Efficient Live Migration of Linux Containers

Radostin Stoyanov and Martin J. Kollingbaum(✉)

University of Aberdeen, Aberdeen, UK
r.stoyanov.14@aberdeen.ac.uk, m.j.kollingbaum@abdn.ac.uk

Abstract. In recent years, operating system level virtualization has grown in popularity due to its capability to isolate multiple userspace environments and to allow for their co-existence within a single OS kernel instance. Checkpoint-restore in Userspace (CRIU) is a tool that allows to live migrate a hierarchy of processes – a container – between two physical computers. However, the live migration may cause significant delays when the applications running inside a container modify large amounts of memory faster than a container can be transferred over the network to a remote host. In this paper, we propose a novel approach for live migration of containers to address this issue by utilizing a recently published CRIU feature, the so-called "image cache/proxy". This feature allows for better total migration time and down time of the container applications that are migrated by avoiding the use of secondary storage.

Keywords: Linux containers · CRIU · Live migration
Cloud computing

1 Introduction

Live migration of containers is the act of detaching a set of processes that run in the context of a container, transfer them to a remote host, and reattach them back to the new OS kernel. Checkpoint-restore in Userspace (CRIU) [4] is a tool that allows such live migration of a hierarchy of processes (container) between two physical computers. Live migration techniques are used for moving a container instance from one physical host to another, while preserving the running state of the containerized applications and maintain open network connections. Live migration across distinct physical hosts has several benefits, such as dynamic load balancing, fault tolerance, data access locality, and it makes low-level system maintenance easy by allowing a clean separation between hardware and software. Migration can be used to improve power efficiency by gathering containers together on a physical machine, and to enable the suspension of currently-unused hardware resources.

During migration, several resources are transferred over the network – CPU state, memory state, network state and disk state. The transfer of the disk state can be circumvented by having a shared storage (SAN, NAS, NFS, etc.) between hosts participating in the live migration process. The size of the memory state depends on the type of applications that are being migrated. For example, an HTTP server such as Nginx[1] might use only a few hundreds megabytes of memory, whereas an in-memory data store such as Redis[2] might be associated with several gigabytes of data. The transfer of the whole memory state during migration can take too long to be a practical solution [14]. This problem is specifically hard when containerized applications modify large amounts of memory faster than a container can be transferred over the network.

The technique used to send the memory state to a remote host is, therefore, a major concern in live migration algorithms.

Improving the performance of live migration algorithms mainly focuses on reducing the *total migration time* and *down time*. The total migration time is the time between the start and the end of the migration process. Down time is the time when the migrated application is not running neither on the source nor on the destination server. Live migration algorithms aim to minimize the down time period, during which the application service is totally unavailable, while keeping the total migration time as small as possible.

Many live migration algorithms have been proposed over the years [18]. In terms of the way they transfer the memory state, these algorithms are classified into three categories:

1. **Pre-copy** migration starts by copying the memory state to the destination host. While copying, the source host remains responsive and keeps progressing all running applications. As memory pages may get updated on the source system, even after they have been copied to the destination system, the approach employs mechanisms to monitor page updates.
2. **Post-copy** first suspends the migrated application at the source host, copies a minimal processor state to the destination host, where the migrated application is resumed, and begins fetching memory pages over the network from the source.
3. **Hybrid-copy** works by combining both pre and post copy algorithms. It first starts pre-copy migration of the application, which keeps running on the source host, while all the memory pages are copied to the destination host. The application is then suspended and its processor state is copied over, without the remaining memory pages. Then, the application is resumed at the destination immediately, and the post-copy algorithm is used to synchronize the rest of the memory pages.

Two main implementations of container migration, Docker [6] and LXD [7] currently only use pre-copy migration. There are no container runtime systems that currently use image-cache/image-proxy for migration. In this paper, we report on our migration implementation that uses Image Cache and Image Proxy for container migration, which are two options in development in the CRIU (Checkpoint Restore In Userspace) tool. We show that live migration of Linux

[1] https://www.nginx.com/.
[2] https://redis.io/.

containers based on the image-cache and image-proxy components of CRIU, currently available in its development branch, provides better performance and time savings in migrating containers. The pre-copy migration algorithm has been implemented by performing one or more pre-dump iterations with CRIU Sect. 2.1. This approach allows the automation of the transfer of image files and reduces the total migration time and down time by keeping all memory pages in a cache buffer rather than storing them on disk. The results of our evaluation show that the performance of pre-copy migration with CRIU depends on the memory intensity of applications, and the total migration time and down time increase proportionally to the amount memory used by the migrating process.

In Sect. 2, we describe in detail the particular features of checkpoint/restore mechanisms relevant to our work. In Sect. 3, we point out how we improve live migration. In Sect. 4, we provide a performance analysis of process live migration. In Sect. 5, we discuss future work and conclude the paper with Sect. 6.

2 Live Migration with CRIU

With the increased interest in Linux container technology, the checkpoint/restore mechanism has attracted more attention. This mechanism can be used for fault tolerance or dynamic load balancing by migrating a running process from one system to another. Live migrating a process is nothing more than checkpointing a process, transferring it to a destination system, and restoring the process back to its original running state. The checkpoint/restore mechanism can be applied to a hierarchy of processes, thus it is a perfect base technology for container migration. Early implementations of checkpoint/restore were implemented as an in-kernel approach [10,16]. As pointed out in [3], these implementations did not focus on upstream inclusion in the Linux kernel. As a result, there was no agreement in the Linux kernel community on the design of a checkpoint/restore mechanism, which led to the adoption of solutions that were not officially accepted by the Linux community [3]. The CRIU project solves this problem by implementing checkpoint/restore in user space, using available kernel interfaces. As pointed out in [3], one of the most important kernel interfaces for checkpointing is the *ptrace* (see ptrace manual pages) system call. It provides means for a process to control the execution, and examine and change the memory allocated to another process and its registers. Another important kernel feature, *last_pid* control, that is used to implement the restore functionality of CRIU allows the restored process to receive the same process identifier (PID) it had during checkpointing. In order to achieve this, CRIU writes one number less of the desired PID to `/proc/sys/kernel/ns_last_pid`. Then, it verifies that the newly created process has the correct PID, otherwise the restoration of this process is aborted.

2.1 Pre-copy Migration with CRIU

Iterative pre-copy live migration is one of the most reliable live migration algorithms. By "*iterative*" we mean that pre-copying occurs in *rounds*, during which the memory pages to be transferred in round *n* are those that are modified after round *n - 1* (all pages are transferred in the first round). The pre-copy support in CRIU is implemented as an incremental pre-dump [1] by using the concept of "soft-dirty bit" on a Page Table Entry (PTE) [13]. The soft-dirty bit feature is implemented in the Linux kernel to enable tracking of memory changes [12]. CRIU starts the tracking of memory by writing the integer 4 to `/proc/$PID/clear_refs`. This operation instructs the kernel to clear the soft-dirty and the writable bits from all PTEs of the specified process. After this, every first write operation on a memory page associated with this process will set the soft-dirty flag. Modified PTEs are identified in a subsequent pre-copy iteration by reading `/proc/$PID/pagemap`, where `$PID` is a process identifier of the migrated process. The modified PTEs are those that have a soft-dirty (the 55'th) bit reported. CRIU enables pre-copy iterations with the `pre-dump` action. This allows CRIU to extract only part of the information (i.e. the memory pages) associated with a container.

2.2 Post-copy Migration with CRIU

Post-copy minimizes the application down time during live migration. In contrast to pre-copy, this algorithm transfers all memory pages until after the CPU state has already been moved and resumed on the destination host. Concurrently, when the migrated application accesses a missing memory page, CRIU handles this page fault by transferring the required page from the source node and injects it into the running task memory address space. This demand paging approach ensures that each memory page is sent over the network *at most once*. However, the network delay might cause a performance degradation of the migrated process, as well introduce additional, high-priority network traffic. Residual dependencies are being removed from the source host as quickly as possible by pro-actively pushing the remaining memory pages to the destination.

The post-copy algorithm is implemented in CRIU by utilizing a recently added user-space page fault feature in the Linux kernel – userfaultfd [2]. The post-copy migration can be started by providing a `--lazy-pages` flag to the *dump* action, as well as during restore to skip the injection of memory pages into the processes address space and register lazy memory areas with userfaultfd [2]. This option instructs CRIU to not extract memory pages during checkpoint, and to allow the lazy-pages daemon to request them via TCP connection. Memory page fault notifications are handled by a *lazy-pages* daemon that receives a userfault file descriptor from the restore process via UNIX socket.

2.3 Automatic Transfer of Image Files

Recently, CRIU has been extended with a new feature that automates the transfer of image files [8]. This feature enables live migration of Java applications from one Java runtime environment (the so-called "Java Virtual Machine") to another without performing expensive I/O operations. This extension introduces the"image-cache" and "image-proxy" actions for CRIU, as well as the `--remote` option for dump and restore. The automatic transfer of image files enables simplified implementation for live migration of containers.

Two main implementations of container migration, Docker [] and LXD [] currently only use pre-copy migration. There are no container runtime systems that would use image-cache/image-proxy for migration. In this paper, we report on a migration implementation that uses Image Cache and Image Proxy for container migration, using image cache and image proxy for live migrating containers. The pre-copy migration algorithm has been implemented by performing one or more pre-dump iterations with CRIU Sect. 2.1. The pre-dump CRIU feature allows to extract relevant memory pages of a hierarchy of processes (container) that is being migrated. The pre-dump CRIU feature is described in the Patent Specification US9621643B1 [11] and is implemented by injecting a parasite code, which is used to execute CRIU service routines inside the address space of the migrated process. The parasite code is a binary blob of code built in PIC (position-independent code) format for execution inside another process address space. PIC is a body of machine code that, being placed somewhere in the primary memory, executes properly regardless of its absolute address.

There are two modes the parasite can operate in - *trap mode* and *daemon mode*. In trap mode, the parasite executes one command at a time. Right after a call is executed, a trapping instruction is placed to trigger a notification to the caller and indicate that the parasite has finished. When the parasite runs in daemon mode, it opens a command socket that is used to communicate with the caller. Once the socket is opened, the daemon will wait for commands in sleep mode. When everything is done, the parasite code is being removed, and the migrated process is left back into the state it was before the injection.

2.4 Design of Post-copy Memory Migration

The current Post-copy implementation in CRIU begins with a “normal” checkpoint (`criu dump`) with provided `--lazy-pages` option. By using this option, the process memory is collected into pipes and non-lazy pages are stored into image files or transferred over to the destination host via page-server. The lazy pages are kept in pipes for later transfer. After the checkpoint is completed (at the dump_finish stage), a TCP server is started to handle page requests from the restore host. In other words, the lazy memory pages are transferred on-demand via TCP connection rather than being stored into image files. At the destination side, a lazy-pages daemon has to be started to create two sockets – a UNIX socket that is used to receive page-faults from the restore; and a TCP socket to forward page requests to the source host. The checkpoint image files are used by the restore action to create memory mappings, registers memory areas with userfaultfd and fill the VMAs that cannot be handled by the userfaultfd mechanism. All lazy memory pages are handled by a dedicated daemon. On page-fault, the restore action sends the userfaultfd to the lazy-pages daemon, which sends a command to the source host. The source side extracts the requested memory pages from the pipe and send them over via the TCP socket, and the lazy-pages daemon copies the received pages into the restored process address space. The lazy-pages daemon knows which pages exist on destination host, and therefore it can identify when all pages have been migrated. CRIU implements the *active pushing* mechanism [15] for post-copy migration by copying the remaining memory pages of the migrated process in the background while no network page faults are being requested [2].

2.5 Combining Pre-copy and Post-copy Migration

Both Pre-copy and Post-copy migration algorithms have advantages and disadvantages. Application down time can be significantly decreased by pre-copying relevant memory pages from the address space of the migrating process hierarchy before the final freeze. However, in the case of write-memory-intensive applications, such pre-copy iterations have negative effects and increase the total migration time without improving the application down time. A solution to this problem is provided with post-copy migration. The post-copy algorithm transfers only the minimal application state that is required during down time in order to resume execution on the destination host. This approach greatly improves the application down time and ensures that each memory page is transmitted over the network at most once. However, this approach has two major drawbacks. First, the performance of the application during migration time is affected significantly due to the network latency that slows down memory access. Second, the reliability of this migration algorithm is reduced due to the impossibility to recover the running state of the application on neither the source or the destination host in case of network problems.

Fortunately, both pre- and post-copy algorithms can be used together with CRIU [19]. This approach is also known as *hybrid-copy*. The benefits of this combination are the following:

1. Application downtime is minimized by transferring memory pages that are being frequently modified on-demand.
2. Performance degradation of the application, after it has been moved over to the destination host, is minimized by providing as much memory pages as possible on the destination side prior the start of the post-copy phase. For example, read-only areas of the application's memory address space are being transfered over prior the post-copy phase.
3. Reliability is significantly improved, in comparison with Post-copy, by reducing the number of pages that are being transfered on-demand.

3 Using Image Cache and Image Proxy for Container Live Migration

Originally, CRIU was designed to store the running state of a checkpointed process as a collection of image files to persistent storage. In a live migration scenario, this approach has two major disadvantages. First, all image files are written to persistent storage twice – once when dumping the container on the source host, and once when receiving the images on the destination host. Second, these image files are read from disk twice – once when sending the images to destination host, and once when performing the restore operation. A simple solution of this issue is outlined in the CRIU documentation [5] as *disk-less migration*, which stores image files on a temporary file system instead of persistent storage.

A better solution has been proposed by introducing two new components to CRIU – image-cache and image-proxy [8]. These components allow a decoupling of the saving/reading of image files from dumping/restoring a process tree. The communication between these two components is achieved over a TCP socket and the running state of the checkpointed/restored process is transferred via Unix sockets from the CRIU process. This approach decreases the total migration time and down time for live container migration by keeping image files in cache rather than on persistent storage. Another major advantage of this approach is the automated transfer of image files, which allows a simplification of the implementation of live migration with CRIU.

4 Evaluation

Qualitative evaluation between pre- and post-copy migration algorithms would give some indication of their potential value for migrating memory-intensive applications. The image-cache/proxy implementation is compared with total-copy, which transfers (using rsync) the entire process state before the process execution resumed on the destination machine.

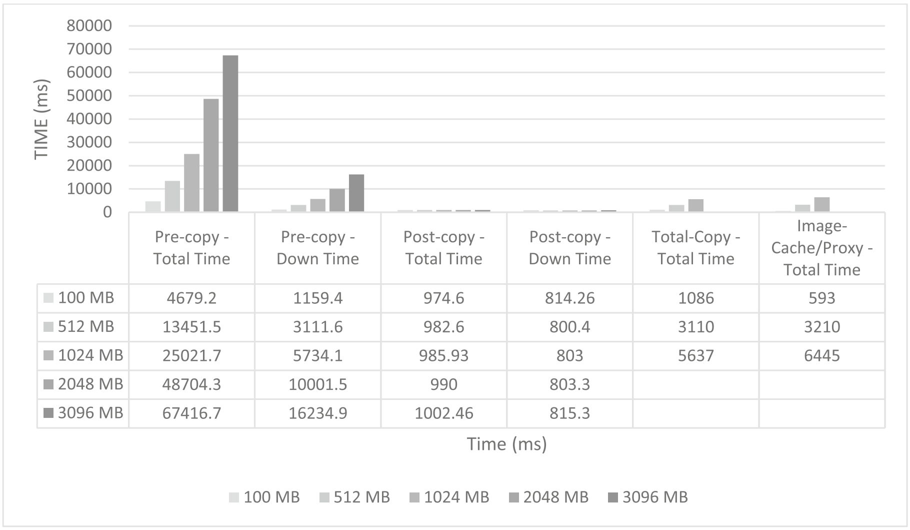
All evaluation tests were performed by live migrating the memhog process between two VMs with pre-installed Fedora 27 Cloud Edition and CRIU compiled from the criu-dev git branch – commit 8340e64137e. Both VMs have an identical hardware configuration – 4 vCPUs (Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20 GHz), 30 MB cache size, and 8 GB RAM.

The results show that the performance of pre-copy migration with CRIU depends on the memory intensity of applications, and the total migration time and down time increase proportionally to the amount memory used by memhog. The total migration time and down time for the post-copy migration do not increase significantly with the increase of memory used by memhog.

The image-cache/proxy mechanism does not have significant improvement in terms of decreased total migration time. In contrast, when compared with the total-copy using rsync, the image-cache/proxy technique shows higher migration time, which increases proportionally with the amount of memory used by memhog.

5 Discussion and Future Work

Several techniques, borrowed from live migration of virtual machines, are applicable for Linux containers. These techniques can be used to extend CRIU to further optimize the pre- and post-copy migration (Fig. 1).



	Pre-copy - Total Time	Pre-copy - Down Time	Post-copy - Total Time	Post-copy - Down Time	Total-Copy - Total Time	Image-Cache/Proxy - Total Time
100 MB	4679.2	1159.4	974.6	814.26	1086	593
512 MB	13451.5	3111.6	982.6	800.4	3110	3210
1024 MB	25021.7	5734.1	985.93	803	5637	6445
2048 MB	48704.3	10001.5	990	803.3		
3096 MB	67416.7	16234.9	1002.46	815.3		

Fig. 1. Comparison of the live migration algorithms for the memory intensive application (*memhog*) used with different size of allocated memory.

In identifying the Writable Working Set, each process will have some (hopefully small) set of memory of pages that it updates very frequently and which are therefore poor candidates for pre-copy migration. This concept of *writable working set* (WWS) was first introduced in [9]. Based on the analysis of the behavior of server workloads (the WWS), the number of pre-copy iterations can further improve the efficiency of the migration. The writable working set can be identified by reading `/proc/$PID/pagemap` between pre-copy iterations and keep track of the modified memory pages since the last iteration that have not been send over to the new host yet. A similar idea was demonstrated with a Markov model applied to forecast the memory access pattern to adjust the memory page transfer order and reduce the number of unnecessary transfers [17].

A crucial concern for live migration is the impact on active services. Resource usage control during live migration may provide performance enhancements. For instance, iteratively extracting and sending memory pages between two hosts in a cluster can easily consume the entire bandwidth available between them and hence starve the active services of resources. This issue needs to be addressed by carefully controlling the network and CPU resources used by CRIU during the migration process, thereby ensuring that it does not interfere excessively with active traffic or processing.

Another improvement may be achieved through Delta Compression Based Memory Transfer. In order to live migrate a virtual environment (container), all memory pages need to be transferred across the network to the destination host. However, a typical memory page occupies 4 KB, a standard Ethernet network packet can only transport 1 KB (including header). Therefore to transfer a unique memory page, we need to send a minimum of 5 packet over the networks. When the memory pages are frequently updated, and the changes must be propagated over some transport medium it is undesirable to transmit the full new page due to its size. An effective solution to this I/O bottleneck is a delta compression algorithm for live migration of KVM virtual machines [14]. By using a simple and fast compression algorithm such as XORed Binary Run Length Encoding (XBRLE) [14] on the original and updated memory pages, and transmit only the delta file, the amount of down time can be reduced drastically, making live migration a suitable for large business applications.

The characteristics of this algorithm of being very small (a few hundred lines of code) and fast, allow the maintenance of both memory pages and the executable code in CPU cache. However, the compression ratio performance is limited since information entropy plays a role as a measure on how well data can be compressed.

6 Conclusion

In this paper, we discuss an implementation of the use of Image Cache and Image Proxy for container migration (CRIU). Originally, CRIU was designed to store the running state of a checkpointed process as a collection of image files to persistent storage. In a live migration scenario, this approach has major disadvantages as image files have to be stored and retrieved multiple times from persistent storage. In this paper, we presented the implementation of a *disk-less migration*, which stores image files on a temporary file system instead of persistent storage, using two new components to CRIU – image-cache and image-proxy. The results show that the performance of pre-copy migration with CRIU depends on the memory intensity of applications, and the total migration time and down time increase proportionally to the amount memory of the migrating process. More work is needed to improve the migration performance of our implementation of an image-cache/proxy mechanism. Although it should perform better then a total-copy migration, the evaluation results show that our implementation is not yet optimized. The pre- and post-copy algorithms were compared by migrating a memory intensive process – memhog. The results show that post-copy performs better for this type of application.

References

1. Memory changes tracking - CRIU documentation. https://criu.org/Memory_changes_tracking
2. Userfaultfd - CRIU documentation. https://criu.org/Userfaultfd
3. CRIU - Checkpoint/Restore in User Space, October 2016. https://access.redhat.com/articles
4. CRIU (2018). https://criu.org/
5. CRIU disk-less migration (2018). https://criu.org/Disk-less_migration
6. Docker, July 2018. https://docs.docker.com/engine/reference/commandline/checkpoint/

7. Lxd, July 2018. https://github.com/lxc/lxd
8. Bruno, R., Ferreira, P.: ALMA: GC-assisted JVM live migration for java server applications. In: Proceedings of the 17th International Middleware Conference, p. 5. ACM (2016)
9. Clark, C., et al.: Live migration of virtual machines. In: Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2, pp. 273–286. USENIX Association (2005)
10. Documentation, O.: Checkpointing and live migration (2018). https://wiki.openvz.org/Checkpointing_and_live_migration
11. Emelyanov, P.: System and method for joining containers running on multiple nodes of a cluster. https://patents.google.com/patent/US9621643
12. Emelyanov, P.: Ability to monitor task memory changes, April 2013. https://lwn.net/Articles/546966/
13. Emelyanov, P.: Soft-Dirty PTEs - Linux Kernel Documentation, April 2013. https://www.kernel.org/doc/Documentation/vm/soft-dirty.txt
14. Hacking, S., Hudzia, B.: Improving the live migration process of large enterprise applications. In: Proceedings of the 3rd International Workshop on Virtualization Technologies in Distributed Computing, pp. 51–58. ACM (2009)
15. Hines, M.R., Deshpande, U., Gopalan, K.: Post-copy live migration of virtual machines. ACM SIGOPS Oper. Syst. Rev. **43**(3), 14–26 (2009)
16. Laadan, O., Nieh, J.: Transparent checkpoint-restart of multiple processes on commodity operating systems. In: 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference, ATC 2007, pp. 25:1–25:14. USENIX Association, Berkeley (2007). http://dl.acm.org/citation.cfm?id=1364385.1364410
17. Lei, Z., Sun, E., Chen, S., Wu, J., Shen, W.: A novel hybrid-copy algorithm for live migration of virtual machine. Future Internet **9**(3), 37 (2017)
18. Milojičić, D.S., Douglis, F., Paindaveine, Y., Wheeler, R., Zhou, S.: Process migration. ACM Comput. Surv. (CSUR) **32**(3), 241–299 (2000)
19. Reber, A.: Combining pre-copy and post-copy migration, October 2016. https://lisas.de/~adrian/posts/2016-Oct-14-combining-pre-copy-and-post-copy-migration.html