TelePod: Live Migration for Stateful Containers

Mingjie Yan, Atharva Ranade, Xin Zhang, Kartik Gopalan
{myan28, aranade, xzhang99, kartik}@binghamton.edu
School of Computing, Binghamton University, Binghamton, NY, USA

***Abstract*—Achieving "true" live migration for stateful containers, with low service interruption for workloads, has been difficult due to memory and state transfer overheads. Numerous container applications are engineered to be entirely stateless to avoid the storage or migration of state, leading to a relative paucity of attention paid to stateful live container migration techniques. However, while stateless containers can simply be destroyed and restarted on new hardware without consequence, stateful container workloads – such as active databases or multi-player game servers – require live migration to preserve volatile in-memory data and active network connections that would be lost during a cold restart. This paper introduces TelePod – a new stateful container migration technique that combines pre-copy iterative memory transfer and an overlapped transmission-restoration process to significantly reduce workload interruption and increase liveness during migration. Allowing the destination node to begin restoring state while the source is still transmitting dirty pages effectively decouples total migration time and downtime. Our prototype of TelePod, based on modified Checkpoint/Restore In Userspace (CRIU), achieves a live migration performance comparable to established hypervisor-based VM migration with the same workload, for both traditional and confidential VMs, making it suitable for migrating stateful containerized applications.**

***Index Terms*—Containers, Live migration, Virtualization.**

I. Introduction

Containers are a popular mechanism to quickly deploy cloud applications. Cloud native applications often use stateless design in which disposable containers do not have any internal state. Although, for a wide range of web services and microservices, this stateless approach is attractive, it cannot be applied to a broader class of mission-critical or time-sensitive workloads [1], [2] where the stateless assumption breaks down. Many common applications, including database servers, game servers, and new workloads such as Large Language Model (LLM) inference and edge computing [3]–[6], rely on complex memory states. For these stateful applications, restarting from scratch is computationally expensive and costly in time and resources. Initializing an LLM or a high-performance database can take a long time [6], [7]. Time sensitive services cannot tolerate the delay of a cold start. In addition, edge computing scenarios, such as 5G base stations [8]–[11], require extremely low latency. One challenge of deployment is ensuring that users have the same experience when they move between different locations.

This makes it important to have a robust mechanism to move stateful containers from one host to another without losing their active memory state and network connections. The commonly used Checkpoint/Restore In Userspace (CRIU) [12] tool relies on a *stop-and-copy* mechanism to migrate container state. In this mode, the application freezes for the entire time that the data is being transferred. This *downtime* grows linearly with the container's memory size. For large applications, this freeze can last tens of seconds or more, which is unacceptable for workloads that need to remain responsive.

When it comes to migrating virtualized computing resources, Virtual Machines (VMs) live migration is more widely accepted for its performance and reliability. In cloud environments, it is also common to run containers within VMs and then migrate the entire VM to achieve stateful container migration. However, live migration of VMs is a coarse-grained solution that transfers the entire guest OS and all its applications as a black box, which may not always be desirable. Another category includes emerging Confidential Virtual Machine (CVM) environments, where the hypervisor cannot directly access the guest OS memory in the CVM. Their reliance on a secure firmware results in existing CVM live migration being slow with long downtimes. In both scenarios, containerizing an application running in a VM and then only live migrating the container to the destination avoids various limitations of live migrating a full VM.

To improve the container migration toolkit to match the efficiency of migrating a VM, this paper proposes *TelePod* a container live migration system. TelePod implements a true pre-copy live migration approach for containers on a combination of CRIU, Podman [13], and runc [14] platforms. Our approach iteratively copies memory pages to the destination while the container continues to run at the source, and freezes it only for a brief final state synchronization. We aim to reduce both total migration time (TMT) and downtime with a target performance of live VM migration approaches. The main contributions of this paper are as follows:

- We identify the primary causes of excessive TMT and downtime in existing CRIU live migration characterized by linear blocking dependencies between components.
- We present TelePod, a pre-copy-based live container migration approach that lowers TMT and downtime via parallel page transmission and processing.
- We demonstrate that our approach outperforms existing container live migration solutions and achieves performance comparable to QEMU-based VM live migration for several real-world applications.

II. Background and Related Work

A. Live Migration

Live migration is the process of moving a running application or an entire guest OS from one host to another

with minimal disruption. Live migration efficiency is typically evaluated using two metrics: TMT and downtime. TMT is the duration from the start of the migration to the resumption at the destination. Downtime is the period during which the service is unavailable to users when syncing the final state.

Two primary live migration techniques are *pre-copy* [15] and *post-copy* [16]. Pre-copy iteratively transfers memory while the application runs, tracking and re-sending dirty pages until a final downtime and handoff to the destination. Post-copy suspends the application immediately, moving only the CPU state before fetching memory on-demand at the destination and pushing memory actively from the source. Pre-copy works well for read-intensive workloads, whereas post-copy works well for write-intensive workloads. TelePod focuses on implementing pre-copy approach.

B. CRIU based Container Migration

CRIU is a popular tool to checkpoint running containers or applications, capturing their exact state at a given moment and saving it. This checkpointed data can later be used to restore the application or container, allowing it to resume from the point where it was checkpointed.

1) CRIU dump: A CRIU dump (checkpoint) of a process begins by freezing the target process tree to ensure a consistent state. CRIU uses the `ptrace` system call to take control of the target process. Once attached and frozen, CRIU injects a "parasite code" into the target's address space. The parasite runs in the context of the target process, enabling CRIU to bypass kernel access restrictions, and transfer memory contents and register states directly into kernel pipes. CRIU then reads the collected data from kernel pipes, serializes it, and writes it to image files using protocol buffers.

2) CRIU pre-dump: CRIU's pre-dump reduces downtime by iteratively extracting memory while the application runs, similar to VM pre-copy migration. It uses Linux kernel's soft-dirty bit [17] to track changes, performing brief freezes to inject a parasite code and drain dirty pages. Although it minimizes service interruption, a final dump is necessary to capture the complete state.

3) CRIU restore: CRIU restore begins by reading the image files and forking new processes to match the original process hierarchy. To restore memory, CRIU copies the dumped memory pages from the image file to the restorer process' address space, and restores resources such as file descriptors, IPC objects, signals, timers, and network connections. Then it injects a parasite code as the restorer for final restoration. The restorer also runs inside the restorer process's address space and performs tasks that cannot be handled externally, such as remapping memory pages to their original virtual addresses, restoring CPU registers, and reconstructing shared resources. Finally, the parasite code transforms the restorer process into the original application and resumes execution at the exact point at which it was frozen at the source.

4) CRIU migration: A pre-dump enabled CRIU migration workflow works as follows: ① The process is paused for a short time. ② A parasite code injected into the process. ③ Application keeps running while CRIU serializes dumped memory pages and writes them to disk. ④ Steps 1-3 are repeated. ⑤ After certain iterations, the final dump starts, which freezes the application for the downtime. ⑥ Pre-dump files and final dump files are sent to destination. ⑦ CRIU restores the process from received image files at destination.

CRIU is designed to operate statelessly. Each operation performs only one action, whether dumping, restoring, performing a pre-dump, or verifying system support and compatibility. Once an operation is complete, CRIU exits without retaining any intermediate state records. Consequently, the aforementioned pre-dump-enabled CRIU migration workflow requires the end user to issue commands multiple times, manually or through pipelined scripts. Although many container migration projects based on CRIU claim to achieve live migration, their actual implementations cannot be considered truly live but closer to stop-and-copy, compared to live VM migration.

C. Podman and `runc`

To implement TelePod, we also use an OCI compatible stack, specifically Podman and *runc*. Podman is a daemonless container engine that manages containers, networking, storage, and images. Unlike Docker, which relies on a central daemon, Podman uses a fork-exec model to achieve daemonless operation. This provides TelePod more flexibility to interact with the underlying runc, the file system layers, and networking during live migration. The `runc` command line interface tool is used for spawning and running containers on Linux that follow the OCI specification. It directly configures Linux kernel features, such as cgroups and namespaces, to provide container isolation. Crucially, `runc` integrates directly with CRIU to facilitate the checkpoint and restore commands.

D. Confidential Virtual Machine (CVM) Live Migration

CVM is a VM-level Trusted Execution Environment (TEE) that protects data in use by transparently encrypting the VM's memory contents. While we consider the AMD SEV [18] platform for CVMs, TelePod is not hardware dependent and can work in any other TEE, such as Intel TDX [19]. Each CVM receives a unique encryption key at boot time, which is inaccessible to the hypervisor. These keys are managed by a Secure Processor (AMD-SP) that serves as a hardware root of trust. Programs running within CVM experience a slight decrease in computational performance, but suffer a notable reduction in I/O performance [20]. The current publicly available CVM live migration approach [21] relies on the hypervisor to transfer CVM's memory contents. To maintain confidentiality, the VM memory is packaged via the AMD-SP, limiting the migration throughput to about 350-450 pages/second [22]. The key problem in this approach is that the CVM migration agent is outside the trusted region. With a container migration approach, TelePod, runs inside the CVM and is also protected by the CVM in the same manner as any other application.

E. Other Container Migration Approaches

PCLive [23] proposed a pipelined restoration system for application containers to achieve a lower downtime, but is limited

to one application at the CRIU level and does not support containers based on the OCI standard. mWarp [24] reduces inter-VM container migration downtime by transferring ownership of pages between VMs residing on the same host. Voyager [25] focuses on persistent storage migration and synchronization. Li et al. [26] and MigrOS [27] both proposed a CRIU based migration system for RDMA application containers. Ma et al. [28] and Sledge [29] focus on end-to-end image migration for containers. Silva et al. [30] use CRIU to address cold start problem for stateless containers. Benjaponpitak et al. [31] leverage NFS and VPN tunnels to enable container migration across multiple cloud vendors. Poggiani et al. [11] migrate both single-container and multi-container instances between different Kubernetes clusters. Techniques developed in TelePod can benefit the above approaches as well.

III. Design

The primary design goal of TelePod is to minimize TMT and downtime of a running container during migration. This requires not only live migrating the active application within the container but also preserving the container environment exactly as configured on the source side. To achieve this, we split the responsibility between the Data Plane (application memory, CPU state, etc.) and the Control Plane (container configuration, namespaces, etc.). TelePod ensures that the source and destination actively "sync" both the Data and Control Planes throughout a live migration event. Figure 1 shows the comparison in timeline of the default Podman container migration and TelePod.

A. Control Plane state migration

The Control Plane includes a container's configuration, namespace settings, network back-end settings, mounted file systems, and open file descriptors. In short, we refer to this collection of information as "container metadata". We use Podman as the Control Plane controller and runc as the underlying engine to manage these components.

Traditionally, the source side checkpoints the container by building and packaging all data from the Control and Data Planes into a single tarball. Then, this package is transferred to the destination machine via third-party tools, such as scp or rsync. The destination side then unpacks the tarball and copies all its contents to a location used to store the container's data. After that, runc reads all container metadata and prepares the namespace, mounted file systems, and other settings, replicating the container environment exactly as it was on the source side. Once runc is ready, it uses Remote Procedure Call (RPC) to start CRIU and perform a restoration from the received Data Plane state.

In contrast to the traditional, centralized package-and-transfer approach, TelePod implements a direct, TCP-based streaming mechanism within Podman, effectively eliminating the dependency on intermediate tarballs. This enables the source to stream container metadata directly to the destination without the overhead of intermediate storage or waiting for Data Plane to be fully consolidated. With this setup, the destination can perform a partial restore as soon as it receives the container metadata. Thus, destination Podman can start rebuilding the container environment at the very beginning of a migration event, in parallel with the source's Data Plane checkpointing. This effectively prepares the destination side system for the subsequent restoration step without delay.

B. Data Plane at Source Side

The Data Plane consists of runtime execution state of the containerized application, including the memory, CPU state, network sockets, open file descriptors, etc. We modified CRIU to collect and transfer these data to the destination side in a way that supports TelePod at both sides.

Conventionally, a container checkpoint uses CRIU to save a single full dump to disk, which is later packed into a tarball with Control Plane data. This means that once the migration starts, the container is no longer able to respond until restoration is complete on the destination side. As a result, the downtime of this container stop-and-copy migration is substantial. It includes the time spent on CRIU checkpoint, Podman container metadata dump, tarball packaging, data transfer, tarball unpacking, environment setup and CRIU restoration. In practice, this entire process takes tens of seconds at best, and often up to a couple of minutes, depending on the workload.

The workflow described above is a strictly sequential procedure. Although CRIU has the pre-dump capability, which is analogous to the pre-copy migration in VM, most container Control Plane tools cannot fully leverage this feature, primarily because none of the existing tools have been able to resolve the transmission issues associated with the generated iterated image files. Which means, pre-dumped data still sit on source side. As for our baseline Podman, it can only use up to one iterative pre-dump for restoration, lacking the logic to handle them during a live transfer. In contrast, TelePod overlaps steps wherever possible, turning the sequential workflow into a highly parallelized process.

On the TelePod source side, Podman first transfers the container metadata to destination, as described in the previous section. It then launches runc to prepare the required parameters and environment for the subsequent CRIU checkpoint process. Once CRIU begins execution and detects a container live migration job initiated by runc, it launches its built-in page-server to transfer memory pages to the destination. We modified the page-server logic to coordinate with the page receiver (one of the components of TelePod on the destination side) waiting at the destination.

Once the CRIU page-server and page receiver establish a connection, CRIU immediately initiates the first round of pre-dump. During all processes inside the container, it reads memory contents while simultaneously transfers them to the destination. As this is the initial round, all memory pages should be transferred. Next, a built-in decision-maker decides whether to start the next pre-dump round or enter downtime phase for the final dump based on several parameters.

From this point onward, CRIU operates in a stateful manner rather than its default stateless mode. Upon completing a pre-

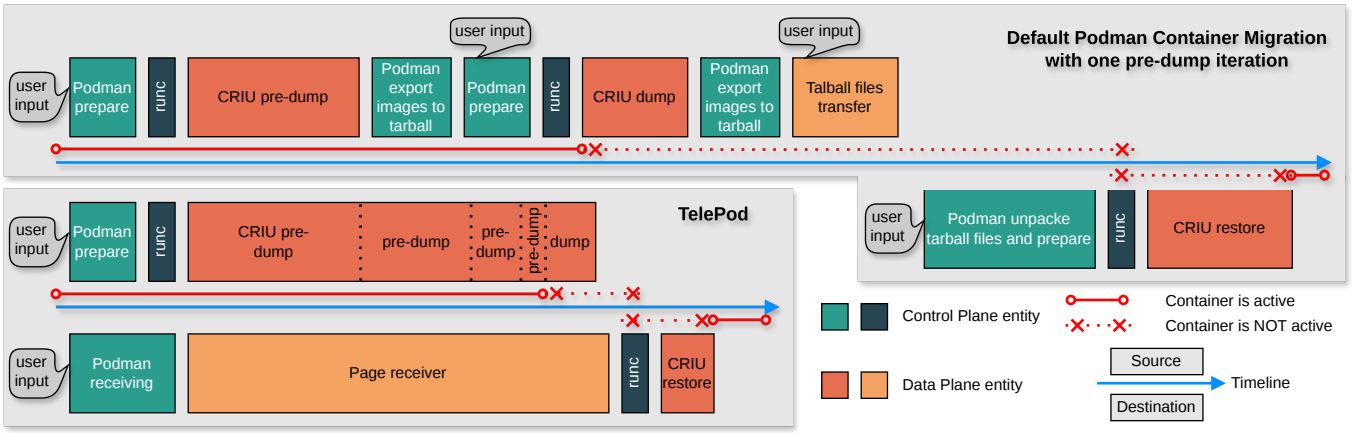

Fig. 1. Podman container migration VS TelePod in the timeline view. The Podman solution requires user to continuously issue commands to complete the container migration process. It also involves extensive repetitive disk I/O and network transfer. Furthermore, restoration at the destination must wait until all files arrive. TelePod once initiated by user, it will proceed as a parallel, state-synchronized transmission and processing between the source and destination.

dump round, CRIU does not exit. Instead, it records per-round statistics, including the number of pages transferred, the transfer bandwidth, the time consumed during this round, and the brief application pause time at the start of the round. If the decision-maker determines, based on the performance metrics of the recently completed pre-dump round, that entering downtime at this point would result in excessive downtime, it would violate the original design intent of TelePod. Therefore, it initiates another pre-dump round, hoping that fewer pages will be dirtied and need to be transferred in the next iteration. Alternatively, if the most recent pre-copy iteration completes quickly, this indicates that only a small number of pages are dirtied and the system can safely enter the downtime phase.

Upon entering downtime, CRIU freezes the processes inside the container, captures the process tree, and collects the namespaces. Next, CRIU dumps the CPU states, memory pages and shared memory, and transfers these dumped data directly to the destination side page receiver. Upon completing the memory dump, CRIU sends a message to the destination page receiver indicating that the memory phase is finished. It then finalizes the process by dumping namespaces, cgroups, network sockets, file locks and file descriptors, writing these states into image files and transferring them to destination page receiver. These image files are typically small, requiring minimal time to dump and transfer. Meanwhile, the destination works concurrently to prepare for the final restoration.

C. Destination side page receiver and CRIU

In the default CRIU behavior, performing multiple pre-dumps and one final dump creates a chain of image directories (e.g., images/0/, images/1/, ... images/N/) corresponding to successive iterations of memory dumping. In this case, the image dump from iteration N is incremental and based on the image captured from iteration N-1. This dependency ensures efficient bandwidth usage by transferring only the dirty pages since the previous round. However, this design forces the restore process to traverse all the chained image files. If the number of pre-dump iterations is large, reconstructing the final overall image by walking the chain can take a long time. Furthermore, if the image files are on a disk or NFS shared mount, the file read latency will incur significant cost. More importantly, the aforementioned process of reconstructing the image occurs entirely within the downtime, and the actual memory recovery steps cannot formally commence until the reconstructed image is fully prepared. The developers of CRIU are well aware of these circumstances. Their documentation recommends using a diskless [32] storage method to temporarily store received image files. This primarily involves setting up a `ramfs` at the destination to achieve fast write and read operations, thereby avoiding disk I/O bottlenecks.

When sending pages, CRIU aggregates contiguous pages from the same virtual memory area (VMA) and sends them together. As TelePod aims to pipeline destination workload with source side to make the entire migration process parallel, we build a dedicated page receiver, whose primary function is to integrate with the source CRIU workflow to achieve the parallel page receiving. Apart from simply receiving pages in the order defined by the image chain mentioned earlier, our page receiver has another major task, which is to consolidate the page content from multiple iterations, along with their corresponding pagemap information, into a single consolidated page content image file (and its pagemap image file) on the fly. In this way, we avoid traversing chained image files during CRIU restoration because our page receiver exposes the received state as a single, consolidated, full-dump image set, even though the source transmits it across multiple iterations.

Resolving the issue of excessive downtime caused by having to read all chained file structures and reconstruct usable data before restoring memory pages is not the end goal. If we adopt the CRIU developers' recommended method for temporarily storing received content for a quicker restoration, this will introduce one major drawback: it reduces the amount of available system memory. If the memory footprint of the program to be restored exceeds half of the system's allocatable memory, CRIU restoration may fail due to insufficient memory. This is because the images temporarily staged in a ramfs can consume

more memory than the program's memory footprint, especially when multiple pre-dump iterations generate additional image data. So in order to restore the process in a quick way, the system available memory must be greater than 2x of the process real memory usage. This generally will not be acceptable for many cloud environments and edge devices. For efficiency and cost reasons, there is often insufficient memory available at the migration destination to temporarily store these image files. Although the CRIU team proposed memory image deduplication [33] to solve this problem, the image file in the ramfs is still organized in a chained format, meaning it still needs extra time to reconstruct the final restore page image. To address this issue in TelePod, the receiver first stores pages in a shared memory object `/dev/shm/` that sits in memory. Then, starting from the 2nd iteration, any page content that sits in the same VMA that we have already received in the previous iteration, we overwrite it in the same location instead of simply appending to the end. Thus, even under write intensive workloads, the consolidated image size remains close to the source process's memory footprint.

There are two major steps in a CRIU restore. First, CRIU loads the memory from the image file into a temporary memory region within the target process's address space. Second, the "parasite code" restorer will remap those memory pages to the same location as the original virtual address.

Since the page content is already in a shared memory object in TelePod design, the first step can be safely skipped. Therefore, we modify the restorer to give it the capability to directly remap pages from the shared memory, written by the page receiver, to the corresponding original virtual address.

IV. Implementation

A. Container metadata dump and transfer

By default, Podman is stateless, which forces the destination Podman to wait until all data transfers are completed before starting the container restoration process. Without the container metadata at the destination, it is impossible to pre-assemble the container skeleton (e.g. the container base image, container storage and volumes, network settings, ID, and startup parameters) needed for a restoration. In TelePod, the destination Podman initially only reads the container ID, the base image, and the startup parameters at the beginning of data exchange. These fields typically remain unchanged and are sufficient to construct the skeleton of the container to be restored. The destination can then safely launch the page receiver to receive pages at the same time. During the source side downtime phase, we perform a second container metadata checkpoint and send it as the final container metadata. The destination applies this final metadata to configure remaining settings before invoking runc to complete restoration.

B. Dirty page tracking and handling

As discussed in Section II-B2, CRIU's pre-dump fully relies on the kernel's soft-dirty bit tracking to perform an incremental dump, saving only the memory pages that have changed since the last iteration. With the corresponding kernel configuration, Linux kernel maintains a soft-dirty bit for each page table entry. During the freeze time, CRIU inspects the container's process memory map from `/proc/<pid>/pagemap`, where the soft-dirty bit information is stored, to determine which pages are dirty. After obtaining the list of all dirty pages, we modified CRIU's behavior so that, instead of performing additional inspections, it immediately drains those pages marked dirty to the kernel pipes, resets the kernel's soft-dirty bit for the next iteration, and resumes the process if still in a pre-dump.

C. Downtime decision maker

Algorithm 1 Downtime Decision Logic

Require: P_{dumped} (Pages dumped), BW (Bandwidth), T_{last} (Last iteration time), T_{frozen} (Frozen time), k (Iteration count)

$T_{threshold} \leftarrow 500ms$ // Base threshold
if $T_{last} < T_{threshold}$ **then**
  return downtime
else if $T_{frozen} > 1000ms$ **and** $k \neq 0$ **then**
  return downtime
else if $BW < 1/10BW_{system}$ **then**
  return downtime
else if $k >= 10$ **then**
  $N \leftarrow \lfloor k/10 \rfloor$
  $T_{current} \leftarrow T_{threshold} + (N \times 100)$
  if $T_{last} < T_{current}$ **then**
    return downtime
  else
    return pre-dump
  end if
else
  return pre-dump
end if

This is a simple and intuitive logical decision step that runs after each pre-dump round. Its primary function is to regulate TMT and downtime during live migration. In certain scenarios, earlier initiation of downtime can result in a substantial reduction in TMT. Conversely, in other scenarios, the execution of an additional pre-dump round can lead to a reduction in the final downtime. Algorithm 1 shows the entire logic of this decision making process. At each round, the system needs to determine whether to start the final downtime as soon as possible, since the container continues to run and may dirty more pages than expected. Each pre-dump reports the current pre-dump iteration count, the number of pages dumped during the current iteration, the page-server transfer bandwidth, the time for last iteration, and the short downtime at the beginning of the pre-dump.

The short frozen time is usually proportional to the container's memory usage. If any iteration (excluding the very first iteration, which dumps the entire memory) takes more than one second, it indicates either an extremely write-intensive workload or significant system pressure on the source side. This is because the injected parasite code drains the pages marked as dirty to the kernel pipes during this frozen time. In such cases, to balance the TMT and downtime, we need to start the downtime phase immediately; otherwise, additional pre-dump rounds will fail to reduce the downtime while unnecessarily increasing TMT.

Under normal circumstances, the CRIU page-server at the source and the page receiver at the destination can typically use the full available network bandwidth of the system. However, after the short frozen time, the container resumes execution and may start consuming network bandwidth. At the same time, the CRIU page-server continues transmitting pages data, competing with the container for available bandwidth. If the actual bandwidth consumed by CRIU drops below 10% of the system's maximum available bandwidth, indicating that the container is consuming roughly 90%, we initiate downtime immediately. This is because high network bandwidth usage often correlates with a high memory dirty rate, reducing the effectiveness of additional pre-dump rounds.

In addition, a third scenario arises when the application inside the container continuously dirties memory, causing each pre-dump iteration to exceed the default threshold. Thus, CRIU may never initiate the downtime needed to complete the migration. To address this, we implemented a dynamic threshold adjustment mechanism. After every ten pre-dump iterations, the default threshold is increased by 100 ms to counter the application memory aggressiveness. This keeps the downtime as low as possible while ensuring the migration eventually completes and preventing excessively long TMT.

D. The Parasite Code restorer

Depending on the image files given to CRIU, the default CRIU restore logic operates in one of two modes. In the first mode, CRIU is provided with multiple pre-dumps and one final dump (described in Section III-C), forming a chain of iteration image files. CRIU first allocates a temporary memory region in the target process's address space that matches the size of the target process's memory usage. It then reads the pagemap file generated during the final dump stage and checks the flags of each VMA region. If a page is marked as `PARENT`, it indicates that the page was transferred in previous iterations. Then, CRIU recursively traverses the pagemap files from previous iteration(s) until it finds the page marked `PRESENT` in the most recent ancestor image. CRIU then copies the data to the temporary memory region. Once all pages of a VMA region are resolved, the VMA is marked as `PREMMAPED` and CRIU injects the restorer until all VMAs are marked. At this point, the restorer scans the pagemap again to verify that all VMAs are pre-mapped. If some pages are missing (usually COW and shared pages), the restorer handles them separately. For those that have been pre-mapped, the restorer first creates VMA regions with empty content using `mmap(..., MAP_ANONYMOUS|MAP_FIXED, fd)` at the correct location. Then, it will move from the pre-mapped region to the original virtual addresses using `mremap()`. In the second mode, CRIU is provided with an image file from a single, non-iterated dump. CRIU does not need to traverse chained image files or pre-map VMAs to the target process address space. Instead, it only parses pagemap and passes the file descriptors directly to the restorer. The restorer uses the same `mmap()` logic and flags as in the other mode to create an empty VMA at the correct location. Then, it uses `preadv()` to read directly from the opened image file descriptor into the newly mapped memory at the original virtual addresses. The key differences between these two modes lie largely in CRIU's work before the restorer begins. In the multi-image mode, CRIU must traverse the chained image files to locate page data and copy it to a temporary memory region in the target address space. The restorer then needs to go through them again to move them to the original virtual addresses. In single-image mode, there is only one dump iteration, so the restorer can populate pages directly from the image file to the original virtual addresses without the extra chain traversal.

To minimize downtime, TelePod aligns its restoration logic with the efficient single-image mode described above. Although the source sends the pages incrementally across multiple iterations, the destination page receiver consolidates them into a single-image mode representation backed by a shared memory object. Then, we modify the restoration logic to register the shared memory object's file descriptor as a service fd within the CRIU context. This ensures that the restorer can reference the data consistently without conflicting with the target process's own file descriptors. Unlike the default restorer in single-image mode, which creates anonymous placeholder mappings and subsequently populates them by reading page contents from an image file, TelePod's restorer maps the received pages in the shared memory object directly at their corresponding virtual addresses in the target process. Because the region is file-backed by the shared memory object that is already resident in memory, TelePod avoids loading the page contents from an image file during restoration, thereby reducing the downtime and avoiding memory usage spikes.

E. The Page Receiver

The page receiver on destination ensures that the shared memory object always reflects the latest version of any given page, keeps the shared memory object size as small as possible, and generates final image following the single-dump mode format. For presentation ease, assume that the migrated container contains only a single process. In the context of containers with multiple processes, both TelePod and CRIU maintain their functionality. Each process has its own corresponding pagemap image file, while all page data is stored in the same pages image file. The pagemap image records the offset to the pages image file, reflecting the physical location of the corresponding page's data.

1) Consolidate page data on the fly: The source CRIU continuously sends pages content to the destination receiver, along with (virtual address, length) pair, also known as an I/O vector (iovec). Within a single pre-dump transmission, pages are always sent in ascending order of virtual address. For those regions that remain unchanged in a given pre-dump round, we only receive a special identifier along with the iovec, without the actual page content. Based on each received iovec, the page receiver maintains a per-process tracker that records the mapping between iovec and the corresponding file offset where the page content is stored within the shared memory object. This tracker is an in-memory list of non-overlapping intervals

sorted by virtual address. During the first pre-dump round, the page receiver only appends new intervals to the tracker and writes the corresponding pages to the end of shared memory object. Starting from the second round, when a new iovec arrives, the page receiver performs the following steps.

①Overlap detection: Checks whether the new iovec overlaps with any existing intervals in the tracker. Overlaps are expected, since live migration proceeds in multiple iterations.

②Write offset calculation: If the new iovec fits entirely within, or exactly matches an existing interval's address range, the page receiver reuses the existing offset that stored in that interval. This is a subset overwrite, and replaces the old data at the existing offset, preventing the shared memory object from unnecessary growth. If the new iovec is entirely new, the page receiver appends this new part to the end of the shared memory object. If the new iovec partially overlaps with existing intervals, the page receiver splits the write operation into distinct parts. It treats the overlapping portion as a subset overwrite and the non-overlapping portions as the "new part".

③Page data write: If there is no partial overlap, the page receiver uses `recv()` to write the page content received from the network socket directly into the shared memory object at the calculated offset. This avoids intermediate user space buffers and reduces the memory footprint. However, partial overlaps cause writes to be fragmented, and the receiver cannot write directly from the socket to the final destination in a single pass, losing the ability to use the zero-copy optimization.

④Tracker update: Once the page content has been written to the correct offset, the page receiver updates the tracker to reflect the new state. Any existing intervals that are fully covered by the new range are removed from the tracker, as their contents are now obsolete. Those that are partially covered are split or trimmed to preserve only the valid, non-overwritten data. The receiver then inserts the new interval into the sorted list and records each iovec together with its corresponding offset in the shared memory object. Maintaining tracker accuracy is essential: it serves as the sole basis for subsequent sorting and de-fragmentation of received page content. Ultimately, this tracker is serialized into the pagemap images, which works as a critical component of CRIU restoration.

2) The final sorting and rewrite step: Once the CRIU source has finished transferring the pages, it sends a special identifier to inform the page receiver that no more pages will arrive. However, the overall dump process is not yet complete; critical non-memory state, such as network device state, TCP connections, namespace entries, signal handling, and other process state, must still be dumped and transferred. At this point, the page receiver starts a couple of background worker threads, while the main thread transitions to receive the remaining image files as they become available. Concurrently, the worker threads perform a final cleanup process to reform the page data as a contiguous and sorted shared memory object, based on the information in the tracker. Because the remaining image file reception on the main thread and the sorting of received pages by worker threads occur in parallel, this further reduces the time spent in the downtime phase. This reordering process is necessary because the page contents in the shared memory object now are likely out of order due to the append writes nature of multiple iterations, even though all the data are valid.

V. Evaluation

To evaluate performance differences across various configuration scenarios, we employed two representative workloads: redis-server (memory-intensive) and llama-cpp-python (large memory footprint, LLM related). We compared the TelePod migration metrics against two baselines: the default Podman container migration (hereinafter Podman migration) and QEMU-based VM live migration (hereinafter VM migration), with the same applications running inside the VM. The migration performance matrices in this section are based on the average of five runs using the same setup.

A. Evaluation environment setup

We deployed three machines with identical configurations to perform all the experiments below. Each machine is equipped with 2×64-core AMD EPYC Milan 7763 processors, 256GB RAM, a 500GB SATA SSD, and a duel port 25Gbps NIC. Two of these three machines serve as the source side and destination side for the migration, while the third machine functions as the destination location. This third machine primarily acts as a third-party client to initiate requests to the application. It runs Redis-benchmark during redis-server testing and serves as the receiver for tokens generated by the LLM inference. This third machine serves another purpose: it acts as a receiver for UDP timestamps, which are used to track migration-related TMT and downtime. Since VM migration only reports TMT and downtime from the source QEMU, our team found through prior research that these reports are inaccurate. Since migration completion occurs at the destination QEMU, both sides must report their status to this third machine. Therefore, we added UDP packet transmission capabilities to QEMU v10.1.3 to achieve more precise reporting of migration metrics. Similarly, we integrated corresponding UDP packet transmission functionality into Podman v5.0.2, CRIU v4.0 and TelePod. The duration of each migration phase is calculated by comparing the timestamp differences of the received UDP packets.

Host running Ubuntu 24.04 and Linux kernel 6.18.0 for non-CVM migration case. In order to finish CVM migration, we have to change the host to Ubuntu 22.04 and Linux kernel 6.3.8. Guest running Ubuntu 22.04 and Linux kernel 5.19.0 with CVM migration related patches enabled.

1) VM configuration: All container experiments run inside a VM, using the same VM image used for VM migration. This ensures that the tested environment remains consistent across all migration techniques. There are two VM sizes for different experiments. The "Large-VM" has 16 cores and 40 GB of memory. The "Small-VM" has 4 cores and 2 GB of memory. All VMs are configured with vhost-net to achieve better network throughput inside the guest. Using iperf, we tested the bandwidth between two VMs running on separate machines and achieved approximately 10 Gbps with a single

thread. This represents the maximum bandwidth for container migration because containers operate within VMs. For VM migration, QEMU can utilize the physical NIC on the host side with a bandwidth of up to 25 Gbps. Therefore, to ensure a fair comparison, we limit the maximum available bandwidth during QEMU migration to align with the container network bandwidth ceiling, with other configures remaining default.

2) Container configuration: The container's IP and MAC address must remain consistent before and after migration. This guaranties that network connections interrupted during the downtime phase will automatically resume once the destination is restored. We have defined a statically IP-assigned network using ipvlan plugin. As for MAC address, Podman has the capability to configure it based on received container metadata. Additionally, we removed the resource restrictions on the container, allowing it to utilize all resources in the VM.

3) Redis-server setup: We use official Redis 7.4.7 and keep the configuration mostly in the default values. The experiment will involve two configurations. The "Redis-Large" configuration was conducted on a Large-VM that was pre-populated with 25 GB of cold data. The "Redis-Small" configuration was conducted on a Small-VM that was pre-populated with 800 MB of cold data. This workload helps us study the effect of different migration methods on memory-intensive workloads. Additionally, we will examine the migration performance differences when the application's memory footprint varies.

4) llama-cpp-python: The LLM application used for testing is simple and intuitive. After loading a specific LLM model, a fixed prompt is provided: "Repeat the following sentence exactly 1000 times: 'The quick brown fox jumps over the lazy dog.' Start now: ". This simplified "repeat phrase" prompt can minimize inference overhead from CPU-bound LLM inference as our primary focus is on migration-related performance.

For this experiment, we also employed two configurations. "Llama" represents our use of the Llama-3-8B-Q4KM model running on a Large-VM, with llama-cpp configured to utilize 12 cores and a larger context window. "Qwen" represents our use of the Qwen-2-1.5B-Q4KM model running on a Small-VM, configured with llama-cpp to utilize 2 cores and a relatively smaller context window. We include the model files into the corresponding container base image to avoid syncing disk and volumes for container cases. This setup helps us study the impact of various migration techniques on memory intensive LLM inference workloads.

B. Total Migration Time (TMT) and downtime

The TMT and downtime performance of various migration techniques for redis-server with different pre-filled sizes is shown in Figure 2. These sizes represent the amount of initial data requiring migration. Here, the third machine keeps running Redis-benchmark in GET command (read intensive) during the entire migration. Although it generates very few dirty pages, it maintains 50 TCP connections and states. As the pre-filled size increases linearly, the TMT and downtime of Podman migration increase in a steep linearly. In contrast, VM migration has nearly constant downtime due to QEMU's

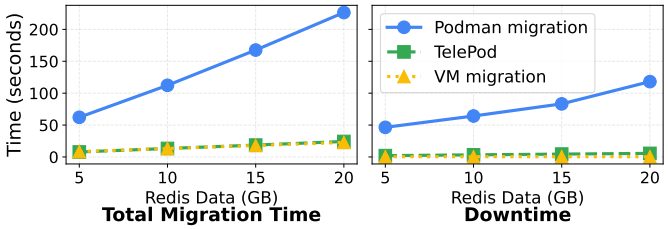

Fig. 2. Performance of various migration techniques in terms of TMT and downtime for redis-server with different pre-filled sizes.

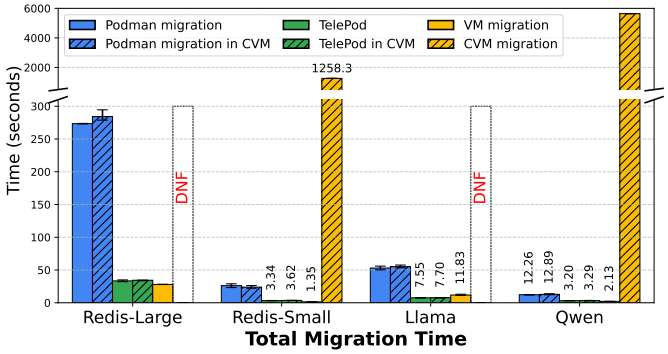

Fig. 3. TMT of various migration techniques while running different types of workloads. DNF refers to "Did Not Finish" for CVM Migration.

default migration algorithm imposing an upper limit on downtime, and there is only a very slight linear increase in TMT. TelePod shows a similar very slight linear increase rate in both TMT and downtime compared to VM live migration. The primary reason for this significant difference in scaling rates is that Podman migration is constrained not only by network bandwidth, but also by disk I/O performance. In contrast, TelePod and VM migration are both limited by network bandwidth. Consequently, TMT exhibits only a very marginal increase. TelePod also must process and transmit additional information required by processes beyond page data during the downtime phase, causing gradual growth for downtime.

Figure 3 and Figure 4 illustrate TMT and downtime for various migration techniques under workloads with higher dirty page rates. For Podman migration, most workloads utilized the one pre-dump followed by a final dump. However, the Redis-Large workload reverted to a stop-and-copy mechanism. This occurred because the initial pre-dump duration was excessive,

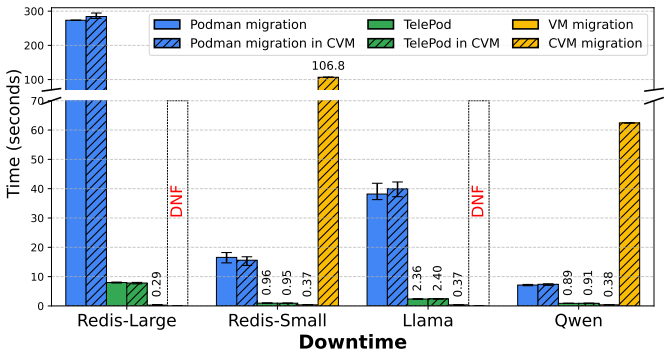

Fig. 4. Downtime of various migration techniques while running different types of workloads. DNF refers to "Did Not Finish" for CVM Migration.

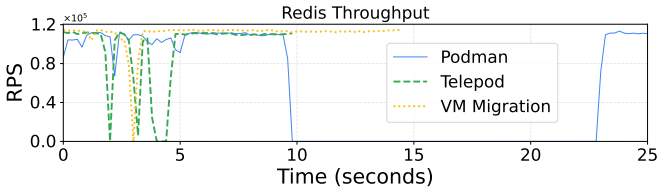

Fig. 5. Redis-benchmark throughput comparison when a migration happens.

allowing the redis-benchmark to generate dirty pages faster than the transfer rate; consequently, the final dump exceeded the original 25 GB pre-filled size. Overall, Podman migration exhibited high TMT and downtime across all scenarios. Even workloads with smaller memory footprints required tens of seconds to migrate, and performance was significantly degraded as memory increased.

The current CVM live migration prototype restricts instances with less than 2.5GB of memory. With a larger memory size, the migration will never finish. Thus, we only present data for the Small-VM related configurations. We can observe a drastic increase in both TMT and downtime. This indicates that migrating an entire CVM is computationally expensive and difficult to converge (due to its low throughput hardware limitation) compared to strictly migrating the internal container, even when utilizing the baseline Podman migration approach. VM migration remains the gold standard for performance. However, because it must transfer the entire guest OS, its transferred data exceed those of container migration. Consequently, TelePod slightly outperformed VM migration in TMT for Llama workload. The performance gap between TelePod and CVM migration highlights the advantages of container migration in confidential environments. CRIU operates within the trusted region of the CVM, TelePod thus avoids the performance overhead imposed by interactions with the AMD-SP. For example, TelePod reduces the TMT of Qwen by more than 99.9% in the case of a CVM. Therefore, TelePod is also a practical solution to the challenges of CVM live migration implementations. Compared to VM migration, TelePod maintains comparable TMT levels but benefits from a smaller amount of data transfer. Moreover, it outperforms Podman migration in all scenarios.

C. Impact on the application

To evaluate the impact of migration techniques on application service quality, we monitored the real-time throughput of a write-intensive Redis-benchmark SET command on a Redis-Small setup. As shown in Figure 5, we synchronized the start of migration for all three techniques. We stopped capturing the data once the migration finished. The benchmark, which runs on the third machine and emulates 50 clients, continuously sends 128-byte length requests to dirty the memory of the redis-server to be migrated. Although Podman migration utilized a strategy of one pre-dump and one final dump to minimize downtime, there still resulted in throughput volatility during the pre-dump phase and a downtime approximately 13 seconds. In contrast, VM migration established the optimal baseline with minimal impact and downtime of about 300ms.

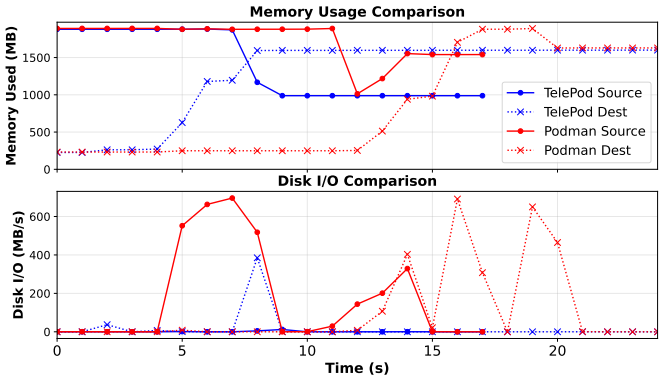

Fig. 6. System memory usage and disk I/O comparison during a migration.

TelePod demonstrated effective multi-iterative live migration. While there were minor fluctuations in throughput (two short dips) during the pre-dump phase, the final downtime was around 900 ms. Through this Redis write throughput comparison, TelePod seems to have a higher impact on application compared to VM migration. However, it is worth noting that the hypervisor can utilize the resources from host side, and in fact, during this experiment, we noticed that VM migration needs three iterations of pre-copy to finish the migration under write-intensive workload. Thus, TelePod maintains much better service availability than Podman migration and is comparable to VM migration techniques.

D. System pressure during migration

To analyze and compare system-level resource pressure, specifically memory usage and disk I/O between TelePod and Podman migration during the migration of Qwen workload in the Small-VM configuration, Figure 6 shows the direct comparison in the real-time resource report using `dstat` tool. Memory usage is calculated by $Memory_{system-max} - Memory_{free}$, which not only includes application memory usage but also includes page cache and system buffer to speed up disk performance and network performance. Disk I/O is the combined value of reads and writes.

Podman migration initially exhibits a rapid surge in disk I/O between 5s-9s, indicating that a pre-dump operation is writing memory pages to the local disk. Following completion of the final dump at 12s, memory usage at the source initially drops but then rebounds. This fluctuation is attributed to the system caching the dumped image files during network transfer. By 15s the destination node has finished receiving the tarball, which is followed immediately by two distinct disk I/O spikes. These spikes correspond to Podman unpacking the tarball to the working directory and then initiating CRIU via RPC to read the image files. Restoration concludes around 21s. The stepwise increase in memory usage at destination reflects the behavior of the system page cache, which is released once the restoration process is complete. This monitoring of memory usage and disk I/O further highlights Podman migration's primary issue: executing operations sequentially forces each step to wait for the previous one to complete. Moreover,

the design incorporates excessive redundant disk read/write operations, resulting in extremely low efficiency.

TelePod starts live migration around 5s, engages the Control Plane to sync the container metadata and Data Plane for memory transfer. Since received pages are written directly to a shared memory object, no disk I/O occurs during this transmission phase. By around 9s, the restoration is finished and the container is resumed. The disk I/O at 8s is the read of container base image and is cached in memory. In particular, the memory usage at the destination does not exhibit spikes during migration, as the memory footprint is progressively populated during the synchronized transfer process. On the source side, the system immediately releases the memory occupied by the container once the handover is finalized.

TelePod avoids the heavy I/O penalty of Podman migration by performing a direct memory-to-memory transfer. Podman migration relies on writing checkpoints to disk, which delays migration and saturates disk I/O, negatively impacting other processes. Hence, TelePod is more efficient in I/O-sensitive and memory limited environments.

VI. Conclusion

We presented TelePod, a stateful container live migration approach to overcome the limitations of traditional sequential, stop-and-copy container migration. By addressing blocking dependencies in existing solution, TelePod achieves true live migration for stateful applications such as active databases and LLM inference engines. Experimental results demonstrate that TelePod significantly outperforms existing container migration tools, delivering performance comparable to established live VM migration techniques. Furthermore, TelePod enables live migration of OCI standard containers with minimal interruption to application service availability for both normal VM and CVM setups.

Acknowledgment

Results in this paper were obtained using Chameleon [34] testbed supported by the US National Science Foundation.

References

[1] J. V. Pandit, "What Workloads Do Customers Run on Kubernetes?" https://www.redhat.com/en/blog/what-workloads-do-customers-run-on-kubernetes, 2021.

[2] L. Abdollahi Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Microservice based architecture: Towards high-availability for stateful applications with kubernetes," in *IEEE Intl. Conf. on Software Quality, Reliability and Security*, 2019.

[3] Y. Fu, L. Xue, Y. Huang, A.-O. Brabete, D. Ustiugov, Y. Patel, and L. Mai, "ServerlessLLM: Low-Latency serverless inference for large language models," in *USENIX OSDI*, 2024.

[4] L. Yu, J. Lin, and J. Li, "Stateful large language model serving with pensieve," in *Eurosys*, 2025.

[5] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *SOSP*, 2023.

[6] Datadog, "10 insights on real-world container use," https://www.datadoghq.com/container-report/, 2023.

[7] L. Ma, S. Yi, N. Carter, and Q. Li, "Efficient live migration of edge services leveraging container layered storage," *IEEE Transactions on Mobile Computing*, vol. 18, no. 9, pp. 2020–2033, 2019.

[8] A. Machen, S. Wang, K. K. Leung, B. J. Ko, and T. Salonidis, "Live service migration in mobile edge clouds," *IEEE Wireless Communications*, vol. 25, no. 1, pp. 140–147, 2018.

[9] S. Ramanathan and et. al., "Live migration of virtual machine and container based mobile core network components: A comprehensive study," *IEEE Access*, vol. 9, pp. 105 082–105 100, 2021.

[10] R. A. Addad, D. L. C. Dutra, M. Bagaa, T. Taleb, and H. Flinck, "Fast service migration in 5g trends and scenarios," *IEEE Network*, vol. 34, no. 2, pp. 92–98, 2020.

[11] L. Poggiani, C. Puliafito, A. Virdis, and E. Mingozzi, "Live migration of multi-container kubernetes pods in multi-cluster serverless edge systems," in *Workshop on Serverless at the Edge*, 2024.

[12] J. Corbet, "Checkpoint/restart (mostly) in user space," https://lwn.net/Articles/452184/, 2011.

[13] P. C. Tools, "Podman: A tool for managing OCI containers and pods," https://github.com/containers/podman/, 2025.

[14] O. C. Initiative, "runc," https://github.com/opencontainers/runc, 2025.

[15] C. Clark and et. al., "Live migration of virtual machines," in *NSDI*, 2005.

[16] M. R. Hines, U. Deshpande, and K. Gopalan, "Post-copy live migration of virtual machines," *SIGOPS Operating Systems Review*, vol. 43, no. 3, p. 14–26, Jul. 2009.

[17] Linux Kernel Driver DataBase, "CONFIG_MEM_SOFT_DIRTY: Track memory changes," https://cateee.net/lkddb/web-lkddb/MEM_SOFT_DIRTY.html, 2025.

[18] D. Kaplan, J. Powell, and T. Woller, "AMD Memory Encryption," https://www.amd.com/system/files/TechDocs/memory-encryption-white-paper.pdf, 2021.

[19] Intel, "Intel® Trust Domain Extensions," https://www.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-v4.pdf, 2023.

[20] M. Yan and K. Gopalan, "Performance overheads of confidential virtual machines," in *Proc. of MASCOTS, Stony Brook, NY, USA*, oct 2023.

[21] A. Kalra, "Live migration of confidential guests," https://lpc.events/event/11/contributions/958/, 2021.

[22] ——, "[PATCH v4 13/14] migration: for SEV live migration bump downtime limit to 1s," https://www.mail-archive.com/qemu-devel@nongnu.org/msg828088.html, 2021.

[23] S. B. Tripathi and D. Mishra, "PCLive: pipelined restoration of application containers for reduced service downtime," in *Proceedings of ACM Symposium on Cloud Computing*, 2024, p. 284–301.

[24] P. K. Sinha, S. S. Doddamani, H. Lu, and K. Gopalan, "mwarp: Accelerating intra-host live container migration via memory warping," in *IEEE INFOCOM 2019 Workshops*, 2019, pp. 508–513.

[25] S. Nadgowda, S. Suneja, N. Bila, and C. Isci, "Voyager: Complete container state migration," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017, pp. 2137–2142.

[26] X. Li, R. Shu, Y. Xiong, and F. Ren, "Software-based live migration for containerized RDMA," in *Asia-Pacific Workshop on Networking*, 2024.

[27] M. Planeta, J. Bierbaum, L. S. D. Antony, T. Hoefler, and H. Härtig, "MigrOS: Transparent Live-Migration support for containerised RDMA applications," in *USENIX ATC*, Jul. 2021.

[28] L. Ma, S. Yi, N. Carter, and Q. Li, "Efficient live migration of edge services leveraging container layered storage," *IEEE Transactions on Mobile Computing*, vol. 18, no. 9, pp. 2020–2033, 2019.

[29] B. Xu, S. Wu, J. Xiao, H. Jin, Y. Zhang, G. Shi, T. Lin, J. Rao, L. Yi, and J. Jiang, "Sledge: Towards efficient live migration of docker containers," in *IEEE CLOUD*, 2020.

[30] P. Silva, D. Fireman, and T. E. Pereira, "Prebaking functions to warm the serverless cold start," in *Middlewaere*, 2020.

[31] T. Benjaponpitak, M. Karakate, and K. Sripanidkulchai, "Enabling live migration of containerized applications across clouds," in *IEEE INFOCOM*, 2020.

[32] CRIU, "Disk-less migration," https://criu.org/Disk-less_migration, 2019.

[33] ——, "Memory images deduplication," https://criu.org/Memory_images_deduplication, 2016.

[34] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, J. Mambretti, A. Barnes, F. Halbach, A. Rocha, and J. Stubbs, "Lessons learned from the Chameleon testbed," in *Proc. of USENIX Annual Technical Conference (ATC)*, July 2020.