mWarp: Accelerating Intra-Host Live Container Migration via Memory Warping

Piush K Sinha, Spoorti S Doddamani, Hui Lu, and Kartik Gopalan
State University of New York (SUNY) at Binghamton
{psinha1, sdoddam1, huilu, kartik}@binghamton.edu

***Abstract*—Live container migration allows containers to roam from one server to another to achieve agility goals like load balancing, tackling machine failures, scaling in/out and reallocating resources. However, migrating a container is also costly mainly due to memory state migration — a large number of memory pages need to be copied from the source server to the destination server. In this paper, we propose a fast and live container migration approach, mWarp, in an intra-host scenario, where both the source and destination virtual machine (VM) servers reside on the same physical host. Instead of copying a container's memory, mWarp relocates the ownership of the container's physical memory pages from the source VM to the destination VM with a highly-efficient memory remapping mechanism. As relocation of memory ownership is light-weight, mWarp leads to fast and live container migration with less service disruption to applications running in containers being migrated. We implement mWarp upon a well-known live container migration tool (CRIU) with key kernel/hypervisor-level support. The evaluation with both micro benchmarks and real-world applications shows that mWarp greatly reduces the total container migration time and downtime (e.g., by an order of magnitude) with significantly improved application-level performance (e.g., by 20%).**

***Index Terms*—Virtualization, Container, Live Migration**

I. Introduction

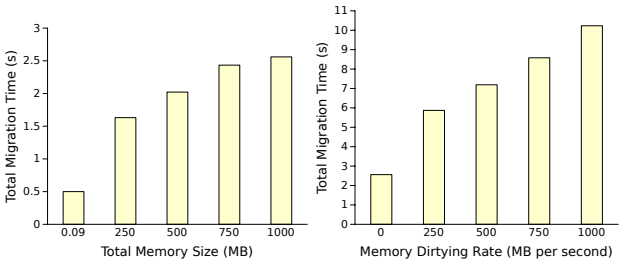
As an alternative to virtual machine (VM) based virtualization (e.g., KVM, VMware, Xen), containers relying on process-based virtualization offer a much flexible way in deploying and executing applications. With containers, a bunch of new use cases have been enabled in clouds such as (micro-)services orchestration, management and just-in-time deployment [1]–[5]. For better isolation and security in multi-tenant public clouds [6]–[10], containers are commonly encapsulated in VMs while running. For example, containerized applications are orchestrated and managed by Google Kubernetes Engine upon a group of Google Compute Engine instances (i.e., VMs) [3].

Like VMs, in-cloud containers need the capability to migrate from one VM to another to achieve agility goals like balancing load, escaping from hardware failures, scaling in/out demands for resources, etc. A "cold" container migration procedure [11] usually involves three main steps: (1) suspending the container on the source VM; (2) copying the state of the container from the source VM to the destination VM; and (3) restoring the container on the destination VM with the same state as that on the source VM. In practice, a *live* container migration is more appealing which keeps the container running during migration — without disrupting the services of applications running within. For example, precopy-based live container migration [12] allows the migrated container to keep running on the source side while its memory gets transferred to the destination in an iterative manner (i.e., only the dirtied memory is transferred in each iteration), until the dirtied portion is small enough when the container is paused and its remaining state is copied (less migration downtime).

Though the above approaches [12], [26] reduce the migration downtime, they in turn lengthen the total migration time (i.e., the time between the start and the end of the migration) due to either iterative memory copying [12] or on-demand memory pages fetching [26]. During the total migration time, the performance of applications running in containers could be negatively impacted, for example, by costly page-fault-based dirty memory tracking or massive migration network traffic.

In this paper, we present **mWarp**, a fast and live container migration approach targeting a common *intra-host* migration scenario in public clouds: When performing container migration, it is preferable to choose/provision a destination VM on the same physical host (as long as the underlying host remains available with sufficient resources), as the intra-host migration can leverage local memory bandwidth for fast state transferring and avoid costly inter-host network communication. Such intra-host container migration is particularly applicable for a VM that needs to be temporarily shut down for maintenance, upgrade, and recovering from failures during which its hosted processes/containers must be migrated.

The intra-host live container migration makes the key idea of mWarp feasible: As the source and destination VMs reside on the same physical host sharing the same memory, mWarp completely avoids memory copying by *relocating* the ownership of a container's memory pages from the source VM to the destination VM — which we call *memory warping*. To realize a highly-efficient memory relocation mechanism, mWarp involves a new page table, *mWarp table*, residing in the host kernel. During a container's migration, the mWarp table keeps track of the mappings from the container's address space to its host physical address on the source VM, and exposes such mapping information for reconstructing the containers memory space on the destination VM. As mWarp's memory relocation is super fast in comparison with traditional memory copying, mWarp leads to sub-second intra-host live container migration, regardless of the size of the containers being migrated. In contrast, the memory copying based intra-host container migration can take several or tens of seconds.



(a) Impact of varying memory size for an idle process.

(b) Impact of varying memory dirtying rate given the 1 GB size.

Fig. 1: Impact on total migration time by varying memory size and varying memory dirtying rate.

We have implemented a prototype of mWarp on CRIU [11], a well-known, user-space process checkpoint/restore tool. mWarp further extends Linux kernel with the support of the mWarp table and implements the memory relocation mechanism for live container migration. Our preliminary evaluation results show that mWarp achieves smaller total container migration time and downtime for both micro benchmarks and real-world applications.

In the rest of the paper, we first show the overhead analysis of existing memory-copying based live container migration, followed by the design, implementation, and evaluation of mWarp, and finally discussion of related work and conclusions.

II. Problem Demonstration

Live Container Migration. To illustrate the overhead of live container migration, we use PHaul [12], a state-of-the-art pre-copy based live container migration tool based on CRIU [11]. PHaul works in the following two stages: checkpointing and restoration. In the checkpointing stage, PHaul copies the state of all processes of the container being migrated from the source VM to the destination VM in an iterative manner. In the restoration stage, PHaul restores all processes of the container in the destination VM with received state.

More specifically, the pre-copy based live container migration in PHaul involves multiple iterations in the checkpointing stage. Each iteration includes three steps (except for the last one): (1) freezing the processes of the container; (2) marking memory pages to be transferred in this iteration; and (3) unfreezing the processes of the container. Note that, in step (2) the dirtied pages since the last iteration are marked to be transferred (all pages are marked in the first iteration). After step (3), the marked memory pages will be copied from the source VM to the destination VM *asynchronously* — without blocking the processes being migrated. The migration algorithm decides to enter the last iteration when either the number of dirty pages is small enough or it reaches a certain number of iterations. In the last iteration, all processes are suspended and all memory pages (i.e., the content) as well as other state of the processes (e.g., register contents, process trees, memory properties, etc.) are copied to the destination VM, after which the container can be gracefully shutdown.

On the destination VM, after receiving the complete state of the container, the restoration procedure restores all the processes of the container by first creating "empty" processes and then loading the received state to these processes making them have exact the same state as those in the source container (right before suspension).

Overhead Analysis. We conduct the overhead analysis of the live container migration using with the following configurations: Each (the source and destination) VM is configured to have sufficient resources (4 virtual CPUs and 4 GB Memory) running on the same physical host (12 physical CPUs and 128 GB memory). The network bandwidth between these two intra-host VMs is set to 10 Gbps. Two main metrics are used for gauging the performance of live container migration: *total migration time* — the time between the start and the end of the whole migration; and *frozen time* — the time during which the migrated container is suspended (i.e., in the last iteration).

First, we initialize a container with a single *idle* process. We create such a process with varying memory sizes from a "minimal" size of 0.09 MB (with only 22 memory pages) to a relatively large size of 1 GB. The process allocates the memory of a given size and dirties the entire allocated memory by performing write operations. After initialization, the process stays *idle* while we perform live migration. As shown in Figure 1 (a), as the memory size increases, the total migration time increases from 0.5 seconds under the size of 0.09 MB to 2.56 seconds under the size of 1 GB.

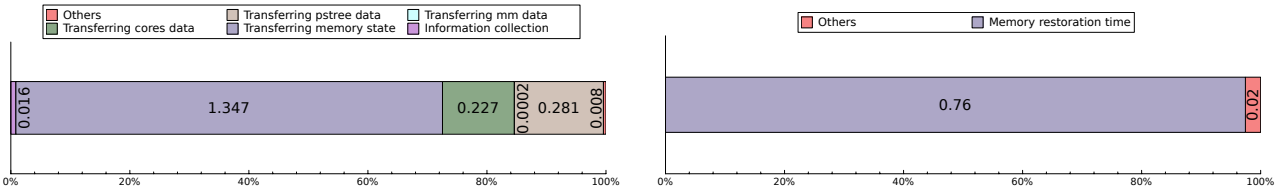
Further, we run a container with a single *busy* process. We fix the memory size of the process to 1 GB, allowing it to keep dirtying memory pages at different rates during migration, from 0% (i.e., no pages are dirtied) to 100% (i.e., all allocated pages are dirtied per second). As plotted in Figure 1 (b), the total migration time keeps increasing from 2.56 seconds (at rate 0%) to more than 10 seconds (at rate 100%).

We also measure the frozen time to analyze the service disruption in the execution of the container. The detailed frozen time breakdown in both checkpointing and restoration stages is shown in Figure 2 — under the case of the 1 GB process with 100% dirtying rate. In Figure 2 (a), we observe that more than 70% of the frozen time in the checkpointing stage is caused by memory state copying, while Figure 2 (b) shows that around 96% of the time during the restoration stage is contributed by restoring memory state.

In summary, in the above intra-host live container migration, the time spent in memory state copying and restoration dominates the total migration time and frozen time. This motivates us to develop a much faster live container migration technique based on memory relocation that does not involve the copying of containers' memory pages.

III. Design of mWarp

The key idea behind mWarp is as follows: Instead of copying the memory of containers being migrated, mWarp relocates the memory ownership of containers' physical memory pages from the source VM to the co-located destination VM via page table remapping. In this Section, we first discuss how mWarp



(a) Frozen time breakdown on the source VM (checkpointing stage).

(b) Frozen time breakdown on the destination VM (restoration stage).

Fig. 2: Breakdown of frozen time at each stage of live container migration.

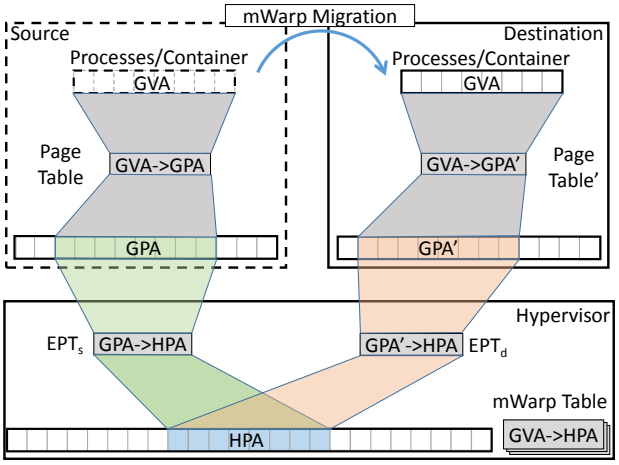


Fig. 3: Architecture of mWarp.

builds upon the existing memory translation mechanism and then present the detailed design of mWarp.

Memory Translation. In the native execution mode (i.e. without virtualization), the mappings between the virtual addresses (VA) of a process (running inside a container) to its physical addresses (PA) is kept in its page table. With such mappings, once a virtual address is accessed by the process, the memory management unit (MMU) walks the page table and does the VA-to-PA translation.

In the virtualization execution mode, an additional level of memory address translation is needed for virtualizing memory management of VMs. As plotted in Figure 3, the page table of a process running in a guest VM stores the mappings between the guest virtual addresses (GVA) of a process to the guest physical addresses (GPA) — the virtualized memory view provided by the host (or the hypervisor). The hypervisor uses another page table, called the extended page table (EPT) to map GPA to its actual host physical addresses (HPA). When a VM tries to access unallocated GPAs, EPT violations are generated, like page faults for a process. These faults are processed by the hypervisor which allocates a new physical page for the faulting GPA. Hence, to access data from GVA, two chained translations are needed as follows:

$$\text{Source Container: } GVA \rightarrow GPA \rightarrow HPA$$

As plotted in Figure 3, during migration the migration restorer on the destination VM constructs the same address space (i.e., GVA) for the container processes as those on the source VM. With the memory-copying based migration technique, the restorer loads the memory page content (received from the source VM) to the above GVA, which map to newly allocated GPA' (in the destination VM). Such GPA' in turn map to the newly allocated HPA' (in the host):

$$\text{Destination Container : } GVA \rightarrow GPA' \rightarrow HPA'$$

Note that, HPA and HPA' are two disjoint sets of physical addresses. However, they store the *same* memory page content. After migration, all HPA of the source VM will be reclaimed.

Memory Relocation. With the above observation — HPA and HPA' store the same memory page content — mWarp avoids copying memory from HPA to HPA', but instead directly relocates the memory ownership of the container's physical memory pages (HPA) from the source VM to the destination VM (i.e., by remapping the GVA of the destination VM to the same HPA of the source VM):

$$\text{Destination Container : } GVA \rightarrow GPA' \rightarrow HPA$$

The key to realizing such memory relocation is to obtain the GVA-to-HPA mappings from the source VM. However, such mapping information is scattered in two separate locations: one in the source VM's page table and the other in the host EPT table. To this end, mWarp involves a new page table, *mWarp table*, in the hypervisor. An mWarp table explicitly stores the GVA-to-HPA mappings of a container's process (identified by its unique process ID) on the source VM. To migrate a container, we need multiple mWarp tables each maintaining the GVA-to-HPA mappings of one process. The mWarp tables are then exposed to the migration restorer on the destination VM to establish the same GVA-to-HPA mappings for the processes of the container being migrated. To further distinguish processes from different VMs, an mWarp table involves a VM's identifier (VMID) as follows:

$$[VMID : PID] : GVA \rightarrow HPA$$

To build an mWarp table, the hypervisor needs to get the GVA of a process from the source VM (i.e., kernel). mWarp provides a new hypercall, `hypercall_set`, to the VM for exposing such mappings. The source VM kernel invokes this hypercall to pass a list of GVA-to-GPA mappings of the process to the hypervisor. For each received GVA-to-GPA mapping, the hypervisor extracts the GPA and translates it to HPA using the EPT table, and then puts the corresponding GVA-to-HPA mapping to the mWarp table (identified by the process's PID and the source VM's VMID).

Memory Reconstruction. Restoring the address space of the migrated container processes on the destination VM needs to build GVA, GPA' and HPA as well as their mappings. The migration restorer on the destination VM first allocates all GVA of the process (e.g., with the mmap system call) and maps them to free guest page frames, to reconstruct the GVA-to-GPA' page table mappings. mWarp provides another hypercall, `hypercall_map`, to allow the destination VM to pass a list of GVA-to-GPA' mappings and the process ID of a container's process being migrated to the hypervisor. In the `hypercall_map` hypercall, for each received GVA-to-GPA' mapping, the hypervisor: (1) looks up the *mWarp table* to find out HPA from the GVA-to-HPA mappings; and (2) creates a new mapping between GPA' and HPA to update the destination VM's EPT with this newly mapped GPA'-to-HPA.

IV. Implementation

To demonstrate our proposed memory relocation based live container migration, we have implemented an mWarp prototype on CRIU [11], a popular user space process checkpointing/restoration tool. The whole live container migration consists of two stages: checkpointing and restoration.

mWarp on Checkpointing. The creation of the mWarp table during the checkpointing stage is summarized in Figure 4 (a). On the source VM, we modify CRIU's checkpointing code to replace the procedure of copying memory content with the one of building the mWarp table. A new system call, `syscall_set`, is added to the source VM. This system call takes an array of GVA-to-GPA mappings along with the process's PID (i.e., we exploit the `/proc/[PID]/pagemap` file to get the GVA-to-GPA mappings). If we make one `syscall_set` system call for each GVA-to-GPA mapping, it will add too much overhead because of excessive system calls. We optimize our implementation to send a large number of mappings within one system call. To achieve this, we allocate page-size (i.e., 4 KB) aligned memory using the `memalign` function and fill these memory with GVA and their GPA. In practice, we allocate two page-sized arrays, one is for storing GVA and the other is for storing GPA. Once we fill up these pages, we invoke one `syscall_set` system call.

The `syscall_set` system call translates the base address of the two arrays into a set of disjoint guest physical addresses. These two disjoint guest physical addresses along with the process's PID are used for invoking the `hypercall_set` hypercall. For one system call containing page-sized arrays (as stated above), only one hypercall is invoked. As stated in Section III, the `hypercall_set` locates the mWarp table of the process, obtains the GVA-to-HPA mappings, and inserts them to the mWarp table.

mWarp on Restoration. Figure 4 (b) summarizes the details of mWarp's restoration stage using the mWarp table. On the destination VM, we modify CRIU's restoration code to replace the procedure of loading memory content with the one of relocating address space. Another new system call, `syscall_map`, is added to the destination VM which takes an array of GVA with PID. More specifically, to reconstruct the address space of the process on the destination VM, the restorer uses `mmap` system call to first restore all the GVA received from the source VM. Using `mmap`, we only reserve the guest virtual addresses without having them mapped to any physical memory (i.e., guest physical addresses on destination VM). However, these guest virtual addresses need to be mapped to free guest physical addresses (i.e., GPA'). To this end, the `syscall_map` system call reserves free memory pages using the kernel function `get_free_pages` (to reduce overhead, we can reserve a batch of free pages by calling one such function), and then establishes the mapping between GVA and these reserved GPA' by manipulating the page table entries of the process.

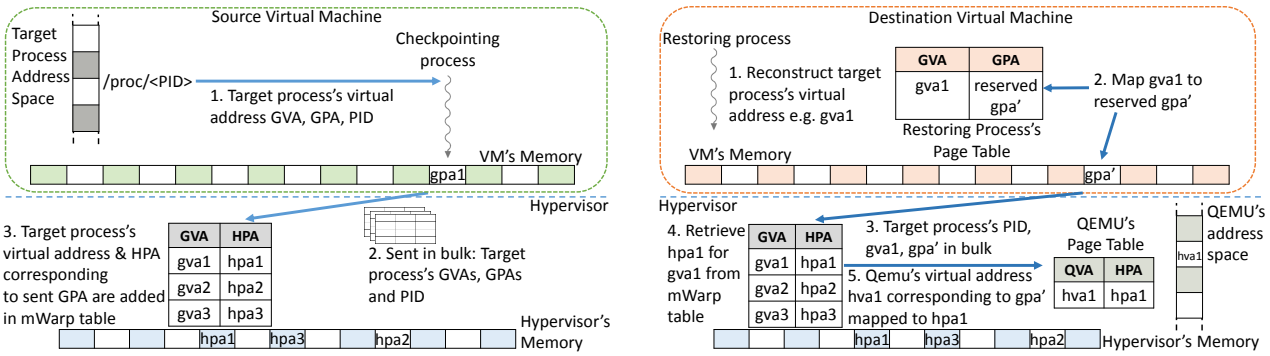
In the end of the `syscall_map` system call, it invokes the `hypercall_map` hypercalls and send the GVA-to-GPA' mappings along with the process ID to the hypervisor. Using the approach similar to that in mWarp's checkpointing stage, we optimize our design to reduce the number of system calls and hypercalls using page-size aligned memory to transfer GVA and GPA'. In the hypercall, it first checks if any of the GPA' received by the hypercall already has an existing EPT mapping. If there is, the hypervisor will invalidate the existing EPT mapping. Next, as stated in Section III, the hypervisor retrieves HPA from the mWarp table for each received GVA, and translates its GPA' to the virtual address of the QEMU process (which controls the destination VM). Last, the QEMU's virtual address, which is the host virtual address (HVA), is mapped to the received HPA, creating the HVA-to-HPA mappings in the QEMU process's page table. Notice that, the GPA'-to-HPA entry in the EPT table will be automatically established from the HVA-to-HPA page table via EPT violations.

V. Preliminary Evaluation

In this section, we present our preliminary evaluation results. We evaluate the effectiveness of mWarp by comparing it with CRIU's pre-copy live migration approach using both micro benchmarks and real-world applications.

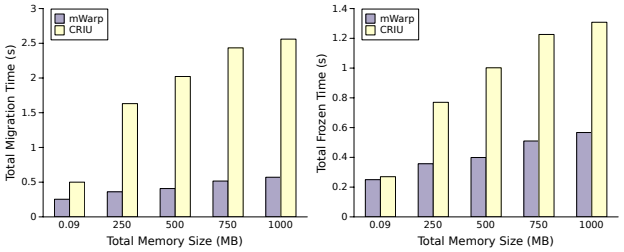
Micro Benchmark. We run a write-intensive benchmark in a container by varying its memory sizes and dirtying rates — the same one that we have used in Section II. This benchmark varies its memory size from a minimal usage of 0.09 MB (22 memory pages) to that of 1 GB in the idle case, and varies its memory dirtying rate from 0 GB/sec to 1 GB/sec with a fixed memory size of 1 GB in the busy case. We use this benchmark to evaluate the total migration time and total frozen time.

Figure 5 (a) and (b) show that, with CRIU, the total migration time and total frozen time (i.e., downtime) increase significantly with the increase of the memory size. For example, the total migration time increases from about 0.5 seconds under the 0.09 MB memory size to more than 2.5 seconds under the 1 GB memory size, while the total frozen time is around 0.3 seconds under the 0.09 MB memory size and increases to around 1.3 seconds under the 1 GB memory size. With mWarp, the changes in the total migration time and



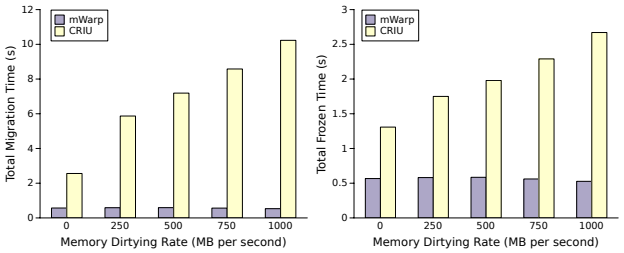
(a) mWarp's checkpointing stage. (b) mWarp's restoration stage.

Fig. 4: The checkpointing and restoration stages of mWarp's live container migration.



(a) Total migration time. (b) Total frozen time.

Fig. 5: Varying memory size on live container migration.



(a) Total migration time. (b) Total frozen time.

Fig. 6: Varying dirtying rate on live container migration.

total frozen time are very little and stay within sub-seconds regardless of the total memory size. More concretely, with mWarp, both the total migration time and total frozen time are about 0.25 seconds for 0.09 MB memory size and increase a little above 0.5 seconds under the 1 GB memory size. We notice that the gap between the CRIU and mWarp results keeps growing with the increase in memory size.

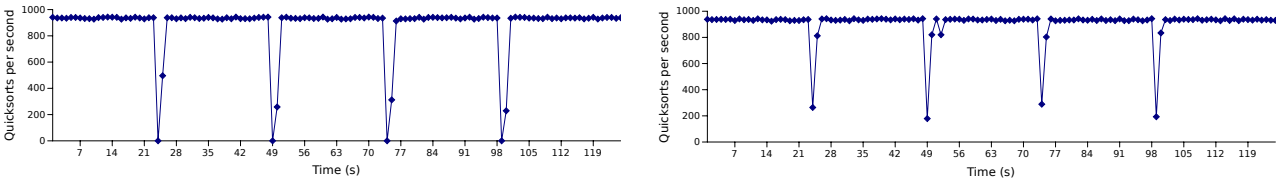
Figure 6 (a) and (b) show the total migration time and total frozen time with varying dirtying rates (fixing the total memory size to 1 GB). Figure 6 (a) shows that, with CRIU, the total migration time keeps increasing as the memory dirtying rate increases. For example, it takes around 2.6 seconds to complete the container migration when the dirtying rate is 0 GB/second, while it takes more than 10 seconds when the dirtying rate is 1 GB/second. In contrast, with mWarp, the total migration time stays constantly around 0.5 seconds even when the memory dirtying rate reaches to 1 GB/second. We have the similar observation for the total frozen time. Figure 6 (b) shows that with CRIU, total frozen time keeps increasing rapidly as the dirtying rate increases. However with mWarp, the total frozen time remains constant around 0.5 seconds.

Quicksort is a CPU and memory intensive benchmark that first fills random data in 1024 MB of allocated memory and then sorts 50 of those allocated pages (200 KB) using the quicksort sorting algorithm. We run the quicksort benchmark in a container. To show the performance impact, we consecutively migrate the container running the quicksort benchmark and observe the total number of quicksorts per second during the migration. Figure 7 (a) shows that the number of quicksorts with CRIU reaches zero during the migration. With mWarp in Figure 7 (b), we keep observing 200∼300 quicksorts during the migration. It is because, again, mWarp leads to very small migration downtime.

Sysbench [14] performs multi-threaded memory test by pre-allocating memory and then reading from or writing to it. We run the sysbench application in a container and use its write test with the varying memory range from 128 MB to 1 GB. We first run the memory write test without any migration to get its completion time. We use this as the baseline and compare it with the cases using mWarp and CRIU separately. Figure 8 shows that, with mWarp the completion time of sysbench application is almost the same as the baseline. However, the completion time increases significantly with CRIU as the pre-acllocated memory size increases.

VI. Related Work

Process/container live migration techniques have been extensively studied [11], [16], [23]–[26]. One representative approach is post-copy technique [15], [26], where pages are migrated only when they are referred on the destination machine. Although such an on-demand page migration reduces the initial cost of the migration, it increases the total migration time. With post-copy technique, having longer total migration time is risky as a network failure can result in migration failure hampering the application's execution. Pre-copy [16] keeps processes/containers running on the source machine while its memory keeps getting transferred to the destination in an



(a) Number of quicksorts with CRIU.

(b) Number of quicksorts with mWarp.

Fig. 7: Comparison of number of quicksort operations during migration.

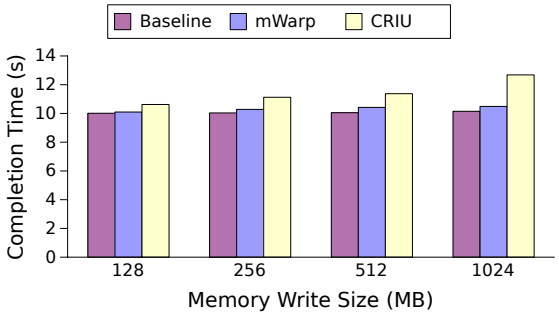


Fig. 8: Sysbench Completion Time.

iterative manner; once the dirtied pages are small enough, the whole process is paused and the remaining pages are transferred. If the page dirtying rate is very high, the migration will take way longer to complete. There has also been work that combines some features of these techniques [17]–[19]. For example, instead of copying entire memory state in the most basic technique, only the minimal dirty pages are transferred [20]. The remaining pages are transferred while the process is running on remote machine. This reduces the transfer cost, but requires the support of the remote paging mechanism. MOSIX [21] and Sprite [22] implements migration by leaving residual dependencies on the source machine. Although, it makes the migration simple but it requires the source machine to be available all the time till the process completes its execution. In contrast, mWarp focuses on an intra-host container migration scenario, which completely eliminates memory copying making the live container migration super fast.

VII. Conclusions and Future Work

We have presented mWarp, a fast and live intra-host container migration approach. In mWarp, instead of copying a container's memory, it relocates the ownership of the container's physical memory pages from the source VM to the destination VM on the same host via a highly-efficient memory relocation mechanism. Our preliminary evaluation shows that mWarp leads to sub-second total container migration time regardless of the container sizes and significant application-level performance improvement for memory intensive applications. In the future, we are going to extend mWarp for a comprehensive design and implementation including live multiple containers migration and conduct further performance optimizations including a more efficient kernel-level implementation. We will also perform detailed cost and benefit analysis with real-world containerized cloud applications.

References

[1] AWS Lambda, "https://aws.amazon.com/lambda/".
[2] Azure Serverless Computing, "https://azure.microsoft.com/en-us/overview/serverless-computing/".
[3] Google: EVERYTHING at Google runs in a container, "https://www.theregister.co.uk/2014/05/23/google_containerization_two_billion/".
[4] W. Li, A. Kanso, and A. Gherbi, "Leveraging linux containers to achieve high availability for cloud services," in Proceedings of the IEEE International Conference on Cloud Engineering (IC2E), March 2015.
[5] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-dusseau, and R. H. Arpaci-dusseau, "Serverless computation with openlambda," in Proceedings of the 8th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud, 2016.
[6] Amazon Elastic Container Service, "https://aws.amazon.com/ecs/".
[7] Kubernetes Engine, "https://cloud.google.com/kubernetes-engine/".
[8] Azure Kubernetes Service, "https://azure.microsoft.com/en-us/services/kubernetes-service/".
[9] IBM Cloud Kubernetes Service, "https://www.ibm.com/cloud/container-service".
[10] VMware Pivotal Container Service, "https://cloud.vmware.com/vmware-pks".
[11] Checkpoint/Restore In Userspace, "https://criu.org/Main_Page".
[12] P.Haul, "https://criu.org/P.Haul".
[13] E. Zayas, "Attacking the Process Migration Bottleneck," in Proceedings of the 11th Symposium on Operating Systems Principles, 1987.
[14] A. Kopytov, "Sysbench Manual," http://imysql.com/wp-content/uploads/2014/10/sysbench-manual.pdf, 2009.
[15] M. R. Hines, U. Deshpande, and K. Gopalan, "Post-copy Live Migration of Virtual Machines," in ACM SIGOPS Operating Systems Review, 2009.
[16] M. Theimer, K. Lantz, and D. Cheriton, "Preemptable Remote Execution Facilities for the V System," in Proceedings of the 10th ACM Symposium on OS Principles, pp. 2–12, 1985.
[17] G. J. W. Van Dijk, and M. J. Van Gils, "Efficient process migration in the EMPS multiprocessor," in Proceedings of the 6th International Parallel Processing Symposium, pp. 58–66, March 1992.
[18] A. Schill, and M. Mock, "DC++: Distributed Object Oriented System Support on top of OSF DCE," in Distributed Systems Engineering, 1993.
[19] S. Petri, and H. Langendorfer, "Load Balancing and Fault Tolerance in Workstation Clusters Migrating Groups of Communicating Processes," in Operating Systems Review, 1995.
[20] E. T. Roush, and R. Campbell, "Fast Dynamic Process Migration," in Proceedings of the International Conference on Distributed Computing Systems, pp. 637–645, 1996.
[21] A. Barak, and A. Litman, "MOS: a Multicomputer Distributed Operating System," in Software-Practice and Experience, 1985.
[22] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch, "The Sprite Network Operating System," in IEEE Computer, 1988.
[23] Przemysaw, Brian N. Bershad Stefan Savage, et al. "Extensibility, Safety and Performance in the SPIN Operating System." Fifteenth ACM Symposium on Operating Systems Principles. 1995.
[24] D. R. Engler, and M. F. Kaashoek, "Exokernel: An operating system architecture for application-level resource management", Vol. 29, no. 5. ACM, 1995.
[25] H. Lu, C. Xu, C. Cheng, R. Kompella, and D. Xu, "vhaul: Towards optimal scheduling of live multi-vm migration for multi-tier applications", In IEEE Cloud, 2015.
[26] E. Zayas, "Attacking the Process Migration Bottleneck," in Proceedings of the 11th Symposium on Operating Systems Principles, 1987.