Picocenter: Supporting long-lived, mostly-idle applications in cloud environments

Liang Zhang[†] James Litton[‡] Frank Cangialosi[‡]
Theophilus Benson[§] Dave Levin[‡] Alan Mislove[†]

[†]Northeastern University
{liang,amislove}@ccs.neu.edu

[‡]University of Maryland
{litton,frank,dml}@cs.umd.edu

[§]Duke University
tbenson@cs.duke.edu

Abstract

Cloud computing has evolved to meet user demands, from arbitrary VMs offered by IaaS to the narrow application interfaces of PaaS. Unfortunately, there exists an intermediate point that is not well met by today's offerings: users who wish to run arbitrary, already available binaries (as opposed to rewriting their own application for a PaaS) yet expect their applications to be long-lived but mostly idle (as opposed to the always-on VM of IaaS). For example, end users who wish to run their own email or DNS server.

In this paper, we explore an alternative approach for cloud computation based on a process-like abstraction rather than a virtual machine abstraction, thereby gaining the scalability and efficiency of PaaS along with the generality of IaaS. We present the design of Picocenter, a hosting infrastructure for such applications that enables use of legacy applications. The key technical challenge in Picocenter is enabling fast swapping of applications to and from cloud storage (since, by definition, applications are largely idle, we expect them to spend the majority of their time swapped out). We develop an *ActiveSet* technique that prefetches the application's predicted memory working set when reviving an application. An evaluation on EC2 demonstrates that using ActiveSet, Picocenter is able to swap in applications in under 250 ms even when they are stored in S3 while swapped out.

1. Introduction

Over the past few years, cloud computing services have commoditized computing resources. Rather than purchasing and managing their own servers, consumers have turned to third parties for redundancy, scale, resilience, and security. Cloud computing has thus had to evolve to meet the increasingly wide range of user applications.

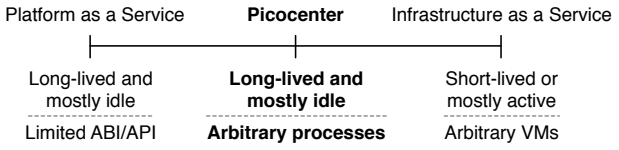

Figure 1: Picocenter represents a unique point in the space between existing abstractions by allowing users to run arbitrary processes that are, in expectation, long-lived yet mostly idle.

To adapt to users' specific needs, there are now two primary models for cloud computing services, each tailored to a particular use case (Figure 1): First, *Infrastructure as a Service (IaaS)* systems, such as Microsoft Azure and Amazon EC2, offer the greatest generality—users can run arbitrary computations and have arbitrary network-facing applications—but are most effectively used by applications that are either mostly active (e.g., a popular web server) or short-lived (e.g., finite but intensive computation). The generality of IaaS comes at the cost of users having to assume the responsibility and overhead of launching, running, and managing an operating system to support their applications. Largely to address these challenges, *Platform as a Service (PaaS)* systems, such as Google App Engine (GAE), Heroku, and Amazon Lambda, emerged to facilitate cloud application development. Inverse to IaaS, PaaS systems limit generality—users must program their applications according to a constrained API using managed languages (typically permitting only event-driven code that cannot bind to new ports)—and are most effectively used by applications that are idle most of the time.

Unfortunately, neither of these models of cloud computation are well-suited for supporting a class of services that are what we call *long-lived but mostly idle* (LLMI). LLMI applications include a variety of services, such as personal email and web servers, decentralized social network nodes [3, 9, 53], per-user mobile cloud offload servers [10, 13, 16, 30, 34], personal cloud-based storage services [5],

EuroSys '16, April 18 - 21, 2016, London, United Kingdom

ACM 978-1-4503-4240-7/16/04…$15.00
DOI: http://dx.doi.org/10.1145/2901318.2901345

or rendezvous services [44]. Because these are LLMI, PaaS would seem to be the best fit, yet we are not aware of any PaaS API that supports such applications. There are many binary executables publicly available that implement these services, making IaaS a reasonable fit, but running them in an always-on VM would unnecessarily waste cloud resources (and thus cost more than necessary), as they are mostly idle. Moreover, in both current offerings, the user overhead to start an application is high: IaaS users must set up and manage a VM, while PaaS users must reimplement their service within the cloud provider's limited API.

In this paper, we present *Picocenter*, an intermediate point between IaaS and PaaS systems that efficiently supports LLMI processes in a cloud environment. More concretely, Picocenter simultaneously (1) allows users to deploy arbitrary binaries (without reimplementing to meet a constrained API) in a more cost-efficient manner than existing services, and (2) allows cloud providers to increase their revenues by enabling a new set of users and applications.

Picocenter provides two key abstractions that make it a unique point in the space of cloud computing solutions. First, to tenants (those who launch applications), it appears that they have sole access to an OS, while in reality there may be many users running in isolation within the same OS. This virtualization is similar to approaches like Docker [14], unikernels [43], and OSv [31]. Unlike these and PaaS systems, however, Picocenter supports arbitrary computation (including forking which is not supported by unikernels and OSv), arbitrary network protocols (AppEngine only supports HTTP), and event handling (Amazon Lambda [1] supports only a pre-defined set of Amazon event triggers). Similar to systems like Docker [14], Picocenter leverages prior work on Linux containers [40] to provide a client execution interface that supports legacy applications, ensures security and isolation, and, most importantly, scales to support a workload where a large number of applications are alive, but not actively running.

Second, to clients (those who interact with applications), Picocenter provides the abstraction that services are always up and running because they quickly respond to requests, though in reality Picocenter may swap entire processes off of a running machine and store it in long-term storage (such as Amazon S3). We achieve this by supporting checkpoint and restore—based on Linux's Checkpoint/Restore in Userspace (CRIU) [38]—as well as dynamic page loading from cloud storage. To avoid costly delays when restoring applications, we develop an *ActiveSet* technique that learns the set of commonly accessed memory pages when requests arrive; we first prefetch those pages from cloud storage before fetching additional pages on-demand. Our ActiveSet technique represents a prediction of the memory pages that are likely to be accessed the next time an application is restored.

While each of these two goals have been addressed separately in prior work, we are the first to achieve them in tandem. Picocenter is therefore the first system that has had to tackle the performance challenges that arise when checkpoint/restore is applied to containers (we detail our implementation in §5).

We deploy Picocenter to Amazon's EC2 service and evaluate it with four approaches. First, using microbenchmarks, we show that Picocenter's overhead due to OS virtualization is quite small. Second, we evaluate the performance of application swapping, showing that Picocenter can swap in and restore real-world legacy applications such as the popular lighttpd web server, the BIND DNS server, and the Postfix email server in approximately 250 ms, even when the application's memory image is completely stored on S3 (conversely, some PaaS systems have load times of 20–30 *seconds* [18]). Third, we demonstrate that the ActiveSet technique can dramatically speed up application restore times by only downloading the memory pages likely to be accessed when an application is restored. Fourth, we roughly estimate the real-world cost of running applications in Picocenter to end users; we demonstrate that Picocenter can represent a savings of over 99% compared to running a dedicated VM.

The remainder of this paper is structured as follows. In §2, we present an overview of LLMI applications and review related work. We describe Picocenter's design in §3, its approach to enabling efficient swapping of applications in §4, and its implementation in §5. In §6, we present microbenchmarks and an end-to-end evaluation on Amazon EC2 which show that Picocenter is able to provide the abstraction that users' processes are always running, even when the limited computing resources demand that they be swapped out when idle. Finally, we conclude in §7.

2. Background and Related Work

In this section, we provide more details on LLMI applications and why they are not well supported by prior work.

2.1 Long-lived mostly-idle applications

Long-lived, mostly idle applications are typically network-based services, such as personalized email servers, individual users' web servers, IRC/XMPP chatting servers, or nodes in decentralized applications [3, 9, 53]. These LLMI applications all have several common characteristics:

- **Long-lived:** The applications typically provide network-based services, and often run for long periods of time.
- **Mostly idle:** The applications are largely I/O-bound, and spend most of the time waiting for incoming messages.
- **Short response time:** The applications may have latency requirements to responding to incoming messages (e.g., web servers or chat servers), meaning incoming messages should not be queued for long periods of time.
- **Low per-request processing:** The time required to process an individual message or event is typically short; the applications do not become CPU-bound for long.

There is no obvious PaaS-style "platform" or constrained API upon which to base all LLMI applications; some bind to multiple ports (such as an HTTP/HTTPS server) or fork many processes (such as BIND and vsftp). Therefore, ideally, any arbitrary binary application could be run in an LLMI fashion. Moreover, to reduce resource consumption for cloud providers (and cost for users), LLMI applications would ideally consume as few resources as possible while they are not actively running. Picocenter provides these two features by drawing from a wide range of prior work.

2.2 Related work

We review prior work in the areas of virtualization, containers, other work that seeks to extend PaaS applications, and migrating and checkpointing VMs and processes.

Hardware virtualization Typical hardware virtualization solutions such as Xen [4], VMWare ESX Server [56], or KVM [39] allow users to run operating systems and applications of their choice. Despite this, many cloud applications today only run one service (e.g., Apache or MySQL) in a virtual machine (often to simplify management and isolate failures). As a result, several prior systems have sought to reduce the overhead of running a full-fledged OS for a single application. For example, OSv [31] and unikernels [43] (e.g., Mirage [42]) improve resource utilization by reducing guest operating system size, providing a library-OS-like environment for user programs. However, these systems do not provide easy support for swap ins. The closest work on supporting swap ins for unikernels is Jitsu [41], which demonstrated that unikernels can be started in a matter of milliseconds. However, Jitsu assumes that the applications are stateless and that new ones can be spawned when requests arrive; with Picocenter, we aim to support the swapping of stateful applications and provide fast restore times.

Operating system containers Operating system containers serve as a lightweight method of virtualization compared to hardware virtualization; examples include Docker [14], LXC [40], VServer [52], BSD Jail [28], and Solaris Zones [49]. This technique modifies the hosting operating system kernel, using kernel-level isolation techniques to make it appear to process groups that they have the machine to themselves. Picocenter leverages containers in order to provide isolation between client processes while allowing backward compatibility with existing applications; Picocenter extends Linux containers with support for checkpoint/restore as well as partial swap ins.

Dedicated runtime Picocenter is distinguished from PaaS systems like Amazon Lambda [1] and Google App Engine [20] in that it permits arbitrary applications written in arbitrary languages. PaaS services typically require that clients write code using a limited subset of managed languages (e.g., Java, Go, or Python); this likely simplifies the provider infrastructure necessary to support these services, but also greatly limits the kinds of services that PaaS systems can support. Another recent trend is hosting applications in a virtual machine; ZeroVM [61], for instance, uses Google's Native Client [59]. However these techniques are not designed to run applications from multiple users, and do not handle resource allocation or swapping.

In parallel, other researchers have developed *picoprocesses* [27], a "stripped-down virtual machine without emulated physical devices, MMU, or CPU kernel mode" [15]. Essentially, picoprocesses are process-like objects that use a small interface to access various hardware resources (network, memory, etc). Picoprocesses have been previously used in the context of redesigning the architecture of Web (in the Embassies [26] project), moving from a Web based on HTML and Javascript to one based on the picoprocess interface [47]. We experimented with a Picocenter design based on picoprocesses, but found them to have significantly higher overhead than our eventual (CRIU-based) approach.

Pre-paging and migration Our ActiveSet algorithm builds on a long line of research on Pre-Paging [54] and VM live migration [11, 25, 45, 50] which predict and prefetch working sets using historical information. Other work [6] has explored consolidating desktop machines onto centralized hardware, and migrating the working set of each VM to the user's machine. Yet others [17, 60] have explored approaches to process migration that lazily copies memory in a similar manner as our reactive page faulting techniques. Unlike these approaches, our execution environment provides us with sufficient visibility into memory access-patterns and allows us to determine working sets at a much finer granularity. This finer granularity reduces the size of data and transfer times. Other approaches to migrating virtualized resources such as networks of VMs and other infrastructure [21, 29, 46] take on the order of minutes to hours to complete, unfortunately the interactive nature of the set of Picocenter's target applications demand faster techniques.

Checkpoint and restore Our work extends the study of process checkpoint and restore [2, 23, 36], which retrieve states of processes in operating systems and restore processes from the saved states. Picocenter leverages this work by using the Checkpoint/Restore in Userspace for Linux project [38], while extending CRIU to be able to restore processes with memory loading on-demand, and to also be able to make incremental checkpoints of Linux containers.

Other work has explored full virtual machine checkpoint and restore, including DreamServer [32] and Sloth [33]. In these systems, idle VMs (e.g., VMs with no incoming requests for 30 seconds) are suspended, and are restored when they receive an incoming request (typically within 2 seconds). While Picocenter shares many goals with these approaches, our underlying approach is fundamentally different; Picocenter deals with sets of processes, while DreamServer deals with entire VMs. Thus, Picocenter is signifi-

cantly more lightweight (our restore times are an order of magnitude lower), and it also relieves clients of the burden of VM management.

Code offloading In parallel with the work above, there has been a line of research investigating the potential for *offloading* computation from less-powerful devices to more-powerful devices (e.g., from mobile devices to the cloud). Notable examples include UCoP [16] (targeting laptop devices) and MAUI [13] (targeting mobile devices). While these share some similarities with Picocenter, the set of assumptions they make and challenges they face are quite different. For example, such offloading systems typically must develop techniques to efficiently divide code between local and remote resources, but are not concerned with swapping idle applications to cloud storage.

3. Picocenter Architecture

In this section, we present the design of Picocenter's architecture and describe its components.

3.1 Interface to cloud tenants

We begin by describing what Picocenter looks like to cloud tenants. To begin using Picocenter, a tenant should provide:[1]

- **Tarball** A filesystem containing any executables, configuration files, and libraries the application needs, as a tar file. Picocenter provides a base Linux image, so the tenant only needs to provide any missing or modified files.
- **Initial process** The initial process within the provided filesystem that should be executed, and its arguments; this process is free to launch any child processes, etc.
- **Pico-configuration** A list of the external protocols and ports the application will bind to.

For example, the application binary might be a web server such as Nginx, and the filesystem image would contain the binary and the set of documents Nginx should serve (along with Ngnix's configuration `/etc/nginx.conf`), and the list of external portocols and ports would be: TCP port 80. It is important to note that the tenant can specify particular ports or simply a number of ports (if a specific port is not needed).

If Picocenter accepts the tenant's application, it will begin executing the application and will return to the tenant a unique DNS name where the application can be reached (on the ports specified at submission time). If the application becomes idle, Picocenter may swap the application (and its filesystem) to either local disk or remote cloud storage (e.g., Amazon's S3) in order to allow for other active applications to be run. We refer to applications that are swapped out as *inactive* applications. However, Picocenter will maintain the illusion to the tenant that the application is always running: when the tenant attempts to access the application again in the future, or a long-lived timer in the application expires, Picocenter will transparently retrieve it from disk or cloud storage and restore the application where it left off. We refer to this as *reviving* the application.

Thus, to a tenant of Picocenter, it appears that their application is alive and running the entire time, but during its lifetime, it may in fact be swapped out and swapped back in on a different physical machine. The tenant is primarily billed for the time when the application is actively running, so LLMI applications are likely to be significantly more affordable when compared to an always-running VM approach.

[1] Our requirements are similar to what is contained inside of a Docker [14] image.

3.2 Challenges

Next, we motivate our design decisions by discussing the challenges with a process-based cloud computing model.

Transparent checkpoint and restore Lowering operational costs for the cloud provider allows for savings to be passed on to clients. Picocenter will need to be able to support application checkpointing and swapping, and later restoring on a potentially different machine. This can happen both as the set of active applications changes, as well as when the load on different hosting machines varies. To ease the burden on cloud tenants, migration must be transparent to application processes.

Backwards compatibility Picocenter needs to work with the rich set of applications that exist today, and not require any source code modifications or recompilation.

Partial swap ins Since applications are swapped-out to cloud storage, downloading an applications's entire memory image before resuming it would likely carry a large latency penalty. Instead, Picocenter must support *partial swap ins*, where the hosting machine can restore the application with only a portion of its memory having been downloaded.

Application storage Applications are presented with a private filesystem, consisting of a base image, user-provided data, and application-created files. This filesystem must be maintained as the application is swapped to cloud storage and restored (potentially on a different hosting machine).

3.3 Architecture overview

We present an overview of the Picocenter architecture in Figure 2, detailing the interactions between Picocenter and external entities. There are two external entities: the application owners, *tenants*, who deploy services in Picocenter and *clients*, who interact with the applications hosted on Picocenter using existing applications and devices.

At a high level, Picocenter is internally composed of two components. The first is a logically centralized component called the *hub* that performs global resource allocation, decides where to push new applications and where to route packets for swapped-in applications (e.g., the location of active applications, as well as metadata about inactive appli-

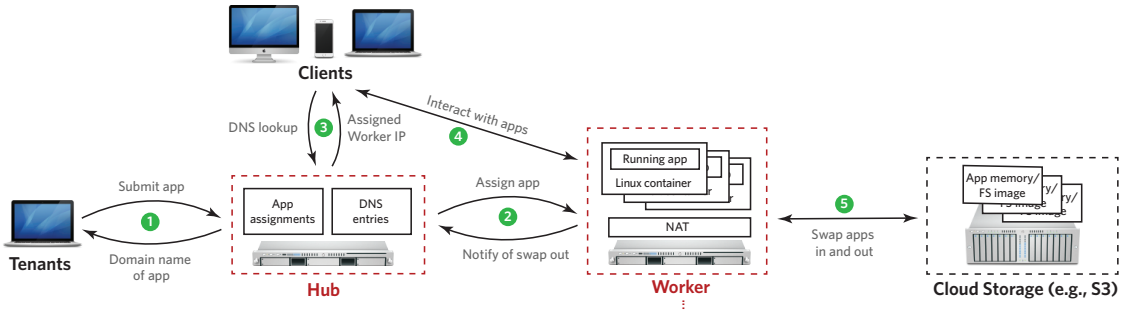

Figure 2: Overview of Picocenter, with Picocenter components shown **in red**. Tenants submit applications via the hub and receive a DNS name ①; the hub then assigns these applications to be run on workers ②. Clients lookup a Picocenter domain via the hub ③ and then interact with applications ④. Workers may swap out inactive applications to cloud storage ⑤, and swap them in again later when requests arrive.

cations). The hub serves as a central point of management control in Picocenter.

The second is a virtualization layer akin to a modern hypervisor called the *worker*, that actually hosts the tenant's applications. There are multiple workers that work with the hub, and each worker performs local resource allocation. The worker is assigned with applications to run by the hub, but determines when to swap out running applications and performs optimizations to ensure that active applications maintain a short response time.

3.4 Hub

The hub manages global resources within Picocenter: allocating IP addresses and ports, assigning domain names, and assigning applications (both new and revived) to workers. To do this, the hub must be informed of any new applications to be launched as well as when applications are swapped to cloud storage: it receives requests from tenants to deploy new applications, it receives DNS requests from clients, and it receives updates from workers when applications are swapped out to cloud storage.

In the discussion below, we describe the hub as a single physical machine, as this is how it is implemented in our prototype. However, all of the functionality of the hub could be easily parallelized across multiple machines for both redundancy and scalability. In essence, the function of the hub in Picocenter is both as a cluster manager (similar to Borg [55], Omega [51], and Mesos [24]) as well as a load balancer (similar to HAProxy [22]). While we did not implement our hub prototype using any of these systems, some of the functionality could likely be outsourced to them. We leave a full exploration of a parallelized hub implementation to future work.

New applications As shown in Figure 2, the hub interfaces with the tenants and accepts new tenant applications subject to the constraints specified in the application's configuration file. The hub searches for a public IP address with the appropriate ports free and for a worker with sufficient resources.[2] For example, if the tenant requests port 80, the hub searches through all IP addresses owned by Picocenter, for one whose port 80 is unassigned. If no available IP address can be found, the hub either launches a new worker or refuses to accept the application.

To guide its decisions of which worker to assign the application to, the hub maintains state for each worker: what IP addresses it offers, what applications it is currently running, and whether or not the machine has asserted that it is capable of accepting new applications. This information is used by the hub's scheduler to determine which worker is available to accept new tasks. The hub's scheduler can support a range of scheduling disciplines from LRU to Lottery Scheduling [57] when deciding which available worker to pick for a task.

Assuming the hub does find an available IP address and a worker with sufficient free resources, it generates a unique DNS name; stores the mapping between the name and the public IP address; informs the worker that it should begin swapping in the application; and returns the DNS name (and any requested, unspecified ports) to the tenant.

To ensure that future incoming requests are always routed to the appropriate worker, the hub contains a DNS resolver that responds to DNS queries. Any network traffic to the application will thus be routed directly to the appropriate worker, and do not need to transit the hub. The TTLs for the DNS responses are set to the minimum amount of time that an application must reside on a worker before being swapped-out. We set this value to one minute in our prototype; while it may seem unlikely that today's cloud-based applications would be idle for more than one minute, recall that we are targeting LLMI applications that are, by definition, idle for long periods of time.

Picocenter swaps a new application in when it is accepted (rather than immediately storing it in cold storage) so that the application can perform its own initialization, such as listening on sockets and starting timers.

[2] This process is similar to systems that provide cluster management and orchestration functionality like Kubernetes [35], except that the hub in Picocenter is managing at the level of *applications* and *network ports*; typical cluster management systems manage at the level of *containers*, and therefore simply need to assign a unique IP address to each container. In Picocenter, multiple applications may be assigned the same IP address as long as their port reservations do not conflict.

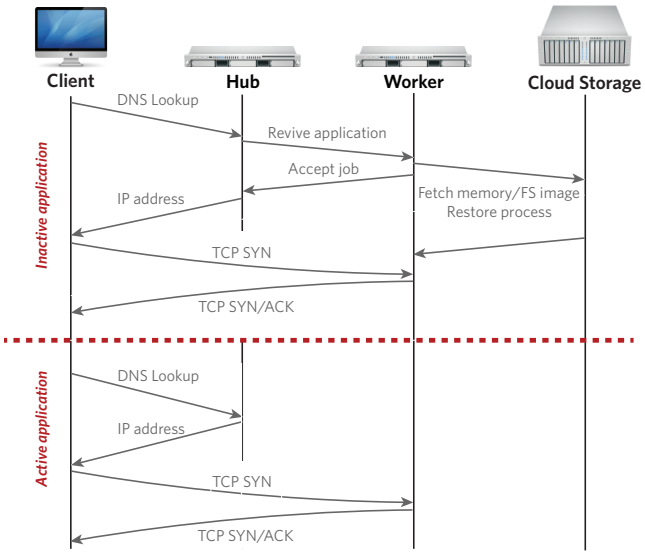

Figure 3: Timeline for serving incoming requests for applications, both inactive and active. Inactive applications are revived while the DNS response is sent to the client; all network traffic for the application goes directly to the worker.

Reviving applications Applications can be revived from cloud storage in one of two ways: when an application timer expires, or when a packet arrives for the application.

In both situations, the hub needs to revive the application by assigning it to a worker. To do this, the hub needs to know where the application is stored in cloud storage and which worker to swap the application to. The hub makes these decisions using the same mechanism as when a new application is launched: the hub looks for a worker that has sufficient free resources and has an IP address with the application's used ports free. Similar to the case above, if no workers are available to accept a recently invoked task, then the hub spins up a new worker (or drops the request).

3.5 Worker

The worker is responsible for running the tenant's applications, swapping in and swapping out applications, performing address translation for incoming traffic, and monitoring application resource utilization for billing purposes.

To accomplish these goals, the workers interact with the hub as follows. The hub informs the workers of new application assignments. In turn, the workers inform the hub periodically as to whether or not they have sufficient capacity to accept new applications, and they also inform the hub when they have swapped out an application they were previously assigned to cloud storage (thereby releasing the application to be assigned to another worker).

Initially, all workers are willing to accept new applications. When an application needs to be run (either a new application, or an application to be revived), the hub chooses a worker (as described above) and provides it the required information for running the application. Namely, the application's network configuration, the location of the application's data in cloud storage, and the Pico-configuration. Using this information, the worker is able to successfully run the application and route its network traffic.

We provide more details on the worker implementation and hosting infrastructure in §5.

4. Quickly Swapping in Processes

Picocenter provides the abstraction that all registered applications are running all the time by providing low response time to client requests, even if the application has been idle for months or longer. However, so as not to unnecessarily waste cloud resources, Picocenter does not keep all registered applications running or even resident on worker machines at any given time; most of the time, an LLMI application's memory state is "swapped out" to long-term cloud storage, such as Amazon's S3. Picocenter maintains this abstraction by swapping in processes from long-term storage to execution on a worker *quickly*: to appear to always be running, our target is to swap in processes on the order of a few client's round-trip-times (RTTs)—the time it takes for the DNS response to reach the client, and for the client to send the request to the application. This way, in the time it takes for the client to receive the DNS response and initiate a TCP handshake, the process would have been loaded (Figure 3).

Challenges While swapping processes and memory pages has received extensive prior study across a wide range of domains, including VM migration [6, 11, 25, 45, 50], "just in time" unikernels [41] and process migration [17, 60], the LLMI applications we consider represent a unique set of challenges: First, whereas most work in VM migration involves moving a running VM from one machine to another, we seek to pull a frozen process into a running state quickly enough to remain unnoticeable to users. Second, unlike most prior work in PaaS and unikernels, we do not constrain the size or type of user applications, and thus we must support applications that can fork and listen on multiple ports. To meet both of these ends, we extend prior work on partial migration [6] to quickly load precisely what a process needs to begin handling a request based on a given port.

4.1 Swapping out applications

The workers are responsible for judiciously swapping out enough applications in order to keep sufficient resources free for active applications. To this end, the worker constantly monitors its load (e.g., memory pressure, CPU utilization, or bandwidth consumption; in our implementation, we use memory pressure, but some combination thereof may be most suitable in practice). When a machine's load goes above a high watermark, the worker informs the hub that it can no longer accept new applications, and it begins swapping out the most idle applications to cloud storage in an LRU manner. When the load drops below a low watermark, the worker then informs the hub that it can again accept new

applications, and stops swapping out applications. By maintaining a separation between high and low watermarks, we ensure that workers do not oscillate between these two states.

Recall that when an application is assigned to a worker, the worker agrees to be responsible for the application for a fixed period of time (one minute in our prototype). This agreement is reflected in the DNS TTL that is returned to the client, mapping the application's DNS name to that worker. At times, a worker may discover that its memory pressure is too great, but that it cannot swap any applications to cloud storage due to this restriction.

In order to allow a worker to relieve its memory pressure without violating this agreement, we allow workers to also swap applications to their local storage instead of cloud storage. The mechanism for doing so is identical to swapping to cloud storage, except that the worker does not inform the hub that it has swapped out the application. As a result, incoming network traffic may arrive at the worker for a locally swapped-out application. In this case, the worker revives the application from local disk, rather than cloud storage.

4.2 Swapping in applications

Picocenter swaps in applications as a result of an invocation event, such as an incoming DNS query for the given application's hostname, or a triggered timer the application registered. Because applications register timers with workers (and with the hub, should a worker swap an application to cloud storage), we can easily predict timer events and simply swap in applications before they fire, so that the application is ready to run precisely when it needs to be. Incoming packets, however, are more difficult to predict, such as an incoming email to a personalized email server; to maintain our abstraction, we must react and swap in the relevant application quickly enough to avoid human perception.

Let us take a step back and ask: how quickly does an application need to be swapped in to be unnoticeable to an external client?

Recall that each application is given a public-facing DNS name, but we may change which public IP address it is NAT-ed to as it moves between being active and inactive. Therefore, the first step a client must take to initiate a new connection is often to issue a DNS query for the hostname. Once the hub selects a worker to revive the application, we have roughly the RTT to the client before there will be any external input to the application. We can leverage this time—typically on the order of tens to hundreds of milliseconds—to begin swapping in (similar to the way TCP handshake offloading is used in some hypervisors [41]), but even this is not enough to swap in the *entire* memory space if the application is on cloud storage.

Partial swap ins A crucial component to maintaining our abstraction of "everything is always running" is *swapping in only the portions of the process's memory that are needed*. For non-trivial applications, this is far less than the entire memory space and it is often somewhat predictable. As a motivating example, we instrumented the Nginx web server in Picocenter to record memory reads and writes when serving uniform requests to a variety of files. We observed that (a) on average, less than 1% of the valid memory space was actually accessed on any request, and (b) the set of pages accessed showed over 90% overlap across all requests.

Picocenter uses two mechanisms to efficiently load only what a process needs: reactive faulting and prefetching.

Reactive page faulting Picocenter monitors all memory pages that each application reads and writes to. When an application tries to access a page that has not been loaded into its runtime memory space, Picocenter captures this as a "Picocenter-level fault," pauses the thread that caused the fault, downloads the data from storage, and resumes the thread. Because handling a Picocenter-level fault involves transferring a small amount of data from cloud storage to a machine (4KB), it is a latency-bound operation. Thus, we can swap in more than one page at a time without significantly impacting the overall performance: we segment the data into sets of B pages, so that, given a page fault on page p, we swap in pages $B \cdot \lfloor p/B \rfloor$ to $(B+1) \cdot \lfloor p/B \rfloor - 1$.

While these Picocenter-level faults are less expensive than loading an application's entire memory, they are far more expensive than traditional OS page faults: the OS needs only pull from local storage, while Picocenter may have to pull from cloud storage, such as Amazon's S3. Thus, Picocenter seeks to minimize the number of Picocenter-level faults (to optimize runtime performance), while still loading as few pages as possible (to optimize swap in performance). To this end, Picocenter employs a preemptive component as well.

Prefetching with ActiveSet When swapping out an application, the local machine stores the page access information in cloud storage, along with whatever kind of event invoked the application in the first place (e.g., the port of an incoming network connection or memory address of the timer that fired). Over time, Picocenter develops a model of which memory regions are often used for a given application when processing different events. When swapping in an application, Picocenter *preemptively* swaps in those memory pages that are most likely to be accessed as a result of serving that request. In the Nginx example, preemptive, partial swapping reduces the amount of application state that needs to be transferred from cloud storage to a machine by two orders of magnitude. Thus, tracking and predicting these accesses in Picocenter is likely to provide significant performance benefits.

Concretely, Picocenter maintains, for each application, what we call the *ActiveSet* of memory pages: data that we believe are most likely to be needed by that application to handle a future invocation event. Workers monitor what memory pages applications access, and use this to update

the ActiveSet. In our prototype, the ActiveSet consists of all pages that have been accessed during the application's most recent invocation. When swapping the application in from cloud storage, the worker first downloads an index file describing which pages are in the application's ActiveSet and then prefetches those pages and makes them available to the application. When swapping out an application, the worker adds or removes pages from the process's ActiveSet as necessary and writes it to the application's index file in cloud storage.

The ActiveSet may have false positives (it believed a page would be needed, but it was not) and false negatives (it failed to anticipate that a page was going to be needed). False positives can result in a slower swap in, as the worker has to download more data from cloud storage than strictly necessary, but do not affect the application as it is running. Updating the ActiveSet over time mitigates false positives. False negatives, on the other hand, result in an application trying to access pages that we have not swapped in. For these pages, we fall back on our reactive scheme.

Advanced ActiveSet Our approach permits more sophisticated definitions of the ActiveSet, such as a dependency graph among pages (encapsulating the notion of order of accesses), or pages that have been accessed k out of the last n times the application was active (for parameters $k \leq n$).

Because Picocenter supports applications that fork and listen on multiple ports, such as an email server or a web server handling HTTP (port 80) and HTTPS (port 443), it may be beneficial to track page accesses as a function of port number. For example, when processing an HTTP request, Nginx accesses $\sim$100 memory pages; for HTTPS requests, Nginx accesses these same pages, plus an additional $\sim$300 pages. Ignoring port numbers may result either in swapping in more (for HTTP) or fewer (for HTTPS) pages than necessary, resulting in slower processing times.

Picocenter could be extended to support multiple ports by maintaining more than one ActiveSet for each application: one for each listening port, and one that summarizes the most common across all ports. When a DNS query arrives, the worker would begin by loading the ActiveSet common across all of the application's ports, as it would not yet know the port number—in the case of Nginx, this would constitute the $\sim$100 pages common to HTTP and HTTPS processing. When the first transport-layer packet arrives, such as a TCP SYN packet, the worker would then swap in the pages corresponding to that port's ActiveSet. There are several possible further extensions, such as estimating the likelihood of a particular port being accessed, but we leave this to future work.

5. Implementation and Discussion

Before presenting the evaluation of Picocenter, we describe the implementation and deployment challenges that we addressed in Picocenter.

5.1 Implementation

We implemented the hub and worker on Linux. The hub implementation consists of 757 lines of Python. The hub also uses Twisted (a Python framework) to implement the DNS functionality, and MySQL to store information about where applications are running or are swapped out. The hub maintains persistent TCP connections to all workers, used for job assignment and load reporting.

The worker leverages Linux containers [40] (LXC) as a basis for hosting client applications on workers. Containers are an operating system-level approach to virtualization that provides each container (essentially, a group of processes) with a virtualized environment. Thus, with containers, processes in different containers are unable to interact other than via the network. Containers have been successfully used—via projects like Docker [14]—as a basis for simplifying deployment of applications to hosting machines.

Limitations Our prototype implementation of Picocenter has a few important limitations. *First*, our prototype does not keep track of application timers, and currently only revives applications when network traffic arrives. *Second*, our prototype does not implement per-port ActiveSet, and because all of the applications in our evaluation were accessed over a single port, it does not affect our evaluation's conclusions.

5.2 Swapping implementation

We make use of several useful tools that have been built on top of Linux containers to facilitate our implementation of swapping. Clients' memory changes over time; it would be wasteful to copy down the entire memory space in order to simply store it back to the cloud. Instead, we build on the existing work of the Checkpoint/Restore from Userspace (CRIU) project to revive processes such that they only require the downloading of pages that are accessed and to only produce new data for subsequent restores for pages that were modified. This, in effect, gives us the ability to provide incremental checkpoints with on-demand loading of pages. When swapping out a previously-revived process, our modified CRIU only writes out those pages that were modified since the application was most recently revived. We store the resulting checkpoint, which necessarily only contains a page-level diff from the previous one, alongside the existing image in cloud storage. This approach enables quick swapping out of processes.

However, successfully leveraging containers for use in Picocenter presented significant challenges. *First,* supporting ActiveSet in Picocenter requires the ability to (1) observe the memory accesses (reads and writes) that clients make; (2) partially load a client's memory image; and (3) dynamically load pages from cloud storage in the case of page faults. We enabled support for all of these by leveraging Linux's Filesystem in User Space (FUSE) [19]. When restoring a client, we invoke the modified CRIU and supply the FUSE-mounted filesystem as the source for memory images. As

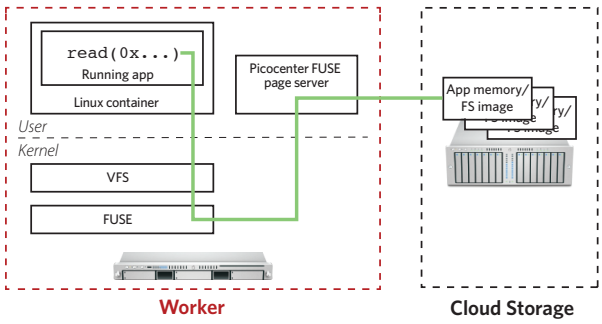

Figure 4: Architecture for supporting partial swap ins, ActiveSet, and on-demand memory loading in Picocenter. Processes are revived with their memory image and filesystem on a FUSE filesystem; as memory requests are made, they are handed to our FUSE process which either serves them locally (for pre-fetched content) or downloads them from S3.

CRIU runs and subsequently when the process is revived, the operating system will load pages from this file whenever a page fault occurs; since it is a FUSE file, our FUSE process will be called whenever this occurs. At this point, our process can fetch the appropriate memory pages from either local disk or from cloud storage. Leveraging FUSE also allows our process to monitor memory usage patterns (since all pages read and written will result in a FUSE call the first time they are accessed). When a client is being swapped out to local or cloud storage, the set of pages accessed are recorded along with the memory image of the process, allowing us to implement ActiveSet. Our FUSE process represents 1,906 lines of C/C++ code, and a diagram of its operation is provided in Figure 4.

Second, support for CRIU [38] with containers is still in "beta" status, and completely implementing Picocenter required us to fix bugs, add basic support for FUSE filesystems, and develop workarounds to enable pages to be lazily loaded; we contributed these modifications back to the respective projects. Overall, our modifications to CRIU comprise 683 lines of C, and, for the interested reader, we describe the details of these modifications in Appendix A.

Third, applications may write to local files, but we may swap an application off of one machine and onto another. We thus needed to implement support for each container to maintain a consistent view, not only of its memory, but of its own filesystem even as it is swapped in and out across different machines. Picocenter efficiently provides each application with a private filesystem based off of a predefined Linux image by using `btrfs` [8]. In essence, `btrfs` allows us to record all changes to the application's filesystem, and the worker stores this diff in cloud storage alongside the application's CRIU state. When reviving an application, the worker first re-creates the application's filesystem by replaying the diff.

5.3 Deployment issues

Before presenting the evaluation of Picocenter, we briefly discuss a few issues that a real deployment of Picocenter would face.

Catching "early" packets One challenge that arises when launching new applications (or reviving existing applications) is that client packets could arrive at the worker before the worker has completed launching (reviving) the application. If not properly handled, the worker may drop the packet or send back a `RST`, potentially confusing the client or causing a timeout. To address this, our worker implementation uses the `iptables NFQUEUE` feature, which allows us to enqueue arriving packets. Once the application is fully alive, the worker releases all packets from the queue.

Resource accounting and billing One of the key benefits of Picocenter is that applications are charged primarily when they are active; this enables largely-idle processes to be supported in a cost-efficient manner. We envision that Picocenter applications would be charged at two rates: R_a for times when the application is active and R_i for times when the application is inactive (on local or cloud storage), with $R_a \gg R_i$. Similar to Amazon Lambda [1], we envision that this rate would be charged per memory GB over time.[3] Thus, as an application's memory demands change over time, it is charged more or less. While we briefly estimate the cost of running a few applications in § 6, we leave a full exploration of accounting and billing to future work.

Applications Picocenter is designed to run most existing applications (including both multi-threaded and multi-process applications), but there are certain types of applications that Picocenter is a poor choice for hosting. Examples of such applications are computationally-intensive applications (scientific computing, Bitcoin mining, etc). While our evaluation focuses on LLMI applications, we note that there may be applications in between these two extremes. We hypothesize that there may be a class of applications, such as an email server that typically services requests once every few minutes, that may be more cost-effectively served on an overprovisioned machine that keeps the ActiveSet cached locally ("warm" in our benchmarks). In the event that the machine is unable to handle requests quickly enough, the tasks that were idle the longest could be moved onto another machine. In this case, tighter guarantees on response time can be provided and sold at an intermediate rate. We leave the exploration of this space to future work.

There are a number of common applications that users are likely to use often. Similar to the way existing cloud provides support VM "images," Picocenter can provide preconfigured Picocenter images for these common applica-

[3] Amazon Lambda has a simple single rate of $0.00001667 per GB-second (as of this writing), because Lambda does not support swapping of inactive applications.

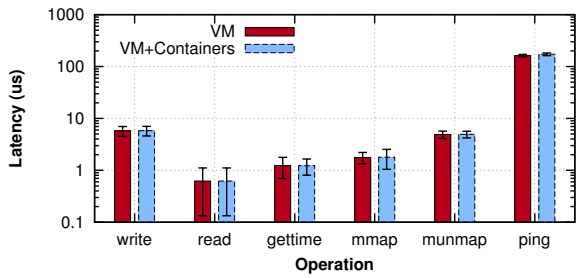

Figure 5: Microbenchmarks for our test application in Picocenter (VM+Containers), compared to a native process running in an IaaS VM. The error bars are standard deviations, and note the *y*-axis is in log scale. Containers (and therefore Picocenter) show 1.4% overhead across all tests, and 6.4% overhead on the UDP ping test.

tions. Doing so would relieve users of the burden of having to create their own configurations for applications to be used in Picocenter.

Competition for ports Because the hosting infrastructure for Picocenter has a limited number of IP addresses, certain ports may be in high-demand for applications (e.g., TCP port 80). We note that most applications today can support applications being run on non-standard ports (e.g., HTTP and HTTPS), but there are a few applications that are required to be run on specific ports (e.g., TCP port 25 for applications that need to interact with legacy SMTP servers). We believe that this non-uniform value of ports can be addressed by charging different rates R_a and R_i for applications that require such high-demand ports.

Security As Picocenter will be supporting applications hosted by mutually distrusting tenants, Picocenter needs to provide the same level of security and isolation between applications as today's cloud infrastructure does. Because Picocenter relies on Linux containers, Picocenter inherits the isolation between applications that containers provide (which is the same way that Docker instances are isolated from each other). Specifically, Picocenter relies on `cgroups` to enforce resource usage limits, Linux `namespaces` to provide isolated namespaces for each container and their filesystems, `seccomp` to limit system call access, and kernel NAT support to provide private network addresses to different applications.

ActiveSet for filesystems As described so far, the ActiveSet technique is limited to applications' memory images; however, it could theoretically also be applied to applications' filesystems. If this were done, it could speed up the time required to revive an application by removing the need to download the entire `btrfs` diff before reviving the application. This could be accomplished by mounting the application's filesystem via FUSE (to monitor accesses and handle faults), but doing so would require carefully indexing the `btrfs` diffs so that the requested data could quickly be located. We leave fully exploring this feature to future work.

6. Evaluation

We now present an evaluation of Picocenter. We frame our evaluation around four key questions: (1) *What is the low-level overhead of running applications in containers in Picocenter?* (2) *How quickly can Picocenter revive real-world processes from cloud storage?* (3) *How does the ActiveSet technique help to reduce application reviving time?* (4) *How does Picocenter perform with a challenging real-world application that has somewhat unpredictable memory access patterns?*

6.1 Evaluation setup

We evaluate Picocenter on a real-world cloud provider, Amazon EC2. We configure both the hub and worker machines to use `c4.large` instance types, with 3.75 GB of memory. All application swapping is done to Amazon's S3.

Unless otherwise stated, our prototype is parameterized (as described in Section 3) to fetch blocks of $B = 32$ pages on every page fault, and we add a page to an application's ActiveSet if it was read from or written to in its most recent previous invocation.

We present results evaluating Picocenter on three different, popular real-world applications: the lighttpd web server (version 1.4.36) [37], the BIND DNS server (version 9.10.2-P4) [7] and the PostFix email server (version 3.0.1) [48]. We choose these three as they represent very different types of applications that users may wish to run in Picocenter. Preparing these applications for use in Picocenter only required preparing a tarball containing the binary, associated libraries, and necessary configuration files (similar to the process for preparing a Docker container).

6.2 Microbenchmarks

We begin by examining the low-level performance characteristics of the hosting infrastructure in Picocenter. Since Picocenter applications are implemented as processes running inside of Linux Containers, we refer to this setup as *VM+Containers*. We also compare Picocenter to the same application running as a normal Linux process in an EC2 virtual machine, representing a prototypical IaaS deployment. We refer to this setup as *VM*. We expect a slight performance hit when compared to VM, due to the overhead of using containers; our goal is to quantify this overhead.

We examine the performance overhead of three broad classes of computation tasks: running without requesting any kernel resources (e.g., math operations and accessing memory), making system calls, and making network requests. To measure these three, we implemented a simple application which (a) reads/writes one 4KB page of memory in a byte by byte fashion, (b) makes system calls for `gettimeofday`, `mmap`, and `munmap`, and (c) performs a UDP ping to a machine on the same network, and waits for a reply. We measure how long each of the operations takes, and repeat the

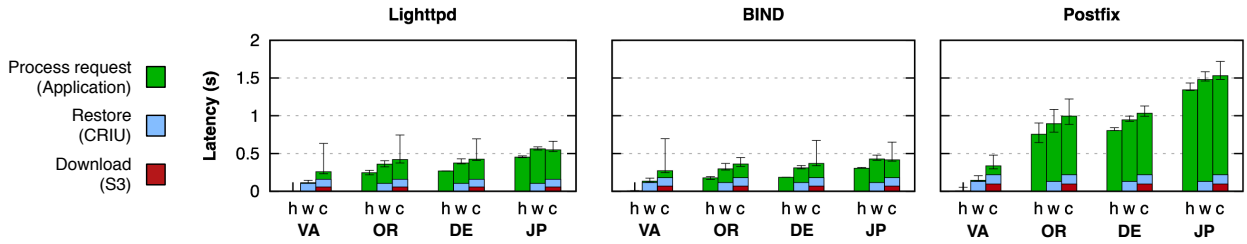

Figure 6: End-to-end client performance for different applications in "(h)ot" (application live and running on a worker), "(w)arm" (application swapped out on a worker's local storage), and "(c)old" (application swapped out to cloud storage) states. The hub and workers are located in Virginia, and the clients are located in different locations around the world, with varying average ping times to the Picocenter instance: Virginia (VA, 0.258 ms), Oregon (OR, 80.9 ms), Frankfurt (DE, 88.7 ms), and Tokyo (JP, 149 ms). Error bars represent the 5th and 95th percentiles of the overall time to complete the client's request. Also shown are the times spent waiting on S3 (downloading checkpoints) and CRIU (restoring the application). Overall, Picocenter provides good end-to-end application performance, with average restore times of 111 ms when applications are "warm" and 186 ms when applications are "cold."

experiments 1,000 times. We run these experiments on EC2 machines.

The results of this experiment are shown in Figure 5, broken down by each experiment and system. We observe that the overhead of using containers in Picocenter is quite low: across all experiments, the average performance overhead is only 1.4%, and in no experiment is the overhead statistically significant. This is consistent with prior work [58] that has shown the overhead of running Linux containers is low. Moreover, we observe that the action that has the highest overhead in Picocenter is the UDP-level ping; this takes an average of 174 us versus 162 us for the VM process (an overhead of 6.4%). The reason for this somewhat higher overhead is the virtual network device that the container uses, necessitating extra processing in the kernel.

Overall, we observe that the usage of containers in Picocenter has very low overhead when compared to running VM-based IaaS processes. Thus, the results indicate that running applications in Picocenter has the potential to provide similar performance to running applications in traditional VMs. In the following section, we take a closer look at end-to-end application performance to see if this holds true for application-level benchmarks.

6.3 Swapping performance

We now turn to examine end-to-end application performance in Picocenter. In these experiments, we set up and run each of our three test applications. For lighttpd, we configure it to serve pages of 1KB and request the pages using `curl` on a separate machine. For BIND, we configure it to serve DNS queries for a test domain, and request a DNS `A` record using `dig` from a separate machine. Finally, for PostFix, we configure it to accept mail for a test domain, and send an email to a test account using Python's `smtplib` from a separate machine.

We want to explore how clients in different locations perceive the latency of swapping in applications from cloud storage, compared to accessing applications that are already running. Thus, we run the hub and workers in EC2's Virginia datacenter, and run clients in EC2's Virginia, Oregon, Tokyo, and Frankfurt datacenters. We repeat each experiment 50 times and report the median, as well as the 5th percentile and 95th percentile performance (shown as error bars).

Figure 6 presents our results from this experiment. For each application, we compare the "cold", "warm", and "hot" performance (recall that cold represents applications that are stored on cloud storage and retrieved, warm represents applications that are stored on worker-local storage and retrieved, and hot represents applications that are already running on a worker). As a reference, we also include in the caption the average ping times between clients in different regions and our workers; we naturally expect that clients who are further away will have slower end-to-end performance characteristics in all applications due to the RTTs required by the various protocols. For example, even when applications are running, completing a PostFix transaction takes 750 ms for the client located in Oregon, while it takes 1,375 ms for the client located in Tokyo.

We make four key observations. *First*, we observe that swapping in applications from cloud storage has a surprisingly low performance penalty. Comparing the "hot" bar to the "cold" bar, which represents Picocenter's overhead of reviving an application, reveals swap in times between 140 ms (lighttpd and BIND) and 200 ms (Postfix). The difference in swapping times is due to the applications having different numbers of processes, memory sizes and layouts, and amounts of process state.

Second, we observe that this overhead can often be dwarfed by the end-to-end performance of the protocol itself. For example, consider the case of PostFix with the client located in Frankfurt. When the application is swapped out to cold storage, sending an email to it takes 1,034 ms; however, even if the applications was already running on the worker, sending the same email would take 805 ms. Thus, in this particular case, the relative client-perceived overhead of swapping the process completely out to cloud storage is low.

Third, we note that there is significant variance in the total restore time, evidenced by the 95th percentile error bars on the cold and warm total bars. This high variability

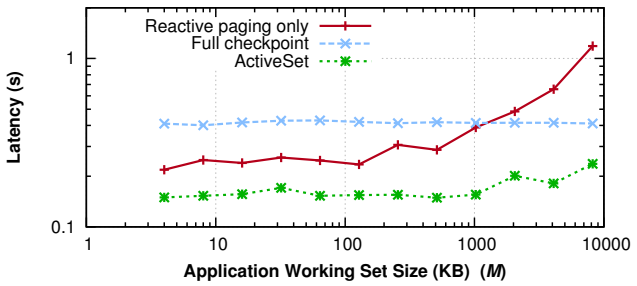

Figure 7: Response latency for our strawman application when comparing Picocenter's ActiveSet technique to two baselines: fetching memory only with reactive paging from cloud storage (without ActiveSet), and downloading full application checkpoints every time. Our strawman application is configured with a 64 MB memory size, and we vary the working set size between 4 KB and 8 MB. Note that both axes are log scale. The ActiveSet technique significantly outperforms the baseline approaches.

is largely due to S3's variance in fetching content; we found that S3 fetch times can occasionally vary by up to an order of magnitude or more, even when fetching data of the same size from the same S3 bucket. Conversely, we found CRIU restore times to be largely consistent.

Finally, we observe that when reviving applications from cloud storage, Picocenter has about 70 ms of overhead not accounted for by CRIU and S3 (these are most clearly seen with the Virginia clients). This overhead is due to the setup cost of the `btrfs` snapshot, which takes about 70 ms on our Linux snapshot. This could potentially be reduced by applying ActiveSet to the application's file system, but we leave this to future work.

Overall, these results show that Picocenter is able to process application-level end-to-end requests quickly, even when applications are swapped out completely to cloud storage. In these experiments, Picocenter restores "cold," swapped-out processes in 158–219 ms and "warm" processes in 101–122 ms. Recall that this overhead only occurs when a client contacts an application that is swapped out; once the application is restored, it is in the "hot" state until it goes idle again and the worker eventually swaps it out.

6.4 ActiveSet

Next, we evaluate the impact that ActiveSet has on the performance of swapping in an application from cold storage. Recall that swapping in must be done quickly in order to maintain the illusion to clients that all applications are running all the time, and that ActiveSet helps achieve this abstraction by transferring the pages that are predicted to be needed to handle a request.

Strawman application To provide for controllable experiments with ActiveSet, we implement a strawman application. This application allocates a block of memory of N pages. Each time it receives an incoming network request, it accesses $M < N$ of these and subsequently returns a response to the client. The value of M represents the application's working set size. To allow for different access patterns (and working set predictability), the application is also configured to "shift" the pages it accesses by S pages on each request. For example, if $S = 0$, the program will access the same M pages every time it is revived; if $S = 1$, it will access $M - 1$ of the same pages it accessed previously, and one new page; and when $S = M$, the process will access different sets of pages on each request. We measure the client-side latency for the application to respond.

ActiveSet performance We begin by exploring how ActiveSet improves the speed of reviving applications. To do so, we create two baseline implementations of Picocenter: *First,* we create a version of Picocenter that disables ActiveSet, and loads each memory page reactively. *Second*, we create a version of Picocenter that performs a full download of the application's memory image before reviving it, obviating the need for reactive page faulting. We run an experiment using our strawman application, configuring it to have a fixed 64 MB memory size (N), and we vary the working set size between 4 KB and 8 MB (M). For these experiments, the application accesses the same set of memory pages each time (i.e., $S = 0$). We repeat each experiment 20 times and report the median latencies in Figure 7.

We make three observations from these results. *First*, we observe that using ActiveSet provides significantly lower latency than the other techniques. The latency does increase with the size of the working set, but stays under 250 ms even when the application has an 8 MB working set size. *Second*, we observe that, as expected, the time to revive an application using the full checkpoint download is independent of the working set size; this is because there is no reactive faulting to cloud storage. However, the cost of downloading and restoring a full checkpoint is quite high: over 400 ms in this experiment.

Third, we observe that the latency for using only reactive paging performs between ActiveSet and full checkpoints for small working set sizes, but eventually becomes more expensive than full checkpoint once our strawman's working set size grows beyond 2 MB. Reactive paging becomes expensive because the memory accesses end up generating a number of sequential round-trips to cloud storage to fetch all of the memory pages required. These results collectively show the benefits of applying ActiveSet to quickly swap in cold applications.

Errors in ActiveSet In the experiment above, the ActiveSet prediction was always correct (i.e., the process always used exactly the memory pages in its ActiveSet). Next, we explore how ActiveSet performs when its prediction is incorrect. To do so, we use the "shift" parameter (S) to vary the overlap between the current working set and the set of memory in the ActiveSet. Because our two baselines are not affected by this shift, we examine only Picocenter with ActiveSet here. For

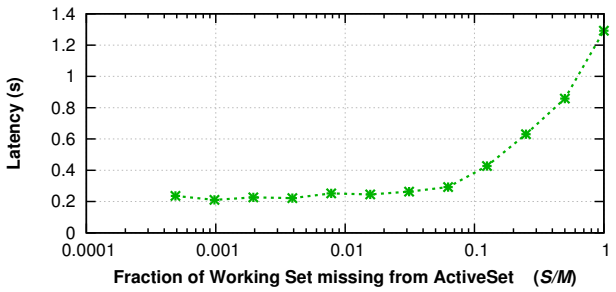

Figure 8: Response latency for our strawman application when different fractions of the application's working set is missing from the ActiveSet. Our strawman application is configured with a 64 MB memory size and a working set of 8 MB. The latency remains low, even when large fractions of the application's memory have to be fetched from cloud storage.

this experiment, we configure our strawman application with a memory size of 64 MB and a working set size of 8 MB.

We vary ActiveSet's prediction error from 0.5% to 100%, and present the results in Figure 8. We can immediately observe that when ActiveSet's prediction is accurate (the left part of the graph), the response time is consistently low. Once ActiveSet misses 10% or more of the true working set, the response time increases linearly as the application is required to make multiple sequential requests to fetch memory from cloud storage. Comparing these results to the previous section's baseline that downloads the entire checkpoint, we note that it is better to download a full checkpoint once the ActiveSet accuracy falls below 80%, for this strawman application. These results collectively show that ActiveSet need not have near-perfect prediction accuracy to significantly improve performance over full or reactive-only checkpointing.

Extreme case: ClamAV As an example of a challenging real-world application, we also evaluated the ClamAV antivirus application. ClamAV is a natural fit for Picocenter; users can direct newly downloaded files to be scanned by the application (thus requests are likely infrequent), and its processing time is typically relatively short (on the order of seconds). At the same time, however, ClamAV effectively stress-tests our ActiveSet design; its memory footprint is 300 MB, and its ActiveSet is typically 54 MB (18% of the total memory footprint). Moreover, as one asks it to scan different files, the memory access patterns will likely differ significantly as the application accesses different parts of its internal index of malicious signatures.

We ran ClamAV in Picocenter on a machine in Virginia, and have a client in Oregon serve files for it to scan. Each time we query ClamAV, we randomly generate 10 files of 50 KB each for it to scan. We repeat the experiment 10 times and report the median. Overall, we find that the latency to scan all files is 5.16 seconds[4] when ClamAV is actively running ("hot") and is 7.49 seconds when ClamAV is swapped out to cloud storage ("cold"). Thus, even with ClamAV's challenging memory access, Picocenter's use of ActiveSet largely hides the fact that the application was fully swapped out to cloud storage (almost all of the additional latency for the latter case is due to requests to cloud storage for memory pages that are not contained in the ActiveSet). Indeed, if we resorted to using the full checkpoint approach (downloading the entire application before reviving it), we found the latency for the same experiment would be 9.61 seconds.

Summary We note that the ultimate success of Picocenter is a function not of how much memory the process allocates in total, but of the process's working set. In practice, many VMs actively use a small fraction of their overall allocation [6]; we have observed similar trends with individual processes. These results show that Picocenter is able to swap a process from cloud storage to running quickly enough to be negligiable for *most* applications—even when the working set of the process is quite large. Moreover, we plan on exploring additional techniques to improve the accuracy of the ActiveSet predictions; the results shown here use only a simple heuristic that predicts an application will use exactly the memory pages it used in the previous request.

6.5 Cost

As a final point of evaluation, we briefly estimate the cost to tenants of running applications in Picocenter. We consider a strawman Picocenter provider that simply uses existing cloud services (e.g., Amazon's AWS) to provide the infrastructure for Picocenter, effectively acting as a reseller. The actual cost will depend on a large number of factors, but our goal here is to simply provide a rough estimate.[5]

We begin by estimating the charging rates for active and inactive applications based on the EC2 and S3 costs of today (i.e., we estimate the cost of leasing EC2 machines to the Picocenter provider). To estimate R_a, the rate charged per GB of memory for active applications, consider the `c4` generation of EC2 instances on AWS: each of these present a cost of $0.00052/GB-minute ($22.46/GB-month), so we expect R_a to be on this order of magnitude. To estimate R_i, the rate charged per GB of memory for inactive applications, we draw directly from the S3 costs of today, providing an expected R_i of $0.0000007/GB-minute ($0.03/GB-month).

The cost to a tenant would be dependent on the memory size of the application, the size of its working set, and the fraction of time that it is active. If we consider a user running an Nginx instance (average memory usage of 65 MB) that is active 10% of the time and uses 10% of its mem-

[4] Recall that with ClamAV actively running, not all of its memory is necessarily resident (some may still be swapped out to cloud storage, and will cause reactive Picocenter-level faults if accessed). For comparison, if ClamAV is entirely memory-resident, we find that the latency to scan all files is an average of 3.36 seconds.

[5] In particular, our estimate does not account for many costs to the provider, including development, personnel, and any profit margin, but the provider would also likely be able to use techniques like Reserved Instances to obtain much cheaper rates.

ory on each invocation, we can derive that the cost of running this application in Picocenter would be approximately $0.0165/month. Compared with a dedicated VM (cheapest cost of $9.36/month), this represents a 99.82% savings.[6]

7. Conclusion

In this paper, we presented a new approach for cloud computation that is designed to support long-lived, mostly-idle (LLMI) applications. Running such applications in today's IaaS cloud is economically inefficient, and prohibitive for typical end-users, as they are charged disproportionately to how much work their jobs perform. On the other end of the spectrum, PaaS models restrain the set of applications that may be run, precluding many useful LLMI applications, such as a personalized email server or a personalized mobile offloading application.

We presented Picocenter, a hosting infrastructure for end user LLMI applications. Our evaluation on a real-world cloud infrastructure shows that Picocenter represents a unique operating point compared to existing service models. Compared to PaaS, Picocenter is not constrained to a particular programming model—we showed that non-trivial applications can be run by just using their off-the-shelf binaries. Compared to IaaS, Picocenter is able to efficiently *swap* applications to and from cloud storage—we showed that Picocenter incurs low latency overhead when swapping applications from cloud storage to running. LLMI applications in Picocenter thus consume fewer resources, allowing users to be charged only for what they use, and allowing cloud providers to accept more jobs.

This paper represents a promising first step towards a new, finer-grained model for cloud computation; many areas of future work remain. Our results on swapping in the ClamAV application demonstrate that, although ActiveSet helps considerably, more optimization may be needed to maintain our always-running abstraction for applications with extremely large working sets.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Evangelia Kalyvianaki, for their helpful comments. This research was supported by NSF grants CNS-1409191, CNS-1319019, CNS-1421444, CNS-1409426, CNS-1409249, and CNS-1345284.

A. Supporting ActiveSet in CRIU

Picocenter uses CRIU for checkpointing and restoring containers. Ideally, swapping in a process would be as simple as loading the process's ActiveSet into memory and restoring the container. Unfortunately, CRIU can only restore a container after all of its pages written during a previous checkpoint are populated into memory. To modify CRIU to support ActiveSet, it was essential that not every page of memory was read before the process was restored.

To restore, stock CRIU spawns a new process and performs a "premap" whose size corresponds to each virtual memory area (VMA) that was a part of the previously checkpointed process. It subsequently reads data from the checkpoint data file to restore the data for these mappings. To support ActiveSet, rather than creating an anonymous mapping and then filling it with data, we perform a file-backed `mmap` with the appropriate offset and length in the checkpoint data file. For pages that were occupied, this conceptually consists of merely changing a `read` into a `mmap` each time the CRIU process attempted to read pages into the VMA. However, we were obligated to retain the unoccupied pages, so we still performed the initial premapping for every VMA. Each time we came across a memory area that traditionally would be read off of consecutive bytes on disk, we perform the file-backed `mmap` over this potential subset of the memory region. The Linux kernel will automatically split the VMA, if necessary, into anonymous unoccupied areas and consecutive file-backed areas. Since CRIU stores consecutive allocations on disk consecutively, the result is few regions.

Since each of these mappings are private, the region is essentially initialized with the contents of the file whenever a fault occurs, whereas private mapping semantics ensures that written pages are no longer associated with the backing file. In these cases, the written pages will be persisted as part of a subsequent checkpoint.

Once all of the memory maps have been completed, as the nascent process finishes restoration, each memory area must be remapped to the memory location it occupied before the previous dump. To accomplish this, stock CRIU performs a `mremap` over each original memory area. However, when the VMA data is sent to the CRIU parasite code [12], which runs in the context of a protean form of the restored process, we no longer have access to all book-keeping (and individual sub-mappings) done in the original restore process. While this could be rectified with additional information passing, our understanding of the mappings is especially confounded because some of these splits took place in the kernel, occassionally in surprising ways. We cannot simply perform a `mremap` over the entire original memory area, because `mremap` is not capable of remapping VMAs that span more than one mapping. To rectify this, we query the kernel to find all extant mappings for the process, which now includes areas that were internally split, and perform `mremap` calls for each area as it exists in the kernel that form subsets of the original mapping data.

The result is we were able to extend CRIU to support checkpoint/restore of containers while simultaneously allowing lazy loading of memory pages as the process runs.

[6] Current IaaS providers charge by the hour; should they move to a usage-based charging model we believe that these IaaS services will still be a poor fit for LLMI due to operational overheads of maintaining/managing VMs.

References

[1] Amazon lambda. http://aws.amazon.com/lambda/.

[2] J. Ansel, K. Aryay, and G. Coopermany. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *Proccedings of the IEEE International Symposium on Parallel & Distributed Processing (IPDPS'09)*, 2009.

[3] R. Baden, A. Bender, N. Spring, B. Bhattacharjee, and D. Starin. Persona: an online social network with user-defined privacy. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'09)*, 2009.

[4] 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. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[5] S. Bazarbayev, M. Hiltunen, K. Joshi, W. H. Sanders, and R. Schlichting. Pscloud: a durable context-aware personal storage cloud. In *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems (HotDep'13)*, 2013.

[6] N. Bila, E. de Lara, K. Joshi, H. A. Lagar-Cavilla, M. Hiltunen, and M. Satyanarayanan. Jettison: Efficient Idle Desktop Consolidation with Partial VM Migration. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12)*, 2012.

[7] BIND DNS Server. https://www.isc.org/downloads/bind/.

[8] btrfs. https://btrfs.wiki.kernel.org/index.php/Main_Page.

[9] S. Buchegger, D. Schiöberg, L. H. Vu, and A. Datta. PeerSoN: P2P Social Networking—Early Experiences and Insights. In *Proceedings of the International Workshop on Social Network Systems (SNS'09)*, 2009.

[10] E. Chen, S. Ogata, and K. Horikawa. Offloading Android applications to the cloud without customizing Android. In *Proceedings of the 2012 IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops'12)*, 2012.

[11] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation (NSDI'05)*, 2005.

[12] CRIU parasite code. https://criu.org/Parasite_code.

[13] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys'10)*, 2010.

[14] Docker. https://www.docker.com.

[15] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging Legacy Code to Deploy Desktop Applications on the Web. In *Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation (OSDI'08)*, 2008.

[16] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. The Utility Coprocessor: Massively Parallel Computation from the Coffee Shop. In *Proceedings of the USENIX Annual Technical Conference (USENIX'10)*, 2010.

[17] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software: Practice and Experience*, 21(8):757–785, July 1991.

[18] Easy way to prevent heroku idling? http://stackoverflow.com/questions/5480337/easy-way-to-prevent-heroku-idling.

[19] Filesystem in Userspace. http://fuse.sourceforge.net.

[20] Google App Engine. https://developers.google.com/appengine.

[21] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'09)*, 2009.

[22] HAProxy: The Reliable, High Performance TCP/HTTP Load Balancer. http://www.haproxy.org.

[23] P. H. Hargrove and J. C. Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. *Journal of Physics: Conference Series*, 46(1):494, 2006.

[24] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)*, 2011.

[25] M. R. Hines and K. Gopalan. Post-copy Based Live Virtual Machine Migration Using Adaptive Pre-paging and Dynamic Self-ballooning. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'09)*, 2009.

[26] J. Howell, B. Parno, and J. R. Douceur. Embassies: Radically Refactoring the Web. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, 2013.

[27] J. Howell, B. Parno, and J. R. Douceur. How to Run POSIX Apps in a Minimal Picoprocess. In *Proceedings of the USENIX Annual Technical Conference (USENIX'13)*, 2013.

[28] P.-H. Kamp and R. N. Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference (SANE'00)*, 2000.

[29] E. Keller, S. Ghorbani, M. Caesar, and J. Rexford. Live migration of an entire network (and its hosts). In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks (HotNets'12)*, 2012.

[30] R. Kemp, N. Palmer, T. Kielmann, and H. Bal. Cuckoo: a computation offloading framework for smartphones. In *Proceedings of the EAI International Conference on Mobile Computing, Applications, and Services (MobiCASE'12)*, 2012.

[31] 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 *Proceedings of the USENIX Annual Technical Conference (USENIX'14)*, 2014.

[32] T. Knauth and C. Fetzer. DreamServer: Truly on-demand cloud services. In *Proceedings of International Conference on Systems and Storage (SYSTOR'14)*, 2014.

[33] T. Knauth, P. Kiruvale, M. Hiltunen, and C. Fetzer. Sloth: SDN-enabled activity-based virtual machine deployment. In *Proceedings of the Third workshop on Hot Topics in Software Defined Networking (HotSDN'14)*, 2014.

[34] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM'12)*, 2012.

[35] Kubernetes: Container cluster manager from Google. `https://github.com/kubernetes/kubernetes`.

[36] O. Laadan and J. Nieh. Transparent Checkpoint-Restart of Multiple Processes on Commodity Operating Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX'07)*, 2007.

[37] lighttpd Web Server. `http://www.lighttpd.net`.

[38] Linux Checkpoint/Restore In Userspace. `http://criu.org`.

[39] Linux kernel based virtual machine. `http://www.linux-kvm.org`.

[40] Lxc. `https://linuxcontainers.org/`.

[41] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, D. Sheets, D. Scott, R. Mortier, A. Chaudhry, B. Singh, and J. Ludlam. Jitsu: Just-in-time summoning of unikernels. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*, 2015.

[42] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*, 2013.

[43] A. Madhavapeddy and D. J. Scott. Unikernels: Rise of the virtual library operating system. *ACM Queue*, 11(11), 2013.

[44] A. Mohaisen, E. Y. Vasserman, M. Schuchard, D. Foo Kune, and Y. Kim. Secure encounter-based social networks: requirements, challenges, and designs. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'10)*, 2010.

[45] B. Nicolae, F. Cappello, et al. Towards efficient live migration of i/o intensive workloads: A transparent storage transfer proposal. In *Proceedings of the ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC'12)*, 2012.

[46] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: a scalable fault-tolerant layer 2 data center network fabric. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'09)*, 2009.

[47] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the library OS from the top down. *ACM SIGPLAN Notices*, 46(3):291–304, 2011.

[48] Postfix Email Server. `http://www.postfix.org`.

[49] D. Price and A. Tucker. Solaris Zones: Operating System Support for Consolidating Commercial Workloads. In *Proceedings of the 18th USENIX Conference on System Administration (LISA'04)*, 2004.

[50] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. *ACM SIGOPS Operating Systems Review*, 36(SI):377–390, 2002.

[51] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys'13)*, 2013.

[52] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys'07)*, 2007.

[53] tent.io. `http://tent.io`.

[54] K. S. Trivedi. On the paging performance of array algorithms. *IEEE Transactions on Computing*, 26(10):938–947, Oct. 1977.

[55] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth ACM/EuroSys European Conference on Computer Systems (EuroSys'15)*, 2015.

[56] C. A. Waldspurger. Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[57] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the USENIX conference on Operating Systems Design and Implementation (OSDI'94)*, 1994.

[58] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 233–240. IEEE, 2013.

[59] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'09)*, 2009.

[60] E. Zayas. Attacking the process migration bottleneck. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles (SOSP'87)*, 1987.

[61] Zerovm. `https://www.zerovm.org/`.