ContainerVisor: Customized Control of Container Resources

Tianlin Li
Computer Science
St. Mary's University
San Antonio, TX, USA
tli2@stmarytx.edu

Kartik Gopalan
Computer Science
Binghamton University
Binghamton, NY, USA
kartik@binghamton.edu

Ping Yang
Computer Science
Binghamton University
Binghamton, NY, USA
pyang@binghamton.edu

***Abstract*—Cloud platforms are increasingly using containers for lightweight virtualization. Unlike full system virtual machines (VMs) that each runs its own operating system, containers share a stateful operating system to reduce their memory footprint and execution overheads. However, mainstream operating systems are currently limited in their ability to customize a container's memory management, since they lack the necessary abstractions and mechanisms to accurately track and isolate a container's memory footprint. We propose a new abstraction, called Container-Level Address Space (CLAS), that provides a unified view of a container's memory across all of its constituent processes. We present the design of *ContainerVisor*, a per-container resource management system that leverages CLAS to provide customized memory management services. We describe a ContainerVisor prototype on Linux for running unmodified applications and demonstrate three proof-of-concept customized services, namely process-level memory limits and reservations, container-specific page replacement policies, and privacy-aware memory de-allocation. Our evaluations show that ContainerVisor can provide these customized services within reasonable overheads.**

***Index Terms*—Containers, Operating System, Virtualization, Cloud Infrastructure**

I. Introduction

Containers [1]–[3] are a lightweight abstraction for isolating a group of correlated processes by limiting the use and addressability of various system resources. A *namespace* refers to the scope of each addressable system resource, such as file systems, process identifiers (IDs), inter-process communication (IPC), network endpoints, and user IDs. For instance, containers in Linux rely on Linux Namespaces [4], [5] to limit resource addressability and Control Groups (Cgroups) [6], [7] to limit resource usage.

All containers in a machine (whether physical or virtual) share a common operating system (OS or kernel), unlike full system virtual machines (VMs or guests) [8]–[10] that each runs its own OS. Generally, multiple containers sharing a common OS consume less system resources and achieve better performance than multiple VMs. Hence containers can be used to run more applications within a single machine.

Despite the above benefits, containers lack some important features provided by full system VMs. First, because containers share a common OS, container-based virtualization does not provide strong isolation. Today's containers provide only namespace isolation and rely on the shared OS to allocate and schedule their resources. For instance, a shared OS may implement uniform memory management policies for all of its containers irrespective of individual application requirements. Also, a security breach in one container that compromises the shared OS also compromises other co-located containers.

Secondly, the ability to easily track an application's memory footprint is necessary for providing a number of services to applications such as guaranteeing memory reservations, enforcing memory usage limits, scrubbing or encrypting confidential data, and performing live migration and checkpointing. A system VM's memory footprint can be conveniently tracked and managed by the hypervisor via a single *guest-physical address space*, i.e. the guest's virtualized view of its physical memory. Typically, a second-level page table [11], such as Intel's extended page table (EPT) or AMD's nested page table (NPT), captures a VM's guest-physical-to-physical page mappings.

In contrast, there is no container-level equivalent of a VM's guest-physical address space that can capture a unified view of a container's entire memory footprint. A container's memory contents are spread across multiple *virtual address spaces* of its constituent processes. Each process is assigned its own separate page table by the OS to track its virtual-to-physical address mappings. This makes it difficult for container-level services to easily and accurately track and manage a container's memory across its multiple processes. For instance, while Linux containers use Cgroups to account and limit a container's memory usage, the underlying implementation requires gathering process-specific information from disparate locations in OS memory, such as multiple process' page tables and system-wide bookkeeping data structures. The memory usage policies enforced by Cgroups are inflexible and uniform across all containers; they do not allow room for customized memory reservation policies for individual containers. Similarly, container checkpointing and migration mechanisms must painstakingly gather a container's page mappings from multiple process-level page tables.

In this paper, we propose *ContainerVisor*, a container resource management system to ease the tracking and management of a container's memory. A traditional OS uses page tables to directly map the virtual address space of each process

to its physical memory. In contrast, a ContainerVisor maps the virtual address space of all processes in a container to a single *Container-Level Address Space* (or CLAS), which is then mapped to physical memory. This new CLAS abstraction provides a convenient single point to track the entire memory footprint of a container and customize its management.

We present the design and implementation of a ContainerVisor prototype in Linux and demonstrate the following proof-of-concept customized services enabled by CLAS.

- **Per-process memory reservations:** ContainerVisor can be used to provide memory reservation (i.e. guaranteed memory availability) for individual processes within a container, besides enforcing memory quotas (maximum limit) as supported by traditional Cgroups.
- **Scrubbing confidential memory after deallocation:** Existing OS mechanisms do not scrub (zero out) deallocated memory pages after a process terminates or otherwise deallocates memory. This leaves the possibility that deallocated pages having confidential data may be reused by other untrusted processes. However, system-wide scrubbing of all deallocated pages can be expensive. ContainerVisor provides a mechanism to scrub deallocated memory pages a container.
- **Per-container page replacement policies:** A traditional OS typically implements a system-wide page swapping policy that may not discriminate between the memory requirements of different containers. ContainerVisor enables one to customize the page replacement policy for each container's application needs over dedicated swap devices.

The rest of the paper is organized as follows. Section II presents ContainerVisor design and the new CLAS abstraction. Section III describes customized services supported by ContainerVisor. Section IV describes the implementation details of a ContainerVisor prototype on Linux. Section V presents experimental evaluation of our prototype's performance and effectiveness. Section VI compares ContainerVisor with related work and Section VII concludes the paper.

II. Design of ContainerVisor

ContainerVisor is a per-container agent that provides customized memory management services to a container. Figure 1 shows the high-level architecture. A ContainerVisor performs two primary functions based on which customized services are implemented. First is to construct and maintain a Container-Level Address Space (CLAS). Second is to intercept and handle memory-related events of interest generated by individual processes. Below, we discuss these two functions.

A. Container-Level Address Space (CLAS)

Each container is associated with a CLAS, which is a virtualized address space representation of the entire memory used by all processes in a container. CLAS enables a ContainerVisor to control the memory usage of individual processes according to container-specific policies. Figure 2 compares

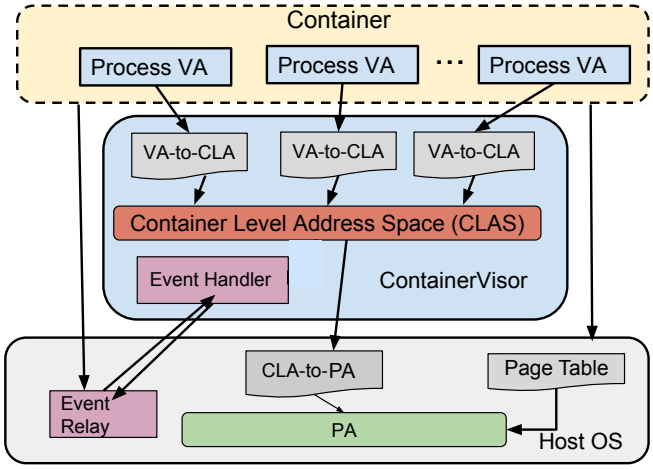

Fig. 1. High-level architecture of ContainerVisor.

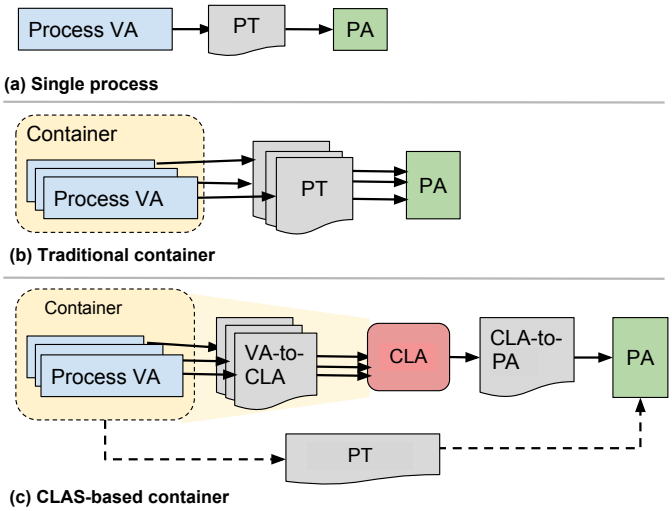

Fig. 2. Memory translation for a single process, a traditional container, and a CLAS-based container. VA: Virtual Address. PA: Physical Address. CLA: Container-Level Address. PT: Page Table.

the memory translation mechanism for a single process, a traditional container, and a CLAS-based container.

For a single process, shown in Figure 2(a), the OS constructs and maintains a page table to translate virtual addresses (VA) to the corresponding physical addresses (PA). The OS uses this page table to manage the memory pages allocated to an individual process whereas the hardware memory management unit (MMU) uses this page table for address translation during execution time without involving the OS.

For a traditional container, shown in Figure 2(b), tracking its memory footprint across its multiple constituent processes is more complicated. Unlike for system VMs, the OS and the MMU hardware do not provide a second-level page table [11] for containers. Instead the OS maintains a traditional page table for each process in a container. Hence, to track and manage a container's memory, the OS must painstakingly gather mapping information spread across multiple page tables. The

drawback of this approach is that the container-wide memory usage may not be tracked synchronously and accurately.

As shown in Figure 2(c), ContainerVisor addresses this limitation by constructing a CLAS – a software-defined container-wide address space. Conceptually, a virtual address of a process in a container maps to a container-level address (CLA), which is then mapped to a physical address. In practice, however, the two-level address translation via CLAS is not recognized by either the OS or the MMU hardware. Hence the ContainerVisor intercepts memory allocation events (page faults) generated by its processes and constructs the VA-to-CLA and CLA-to-PA mappings. The resulting VA-to-PA mapping is then populated in the OS-maintained single-level page table of the faulting process and used by the MMU hardware for address translation. Thus CLAS provides a convenient single point for easy tracking and customized management of all memory pages used within a container. We next discuss the details of memory-event interception and processing by a ContainerVisor.

B. Event Interception and Handling

ContainerVisor intercepts all memory-related events triggered by its processes such as page allocation, deallocation, and sharing, to accurately track the container's memory in the CLAS. An event relay in the OS intercepts and synchronously redirects these events to an event handler in user space. The event handler processes each event by updating the CLAS mapping tables and returns its response back to the OS, which completes updating the traditional process page table. Below, we discuss the details of individual memory events.

Page allocation event: A page allocation event occurs when a process tries to access a valid virtual address in a page that is not currently mapped by the OS to a physical page, either because this is the first access to the page by the process or if a previously swapped out page is accessed again. Upon intercepting a page allocation event, a ContainerVisor updates the corresponding VA-to-CLA and CLA-to-PA mappings, called simply the CLAS mappings.

Process creation and copy-on-write events: When a new process (child) is created by forking an existing process (parent), the OS populates the initial page table of the child by copying the memory mappings from the parent's page table and marks all page table entries as copy-on-write. A ContainerVisor intercepts process creation events in its container to construct the initial CLAS mappings for the child that reflect copy-on-write sharing with the parent.

The ContainerVisor also intercepts write fault events which are triggered when a child or parent subsequently writes to a page that is marked copy-on-write. The ContainerVisor then updates CLAS mappings to reflect the new page allocation followed by copy-on-write processing by the OS.

Page deallocation and replacement events: When a process terminates, its memory pages are deallocated by the OS. A page is also deallocated when an active process explicitly unmaps a page from its memory, such as when the `malloc` library frees memory greater than 128KB. The ContainerVisor intercepts all process termination and memory unmap events from processes in its container and deletes all CLAS mappings of the affected pages.

ContainerVisor also optionally maintains a per-container swap device. When a memory page must be evicted due to memory pressure, the ContainerVisor evicts the page to the container's swap device and updates the page's CLAS mappings to reflect its location on the swap device.

The ContainerVisor also retrieves an evicted page from swap device as a result of a page fault (discussed earlier) or during pre-fetching. In either case, the CLAS mapping of the retrieved page are updated to map to the newly allocated physical page.

III. Customized Services

ContainerVisor provides customized memory management services for each container that complement or augment the system-wide OS-level mechanisms. Below, we present three such use cases of ContainerVisor.

Per-process memory usage limits and reservation: A ContainerVisor tracks up-to-date memory usage of each process running inside a container. Thus, it can enforce memory usage limits, not just for an entire container, but also for individual processes within a container. In addition, ContainerVisor can also reserve memory for important processes within a container, i.e. guarantee minimum availability as opposed to limit maximum usage. This level of intra-container memory allocation granularity differs from traditional Linux Cgroups mechanism which can limit maximum memory usage but not to guarantee minimum availability.

Privacy-aware memory scrubbing: By tracking memory pages of processes within a container, a ContainerVisor can provide container-specific services not typically provided by a traditional OS. For instance, for processes that handle confidential data such as credit card numbers, passwords, and other confidential information, ContainerVisor can scrub (or zero out) deallocated pages that contain such data. Such services can help to limit the lifetime of confidential data for applications running within containers. In addition, a ContainerVisor could potentially use CLAS to perform checkpointing to save a container's execution state, or live migration for system maintenance and load balancing.

Customized page replacement policies: A traditional OS applies uniform page replacement policies for all containers and processes using a common swap device. Such an application-agnostic swapping mechanism can adversely affect critical applications running within containers. ContainerVisor allows for customized page replacement policies for different containers over dedicated swap devices. For example, one ContainerVisor may implement the traditional least recently used (LRU) page replacement policy while another may prioritize evicting pages of less important processes in a container over its important ones.

IV. Implementation

Figure 3 shows the implementation details of ContainerVisor. ContainerVisor consists of a user-level component called

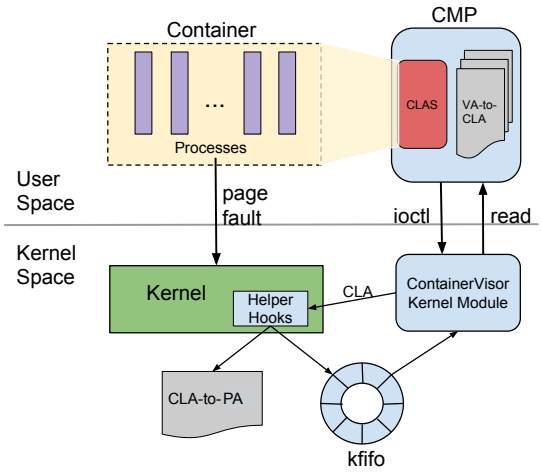

Fig. 3. Components of ContainerVisor.

the *container management process* (CMP), a kernel module, and several helper hooks in the kernel code. Each container is associated with a CMP, which virtualizes the memory address space of a container. CMP received memory-related events from the OS, and updates CLAS mappings. CMP was implemented in the user-level to enable ContainerVisor to more easily implement different memory management policies.

We use an *event redirection* mechanism to track memory pages of a container. The helper hooks in the kernel monitor page faults and other events that need CMP's response. If such an event occurs, the helper hooks redirect the event to the CMP. Once the event is handled by the CMP, the control is returned to the kernel context of the process on which the event occurs. In our current implementation, we use the kernel's built-in data structure *kfifo* (an FIFO buffer) and a kernel module to communicate between the kernel and the CMP. When a page fault occurs, the helper hooks in the kernel write messages into a `kfifo` buffer. The CMP then uses the read operation of the kernel module to read messages from the `kfifo` buffer. The message contains the faulted address, the address of a local variable for storing the CLA value from CMP, the ID of the faulted process, and a flag indicating the event type. The CMP also uses `ioctls` to send event responses from CMP to the kernel or invoke functions in kernel space.

A. Event Handling

A CMP intercepts and processes the following memory-related events that are generated by processes within its associated container.

Page allocation event: ContainerVisor uses the page fault handler to track newly allocated memory pages. Memory pages of an application are allocated through page faults. In traditional Linux operating systems, when a running process needs to access a memory page which is not yet allocated, a page fault occurs and the page fault handler in the kernel is triggered to handle the page fault. The page fault handler then fills the page table entry (PTE) with the page frame number (PFN) of the allocated physical page. For every faulted virtual address va_p received by CMP, if CMP already has a page table entry for va_p, then CMP retrieves the corresponding CLA from the page table. Otherwise, CMP allocates a new CLA cla', and adds record (pid, va_p, cla') to CMP's page table, where pid is the ID of the faulted process. Meanwhile, the process waits in the kernel space until CMP passes cla' to the kernel. Next, CMP uses the $ioctl$ function to trap into the kernel space to pass cla' to the process' kernel context, and the page fault handler continues to run in the kernel. Once the kernel allocates a physical page and fills the PTE of a faulted address in the kernel's page table, the helper hook of ContainerVisor copies the PTE from the process' kernel page table to CMP's page table to fill the PTE of cla', and increments the page map count of the page.

Process creation event: New processes are often created by `fork()` or `exec()` system calls. When a process calls `fork()` to create a child process, function `copy_process()` is invoked in the kernel, which copies the parent process' page table to the child process' page table. The parent and child processes share the memory using copy-on-write (COW), i.e., the page tables of both processes are read-only and both processes share the same address space until one of the processes tries to modify a page. ContainerVisor keeps track of the fork operations performed by a process using hooks in the `copy_process()` function and notifies the CMP once a child process is created. CMP then copies all VA-to-CLA mappings from the parent process' page table to the child process' page table, marks them as COW in CMP, and increases the reference count number for each CLA. When the COW page is modified by either the parent or the child process, the kernel informs the CMP about the write-protection fault. CMP then checks if the faulted address is marked COW. If so, CMP allocates a new CLA cla', maps the virtual address of the faulted page to cla', and decreases the old CLA's reference number by 1. The kernel then continues to handle the page fault and maps cla' to the physical address of the faulted page.

System call `exec()` is handled differently. `exec()` replaces the current process image with a new process image. When `exec()` is invoked, all existing page tables of the current process are destroyed and the new process' address space is loaded. As a result, CMP deletes all VA-to-CLA mappings of the current process, decreases the reference number of the corresponding CLAs, and clears the COW flag of pages that are no longer shared by multiple processes.

Page access change event: When a write protection page fault is triggered, function `do_wp_page()` is invoked to resolve the page fault. If the reference count of the faulted page is more than 1 (which means that the page is shared by two or more processes), then the kernel allocates a new page to resolve the page fault. Otherwise, the kernel simply updates the access permission of the page in PTE. Our ContainerVisor uses helper hooks to intercept `do_wp_page()` and uses

the updated PTE of the faulting page to update the CLAS mappings.

Page reclamation event: When a process terminates, CMP no longer needs to keep track of its memory. As a result, CMP deletes all page table entries related to the process. When a memory page is freed and returned to the kernel, CMP updates its page entries. Because function `do_exit()` is always called when a process terminates, we inserted a helper hook into this function to inform the corresponding CMP about the termination of the process. CMP then zeroes out all memory pages of the process, erases VA-to-CLA mappings from CMP's page table, and deallocates the corresponding CLAs. Similarly, because the function `munmap()` is always called when a memory page is released to the kernel, we inserted a helper hook into this function to inform the corresponding CMP about the deallocation event.

B. Customized Per Container Swapping

We have implemented a user-space customized swapping mechanism in CMP to enforce memory quota and handle page swap-in and swap-out.

Page swap-out: When a container exceeds its memory limit (quota), CMP starts swapping its memory pages out. Currently, we use a `swap_array` in CMP's userspace as a swap device. When a page needs to be swapped out, CMP saves the content of the page in the array, records the corresponding index, and stores the tuple $(pid, va, index)$ in the CMP, where pid is the ID of the process whose page is swapped out, va is the virtual address of the page, and $index$ is the index in the array where the page is stored. CMP then deletes the corresponding VA-to-CLA mapping in its userspace page table and updates the memory usage. CMP also traps into the kernel to update the page tables of the process whose page has been swapped out, locates the corresponding PTE entry in the page tables of CMP and the process, deallocates the page, and sets the corresponding PTE as non-present.

Page swap-in: If a page fault occurs and the corresponding PTE is not empty, then the faulted page has been swapped out. In this case, the kernel invokes function `do_swap_page()` to swap in the page. PTE contains the offset of the page in the swap device and kernel uses it to locate the page. ContainerVisor handles page swap-in in a different way. Instead of having the kernel to handle the swap-in operation, ContainerVisor transfers the control to CMP and sends the process ID pid and the faulted address va to CMP. CMP then obtains the index of the page from `swap_array` and retrieves the page content based on the index. Next, CMP allocates a new CLA cla' for the faulted address va, adds the mapping (pid, va, cla') to CMP's page table, reads the page content from `swap_array` into cla' using function $memcpy()$, and deletes the corresponding record in `swap_array`. The kernel then allocates a new physical page for the page being swapped in. Next, CMP traps into the kernel using `ioctl` to pass cla' to the kernel. The control is then returned to the faulted process' kernel context. Finally, the helper hook of ContainerVisor extracts the PFN from the PTE of cla', and fills the PTE of va with the PFN and the PRESENT flag.

Handling COW memory: A page being swapped out may be shared by multiple processes. In such a case, the kernel uses a reverse mapping mechanism to locate all PTEs of processes that share the page, and changes their status from present to swapped-out.

We have implemented our own reverse mapping table to facilitate efficient lookup of process IDs and virtual addresses. When a fork operation is performed, CMP copies all VA-to-CLA mappings from the parent process' page table to the child process' page table, increases the CLA reference number, and inserts mappings $(cla, ppid, va)$ and $(cla, cpid, va)$ into the reverse mapping table for every cla of the parent process, where $ppid$ is the ID of the parent process, $cpid$ is the ID of the child process, and va is the virtual address of the page that is mapped to cla.

When a write protection fault occurs or when function `exec()` is invoked, CMP updates the corresponding entry in the reverse mapping table (e.g. deletes a reverse mapping if the corresponding page is not COW anymore). When a COW page is swapped out, CMP uses the reverse mapping table to get the ID of the process that owns the page and marks the present bit of the corresponding PTE in all page tables as swapped-out. When a page is swapped in, CMP not only handles the swap-in operation of the corresponding processes, but also updates the VA-to-CLA mapping with new CLA and update PTEs with the new PFN for all other processes that own the page.

C. Handling Multiple Processes and Multi-threaded Programs

Multiple containers may run on the same host machine and the host machine should forward only page faults of processes inside the target container to CMP. Because each container has a separate PID namespace from the host and processes within the same PID namespace have the same inode numbers, we use inode numbers to identify processes running inside a container. When a container starts, the CMP passes the inode number of processes running inside the container to the kernel. When a page fault occurs, the kernel checks if the inode of the faulted process is the same as that of the target container. If so, the kernel redirects the page fault to CMP; otherwise, the kernel handles the page fault as normal.

Multi-threaded programs are handled similarly as non-multi-threaded programs except that functions `fork()` and `exec()` are handled differently. When function `fork()` or `exec()` is called to create a process or a thread, function `do_fork()` is invoked in the kernel. To distinguish process creation and thread creation, when `fork()` is invoked, ContainerVisor compares the parent's TGID and the child's TGID, and if they are the same (which means that a thread is created), then CMP does not copy the mappings from the parent's page table to the child's page table. Otherwise, CMP copies the parent's mappings to the child's page table. When a process or a thread terminates, function `do_exit()` is invoked. In this case, ContainerVisor compares the current process' TGID

and PID. If they are the same, then it is a process termination; otherwise, it is a thread termination.

D. Memory Scrubbing

It is important to scrub memory pages that contain confidential data in order to reduce the lifetime of such data. However, scrubbing all deallocated pages is expensive. To address this issue, we implemented an application programmer interface (API) `void *Palloc(size_t size)`, which enables programmers to allocate a piece of memory to store confidential data. `Palloc()` records the start and end virtual addresses of confidential memory region. When a process terminates, CMP scrubs only confidential memory region allocated using `Palloc()`. In addition, if a page that has been swapped out belongs to a confidential memory region, then the page is also scrubbed as soon as the swap-out operation is complete.

V. Evaluation

We now evaluate the effectiveness and performance of ContainerVisor. All experiments were conducted on a host system with 16-core Intel Xeon 2.27GHz processor and 70GB of RAM, running Ubuntu 14.04 with Linux kernel 4.4.2. When measuring the performance overhead of ContainerVisor over a specific process, the process is assigned a separate PID namespace.

A. Effectiveness of Tracking Container Memory

We first evaluate whether ContainerVisor accurately tracks memory pages used by all processes in a container using CLAS. We compared the set of memory pages tracked by ContainerVisor against a popular tool called Checkpoint/Restore In Userspace [12] (CRIU v2.2), which is used to checkpoint a container. Unlike ContainerVisor, however, CRIU does not actively track a container's memory footprint during its execution or provide customized memory management services.

We created a thread in the CMP to gather virtual addresses collected by CRIU during container checkpointing. We also modified the CRIU code so that when it checkpoints the memory pages of a container using function `page_xfer_dump_page()`, it establishes a TCP socket connection with the CMP thread and sends all virtual addresses of the checkpointed memory pages. The CMP thread then compares the virtual addresses received against those collected by CMP on its own.

We checkpointed an idle LXC container on which 16 default system processes were running and several busy containers running various workloads such as Parsec [13], Sysbench [14], and the `vim` text editor. Our comparison showed that ContainerVisor accurately tracked all memory pages of the above containers. We were also able to successfully restore all processes in these containers using CRIU's restoration mechanism, but using memory pages gathered by CMP. Having established the correctness of ContainerVisor's memory tracking mechanism, we next proceed with evaluating the execution time overheads of ContainerVisor.

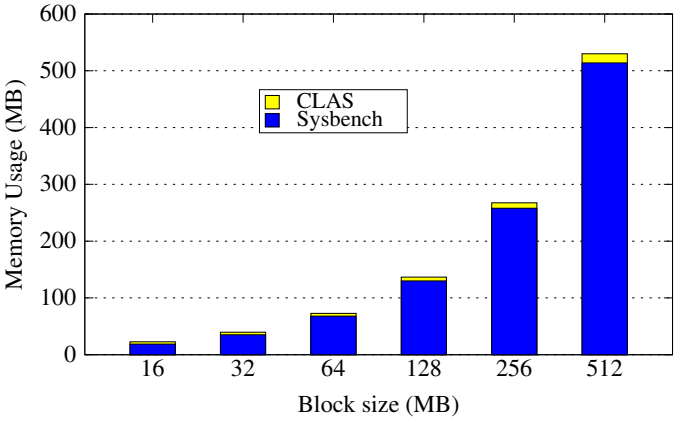

Fig. 4. Memory overhead of ContainerVisor

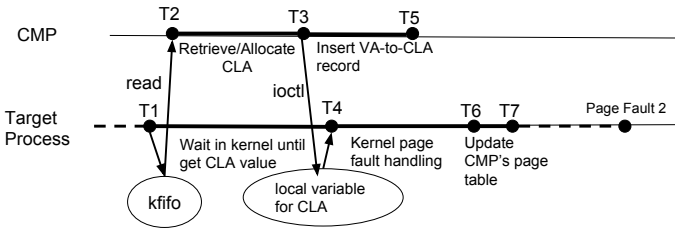

Fig. 5. Page fault handling in ContainerVisor.

B. Memory Overhead

We measured the memory usage of ContainerVisor when the container runs the Sysbench benchmark with different block sizes. Figure 4 shows that ContainerVisor adds low memory overhead across all block sizes. When the memory usage of Sysbench increases, the memory usage of ContainerVisor also increases slightly because the CMP process must record more CLAS mappings. With a block size of $512MB$, the total memory usage of Sysbench is $514MB$ whereas the additional the memory usage by CMP is only $15MB$.

C. Page Fault Handling

This section measures the overhead of ContainerVisor during page fault handling. Figure 5 shows the timeline of page fault event interception and processing in the execution contexts of the faulting process and the CMP. When a page fault occurs at time T_1, the helper hook in the OS inserts a notification message into a kernel data structure called `kfifo` and waits to receive the corresponding CLA from the CMP process. The notification message contains the virtual address of the faulted page, the process ID of the faulted process, and the event type. The CMP retrieves the message from the `kfifo` at time T_2. If the faulted page is already assigned a CLA, the CMP retrieves the CLA from its stored CLAS mappings, else it assigns a new CLA to the faulted page. The CMP then passes the CLA to its kernel module using the `ioctl` system call at time T_3. At T_4, the OS resumes handling the page fault as usual while the CMP records the new VA-to-CLA mapping in the container's CLAS. After the page fault

Target waiting	Page fault handler	Updating CMP PTE	Total
$1.50\mu s$	$1.087\mu s$	$0.208\mu s$	$2.825\mu s$

TABLE I
AVERAGE PAGE FAULT HANDLING TIME AND ITS COMPONENTS.

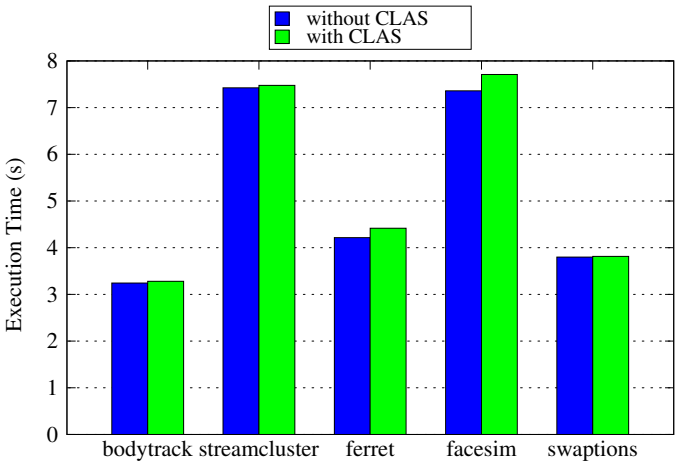

Fig. 6. Parsec benchmarks running individually in a container.

is resolved at T_6 by allocating a new physical page, the OS also updates the CMP's page table to map the CLA received at T_4 to the physical address of the newly allocated page, which corresponds to establishing the CLA-to-PA mapping. We call the time between T_1 and T_4 the *target waiting time* and the time between T_6 and T_7 the *CMP page table updating time*.

We wrote a target program to stress test ContainerVisor. The program repeatedly allocates memory using `malloc()` and calls `memset()` to write to the allocated memory. In our experiments, the program allocates between 100MB to 500MB of memory by allocating one 4KB page in each iteration. We measured the average time spent in handling each page fault in the OS. Without ContainerVisor, the page fault handling time is the execution time of the fault handler routines, such as `do_anonymous_page()`, `do_fault()`, and `do_wp_page()`. With ContainerVisor, the additional time for page fault handling is the sum of target waiting time and the CMP page table updating time.

Table I shows that the average page fault handling time is about $2.8\mu s$ for all workload sizes. Of this, ContainerVisor's additional overhead of $1.7\mu s$ is primarily due to the latency of communicating the page fault event and response with the user space CMP process. Future work can eliminate this user-kernel communication latency by moving the CLA assignment to the CMP's kernel module.

D. Application Performance Overhead

This section compares the performance of applications in a container, with and without ContainerVisor.

Parsec: We measured the performance overhead of five benchmark programs in Parsec [13], namely *bodytrack*, *streamcluster*, *ferret*, *facesim*, and *swaptions*. All except *facesim* are multi-threaded programs. Figure 6 shows the

Benchmark	Number of page faults	Number of Threads
bodytrack	6011	3
streamcluster	2550	3
ferret	31284	7
facesim	88373	1
swaptions	1621	16

TABLE II
NUMBER OF PAGE FAULTS AND THREADS GENERATED BY VARIOUS PARSEC BENCHMARKS FOR FIGURE 6.

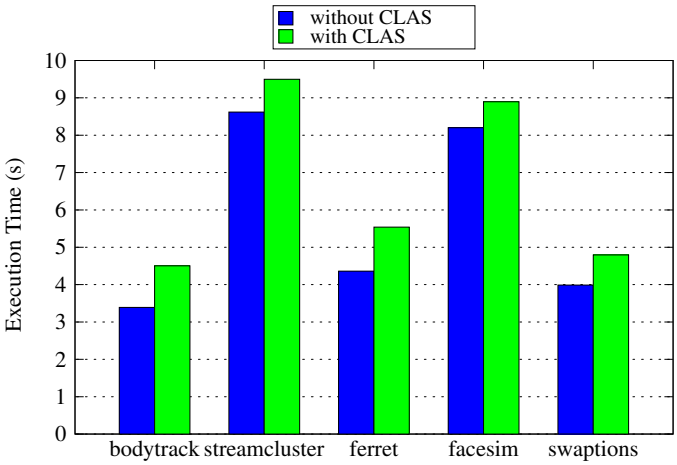

Fig. 7. Multiple Parsec benchmarks running simultaneously in a container.

execution time of these benchmarks with and without ContainerVisor. Table II lists the number of page faults and the number of threads created in these benchmarks. We used the default argument "simlarge" (large-sized input) to measure the performance of these benchmarks. ContainerVisor barely affects the execution times of *bodytrack*, *streamcluster*, and *swaptions* due to lower number of page faults. However *ferret* and *facesim* take about 0.2 seconds and 0.34 seconds longer with ContainerVisor because these two incur significantly more page faults. The number of threads created does not affect the execution time in this experiment.

To assess the impact on a mix of workloads, we executed all five Parsec benchmarks simultaneously inside a container, with and without ContainerVisor. Figure 7 shows that ContainerVisor increases the execution time of different benchmarks between 0.7 seconds to 1.2 seconds, which is higher than the uniform workload experiment above. When running multiple benchmark processes simultaneously, many concurrent page faults are generated. However, our current CMP implementation sequentially handles these page fault events one at a time. Future work can eliminate this sequential bottleneck by parallelizing the processing of concurrent page fault events in CMP.

Sysbench: We used Sysbench [14] to measure the performance of memory write operations with and without ContainerVisor. Sysbench repeatedly writes to a memory block of a given *"block size"* bytes till it writes a specified *"total memory size"* number of bytes. We study the impact of varying these two parameters using the following three experiments.

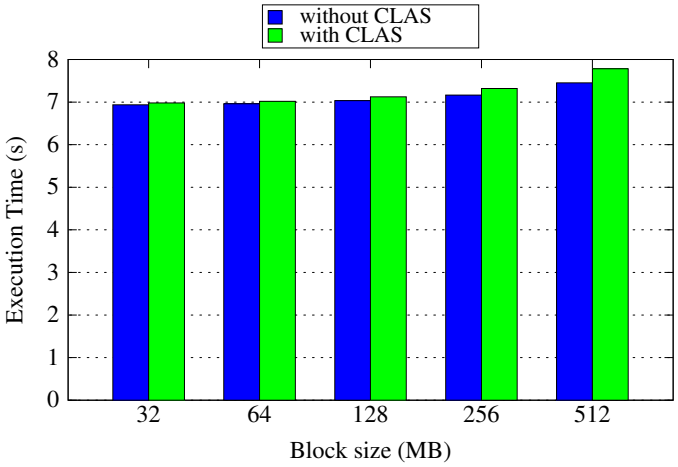

Fig. 8. Sysbench with different block sizes. Total memory size = 32GB.

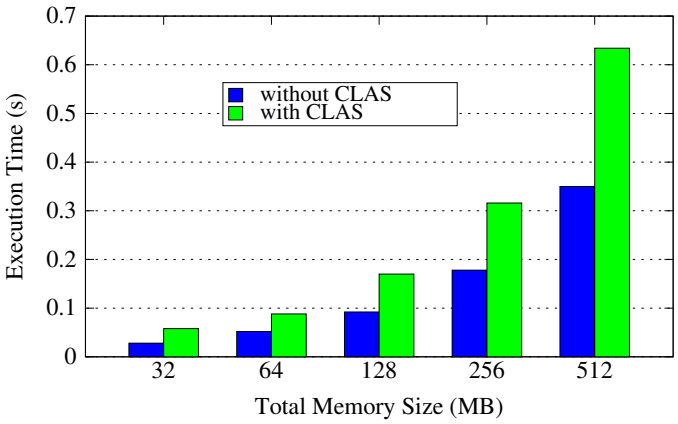

Fig. 9. Sysbench with different total memory sizes. Block size = total memory.

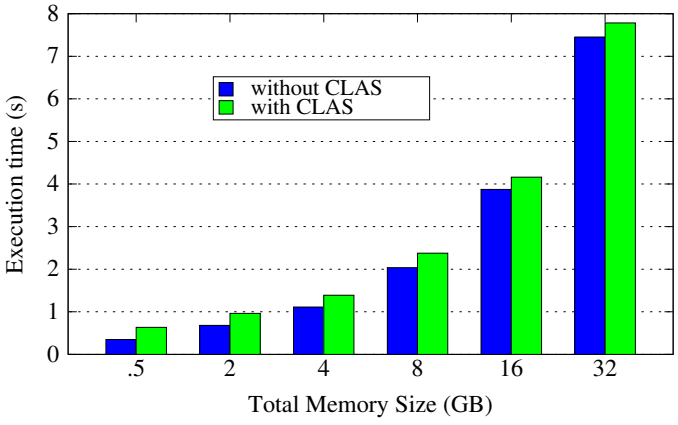

Fig. 10. Sysbench with different total memory sizes. Block size = 512MB.

In our first experiment (Figure 8), the total memory size is $32GB$ while the block size is varied. Sysbench uses one thread to perform repeated writes over the memory block. As the block size increases, Figure 8 shows that ContainerVisor incurs higher overhead because of increasing number of page faults – from 8598 page faults for 32MB block size to 131486 page faults for 512MB block size.

Figure 9 shows the performance of Sysbench in a worst-case scenario, in which the memory block size and the total memory size are the same, which means that the memory block is written to only once. Execution time with ContainerVisor almost doubles compared to without ContainerVisor as the number of page faults generated increases from 8643 for 32MB total memory size to 131525 for 512MB total memory size. However this experiment represents an extreme scenario where a page fault is triggered for every page access and each memory location is not accessed more than once. Typical applications with higher locality would reuse a page multiple times after first access (which triggers a page fault) and thus amortize the cost of page faults.

In the third experiment, we show that ContainerVisor's overhead is much smaller with a lower rate of page faults. We measured the performance of Sysbench with a fixed $512MB$ block size, while the total memory size increases from $512MB$, to $32GB$. Figure 10 shows that, with ContainerVisor, Sysbench takes about 0.3 second longer irrespective of the total memory size. The proportion of this overhead becomes lower as the total memory size increases because Sysbench writes to the same memory block more times. The number of page faults generated for different memory sizes remains stable, between 131483 and 131523, because the block size remains fixed and only the first page access triggers a page fault.

E. Memory Scrubbing

ContainerVisor provides the ability to scrub the confidential memory of a process as soon as the process terminates, which helps to reduce the lifetime of confidential data stored in the memory. In our experiment, we opened a terminal inside a VM, executed the `vim` command in the terminal to open a file, attached the `vim` process to a ContainerVisor, typed "securityword" in the file, and terminated `vim`. Even though the application is terminated, the contents of its memory (in this case with the string "securityword") can linger around in deallocated memory pages inside OS buffers. To detect the presence of such residual application data in the memory of guest OS, we then used the `pmemsave` command from the host OS to dump the entire memory contents of the VM to a file. Without ContainerVisor, the string "securityword" appears 8 times in the VM's memory dump file. With ContainerVisor, the string appears only 3 times. Since the CMP process scrubs (zeros out) the memory pages of terminated processes before returning them to the OS, all user space occurrences of the confidential data are erased. However, the string still appears 3 times in the kernel memory of the guest OS because our current implementation does not track and scrub kernel pages that may contain residual application data.

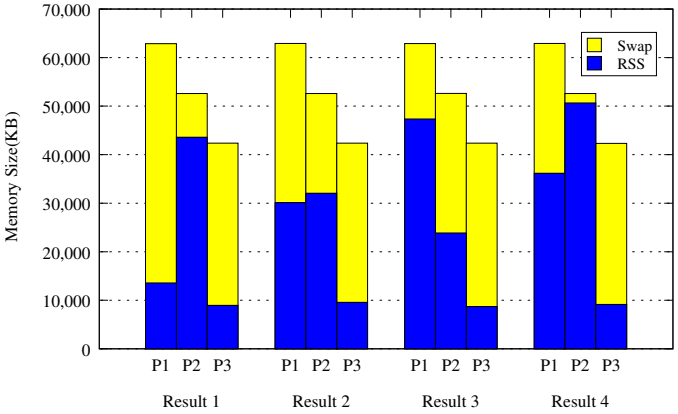

Fig. 11. With Cgroups: RSS and Swap-out sizes for P_1, P_2, P_3.

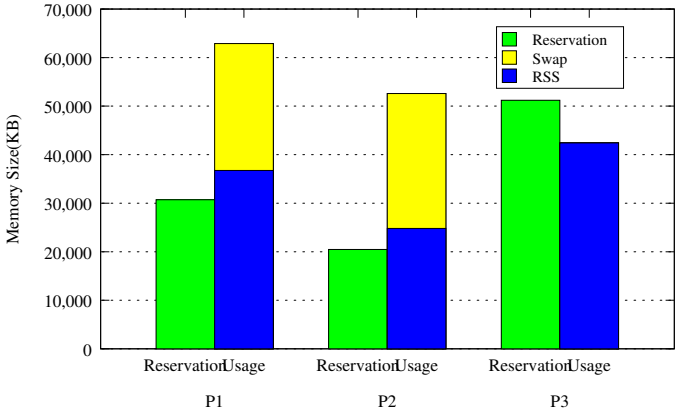

Fig. 12. With CMP: Reservation, RSS and Swap-out sizes for P_1, P_2, P_3.

F. Memory Reservations and Customized Page Replacement

We now demonstrate ContainerVisor's per-process memory reservation and customized page replacement policies.

In our experiment, we created a Cgroup *memtest* with three processes P_1, P_2, and P_3. We set $memory.limit\_in\_bytes$ parameter of *memtest* as $100MB$. Process P_1 used $60MB$ memory, P_2 used $50MB$, and P_3 used $40MB$. Together these three processes consumed about $150MB$ memory, which exceeded the $100MB$ limit of *memtest*. As a result, swapping was triggered by the OS to evict excess memory used. We used the command `smem` to obtain the amount of memory swapped out and the current resident set size (RSS). Figure 11 shows the results over four rounds of the experiment. The figure shows that different amount of memory is swapped out for different processes in different rounds. This is because Linux maintains a per-Cgroup LRU (least recently used) list for swapping pages and, in different rounds, the least recently used memory pages are different. Cgroups does not provide a way to specify guaranteed memory reservation for individual processes in a container, making it hard to predict which process' pages would be evicted during memory pressure, and by how much.

To address this problem, ContainerVisor implements a simple memory reservation mechanism for individual processes in a container along with a customized swapping policy, as a proof-of-concept. Page eviction decision is made based on a processes' current memory usage and its memory reservation. Say, there are n processes P_1, P_2, ..., and P_n. We specify R as the total memory limit of a container and r_i as the memory reservation for process i. Let u_i be the memory usage of process i at any instant. When the total memory usage $\sum_{i=1}^{n} u_i$ is greater than R, swapping is triggered. Let $\Delta i = \max(u_i - r_i, 0)$. In our customized swapping policy, the pages of the process that has the largest $\frac{\Delta i}{r_i}$ will be swapped out. The above procedure repeats until $\sum_{i=1}^{n} u_i$ is below R.

We repeated the earlier experiment of three processes, this time with ContainerVisor applying the above reservation and custom swapping policy. The memory reservation for P_1, P_2, and P_3 are set to $30MB$, $20MB$, and $50MB$, respectively. Figure 12 shows that with ContainerVisor, only memory pages of processes P_1 and P_2 are swapped out because their memory usage exceeds their reservation. However, pages of process P_3 are not evicted because its usage is below its reservation. The result is the same however many times the experiment is repeated. ContainerVisor also allows a process to use more memory than its reservation as long as the total memory used by the container does not exceed its assigned limit.

VI. Related Work

Container-based virtualization: Linux Container (LXC) [2] and Docker [3] are two widely used container platforms. The Linux Containers project provides a virtualized environment for executing groups of applications while avoiding the overhead of system VMs running a separate OS over emulated hardware. Various accounting and tracking operations were implemented in Linux Containers by scanning the address space of each individual process in the container. For instance, CRIU [12] relies on the `/proc` pseudo file system to implement checkpointing and restoration mechanisms. However, CRIU does not provide run-time tracking and monitoring of memory pages. Instead, it collects each process' memory mapping information separately through `proc` and dumps the pages into files using a parasite code inserted into each target process. In contrast, ContainerVisor uses the information in `/proc` file system only when initializing the CLAS mappings of a process. After the initiation, ContainerVisor does not rely on `/proc` to monitor container memory. FreeBSD jail [15] provides an OS-level virtualization mechanism that allows separate virtual environments called jails to be hosted on a single machine. Each jail has its own files, processes, and user accounts. Solaris Zones [16] is a lightweight technology for creating a virtualization layer for applications. Unlike ContainerVisor, neither jails nor Solaris Zones provides run-time tracking or customized management of a container's memory.

Kubernetes [17] is a container management system for automating the deployment of containers. Osman et al. proposed Zap [18], a virtualization layer containing a group of processes within a private namespaces. It provides transparent

checkpoint-restart of unmodified applications. Potter et al. proposed AutoPod [19] to provide a group of processes host-independent virtualized environment. AutoPod also uses a checkpoint-restart mechanism to enable unscheduled operating system updates while preserving application service availability. Both Zap and AutoPod reply on the `/proc` pseudo file system to collect memory pages of processes and do not provide run-time monitoring of container memory.

Intel Clear Container [20] is a lightweight system VM with its own guest OS implemented on top of kvmtool [21], a mini-hypervisor that provides minimal amount of device emulation. In contrast, ContainerVisor does use system VMs but targets customized services for native containers.

Interception of system calls and page faults: Prior work has proposed techniques to improve the performance of virtual machines and enable applications to directly process page fault and system call events by exposing virtual machine hardware to applications directly. Belay et al. proposed Dune [22] that leverages the hardware supported Extended Page Table (EPT) and syscall redirection handlers to provide applications with direct and safe access to privileged CPU features. However, Dune does not support the execution of multiple processes in Dune mode, while ContainerVisor does. In addition, in order to use Dune, applications need to be modified to include Dune APIs. ContainerVisor, in contrast, does not require the modification of applications. ContainerVisor also does not use second-level hardware page tables, such as EPT. On one hand, this provides more flexibility, but also introduces performance overheads due to shadow maintenance of OS page tables. Our future work will investigate the use of second-level page tables for maintaining CLAS mappings.

Srinivasan et al. presented process out-grafting [23], an out-of-VM solution for fine-grained process execution monitoring. Out-grafting uses kernel modules to forward page faults between VMs and uses the hypervisor to capture page faults. ContainerVisor, in contrast, uses helper hooks in the kernel to intercept relevant events.. Nova [24] is a microhypervisor-based secure virtualization architecture that minimizes the amount of code in the privileged hypervisor. It also provides user-space page table management. However, Nova relies on the hardware support for full virtualization, such as VT-x and AMD-V, to intercept page faults, but ContainerVisor does not. OSv [25] is a mere guest operating system designed specifically for a single application running inside an VM. In contrast, ContainerVisor is not an OS, but a container-specific service agent for memory tracking and service customization. Ptrace [26] is a system call that enables one process to observe and control the execution of another process. Compared to ContainerVisor's memory event interception mechanism, Ptrace is less efficient as it incurs more user-kernel context switches.

VII. Conclusion

In this paper, we proposed a Container-Level Address Space (CLAS) abstraction to encapsulate and track a container's memory footprint across all of its constituent processes. We presented ContainerVisor, a container resource management system that uses CLAS to provide customized memory management services to a container. We implemented and evaluated a prototype of ContainerVisor in Linux along with three proof-of-concept customized services, namely, memory reservations for individual container processes, scrubbing confidential memory after memory deallocation, and customized per-container swapping. Evaluations show that our ContainerVisor prototype using CLAS can effectively provide customized memory management services within reasonable overheads.

Acknowledgement

We would like to thank our shepherd, Gerd ZellWeger, and anonymous reviewers for their feedback to improve this paper. This work has been funded in part by the National Science Foundation through awards 1527338 and 1320689.

References

[1] "OS-level virtualisation," https://en.wikipedia.org/wiki/OS-level_virtualisation.
[2] LXC, https://linuxcontainers.org/lxc/.
[3] Docker, https://www.docker.com.
[4] Linux Namespaces, https://en.wikipedia.org/wiki/Linux_namespaces.
[5] M. Kerrisk, "Namespaces in operation," https://lwn.net/Articles/531114/.
[6] Cgroups, https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt.
[7] Cgroups wiki, https://en.wikipedia.org/wiki/Cgroups.
[8] A. Kivity, Y. Kamay, and D. Laor, "kvm: the Linux virtual machine monitor," in *Proc. of the Ottawa Linux Symposium*, 2007.
[9] Xen Hypervisor, http://http://www.xen.org/.
[10] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proc. of ACM Symposium on Operating Systems Principles*, 2003.
[11] "Second level address translation," https://en.wikipedia.org/wiki/Second_Level_Address_Translation.
[12] Checkpoint/Restore in Userspace, *http://criu.org*.
[13] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," Princeton University, Tech. Rep. TR-811-08, January 2008.
[14] A. Kopytov, "Sysbench," https://github.com/akopytov/sysbench.
[15] FreeBSD Jails, https://www.freebsd.org/doc/handbook/jails.html.
[16] "Solaris Containers," https://en.wikipedia.org/wiki/Solaris_Containers.
[17] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *Communications of ACM*, vol. 59, no. 5, pp. 50–57, 2016.
[18] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The design and implementation of Zap: A system for migrating computing environments," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 361–376, Dec. 2002.
[19] S. Potter and J. Nieh, "Autopod: Unscheduled system updates with zero data loss," *Second International Conference on Autonomic Computing (ICAC'05)*, pp. 367–368, 2005.
[20] Intel Clear Containers, https://clearlinux.org/features/intel-clear-containers.
[21] J. Corbet, "The native KVM tool," https://lwn.net/Articles/438182/.
[22] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis, "Dune: Safe user-level access to privileged CPU features," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, 2012, pp. 335–348.
[23] D. Srinivasan, Z. Wang, X. Jiang, and D. Xu, "Process out-grafting: An efficient "out-of-VM" approach for fine-grained process execution monitoring," in *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*, 2011.
[24] U. Steinberg and B. Kauer, "NOVA: A microhypervisor-based secure virtualization architecture," in *Proceedings of the 5th European Conference on Computer Systems (EuroSys)*, 2010, pp. 209–222.
[25] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov, "OSv – Optimizing the operating system for virtual machines," in *USENIX Annual Technical Conference*, 2014.
[26] "ptrace," https://en.wikipedia.org/wiki/Ptrace.