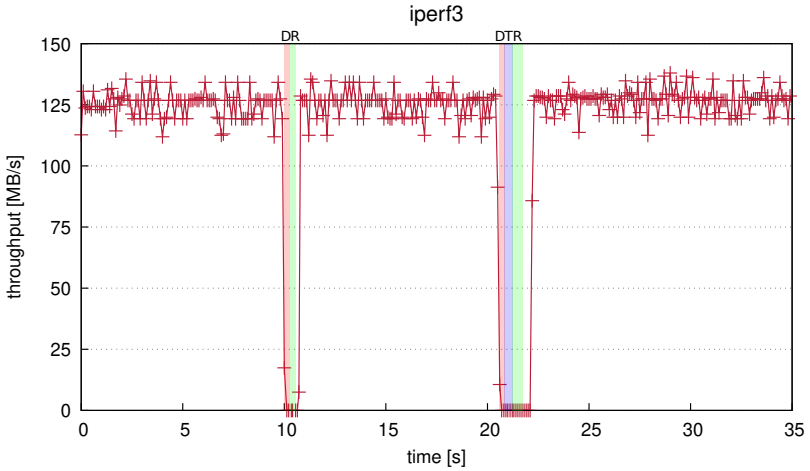


Figure 19: Throughput of a runC container running iPerf3 in the event of a migration to the same host at time 10, and a migration to a different host 10 seconds later.

We present the client's perceived throughput as a function of time in Figure 19. During the downtime, we also differentiate the time taken to dump the container ('D'), perform the *rsync* transfer of the dump files ('T'), and restore the container ('R'). As shown in the figure, during the local migration the measured throughput downtime does not exceed 0.5 seconds. During the remote migration the downtime increases to 1.2 seconds, obviously because the dump files must be now transferred to the destination host, but also because the network namespace must be also created in that node before restoring the container.

Given the behavior of the TCP protocol, we also hypothesize that the established TCP connections might need some additional time to be fully recovered after the restoration of the container if the downtime was long (as could occur when the checkpoint files are huge and/or the network bandwidth is low). In order to confirm this, we repeated the previous experiment, but with an added delay of 3 seconds before starting the container's restoration. As shown in Figure 20, which displays the corresponding client's throughput for this experiment, it takes more than 2 seconds after the restoration to get the connection back to full speed. This is because of the behavior of the iPerf client at the TCP layer. The client is sending as many packets as possible and reporting the measured capacity. During the downtime, those packets are just discarded by the network filters. But for the client, as the socket is

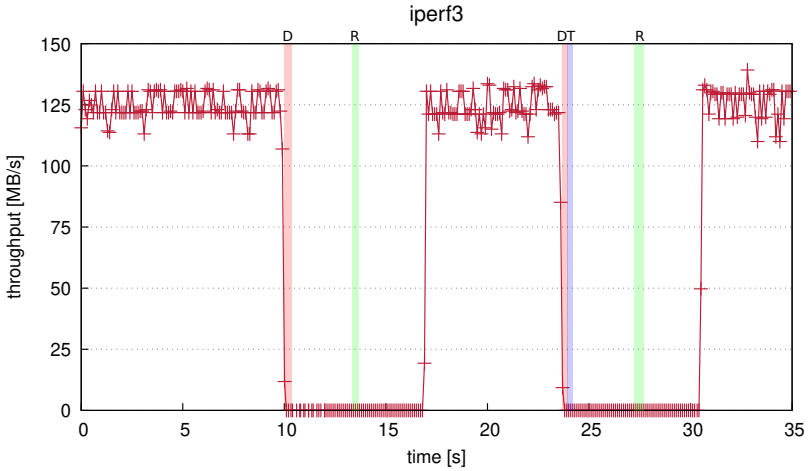


Figure 20: Throughput of a runC container running iPerf3 in the event of a migration to the same host at time 10, and a migration to a different host 10 seconds later, both with an added 3-second delay before restoring.

never closed, those packets remain as unacknowledged, and hence the client will request them to be retransmitted. The specification of the TCP protocol [20] determines that the retransmission timeout must be doubled every time a packet is not acknowledged, therefore the recurrent outage of ACKs might cause the client to back-off for increasing periods.

Summarizing, if the container can be restored soon after the memory dump, the downtime values are small. We think that given these low values together with the fact that the iPerf3 benchmark is very network-intensive, this migration can be considered really 'live' and suitable for most client-server scenarios with negligible impact in the overall quality of service.

5.5. Impact of Migration on Application Performance

In this section, we put all the building blocks together, and we evaluate the impact of diskless, iterative (pre-copy and post-copy), and connection-persistent migration on the performance of the applications.

We choose the most favorable configurations according to the previous experiments and, consequently, we configure live migrations with the *tmpfs* filesystem and with page-server (w/ PS), and the iterative pre-copy migration with one pre-dump and without auto-deduplication. We also configure all the migrations to restore established TCP connections.

We set up an experiment in which each application runs for 10 seconds, then we dump it and restore it immediately in another host after transferring the dump files. There, the application runs until completion. For Redis, this means serving all the queries from a redis-benchmark asking to SET 1000000 keys. As before, the client's connection is never closed.

Table 1 displays the downtime and the elapsed time (mean, standard deviation, and 95% confidence interval of 5 executions) in seconds to execute LU, SP, BT, and Redis applications with several live migration scenarios, namely 'No migration', 'Live migration (w/ PS)', 'Pre-copy live migration (w/ PS)', and 'Post-copy live migration'. As shown in the table, NAS NPB applications are only between 1.1 and 2.4 seconds slower when performing the 'live migration w/ PS'. Most of this time (between 1.3 and 1.9 seconds) corresponds to the downtime due to the migration, which is higher as we increase the size of the application. NPB applications are between 1.5 and 2.9 seconds slower when performing the 'pre-copy live migration w/ PS'. Between 1.3 and 1.7 seconds are the actual downtime due to the migration, which is lower than with 'live migrations w/ PS', while the rest come from the performance interference while pre-dumping the application. Finally, NPB applications are between 1.8 and 8.5 seconds slower when performing the 'post-copy live migration'. Whereas this type of migration shows the lowest downtime (between 1.1 and 1.2 seconds), there is a notable performance penalty, especially for memory-intensive applications, to transfer the memory pages from the source to the destination host when there are remote page faults.

Redis shows a similar behavior, but with slightly higher downtimes and performance impact of the migration. In particular, it is between 2.5 and 4 seconds slower when performing the 'live migration w/ PS', between 3.7 and 6.6 seconds slower when performing the 'pre-copy live migration w/ PS', and between 3 and 4 seconds slower when performing the 'post-copy live migration'. In this case, apart from the impact of the actual downtime (between 1.2 and 2.5 seconds), the pre-dump interference, and the added overhead to transfer memory pages from the source host when doing a post-copy migration, an additional period to bring the connection to full speed again is needed (as explained in Section 5.4).

Table 1: Downtime and elapsed time (mean (M), standard deviation (SD), 95% confidence interval (CI)) in seconds to execute LU, SP, BT, and Redis applications comparing different migration approaches.

	downtime			**elapsed time**		
	M	SD	95% CI	M	SD	95% CI
LU (class A)						
No migration				12.46	0.21	(12.20, 12.72)
Live migration (w/ PS)	1.36	0.08	(1.26, 1.46)	13.51	0.12	(13.36, 13.66)
Pre-copy live migration (w/ PS)	1.26	0.05	(1.19, 1.32)	13.97	0.12	(13.82, 14.12)
Post-copy live migration	1.14	0.05	(1.07, 1.20)	14.47	0.15	(14.28, 14.66)
LU (class B)						
No migration				50.02	0.05	(49.96, 50.08)
Live migration (w/ PS)	1.72	0.04	(1.66, 1.77)	51.6	0.13	(51.44, 51.77)
Pre-copy live migration (w/ PS)	1.63	0.04	(1.58, 1.67)	52.3	0.2	(52.06, 52.55)
Post-copy live migration	1.24	0.03	(1.21, 1.28)	57.04	0.4	(56.55, 57.53)
SP (class A)						
No migration				10.98	0.05	(10.92, 11.04)
Live migration (w/ PS)	1.46	0.04	(1.42, 1.51)	12.34	0.03	(12.31, 12.37)
Pre-copy live migration (w/ PS)	1.28	0.08	(1.18, 1.38)	12.72	0.25	(12.40, 13.03)
Post-copy live migration	1.16	0.06	(1.08, 1.24)	12.78	0.29	(12.42, 13.13)
SP (class B)						
No migration				48.32	0.27	(47.99, 48.66)
Live migration (w/ PS)	1.94	0.09	(1.83, 2.04)	50.08	0.15	(49.90, 50.27)
Pre-copy live migration (w/ PS)	1.68	0.08	(1.59, 1.78)	51.06	0.32	(50.66, 51.46)
Post-copy live migration	1.21	0.1	(1.08, 1.34)	56.83	0.88	(55.74, 57.91)
BT (class A)						
No migration				20.46	0.38	(19.98, 20.94)
Live migration (w/ PS)	1.46	0.02	(1.43, 1.49)	22.11	0.12	(21.96, 22.27)
Pre-copy live migration (w/ PS)	1.33	0.03	(1.30, 1.37)	22.22	0.08	(22.12, 22.33)
Post-copy live migration	1.12	0.04	(1.07, 1.17)	22.54	0.12	(22.39, 22.69)
BT (class B)						
No migration				85.38	0.39	(84.90, 85.86)
Live migration (w/ PS)	1.88	0.07	(1.80, 1.96)	87.76	0.36	(87.30, 88.21)
Pre-copy live migration (w/ PS)	1.73	0.05	(1.67, 1.80)	88.31	0.29	(87.95, 88.68)
Post-copy live migration	1.17	0.06	(1.09, 1.24)	91.74	0.73	(90.83, 92.65)
REDIS (initial size 1000000 keys, keyspace 1200000, 1000000 SETs)						
No migration				98.66	0.6	(97.92, 99.40)
Live migration (w/ PS)	2.15	0.07	(2.06, 2.23)	101.19	0.87	(100.11, 102.27)
Pre-copy live migration (w/ PS)	1.89	0.08	(1.80, 1.99)	102.35	0.28	(101.99, 102.70)
Post-copy live migration	1.18	0.09	(1.06, 1.30)	101.72	0.91	(100.60, 102.85)
REDIS (initial size 2000000 keys, keyspace 2400000, 1000000 SETs)						
No migration				98.78	0.77	(97.82, 99.73)
Live migration (w/ PS)	2.48	0.07	(2.39, 2.57)	102.79	0.36	(102.34, 103.24)
Pre-copy live migration (w/ PS)	2.36	0.07	(2.27, 2.44)	105.4	0.45	(104.84, 105.95)
Post-copy live migration	1.20	0.06	(1.12, 1.28)	102.83	0.44	(102.29, 103.37)

6. Conclusions

In this paper, we have explored the use of CRIU's advanced features to implement diskless, iterative (pre-copy and post-copy) migrations of runC containers with external network namespaces and established TCP connections, so that memory-intensive and connection-persistent HPC applications can live-migrate. Our extensive experiments to characterize the performance impact of those features have demonstrated that using a page-server is determinant to reduce the downtime as well as the disk space used in the source host, although using a diskless approach based on *tmpfs* can also yield benefit when the application's memory footprint is high. Iterative migration also minimizes the downtime of computation- and network-intensive applications by reducing the size of the last dump. It can also help with memory-intensive applications depending on their memory usage footprint. Likewise, memory deduplication minimizes the amount of disk space used by ensuring that each memory page is store only once but at the cost of increasing the downtime. Finally, the ability to migrate external network namespaces and established TCP connections allows hiding the small downtimes to the clients, especially when the container can be restored soon after the memory dump. All in all, we have shown that properly-configured live migrations incur low application downtime and memory/disk usage and are indeed practicable in containerized HPC Clouds.

As future work, we plan to experiment with live container migrations in larger scale HPC Clouds. We also aim to take advantage of live migration to implement resource management and scheduling policies for HPC Clouds. We will also explore the problem of coordinated checkpoint/restore using CRIU for distributed computations.

Acknowledgment

We acknowledge Carlos Segarra for his work in the prospective prototype of our migration system [22]. This research was partially supported by the Spanish Government under contract PID2019-107255GB-C22, by the Generalitat de Catalunya under contract 2021-SGR-00478, and by the EU-HORIZON programme under grant agreement 101092646.

Competing Interests

The authors declare that they have no competing interests.

Appendix A. Detailed Numeric Experimental Results

Table A.2: Downtime in ms (mean (M), standard deviation (SD), 95% confidence interval (CI)) when migrating a runC container running LU, SP, and BT comparing settings diskful/diskless and with/without a page-server (PS).

	dump			**rsync**			**restore net-ns**			**restore runc**		
	M	SD	95% CI	M	SD	95% CI	M	SD	95% CI	M	SD	95% CI
LU (class A)												
diskful w/o PS	495.2	19.2	(471.3, 519.1)	750.2	51.5	(686.2, 814.2)	248.4	14.8	(230.0, 266.2)	114.4	4.9	(108.3, 120.5)
diskful w/ PS	701.2	19.9	(676.5, 725.9)	184.6	10.8	(171.2, 198.0)	237.4	8.7	(226.6, 248.2)	116.2	2.9	(112.6, 119.8)
diskless w/o PS	416	48.2	(356.1, 475.9)	732.8	39.7	(683.5, 782.1)	259.6	10.1	(247.0, 272.2)	114.6	8.2	(104.5, 124.7)
diskless w/ PS	686	10.8	(672.6, 699.4)	181.2	3.9	(176.4, 186.0)	247.8	12.1	(232.8, 262.8)	117.2	7.5	(107.9, 126.5)
LU (class B)												
diskful w/o PS	608.8	36.6	(563.4, 654.2)	1585.6	35.7	(1541.3, 1629.9)	271.6	6.4	(263.6, 279.6)	185.8	2.3	(182.9, 188.7)
diskful w/ PS	1041	40.6	(990.6, 1091.4)	182.4	5.2	(176.0, 188.8)	235.8	13.6	(219.0, 252.6)	176.8	7.2	(167.9, 185.7)
diskless w/o PS	552.8	21	(526.7, 578.9)	1512.6	124.6	(1357.9, 1667.3)	251.6	17.2	(230.3, 272.9)	185.8	5.2	(179.3, 192.3)
diskless w/ PS	994.8	51.5	(930.9, 1058.7)	180	5.3	(173.4, 186.6)	238	5.4	(231.3, 244.7)	191.6	9	(180.4, 202.8)
SP (class A)												
diskful w/o PS	505	24.4	(474.7, 535.3)	836.6	26.8	(803.4, 869.8)	251.4	8	(241.5, 261.3)	126.2	7.3	(117.1, 135.3)
diskful w/ PS	682.6	67.2	(599.2, 766.0)	181	4	(176.0, 186.0)	236	5.9	(228.6, 243.4)	119	4.3	(113.6, 124.4)
diskless w/o PS	424.4	84.1	(319.9, 528.9)	769	52.8	(703.4, 834.6)	235.8	15.7	(216.4, 255.2)	118.6	8.8	(107.7, 129.5)
diskless w/ PS	664.8	71.7	(575.8, 753.8)	186.2	9.6	(174.2, 198.2)	239.2	15.5	(219.9, 258.5)	129.6	7.2	(120.7, 138.5)
SP (class B)												
diskful w/o PS	709.8	7.3	(700.7, 718.9)	1919	112.7	(1779.1, 2058.9)	261	24.6	(230.5, 291.5)	209	10.7	(195.7, 222.3)
diskful w/ PS	1161.4	24.4	(1131.1, 1191.7)	181.8	6.8	(173.4, 190.2)	240.2	10	(227.8, 252.6)	214	6	(206.6, 221.4)
diskless w/o PS	568.4	30.3	(530.7, 606.1)	1643.4	76.3	(1548.7, 1738.1)	262.6	12.2	(247.4, 277.8)	212	11.4	(197.8, 226.2)
diskless w/ PS	1144.8	51.8	(1080.5, 1209.1)	179	4.1	(173.9, 184.1)	247.6	10.2	(234.9, 260.3)	232.8	8.3	(222.5, 243.1)
BT (class A)												
diskful w/o PS	487.6	27.4	(453.6, 521.6)	803.8	41.7	(752.1, 855.5)	246.8	13	(230.7, 262.9)	117	10.1	(104.4, 129.6)
diskful w/ PS	729.8	15.7	(710.3, 749.3)	179.8	2.9	(176.2, 183.4)	235	14.3	(217.2, 252.8)	113.4	8	(103.5, 123.3)
diskless w/o PS	394.6	36.9	(348.8, 440.4)	725	43.1	(671.5, 778.5)	252.6	12.4	(237.2, 268.0)	117.4	6.9	(108.8, 126.0)
diskless w/ PS	668.4	47.2	(609.8, 727.0)	187.6	7.9	(177.8, 197.4)	246.4	8.6	(235.7, 257.1)	127.8	2.3	(124.9, 130.7)
BT (class B)												
diskful w/o PS	651.2	59.1	(577.8, 724.6)	1910.4	125.6	(1754.4, 2066.4)	252	10.8	(238.6, 265.4)	205.2	6.7	(196.9, 213.5)
diskful w/ PS	1093.2	62.8	(1015.2, 1171.2)	183.6	10.6	(170.4, 196.8)	243.6	14.9	(225.2, 262.0)	198.6	9	(187.4, 209.8)
diskless w/o PS	582	30.3	(544.4, 619.6)	1580	53.7	(1513.4, 1646.6)	269.6	14.3	(251.9, 287.3)	211.6	13.9	(194.3, 228.9)
diskless w/ PS	1019.6	63.7	(940.5, 1098.7)	181.6	6	(174.2, 189.0)	245	8.3	(234.7, 255.3)	219.6	7.2	(210.6, 228.6)

Table A.3: Downtime in ms (mean (M), standard deviation (SD), 95% confidence interval (CI)) when migrating a runC container running Redis (without clients) comparing settings diskful/diskless and with/without a page-server (PS).

	dump			**rsync**			**restore net-ns**			**restore runc**		
	M	SD	95% CI	M	SD	95% CI	M	SD	95% CI	M	SD	95% CI
REDIS (initial size 1000000 keys)												
diskful w/o PS	436.2	14.1	(418.6, 453.8)	1083.4	39.6	(1034.3, 1132.5)	292.6	9.9	(280.3, 304.9)	159.4	7.2	(150.5, 168.3)
diskful w/ PS	756.8	21.0	(730.7, 782.9)	95.4	3.4	(91.2, 99.6)	251.6	7.9	(241.7,261.5)	153.2	7.5	(143.9, 162.5)
diskless w/o PS	373.0	20.7	(347.3, 398.7)	903.6	33.4	(862.1, 945.1)	294.4	11.1	(280.6, 308.2)	156.8	9.2	(145.4, 168.2)
diskless w/ PS	744.6	33.8	(702.6, 786.6)	93.6	2.6	(90.4, 96.8)	245.8	12.4	(230.4, 261.2)	162	6.8	(153.5, 170.5)
REDIS (initial size 2000000 keys)												
diskful w/o PS	532.0	13.4	(515.3, 548.7)	1811.8	34.6	(1768.8, 1854.8)	291.6	13.6	(274.8, 308.4)	214.4	7.1	(205.6, 223.2)
diskful w/ PS	1062.8	27.8	(1028.3, 1097.3)	95.4	2.7	(92.0, 98.8)	245	11.1	(231.2,258.8)	216.2	7	(207.5, 224.9)
diskless w/o PS	392	68.5	(306.9, 477.1)	1645.4	50.6	(1582.6, 1708.2)	279.2	13.6	(262.3, 296.1)	214.4	11.4	(200.2, 228.6)
diskless w/ PS	1058.4	30.7	(1020.3, 1096.5)	93.6	1	(92.3, 94.9)	245.2	9.3	(233.7, 256.7)	240.4	5.3	(233.8, 247.0)

Table A.4: Downtime in ms (mean (M), standard deviation (SD), 95% confidence interval (CI)) when migrating lazily a runC container running LU, SP, BT, and Redis (without clients).

	dump			**rsync**			**restore net-ns**			**restore runc**		
	M	SD	95% CI	M	SD	95% CI	M	SD	95% CI	M	SD	95% CI
LU												
class A	508.6	10.0	(496.2, 521.0)	211.8	9.2	(200.3, 223.3)	246.4	11.9	(231.7, 261.1)	90.4	3.6	(85.9, 94.9)
class B	528.8	34.6	(485.8, 571.8)	212.4	10.6	(199.3, 225.5)	249.2	8.1	(239.2, 259.2)	89	4.3	(83.6, 94.4)
SP												
class A	470.4	57.5	(399.0, 541.8)	216.8	9.6	(204.9, 228.7)	260.6	9.7	(248.6, 272.6)	93.4	2.6	(90.2, 96.6)
class B	472.4	86.5	(365.0, 579.8)	222	17.3	(200.5, 243.5)	256	5.8	(248.8, 263.2)	94.6	5.9	(87.3, 101.9)
BT												
class A	447.2	72.3	(357.5, 536.9)	211.2	5	(205.0, 217.4)	254	11	(240.3, 267.7)	90.6	2.9	(87.0, 94.2)
class B	455.6	79.4	(357.0,554.2)	206.2	2.1	(203.5, 208.9)	257	13.1	(240.7, 273.3)	88.4	4.4	(82.9, 93.9)
REDIS												
1000000 keys	328.4	15.4	(309.3, 347.5)	133.4	4.6	(127.7, 139.1)	267.6	10.5	(254.6, 280.6)	94.8	6.4	(86.9, 102.7)
2000000 keys	326.8	14	(309.5, 344.1)	133.8	3	(130.1, 137.5)	261.2	12.9	(245.2, 277.2)	94.2	6.6	(86.0, 102.4)

Table A.5: Downtime in ms (mean (M), standard deviation (SD), 95% confidence interval (CI)) when migrating iteratively a runC container running LU, SP, and BT comparing settings with/without auto-dedup (AD) and page-server (PS).

	dump			**rsync**			**restore net-ns**			**restore runc**		
	M	SD	95% CI	M	SD	95% CI	M	SD	95% CI	M	SD	95% CI
LU (class A)												
w/o PS w/o AD	356	20.5	(330.6, 381.4)	449.2	22.4	(421.3, 477.1)	258	12	(243.1, 272.9)	116.8	5.3	(110.3, 123.3)
w/ PS w/o AD	597.8	11.4	(583.6, 612.0)	175.8	5.6	(168.8, 182.8)	233.2	10.9	(219.7, 246.7)	111.2	6.2	(103.5, 118.9)
w/o PS w/ AD	455.6	20.7	(429.8, 481.4)	1280.4	45.6	(1223.8, 1337.0)	252.2	13	(236.1, 268.3)	115.4	7.3	(106.3, 124.5)
w/ PS w/ AD	635.2	44.7	(579.7, 690.7)	181.4	4.8	(175.4, 187.4)	246.8	13.1	(230.6, 263.0)	117.2	5	(110.9, 123.5)
LU (class B)												
w/o PS w/o AD	490.4	39.3	(441.6, 539.2)	1064.4	81.3	(963.4, 1165.4)	284.0	27.3	(250.1, 317.9)	177.2	5.3	(170.6, 183.8)
w/ PS w/o AD	992	82.4	(889.7, 1094.3)	181.4	15.7	(162.0, 200.8)	238.4	16.9	(217.4, 259.4)	176	3.5	(171.6, 180.4)
w/o PS w/ AD	622.8	77.1	(527.1, 718.5)	4609.8	93.7	(4493.5, 4726.1)	245.6	17.4	(224.1, 267.1)	187.2	6.2	(179.5, 194.9)
w/ PS w/ AD	1093.6	38.4	(1045.9, 1141.3)	180	4.9	(174.0, 186.0)	237	7.6	(227.5, 246.5)	182.8	6.8	(174.4, 191.2)
SP (class A)												
w/o PS w/o AD	406.6	16.9	(385.6, 427.6)	571	50.6	(508.2, 633.8)	236	11.3	(222.0, 250.0)	121.4	5.3	(114.8, 128.0)
w/ PS w/o AD	661	23.6	(631.7, 690.3)	181.2	6.1	(173.6, 188.8)	241.6	12.1	(226.5, 256.7)	120.8	7.1	(111.9, 129.7)
w/o PS w/ AD	463.2	12.8	(447.3, 479.1)	905.4	54	(838.3, 972.5)	240.4	6	(232.9, 247.9)	119.8	10.1	(107.2, 132.4)
w/ PS w/ AD	644.8	78.1	(547.9, 741.7)	183.2	3.1	(179.4, 187.0)	242	14.7	(223.8, 260.2)	128.8	11.5	(114.5, 143.1)
SP (class B)												
w/o PS w/o AD	557.8	51.3	(494.2, 621.4)	1592.4	222.4	(1316.3, 1868.5)	252	22.8	(223.7, 280.3)	207	8	(197.0,217.0)
w/ PS w/o AD	1594.2	121.2	(1443.7, 1744.7)	177.4	4	(172.4, 182.4)	232.6	21.9	(205.4, 259.8)	204.4	5.6	(197.4, 211.4)
w/o PS w/ AD	790	57.6	(718.4, 861.6)	2616.6	118.4	(2469.6, 2763.6)	244.4	18.7	(221.2, 267.6)	210	10.6	(196.9, 223.1)
w/ PS w/ AD	1388.2	56.1	(1318.6, 1457.8)	183.4	5	(177.2, 189.6)	247.4	8.5	(236.9, 257.9)	215.6	6.7	(207.3, 223.9)
BT (class A)												
w/o PS w/o AD	378	10.6	(364.9, 391.1)	547.6	26.2	(515.1, 580.1)	245	12.2	(229.8, 260.2)	118.4	4.4	(112.9, 123.9)
w/ PS w/o AD	646.6	23.8	(617.0, 676.2)	178.8	9.8	(166.6, 191.0)	226.4	13.9	(209.2, 243.6)	111.8	8.1	(101.7, 121.9)
w/o PS w/ AD	466.4	10.7	(453.2, 479.6)	852.2	25.6	(820.4, 884.0)	248	5.8	(240.8, 255.2)	120.4	4.8	(114.4, 126.4)
w/ PS w/ AD	700.8	11.2	(686.9, 714.7)	184.8	9.1	(173.5, 196.1)	238.2	10.5	(225.2, 251.2)	118.2	2	(115.7, 120.7)
BT (class B)												
w/o PS w/o AD	529.4	32.3	(489.3, 569.5)	1491.2	162.3	(1289.7, 1692.7)	240.4	10.7	(227.1, 253.7)	202.2	6.3	(194.3, 210.1)
w/ PS w/o AD	972	53.4	(905.7, 1038.3)	181	7.2	(172.0, 190.0)	242.4	10.1	(229.9, 254.9)	210.4	19	(186.8, 234.0)
w/o PS w/ AD	762	19.5	(737.8, 786.2)	2526.2	142.7	(2349.0, 2703.4)	248	12.8	(232.1, 263.9)	202.4	4.6	(196.7, 208.1)
w/ PS w/ AD	1398.4	94.9	(1280.5, 1516.3)	181.6	3.1	(177.8, 185.4)	259.6	13.9	(242.4, 276.8)	212.4	5.5	(205.6, 219.2)

Table A.6: Downtime in ms (mean (M), standard deviation (SD), 95% confidence interval (CI)) when migrating iteratively a runC container running Redis (with/without clients) comparing settings with/without auto-dedup (AD) and page-server (PS).

	dump			**rsync**			**restore net-ns**			**restore runc**		
	M	SD	95% CI	M	SD	95% CI	M	SD	95% CI	M	SD	95% CI
REDIS (initial size 1000000 keys)												
w/o PS w/o AD	58.6	7.9	(248.7, 268.5)	90.8	5.2	(84.4, 97.2)	250.4	5.9	(243.1, 257.7)	157.4	7.7	(147.8, 167.0)
w/ PS w/o AD	429.0	9.1	(417.7, 440.3)	98	5	(91.8, 104.2)	245.6	12.5	(230.1, 261.1)	159	7.8	(149.3, 168.7)
w/o PS w/ AD	246.2	5.5	(239.4, 253.0)	820.2	41.2	(769.1, 871.3)	249.6	5.7	(242.5, 256.7)	155.6	4.9	(149.5, 161.7)
w/ PS w/ AD	426.8	18.2	(404.2, 449.4)	97.2	1.9	(94.8, 99.6)	257.8	12.0	(242.9, 272.7)	160.2	5	(154.0, 166.4)
REDIS (initial size 2000000 keys)												
w/o PS w/o AD	262	5.9	(254.6, 269.4)	94.8	5.8	(87.5, 102.1)	250	13.2	(233.6, 266.4)	209.4	4.9	(203.3, 215.5)
w/ PS w/o AD	443.2	15.5	(424.0, 462.4)	96	3.3	(91.9, 100.1)	240.8	8.4	(230.4, 251.2)	210.4	4.1	(205.3, 215.5)
w/o PS w/ AD	264.4	11.1	(250.7, 278.1)	1632.2	100.7	(1507.1, 1757.3)	252	19.4	(227.9, 276.1)	208.8	2.8	(205.3, 212.3)
w/ PS w/ AD	442	6.5	(433.9, 450.1)	94.8	1.5	(93.0, 96.6)	236	9.9	(223.7, 248.3)	212.4	9.7	(200.3, 224.5)
REDIS (with clients) (initial size 1000000 keys, keyspace 1200000, 10000 SETs)												
w/o PS w/o AD	442.2	52.6	(376.9, 507.5)	571.8	28	(537.0, 606.6)	248.6	7.3	(239.6, 257.6)	397.8	12	(382.9, 412.7)
w/ PS w/o AD	651.6	58	(579.6, 723.6)	96.6	3.5	(92.3, 100.9)	245.6	15	(227.0, 264.2)	453.6	39.5	(404.5, 502.7)
w/o PS w/ AD	2821.2	141.7	(2645.3, 2997.1)	2133.4	95	(2015.4, 2251.4)	253	9.4	(241.4, 264.6)	395.4	10.6	(382.2, 408.6)
w/ PS w/ AD	2652.2	120.6	(2502.4, 2802.0)	95.8	4.4	(90.3, 101.3)	252.6	17.5	(230.8, 274.4)	387.8	10.6	(374.6, 401.0)
REDIS (with clients) (initial size 2000000 keys, keyspace 2400000, 20000 SETs)												
w/o PS w/o AD	582.6	38.2	(535.2, 630.0)	1719.4	201.7	(1468.9, 1969.9)	244	12.9	(228.0, 260.0)	368.2	6	(360.7, 375.7)
w/ PS w/o AD	1570.2	153.4	(1379.8, 1760.6)	94.4	2.6	(91.2, 97.6)	233	6.2	(225.3, 240.7)	369	6	(361.6, 376.4)
w/o PS w/ AD	2101.2	192.9	(1861.8, 2340.6)	4960	92.7	(4844.9, 5075.1)	246.2	10.1	(233.7, 258.7)	372.4	15.1	(353.7, 391.1)
w/ PS w/ AD	2772.2	34.1	(2729.9, 2814.5)	92.4	2	(90.0, 94.8)	242	5.2	(235.6, 248.4)	367.6	8.7	(356.8, 378.4)

References

[1] J. Ansel, K. Arya, and G. Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *Proc. of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–12, 2009. doi: 10.1109/IPDPS.2009.5161063.

[2] M. Asch, T. Moore, R. Badia, M. Beck, P. Beckman, T. Bidot, F. Bodin, F. Cappello, A. Choudhary, B. de Supinski, E. Deelman, J. Dongarra, A. Dubey, G. Fox, H. Fu, S. Girona, W. Gropp, M. Heroux, Y. Ishikawa, K. Keahey, D. Keyes, W. Kramer, J.-F. Lavignon, Y. Lu, S. Matsuoka, B. Mohr, D. Reed, S. Requena, J. Saltz, T. Schulthess, R. Stevens, M. Swany, A. Szalay, W. Tang, G. Varoquaux, J.-P. Vilotte, R. Wisniewski, Z. Xu, and I. Zacharov. Big data and extreme-scale computing: Pathways to convergence-toward a shaping strategy for a future software and data ecosystem for scientific inquiry. *The International Journal of High Performance Computing Applications*, 32(4):435–479, 2018. doi: 10.1177/1094342018778123.

[3] G. Berg and M. Brattlof. Distributed Checkpointing with Docker Containers in High Performance Computing. Bachelor's thesis, Bachelor in Computer Engineering, University West, 2017.

[4] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proc. of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286. USENIX Association, 2005.

[5] Cloud Native Computing Foundation. Container Network Interface (CNI) Specification, 2021. URL `https://github.com/containernetworking/cni/blob/spec-v1.0.0/SPEC.md`.

[6] J. Corbet. TCP connection repair, 2012. URL `https://lwn.net/Articles/495304/`.

[7] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, sep 2002. ISSN 0360-0300. doi: 10.1145/568522.568525.

[8] R. Garg, G. Price, and G. Cooperman. MANA for MPI: MPI-Agnostic Network-Agnostic Transparent Checkpointing. In *Proc. of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC'19, pages 49–60. Association for Computing Machinery, 2019. ISBN 9781450366700. doi: 10.1145/3307681.3325962.

[9] M. R. Hines, U. Deshpande, and K. Gopalan. Post-Copy Live Migration of Virtual Machines. *SIGOPS Oper. Syst. Rev.*, 43(3):14–26, jul 2009. ISSN 0163-5980. doi: 10.1145/1618525.1618528.

[10] S. Hykes. Introducing runC: a lightweight universal container runtime, 2015. URL `https://www.docker.com/blog/runc/`.

[11] L. Ma, S. Yi, N. Carter, and Q. Li. Efficient Live Migration of Edge Services Leveraging Container Layered Storage. *IEEE Transactions on Mobile Computing*, 18(9):2020–2033, 2019. doi: 10.1109/TMC.2018.2871842.

[12] S. Nadgowda, S. Suneja, N. Bila, and C. Isci. Voyager: Complete Container State Migration. In *Proc. of the 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 2137–2142, 2017. doi: 10.1109/ICDCS.2017.91.

[13] M. A. S. Netto, R. N. Calheiros, E. R. Rodrigues, R. L. F. Cunha, and R. Buyya. HPC Cloud for Scientific and Business Applications: Taxonomy, Vision, and Research Challenges. *ACM Comput. Surv.*, 51 (1), jan 2018. ISSN 0360-0300. doi: 10.1145/3150224.

[14] Open Containers Initiative (OCI). Open Container Initiative Runtime Specification, 2023. URL `https://specs.opencontainers.org/runtime-spec/?v=v1.0.2`.

[15] M. Parashar, M. AbdelBaky, I. Rodero, and A. Devarakonda. Cloud Paradigms and Practices for Computational and Data-Enabled Science and Engineering. *Computing in Science & Engineering*, 15(4):10–18, 2013. doi: 10.1109/MCSE.2013.49.

[16] C. Puliafito, C. Vallati, E. Mingozzi, G. Merlino, F. Longo, and A. Puliafito. Container Migration in the Fog: A Performance Evaluation. *Sensors*, 19(7), 2019. ISSN 1424-8220. doi: 10.3390/s19071488.

[17] M. Raynal. *Distributed Algorithms for Message-Passing Systems*. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-38123-2. doi: 10.1007/978-3-642-38123-2.

[18] A. Reber. Container Live Migration Using runC and CRIU, 2016. URL `https://www.redhat.com/en/blog/container-live-migration-using-runc-and-criu`.

[19] A. Reber. Container Migration Around The World, 2017. URL `https://www.redhat.com/en/blog/container-migration-around-world`.

[20] M. Sargent, J. Chu, D. V. Paxson, and M. Allman. Computing TCP's Retransmission Timer. RFC 6298, June 2011. URL `https://www.rfc-editor.org/info/rfc6298`.

[21] M. Schulz. *Checkpointing*, pages 264–273. Springer US, Boston, MA, 2011. ISBN 978-0-387-09766-4. doi: 10.1007/978-0-387-09766-4_62.

[22] C. Segarra. Transparent Live Migration of Container Deployments in Userspace. Master's thesis, Master of Science in Advanced Mathematics and Mathematical Engineering, Universitat Politecnica de Catalunya, 2020.

[23] M. Sindi and J. R. Williams. Using Container Migration for HPC Workloads Resilience. In *Proc. of the 2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–10, 2019. doi: 10.1109/HPEC.2019.8916436.

[24] P. Snyder. tmpfs: A virtual memory file system. In *Proc. of the autumn 1990 EUUG Conference*, pages 241–248, 1990.

[25] M. Terneborg. Enabling container failover by extending current container migration techniques. Master's thesis, Master of Computer Science and Engineering, Luleå University of Technology, 2021.