Fast In-Memory CRIU for Docker Containers

Ranjan Sarpangala Venkatesh
Georgia Institute of Technology
ranjansv@gatech.edu

Till Smejkal
TU Dresden
till.smejkal@tu-dresden.de

Dejan S. Milojicic
Hewlett Packard Labs
dejan.milojicic@hpe.com

Ada Gavrilovska
Georgia Institute of Technology
ada@cc.gatech.edu

ABSTRACT

Server systems with large amounts of physical memory can benefit from using some of the available memory capacity for in-memory snapshots of the ongoing computations. In-memory snapshots are useful for services such as scaling of new workload instances, debugging, during scheduling, etc., which do not require snapshot persistence across node crashes/reboots. Since increasingly more frequently servers run containerized workloads, using technologies such as Docker, the snapshot, and the subsequent snapshot restore mechanisms, would be applied at granularity of containers. However, CRIU, the current approach to snapshot/restore containers, suffers from expensive filesystem write/read operations on image files containing memory pages, which dominate the runtime costs and impact the potential benefits of manipulating in-memory process state.

In this paper, we demonstrate that these overheads can be eliminated by using MVAS – kernel support for multiple independent virtual address spaces (VAS), designed specifically for machines with large memory capacities. The resulting VAS-CRIU stores application memory as a separate snapshot address space in DRAM and avoids costly file system operations. This accelerates the snapshot/restore of address spaces by two orders of magnitude, resulting in an overall reduction in snapshot time by up to 10× and restore time by up to 9×. We demonstrate the utility of VAS-CRIU for container management services such as fine-grained snapshot generation and container instance scaling.

MEMSYS '19, September 30-October 3, 2019, Washington, DC, USA

ACM ISBN 978-1-4503-7206-0/19/09…$15.00
https://doi.org/10.1145/3357526.3357542

CCS CONCEPTS

• **Software and its engineering → Virtual memory**; **Checkpoint / restart**; *Main memory*; • **Hardware →** *Non-volatile memory*.

KEYWORDS

In-memory Address Space Snapshot/Restore, Container snapshot/restore, CRIU, Multiple Virtual Address Spaces (MVAS), Memory-Centric Computing

ACM Reference Format:
Ranjan Sarpangala Venkatesh, Till Smejkal, Dejan S. Milojicic, and Ada Gavrilovska. 2019. Fast In-Memory CRIU for Docker Containers. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '19), September 30-October 3, 2019, Washington, DC, USA.* ACM, Washington, D.C., USA, 13 pages. https://doi.org/10.1145/3357526.3357542

1 INTRODUCTION

Container technology, with Docker as a front runner, is widely adopted for management of datacenter and cloud applications and infrastructure resources [16, 26]. A key mechanism underpinning many of the operations in the container life-cycle management is snapshot-restore. It permits for the entire state of a running container to be snapshotted, temporarily suspended, or saved and later restored in an entirely different container [9, 27]. On machines with large amounts of memory, many snapshot-based services, such as scaling of workloads via new container instances [15], rewind-restore debugging [24], or even scheduling [12], can be accelerated via in-memory snapshots; after all, these services do not require persistent snapshots that need to survive node crashes and reboots.

For Docker containers, the current approach uses CRIU [3], a widely-used open-sourced snapshot/restore tool in userspace. CRIU provides application-transparent snapshotting for a wide range of applications, including LXC and OpenVZ containers. It operates by suspending the running process container, and saving all the process state, including the process memory, into files.

CRIU captures container application states of the CPU, registers, signals and also memory in image files. The cost of writing/reading memory pages from image files

is expensive for applications with large memory footprints. Even on ramfs, which is an in-memory filesystem, the time to write/read memory pages and associated metadata dominate snapshot/restore time. Our experiments show memory restore time accounts for 83% of the total restore time for a 8GB address space. Such overheads can obviate the benefits sought after from in-memory snapshots. Moreover, they are not necessary for in-memory snapshots which do not require the persistence property typically associated with writing data into files.

In this paper, we show that these overheads can be eliminated through use of recent kernel-level support for multiple independent address spaces (MVAS) [4, 23]. MVAS is a modified operating system process abstraction which can support multiple virtual address spaces (VAS). These address spaces are independent of the process and can be detached, saved, and again attached in the same or different process. If saved in persistent memory they can live across reboots or even operating system versions.

We present VAS-CRIU, a new in-memory snapshot/restart mechanism which uses MVAS. In VAS-CRIU, we snapshot process memory into a process independent virtual address space. A VAS contains a page table and a memory descriptor not tied to any process. The VAS and the snapshotted application share pages in a Copy-On-Write (COW) manner, reducing snapshot creation time and the overall memory footprint of the snapshots. The VAS can later be remapped to restore a container's process memory using COW, accelerating restore and rewind operations. VAS-CRIU speeds up both snapshot and restores of containers.

We demonstrate the utility of VAS-CRIU with two use cases. The first one stresses the snapshot side of VAS-CRIU, by creating container snaphots at fine granularity. Frequent snapshotting of application containers is very useful in various scenarios ranging from crash recovery to replay debugging. In current approaches to snapshot application containers using CRIU, memory pages are written into image files, causing application containers to be paused while a snapshot is created. If the application is snapshotted frequently, the total runtime will be dominated by the snapshot time, limiting the number of snaphots and their granularity. Figure 1 shows the split cost of snapshotting an application container. The figure shows that snapshotting memory accounts for 80% of the snapshot time. VAS-CRIU accelerates the snapshot time by storing memory pages as separate address spaces in DRAM. As a result, VAS-CRIU acheives reduced snapshot time, which in turn allows for frequent snapshot generation.

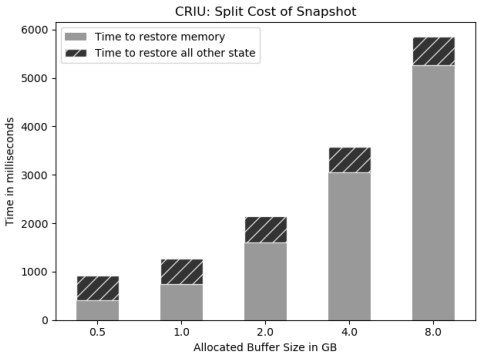


Figure 1: CRIU: Split cost of snapshot time

The second use case relies on restore as a mechanism to scale container instances, where the restore performance is critical for achieving high responsiveness. We show that with VAS-CRIU, new container instances are restored faster than with CRIU. To summarize, both of these use cases can benefit from availability of large physical memory capacity. However, to fully operate at memory speeds, they require a mechanism such as VAS-CRIU, which removes the unnecessary overheads of existing CRIU implementations.

This paper makes the following contributions:

- We present the implemention of the support for multiple virtual address in the Linux Kernel (Section 3.2).
- We present VAS-CRIU, a new in-memory address space snapshot/restore mechanism for containers (Section 3.3). By operating on native address space abstractions (i.e., page tables) and by avoiding unnecessary file system operations, VAS-CRIU acheives two orders of magnitude savings in the time to snapshot/restore memory pages.
- Experimental evaluations of the VAS-CRIU-based snapshot and restore mechanisms in the context of supporting fine grained snapshotting of containers demonstrate that VAS-CRIU results in 9-10× reduction in overall snapshot/restore time.

2 BACKGROUND

Before providing detail on the design and benefits of VAS-CRIU, we provide a brief overview of the existing mechanisms available in Linux through the CRIU utility (Checkpoint Restore In Userspace), and on the original design of MVAS.

2.1 CRIU

CRIU is an application transparent snapshot/restore tool in userspace for Linux. The CRIU project has extended the mainline Linux kernel with system calls that support snapshot/restore. CRIU stores application snapshots as protobuf image files in a filesystem. CRIU is also used for migrating applications (e.g., Docker containers) across

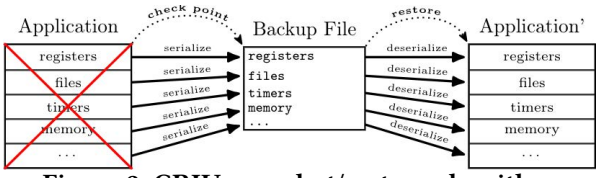


Figure 2: CRIU snapshot/restore algorithm

nodes in a network. However, in the context of this paper, we are only concerned with the use of CRIU for snapshot/restore on a single node.

The basic CRIU operations are:

- **Snapshot** a process into an image file in $CWD: `criu dump -t PID -vvv -o dump.log`
- **Restore** a process from an image file from $CWD: `criu restore -vvv -o restore.log`

Snapshot procedure. CRIU takes the PID of the application to be snapshotted and freezes the entire process tree rooted at PID to ensure state information does not change during a snapshot. CRIU's snapshot procedure relies on the `/proc` filesystem to capture the process file descriptor information, pipes, memory maps, etc., and write all of this in a file (as shown in Figure 2). Our main focus in this work is on capturing memory mappings and associated pages, which dominate the cost. The following steps happen during a memory snapshot in standard CRIU:

1. CRIU captures Virtual Memory Areas (VMA) information by parsing `/proc/pid/smaps/` (anonymous process memory) and `/proc/pid/map_files/` (memory mapped files).
2. Bitmap of pages to be written out is created.
3. Pipes for each VMA region are created.
4. Position independent parasite code is injected into the process in two steps using `ptrace(PTRACE_SEIZE, ...)`. First, only a few bytes required for the `mmap` syscall are inserted at the current instruction pointer location. Second, `mmap` allocates memory and loads the parasite code blob, which once instantiated, executes commands sent by CRIU. For snapshotting, the parasite code maps memory segments into a pipe using `vmsplice()`.
5. Using `splice()` these pages are written out from the pipe into image files.

Restore procedure. On restore, CRIU forks and transforms itself into the process it restores. Shared resources are restored once, and later implicitly shared upon `fork()`. Since CRIU is transforming itself, it restores all resources except for memory mapping, timers and threads. To restore memory, a new context is required, which is provided by a restorer code blob. This is also position-independent, similar to the parasite code. The restorer code blob is loaded at an address which does not conflict with the current CRIU memory mapping and the memory mapping of the process being restored. CRIU jumps into the restore context and does the following:

1. Unmaps all CRIU memory mappings.
2. Performs virtual dynamic shared object (VDSO) proxification. This enables CRIU to make appropriate modifications if a restore is made on a different kernel version, such as on another machine, since the VDSO ELF object could have changed.
3. Creates new memory mapping for the restored process.

Limitations in CRIU. There are two design limitations of CRIU which affect efficiency and reliability of restores. First, during restore CRIU cannot `mmap` snapshotted VMAs in place since they could conflict with existing CRIU mappings. Instead, CRIU allocates a large contiguous buffer which is later remapped into different VMAs by the restorer blob. This adds complexity and time overhead in the restore code path. Second, CRIU assumes a child process inherits anonymous private mapping if the start and end addresses of a VMA are the same. However, this approach fails if the VMA has been remapped.

2.2 Original MVAS

MVAS [4] was designed for large memory-centric computing [5] addressing more physical memory than even the size of the virtual address space. Conventional methods using `mmap()` and `munmap()` become inefficient for saving pointer-based data structures across process lifetimes and for providing shared memory between processes. To overcome these limitations, MVAS provides virtual address spaces as first-class citizens which live beyond the lifetime of processes and can be dynamically attached, detached and switched into by any process with appropriate permissions. The original work was prototyped in FreeBSD and Barrelfish.

For VAS-CRIU we leverage a Linux MVAS kernel [23] to quickly create and restore memory as separate virtual address spaces. In particular, we leverage the following functions of Multiple Virtual Address Spaces.

`vas_create(name, access mode)` creates a new VAS with the given name and access mode. The VAS will belong to the current user and the user's main group (uid and gid).

`vas_attach(pid, vid, type)` attempts to attach the VAS with the given `vid` to the process with the given `pid`. The `type` defines how the attached VAS can be used. Allowed values are read-only attaching or read-write.

Other functions, such as `detach`, `find`, and `delete` are straightforward. We do not use `switch` in this work [4].

3 DESIGN AND IMPLEMENTATION

We summarize the assumptions that underpin the design of our solution, and provide details for the MVAS-specific extensions in Linux and the implementation of VAS-CRIU.

3.1 Design Assumptions

We have made the following design assumptions while developing VAS-CRIU.

Snapshot all process state in DRAM, leveraging VAS for the process address space. Fast snapshotting can be achieved by snapshotting memory state stored in DRAM. In the future, we expect to use NVM to persist VASes across reboots/system crashes. For now, we choose fast snapshot in memory over persistent snapshot on disks/SSDs. We argue that there are many important use cases where the ability to perform fast in-memory snapshots will significantly impact performance. Saving memory pages into files is expensive in terms of performance, even with fast access to a storage medium. We show this by a relatively poor performance of CRIU snapshot/restores over ramfs, as compared to VAS-CRIU. The cost of funneling snapshot-writes and restore-reads through the VFS layer is high.

Optimize both snapshotting and restore in support of our use cases. In traditional use of snapshot/restore, snapshot time is more important than restore time due to a higher frequency of snapshots and rare use of restore, typically on failure. However, the motivating use cases for in-memory snapshotting of containers, such as scheduling, memory-to-memory migration, debugging, etc., demand both – fast snapshot and fast restore. VAS-CRIU maintains memory snapshots in DRAM which can be easily mapped into the restore process address space.

Trade-off COW costs for snapshot/restore time. COW costs affect application execution due to copying pages on write. However, we focus on applications where these costs are not critical. For example, this may be unacceptable for high performance applications where any noise during application execution is detrimental to performance. For the use cases we focus on this is an acceptable trade-off.

3.2 Linux MVAS

VAS-CRIU depends on an implemention of MVAS support for Linux. MVAS is open sourced, available at [23], and being discussed for integration in the mainline Linux. We introduced several new data structures to manage the necessary MVAS data.

One of the new data structures is `struct vas` which is the kernel's representation of an address space created with the MVAS kernel feature. This data structure is allocated in the kernel every time a program wants to create a new VAS. Since these address spaces are first class citizens in the OS (i.e., they are not bound to the lifetime of the process, or even operating system if stored in non volatile memory), they have to be managed independently in the kernel. To achieve this, we are using a radix tree indexed with the id of the VAS. This radix tree is a global data structure in the kernel and access to it is protected using the Read-Copy-Update mechanism (*RCU*). Hence, the frequently happening read accesses do not suffer any bottlenecks.

The second data structure we introduced represents an address space that is currently attached and used by a process – `struct att_vas`. Since this data structure is actually bound to the lifetime of a process and the VAS, it is not managed in a global list, but instead in a list per VAS (`vas_link`). For performance reasons, the `struct att_vas` data structure is additionally also managed in a second per-process list (`tsk_link`). Having the data structure added to two lists allows easier search for a VAS attached to a process and for which processes currently use a specific VAS.

The third data structure we introduced into the kernel is `struct vas_context`. This data structure is added to the in-kernel representation of a process (`struct task_struct`) and contains the list of all VASes that are attached to a process. It allows sharing of attached VASes between `struct task_struct` data structures. The main reason for this additional indirection in the kernel is that in Linux, `struct task_struct` represents both processes and threads. However, we wanted VAS to be attached to all threads of one process and not only to the particular thread that performed the attach operation. Hence, the information which VASes are attached to a process is not directly kept in the `struct task_struct` but instead in a `struct vas_context` to which the `task_struct` only has a pointer. This design facilitates easier sharing of information about attached VASes among all threads of one process.

In Linux, segments have been deprecated based on feedback from the Linux community. As a result, the Linux MVAS implementation uses lazy attach, to defer copying PTE entries until a page fault. A few other optimizations were introduced in support of `switch` (e.g., no automatic syncing of common areas) which are less relevant for VAS-CRIU.

3.3 VAS-CRIU Implementation Details

During a snapshot, CRIU freezes the application to save state, such as CPU registers, file descriptors, pending signals and memory onto files. As shown in Section 4,

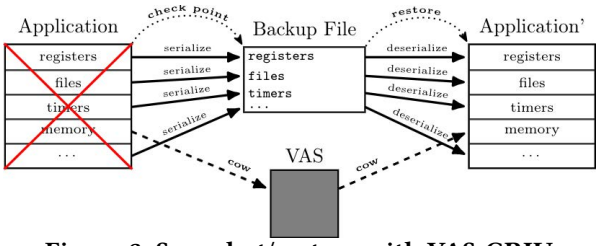


Figure 3: Snapshot/restore with VAS-CRIU

Table 1: System calls introduced in VAS-CRIU.

Name	Parameters	Description
`vas_snapshot`	`pid`	Creates a Copy-On-Write VAS based memory snapshot of a given process. Upon success it returns a vid of the newly created VAS. Else returns negative integer.
`vas_restore`	`pid, vid`	Restores memory mapping of a particular process, based on a given VAS. Returns 0 on success.

for most applications the time to snapshot memory dominates the total snapshot time, even for in-memory files via ramfs.

The goal of VAS-CRIU is to reduce the memory snapshot/restore time by snapshotting process memory in DRAM. We use MVAS to acheive this. VAS-CRIU is based on CRIU and requires kernel support for MVAS. Figure 3 illustrates that VAS-CRIU defers to the original CRIU implementation for dealing with non-memory related states.

VAS-CRIU eliminates the `vmsplice()` and `splice()` steps from CRIU, and snapshots the current application memory state into a process independent address space which lives in DRAM. This address space is created in a Copy-On-Write (COW) fashion, which reduces memory snapshot time by orders of magnitude.

We introduce the new functionality in CRIU using two new operations, `vas_snapshot` and `vas_restore`, summarized in Table 1. In the CRIU snapshot code path, after freezing the process, we call `vas_snapshot(pid)` to create a VAS snapshot. With this approach, there is no need to send a memory snapshot daemon command to the parasite code in the snapshotted process. Commands sent to the daemon are of much higher costs as compared to a simple `vas_snapshot()`.

For VAS restore, in the restorer blob, after unmapping the CRIU mappings, we simply call `vas_restore()`. We just copy the VMA data structures for all regions, except for the VDSO and VVAR regions. VAS-CRIU eliminates the two limitations of CRIU highlighted in Section 2. Since VAS-CRIU captures the entire VAS, including all VMAs, there is no need to allocate and remap memory on restore, and anonymous page mappings are correctly inherited.

VAS-CRIU snapshot. Figure 4 illustrates the behavior of VAS snapshot. VAS-CRIU pauses the snapshotting process and calls `vas_snapshot()` with its PID. The kernel checks if the calling process has the appropriate permissions and valid parameters and creates a VAS snapshot. A VAS is an address space that is not tied to any process. Each VAS is primarily consisting of the `mm_struct` data structure, which has a listing of all mapped VMAs and associated page tables. The kernel copies these data structures from the process into the VAS, instead of actually copying the pages. The PTE for the VAS and the process memory mapping are marked COW. On the left-hand of the figure, we have a process which is being snapshotted. On `vas_snapshot()` new address space data structures are created for the process, and the PTEs are shared in a COW manner. Subsequent writes will create new PTEs. Subsequent snapshots behave similarly, continuing to share address space state with the snapshotted process in a COW manner.

VAS-CRIU restore. Figure 5 illustrates the steps involved in a VAS restore. The left part of the figure illustrates a process with two previous snapshots, i.e., two VASes. On a debug trigger, the process is to be restored from the earlier snapshot, VAS 1. During the restore, CRIU transforms itself into the snapshotted process using VAS 1. Like in the case of standard CRIU, memory is restored at the end of the restore procedure. As with CRIU, the process has all other resources, such as file descriptions and network connections, in place when memory is finally restored. Similarly to standard CRIU, we load the position independent restorer blob at an appropriate place. Before loading the restorer blob, we unmap all of the previous mappings of CRIU. From this blob we call `vas_restore(0, VID1)`. Since the PID is set to 0 for the first argument, it will transform the memory mapping of the calling process. The second argument – corresponding to the VAS ID – is used to identify the memory snapshot to restore from. Similarly to the VAS snapshot procedure, the kernel copies the `mm_struct`, associated virtual memory areas and page tables from the VAS to the new process' memory map. This achieves the same COW sharing as in the snapshot procedure. We notice that the restore blob remains in CRIU's address space after `vas_restore()`. It is unmapped eventually at the end of the restore procedure.

Why not use fork? `vas_snapshot` duplicates page tables in a Copy-On-Write fashion similar to fork. However, fork replicates only the state of the calling thread, not of all threads in the process. In our approach we defer to facilities provided by CRIU to save all thread states and optimize memory snapshotting using MVAS.

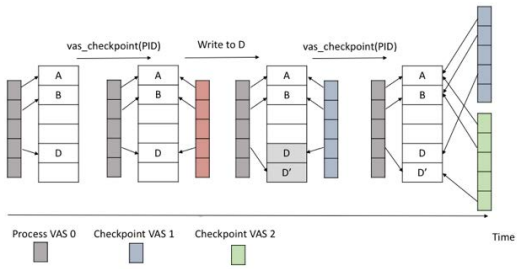


Figure 4: VAS snapshot and COW mechanism

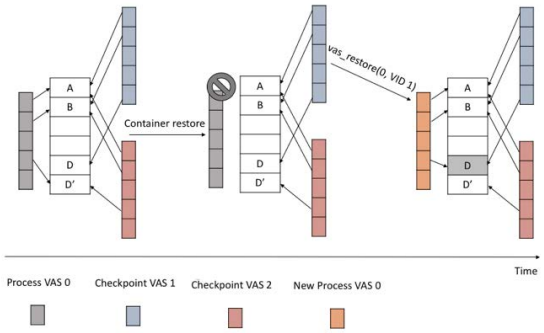


Figure 5: VAS restore mechanism

3.4 Limitations

In the current implementation, VAS-CRIU can only snapshot VASes into DRAM to create in-memory snapshots of the execution. Since these VASes are not flushed to persistent storage, they do not survive system reboots/crashes.

However, for the previously mentioned use cases such as debugging, scheduling and scaling: a) persistence is not essential and we trade fast restore from DRAM for persistence. Hence, snapshotting into DRAM is acceptable; and b) the technology trend towards the use of NVRAM as operating memory is consistent with our approach, our design and implementation will be applicable to snapshotting in NVRAM in the future.

Writes after snapshot/restore cause page faults since the VASes and the process COW-share page frames. We evaluate the effects of this in the following section.

4 EVALUATION

Next, we present the experimental methodology and the results from the evaluation of VAS-CRIU. Our goal is to provide insights into the following questions.

- What are the gains in memory and total snapshot/restore time provided by VAS-CRIU compared to CRIU?
- What is the runtime application performance and Copy-On-Write overhead post VAS-CRIU container restore?
- How does VAS-CRIU contribute to enhacing memory-centric services such as fine-grained in-memory snapshot generation?

4.1 Experimental Methodology

In all experiments we compare the performance of VAS-CRIU with the default implementation of CRIU over ramfs. We choose ramfs over tmpfs, which is also an in-memory filesystem, because ramfs provides no swapping of pages; therefore all file content is guaranteed to be in memory. In the measurements for VAS-CRIU, all state other than the address space memory snapshot, is also stored in ramfs.

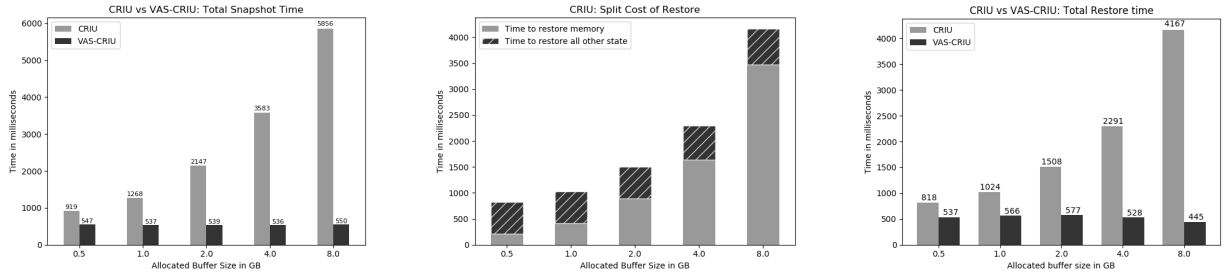
For both the CRIU and VAS-CRIU configuration, we measure the time required to create an in-memory snapshot of the address space, and the total direct costs associated with the snapshot and restore operations, which result in a stall in the application execution. For the application benchmarks, we also measure the indirect costs resulting from the COW effects, and discuss the aggregate impact of these costs on the application execution time and on the ability to perform high-frequency snapshots.

Testbed. All experiments are performed on a dual-socket 12 core 2.67 GHz Intel® Xeon® CPU X5650, with 48 GB memory. VAS-CRIU runs only on a Linux kernel with support for MVAS. Currently, our implementation uses the MVAS kernel based on 4.10. VAS-CRIU itself is based on CRIU v2.12.1 (stable release). VAS-CRIU is linked with *libmvas* [22], a userspace library which provides wrappers for the MVAS system calls (`vas_create`, `vas_attach`). We use the *mvas* [21] CLI client to enumerate VASes.

4.2 Microbenchmark-based Evaluation

Snapshot/Restore microbenchmark. To evaluate the factors that impact the performance of VAS-CRIU, we developed a synthetic microbenchmark with the following behavior. The application allocates a specified amount of memory, touches every page, and waits for snapshot to begin. After a snapshot completes, the original application is killed. The restore operation completes (using CRIU or VAS-CRIU), and the snapshot and restore timing information is returned.

CRIU vs VAS-CRIU (over ramfs). `vas_snapshot()` reduces the time to create an in-memory snapshot of an address space by up to two orders of magnitude. These gains translate into speedup of the snapshot and restore times, shown in Figures 6a and 6c. Figure 6b shows the high cost incurred by CRIU in reading memory pages



(a) Comparison of total snapshot time CRIU vs VAS-CRIU

(b) CRIU: Split cost of restore time

(c) Comparison of total restore time CRIU vs VAS-CRIU

Figure 6: Comparison of CRIU vs VAS-CRIU when snapshotting an application with different memory working set size.

from a filesystem. In contrast, in the experiments performed for Figure 6c, VAS-CRIU incurs less than a millisecond, and no more than 5ms, for restoring the container address space. The remaining restore time costs are common to both CRIU and VAS-CRIU. The results show nearly an order of magnitude lower execution time for these operations in VAS-CRIU compared to CRIU. In addition, unlike CRIU, they are almost constant with respect to increase in the address space size; hence the benefits of VAS-CRIU can increase for larger address spaces.

The trade-off in achieving these speedups is that in VAS-CRIU the COW costs affect the application execution time after a snapshot or restore operation. We evaluate the COW effects of VAS-CRIU in the following section using the application benchmarks.

4.3 Evaluation with Application Kernels

Docker benchmark. To compare the performance of CRIU and VAS-CRIU for snapshotting Docker containers we use the K-means and WordCount applications using Metis, an in-memory MapReduce library designed for multicore architectures [14]. We use the applications provided with Metis.

Using these applications we designed the following benchmark. We built distinct Docker images for each application input size. Application input size determines the memory footprint of the container processes. For each Docker image built we copy the application executable file and docker-entrypoint script into the container's filesystem. For WordCount, we copy the input file as well. Once the container is launched it executes only the MAP phase. Next, the container is snapshotted, and all relevant files, depending on using CRIU or VAS-CRIU, are written into ramfs.

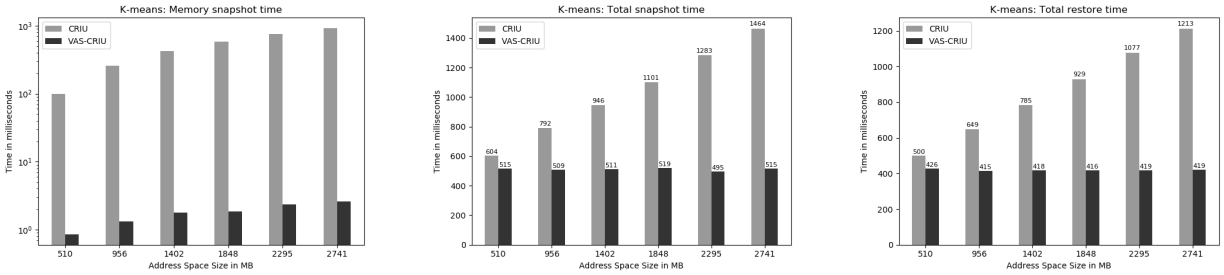
In the experiments measuring the direct costs of the snapshot and restore operations, snapshot results in stopping of the container. Next, we restore the snapshotted container using the latest snapshot, and signal it to continue executing the REDUCE phase. After the container has finished execution we collect the statistics, and delete the snapshot image files before the next test iteration. For evaluating the COW effects associated with VAS-CRIU, we use a similar benchmark configuration, however we do not stop the container; instead after a snapshot completes, the application continues its execution (e.g., with the REDUCE phase).

Although CRIU snapshots are application transparent and can be issued anytime, for our purposes we choose to snapshot after the MAP phase. This approach allows application execution state and address space sizes to be comparable across runs. Address space size impacts the snapshot and restore time, whereas execution state determines the number of future writes, resulting in COW overhead.

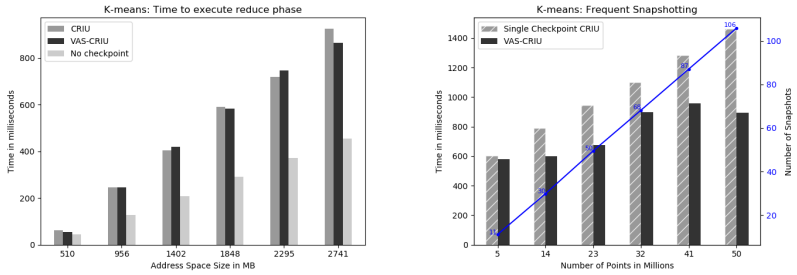
At the time of the snapshot we measure the total address space of all snapshotted processes. CRIU writes a `mm.img` image file for each process which contains a list of VMAs. Each VMA has a start and end address. We add the size of each VMA to obtain the total address space size.

K-means. K-means creates 16 clusters using 10 vector dimensions for a given number of points. We increase the number of points from 5M to 50M to increase the memory footprint of the application. Figure 7a shows the time to snapshot only memory pages after the MAP phase. This increases linearly for CRIU. In contrast, VAS-CRIU has almost the same memory snapshot time for the given set of address space sizes. Figures 7b and 7c show that the gains in memory snapshot time translate to $1.1\times$ to $3\times$ benefits in the total snapshot and restore time for K-means with CRIU vs VAS-CRIU.

The reduction in the direct overheads associated with snapshot and restore in VAS-CRIU leads to consequent runtime costs due to COW page sharing. We quantify the effects of this with the following measurements.



(a) Memory pages snapshot **(b) Container snapshot time** **(c) Container restore time**



(d) Execution of REDUCE phase **(e) Frequent snapshotting**

Figure 7: K-means: Comparison of CRIU vs VAS-CRIU

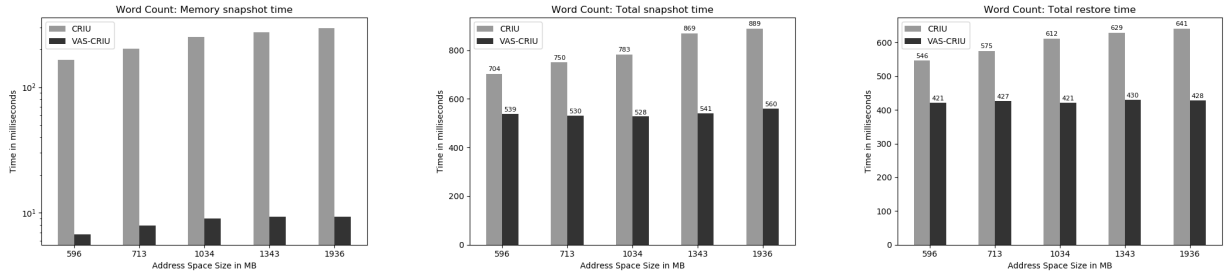
First, once the container has been restored, we allow it to continue executing. Figure 7d shows the execution time of the REDUCE phase in K-means. This time includes the COW overhead of handling page faults and copying the actual page data. We observe that for K-means the execution time is comparable to that of CRIU. However, for VAS-CRIU, the reduction in restore time alone is up to 2× larger, reducing the overall impact of restore by up to 42.3 % compared to CRIU, for this application.

To further justify the COW-related overheads, we perform the following experiment. We execute K-means for each problem size while asynchronously triggering periodic snapshots. On each snapshot, `vas_snapshot` creates a VAS, and the application execution continues. When the application completes its execution, we report the total number of generated snapshots. For this experiment, we set the frequency of the snapshots at an aggressive 100 ms value.

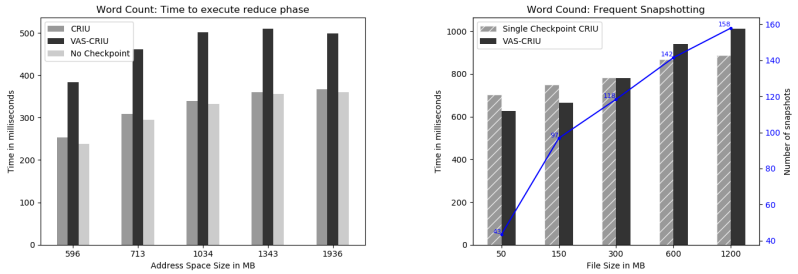
The numbers above each of the bars in Figure 7e correspond to the number of snapshots generated with VAS-CRIU for K-means at each problem size. We do not report the number of snapshots with CRIU for two reasons. First, the default implementation of snapshot in CRIU does not support incremental snapshots, instead the application is stopped and then restoreed. Furthermore, after even a small number of subsequent snapshots, the CRIU experiment terminates prematurely due to lack of memory. In contrast, with VAS-CRIU, K-means was able to complete for all inputs, while also generating 11 to 105 incremental snapshots, depending on problem size.

The total execution time of the application when triggering high-frequency snapshots substantially exceeds the time without snapshots. For brevity, we do not report the raw numbers, but we measured it to be 2× to 5× slower, where 5× slowdown corresponds to generating 105 snapshots during execution. Instead, the height of the bars in Figure 7e corresponds to the normalized overhead on execution time per snapshot. Each value is computed as the difference between the execution time with and without snapshots, divided by the total number of snapshots generated during the application run. Since we cannot compute this value for CRIU, as a reference we plot with dashed lines the measured snapshot time only for CRIU(from Figure 7b). Note that this dashed line is idealistic, as it does not capture the runtime overheads from repeated CRIU snapshots. Regardless, for all cases of K-means, the effective per-snapshot cost is substantially lower for VAS-CRIU vs. CRIU. This illustrates the value of VAS-CRIU for fine-grained, high-frequency incremental snapshots, which can be useful in debugging, migration, and other container management services.

WordCount. The second Metis application we use in the evaluation of VAS-CRIU is WordCount. It returns the top five words in a text file taken as input. Unlike K-means, which iterates over the dataset repeatedly, WordCount has a spread out memory access pattern. Hence, it is expected to have a higher COW overhead



(a) Memory pages snapshot **(b) Container snapshot time** **(c) Container restore time**



(d) REDUCE phase execution **(e) Frequent snapshotting**

Figure 8: WordCount: Comparison of CRIU vs VAS-CRIU

than K-means for similar address space size and snapshot intervals. We increase the size of the text file from 50 MB to 120 MB to increase the memory footprint of the application. Figure 8a shows the memory snapshot time for CRIU and VAS-CRIU. Again we see that the memory snapshot time of VAS-CRIU is two orders of magnitude less than CRIU. Figure 8b shows the total snapshot time using CRIU and VAS-CRIU. We see that the saving in memory snapshot time positively impacts the total snapshot time. VAS-CRIU's total snapshot time remains almost constant for the given set of address space ranges. Similar observations can be made for the total restore time, shown in Figure 8c.

Figure 8d shows the COW overhead for the REDUCE phase for WordCount. Although VAS-CRIU pays an extra 70 ms for the largest address space size (1936 MB), when considering the 200 ms to 400 ms gains in snapshot or restore time, compared to CRIU, it still provides a net gain in execution time of 15.8 %.

Figure 8e shows the measurements from performing high-frequency snapshots on WordCount, at 100 ms periods. As in the case with K-means, VAS-CRIU makes it possible to perform such fine-granularity incremental snapshots, generating up to 157 snapshots during the application execution for the largest problem size. With CRIU, this is not possible. This highlights the utility that VAS-CRIU provides for services such as debugging, which can benefit from frequent snapshots. Unlike K-means, the results for WordCount show that the effective per-snapshot cost is higher, when compared to the

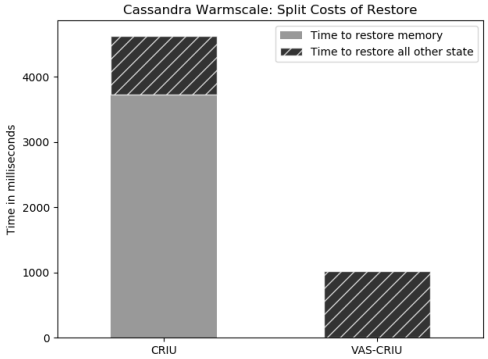


Figure 9: Cassandra total restore time: CRIU vs VAS-CRIU

CRIU snapshot time (in an ideal case, not when considering possible effects of concurrent snapshotting and execution). We suspect that the ratio of the triggered COW events relative to the overall write count for this application is such that it makes the COW overheads significant for this scenario. We plan to perform a more detailed characterization of the write access pattern of the applications to further understand how to exercise the trade-offs.

4.4 COW overhead and Post-restore performance

For the high frequency snapshotting use case, the COW effects are observed continuously, since the application proceeds executing as snapshots are generated, thus triggering copies on each write. However, in other use

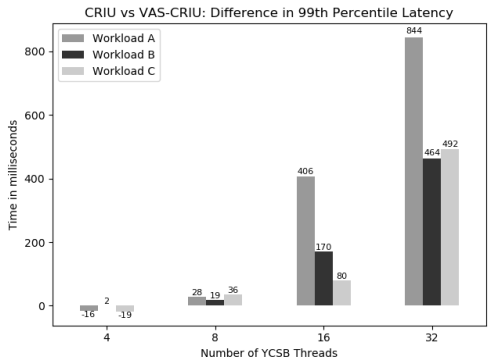


Figure 10: Post-restore YCSB: Difference in 99th-Percentile Latency between VAS-CRIU and CRIU

cases, the COW costs appear only once an application is restored, and are amortized during the application execution time. Once page copies are created, the application execution proceeds at the original performance level. Moreover, the reduction in the overall restore time, can still lead to net benefits.

To demonstrate this, we construct a use case where snapshot restore is used as a mechanism to quickly create new container instances, as needed to scale an application to handle higher load. We use Cassandra [1], a popular open source object store, configured as an in-memory cache. A new docker instance of Cassandra is loaded with 190MB of data using YCSB. There are 50K rows each having 10 fields with 400 byte elements. All in-memory Cassandra rows are flushed to permanent storage. After this the instance is snapshotted using CRIU or VAS-CRIU. A new Cassandra container is restored using the snapshot of the earlier instance. The total size of the container address space is about 6.5 GB. Figure 9 shows the total process restore time of Cassandra. VAS-CRIU provides 3.6 s saving in total restore time against CRIU.

We run YCSB workloads post restore to measure the impact of COW on the Cassandra data serving performance. YCSB and Cassandra containers were pinned to cores on separate sockets. The number of YCSB worker threads was throttled and we show results for 32 and 16 threads. Figure 10 shows the difference in the 99th percentile latency of CRIU and VAS-CRIU. For the write-heavy workload A involving 32 threads, we see at most 800 µs of additional delay in the immediate post-restore request processing time in the case based on VAS-CRIU compared to CRIU. However, the difference in average latency is only 22 µs. The additional delay is due to the Copy-On-Write overheads of the initial write operations. The pre initialized snapshot does not have any application cache since no workload has been run. Therefore, both CRIU and VAS-CRIU will encounter the similar demand paging overhead. Hence, the performance measurements are comparable.

To further understand the impact of using COW in VAS-CRIU, we measure the time series response latency of YCSB workloads. The X-axis shows time in seconds and Y-axis shows average latency in microseconds for the last one second. Figure 11 shows that the significant gains made in container restore time outweigh the runtime performance overhead of COW. These figures show that VAS-CRIU based container starts responding after 2.27 s whereas CRIU based container does not respond to client requests until 8.08 s. For all cases, the total time to complete executing the workloads is always better for VAS-CRIU compared to CRIU. We see upto 6.81 s of total runtime gain for these workloads.

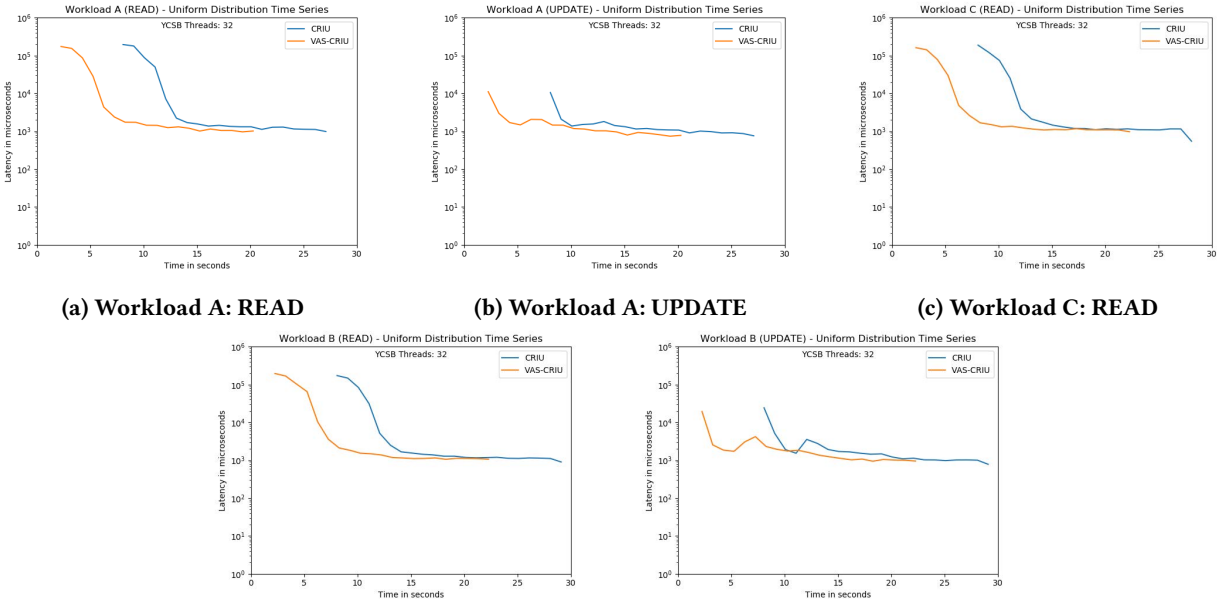
4.5 Lessons Learned

We summarize the following lessons learned:

- By eliminating the VFS overheads and allowing for fast creation of VASes through few kernel-level data structures, VAS-CRIU accelerates the time to create a snapshot of an address space by 1 to 2 orders of magnitude. This translates to a speedup for snapshot or restore operations of 10 % to 200 % for the application benchmarks used in the evaluation, and up to an order of magnitude for the synthetic microbenchmark. Such speedup can accelerate services which rely on snapshot/restore operations.
- VAS-CRIU trades the speed of snapshot/restore for runtime overheads due to COW. We demonstrate that the COW-related overheads can be outweighed by the reduction of the snapshot or restore time from the critical path, leading to a net gain of up to 42.3 % in execution time.
- Finally, VAS-CRIU simplifies the design of process memory snapshotting. VAS-CRIU handles process memory as an address space instead of treating them as individual memory segments. With this VAS-CRIU addresses certain limitations associate with CRIU. VAS-CRIU removes 720 lines of code since there is no need to parse memory maps in `procfs` and write individual page frames into a filesystem.

5 RELATED WORK

Page granular COW based snapshot/restore is accomplished by `fork()` based approaches [29][19][7][24][17]. A new child process is created to snapshot a process state. Fork based snapshotting mechanisms suffer from the same COW overhead as VAS-CRIU. However, `fork()` is much slower than `vas_snapshot()` since `fork()` creates a new process control block (`task_struct`) and all associated data structures. When using frequent snapshots, this could cause a bloat of metadata stored in DRAM even for processes with small address spaces.



(a) Workload A: READ **(b) Workload A: UPDATE** **(c) Workload C: READ**

(d) Workload B: READ **(e) Workload B: UPDATE**

Figure 11: Timeseries response latency with 32 YCSB threads: CRIU vs VAS-CRIU

`mprotect()` based approaches have been proposed to provide page-granular COW based snapshot/restore [2] [20][17]. `mprotect()` is used on all memory regions of a snapshotted process. Subsequent writes result in a minor page fault causing a SIGSEV signal. Most `mprotect` based implementations handle this in userspace resulting in high page fault overhead. Further, the snapshotted data is stored in the same address space. Hence, it does not provide strong data integrity for the snapshotted process state. `mprotect()` based approaches reduce COW overhead since they do not have to copy metadata/data for pages which have not changed between snapshots.

Most COW snapshotting mechanisms use write protection provided by the paging hardware. However, this isn't accessible for userspace software. The Linux kernel exports soft dirty bits to userspace via process memory maps exported via `procfs`. Several snapshot/restore methods have been proposed based on soft dirty bits [13] [8] [25]. The soft dirty bit is set for pages which were written during a snapshot interval. Reading soft dirty bits requires scanning of all maps and this does not scale well for large address spaces. Further, it does not provide snapshot data protection.

Undo-log based snapshot/restore approaches use compiler based static or dynamic instrumentation to log all writes into an append-only log [11][18][30]. They store data, size and target addresses of the writes issued after the snapshot interval. Upon a restore request, all updates can be unwound to get the snapshotted memory state. The logs are in the same address space as the snapshotted process, therefore, they do not provide strong snapshot data protection guarantees. Bounds checking writes could be done during instrumented writes. However, this causes extra overhead for every write. Further, most undo-log based approaches suffer from huge memory usage for the log due to duplicate writes to the same location. Undo-log based approaches are unaware of the spatial locality of memory accesses.

6 SUMMARY AND FUTURE WORK

We presented the design and implemention of VAS-CRIU, a new approach to reduce memory snapshot/restore time using process independent address spaces. Both VAS-CRIU and the Linux-MVAS patches are prepared to be open-sourced [23, 28]. VAS-CRIU accelerates the time to create a snapshot of an address space by 1 to 2 orders of magnitude, resulting in overall speedup of snapshot/restore operations of 10 % to 200 %. This is especially relevant for applications and services which heavily rely on snapshot/restore operations, such as scheduling or debugging. VAS-CRIU accomplishes this by trading the speed of snapshot/restore for run-time overheads due to Copy-on-Write. We demonstrated that these COW-related overheads can be justified by reducing the snapshot/restore overhead in the critical path, leading to a net gain of up to 42.3 % in execution time. In addition, Copy-on-Write results in lower memory overhead. For some pathological cases, such as Word-Count, where the ratio of memory writes to COW is unfavorable, the runtime overheads of such high-frequency snapshots can be significant, however VAS-CRIU still

provides benefits over the traditional CRIU implementation by making fast incremental snapshotting possible due to its lower memory demand.

In terms of future work, VAS snapshotting in NVM can provide both fast snapshotting and persistence. We will extend VAS-CRIU with asynchronously flushing snapshot VASes into NVM. The sync with NVM would not increase snapshot time. The DRAM copy can be retained or discarded. We also plan to explore VAS-CRIU for quick launch of Dockerized applications. Applications can be snapshotted into a VAS after their initial startup phase and then quickly restored into new container instances. Use cases we consider are microservices and NFVs.

For write-heavy workloads, COW has a significant runtime overhead. We will explore pre-copying pages, based on write set estimation algorithms, into VAS page tables before the process restore. Our prior research has already demonstrated the practicality of such solutions for different workload classes [6, 10]. In addition, information about hot pages could be stored in the snapshot, and could be used for page pre-copy.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their valuable feedback and comments. This work was partially supported by research grants from HPE, NSF award SPX-1822972 and the German Research Foundation (DFG) within CRC 912.

REFERENCES

[1] Apache. 2019. Cassandra. http://cassandra.apache.org/.

[2] Edouard Bugnion, Vitaly Chipounov, and George Candea. 2013. Lightweight snapshots and system-level backtracking. In *Proceedings of the 14th Workshop on Hot Topics on Operating Systems*. USENIX.

[3] CRIU community. 2019. Checkpoint/Restart in Userspace(CRIU). https://criu.org/.

[4] Izzat El Hajj, Alexander Merritt, Gerd Zellweger, Dejan Milojicic, Reto Achermann, Paolo Faraboschi, Wen-mei Hwu, Timothy Roscoe, and Karsten Schwan. 2016. SpaceJMP: Programming with Multiple Virtual Address Spaces. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 353–368. https://doi.org/10.1145/2872362.2872366

[5] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojicic. 2015. Beyond Processor-centric Operating Systems. In *15th Workshop on Hot Topics in Operating Systems*.

[6] Pradeep Fernando, Sudarsun Kannan, Ada Gavrilovska, and Karsten Schwan. 2016. Phoenix: Memory speed hpc i/o with nvm. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. IEEE, 121–131.

[7] Qi Gao, Wenbin Zhang, Yan Tang, and Feng Qin. 2009. First-aid: Surviving and Preventing Memory Management Bugs During Production Runs. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys '09)*. ACM, New York, NY, USA, 159–172. https://doi.org/10.1145/1519065.1519083

[8] Roberto Gioiosa, Jose Carlos Sancho, Song Jiang, Fabrizio Petrini, and Kei Davis. 2005. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 9.

[9] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center.. In *NSDI*, Vol. 11. 22–22.

[10] Sudarsun Kannan, Ada Gavrilovska, Karsten Schwan, and Dejan Milojicic. 2013. Optimizing checkpoints using nvm as virtual memory. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 29–40.

[11] Andrew Lenharth, Vikram S Adve, and Samuel T King. 2009. Recovery domains: an organizing principle for recoverable operating systems. In *ACM SIGARCH Computer Architecture News*, Vol. 37. ACM, 49–60.

[12] Jack Li, Calton Pu, Yuan Chen, Vanish Talwar, and Dejan Milojicic. 2015. Improving preemptive scheduling with application-transparent checkpointing in shared clusters. In *Proceedings of the 16th Annual Middleware Conference*. ACM, 222–234.

[13] Yawei Li and Zhiling Lan. 2011. FREM: A fast restart mechanism for general checkpoint/restart. *IEEE Trans. Comput.* 60, 5 (2011), 639–652.

[14] Yandong Mao, Frans Kaashoek, and Robert Morris. 2010. *Optimizing MapReduce for Multicore Architectures*. Technical Report MIT-CSAIL-TR-2010-020. MIT.

[15] S. Nadgowda, S. Suneja, and A. Kanso. 2017. Comparing Scaling Methods for Linux Containers. In *2017 IEEE International Conference on Cloud Engineering (IC2E)*. 266–272. https://doi.org/10.1109/IC2E.2017.42

[16] Tom Nolle. 2018. Expect more container evolution, growth in 2019. https://searchitoperations.techtarget.com/opinion/Expect-more-container-evolution-growth-in-2019.

[17] James S Plank, Micah Beck, Gerry Kingsley, and Kai Li. 1994. *Libckpt: Transparent checkpointing under unix*. Computer Science Department.

[18] Georgios Portokalidis and Angelos D Keromytis. 2011. REASSURE: A self-contained mechanism for healing software using rescue points. In *International Workshop on Security*. Springer, 16–32.

[19] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. 2005. Rx: Treating Bugs As Allergies—a Safe Method to Survive Software Failures. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP '05)*. ACM, New York, NY, USA, 235–248. https://doi.org/10.1145/1095810.1095833

[20] Joseph F Ruscio, Michael A Heffner, and Srinidhi Varadarajan. 2007. Dejavu: Transparent user-level checkpointing, migration, and recovery for distributed systems. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. IEEE, 1–10.

[21] Rodrigo Siqueira. 2017. MVAS-CLI. https://github.com/LSS-USP/mvas-cli.

[22] Till Smejkal. 2016. Userspace MVAS library-libmvas. https://github.com/l3nkz/libmvas.

[23] Till Smejkal and Ranjan Sarpangala Venkatesh. 2017. MVAS Linux kernel. https://github.com/ranjansv/MVAS-CP-Linux-kernel.

[24] Sudarshan M Srinivasan, Srikanth Kandula, Christopher R Andrews, Yuanyuan Zhou, et al. 2004. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conference, General Track*. Boston, MA, USA, 29–44.

[25] Manav Vasavada, Frank Mueller, Paul H Hargrove, and Eric Roman. 2011. Comparing different approaches for incremental checkpointing: The showdown. In *Linux Symposium*. 69.

[26] Steven J. Vaughan-Nichols. 2017. What is Docker and why is it so darn popular? *ZDNet.com* (2017).

[27] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. 2013. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 5.

[28] Ranjan Sarpangala Venkatesh. 2017. VAS-CRIU. https://github.com/ranjansv/VAS-CRIU.

[29] Angeliki Zavou, Georgios Portokalidis, and Angelos D. Keromytis. 2012. Self-Healing Multitier Architectures Using Cascading Rescue Points. In *Annual Computer Security Applications Conference (ACSAC)*.

[30] Chuck Chengyan Zhao, J Gregory Steffan, Cristiana Amza, and Allan Kielstra. 2012. Compiler support for fine-grain software-only checkpointing. In *International Conference on Compiler Construction*. Springer, 200–219.