Nephele: extending virtualization environments for cloning unikernel-based VMs

Anonymous Author(s)

Abstract

Unikernels gained an increasing interest in the recent years because they provide efficient resource allocation and high performance for cloud services by bundling the application with a minimal set of OS services in a guest VM. Although a unikernel is by design small and lightweight, fleets of unikernels based on the same image are not necessarily more efficient than containers because the latter can rely upon OS primitives for sharing memory. Futhermore, porting POSIX applications on top of unikernels brings a new challenge: what does `fork()` mean in the world of unikernels where there is memory isolation within a VM? Lacking `fork()` support significantly reduces the applicability of unikernels in popular cloud applications.

In this paper we address these shortcomings and show that cloning unikernels makes way to further improvements and enables full functionality of popular cloud applications, such as NGINX and Redis. Our solution, Nephele, extends the Xen virtualization platform and provides autoscaling capabilities to unikernel based VMs. Nephele provides 8x faster instantiation times and can run 3x more active unikernel VMs on the same hardware compared to booting separate unikernels.

1 Introduction

Virtualization is the key enabler of cloud computing as it securely shares hardware between untrusted tenants. Mechanisms such as paravirtualization together with hardware support for virtualization (e.g. EPT for memory management) have enabled the compute and I/O performance of virtual machines to near those of bare-metal deployments, further boosting cloud adoption.

Since most VMs run one or few applications, there is an ongoing trend to specialize such VMs for their target application, with two goals: increasing performance and reducing resource consumption (RAM, storage, CPU cycles) and thus cloud costs. Such specialization spans a range of options, from using minimalistic Linux distributions (such as Alpine) to using unikernels ([33, 36]), and offers massive resource reductions: VM images and runtime memory usage of a few megabytes and boot times on the order of tens of milliseconds. However, porting existing applications to run as unikernels depends on the set of system services those applications use and today there is no support for the popular `fork()` call. Fork-based cloning not only helps portability, but can significantly lower resource consumption when multiple VMs are run using the same base image.

To understand why this matters let's consider a top-three cloud provider that keeps a fairly large idle pool of running VMs for Function as a Service workloads to handle new requests, simply because booting a new VM on demand would take too long. This solution however wastes significant resources to achieve its goals. Although memory sharing mechanisms such as Copy-on-Write (COW) do offer benefits for efficient memory within a VM, little support exists in the case of inter-VM resource sharing.

Existing solutions for memory deduplication [7, 12] used for merging pages with identical content between VMs of different tenants were proved to be vulnerable when the attacker VM can guess the contents of victim VM pages [54, 61]. However, this can be overcome by controlling the set of VMs that are subject to same memory sharing, i.e. by removing the eventuality of an attacker VM. Allowing memory sharing only between the clones of a trusted VM, all of them belonging to the same tenant, similarly to related processes on traditional OSes, is one example following this security constraint.

Porting POSIX applications on top of unikernels gives rise to a new challenge: how should the `fork()` call behave inside a unikernel? The first and main result of the call, creating a new process, is in total opposition with the definition of unikernels which outline that unikernels are single-process OSes [41]. For this reason, porting popular cloud applications that make use of `fork()` did not yet enable the full potential of those applications. NGINX, a popular web server, relies on `fork()` to scale up its throughput with every worker process it creates, while Redis, a similarly popular key-value store, depends on `fork()` for fault tolerance by making COW snapshots of its in-memory database which are further saved on persistent storage.

In this paper we propose a solution for cloning unikernel-based VMs and that addresses these problems. We aim to follow the minimalistic principles that cover the design of unikernels and enable cloning with minimal changes on guests code base, without imposing any programming model. Another goal we aim to achieve is to remain true to the definitions of both unikernels and process-related POSIX primitives, such as process creation functions and inter-process communication mechanisms, and to reconcile and integrate them in order to close the gap towards full POSIX compatibility. Our solution keeps the transparency of the `fork()` call for guests by leaving most of the work for the virtualization environment.

Nephele extends the hypervisor interface only with a single new hypercall that handles cloning related operations, leaving configuration options to be controlled only by the

toolstack. Nephele extends I/O paravirtualization to enable multiplexing of physical devices and enables shares of both system and application memory regions, bringing unikernel memory consumption to be better than with containers.

Our evaluation shows that Nephele provides 8x faster instantiation times and can run 3x more VMs on the same hardware compared to booting separate unikernels.

2 Problem statement

Unikernels are guest virtual machines that bundle a single application and the OS services required to support it in a single address space, removing the traditional user space and kernel space separation used in mainstream operating systems; this enables a performance boost and the ability to simply remove unused OS code, which is typically linked statically against the application [36]. Applications running in unikernels can use threads but cannot spawn new processes.

This means running applications that use multiple processes in a unikernel is currently difficult; simply spawning multiple threads is insufficient as they do not provide memory isolation and other process semantics (such as copy-on-write) that applications need to do their work. NGINX and Redis, for instance, are very popular cloud applications that rely on `fork()` to do their job.

In this paper we add cloning support for unikernel VMs, thus enabling the implementation of `fork()` alongside other compelling use-cases that we discuss in §6. We first note that implementing cloning for generic virtual machines has been explored in some depth (e.g. see Snowflock [38]), but it hasn't gained much popularity in production mostly because its main use case was cross-machine scaling; in this context, datacenter-scale load balancers coupled with identical backends proved to be the winning solution [22, 46, 48].

Lightweight applications do need multiple address spaces for a myriad of reasons, and the solution is not to retrofit processes in unikernels [40] as it adds increased complexity and reduces performance. Instead, we can exploit the fact that unikernels are guest VMs each in their own address space; to support multiple address spaces we can simply clone the existing unikernel, much like `fork()` replicates the calling process, resulting in a new VM that is identical to the existing one and runs on the same physical machine.

In Unix operating systems, starting a new process requires running `fork()` to create a process and then replacing the image of this process via an `exec()` call. For unikernels, we can already `boot` a new VM using the virtualization toolstack, thus we focus only on supporting cloning in this paper.

The main requirements for our solution are:

- **Improve performance**: cloning should be faster than booting new VMs.
- **Transparent operation**: both parent and child VMs (or domains) should continue to work seamlessly after the completion of the cloning operation, without requiring any code changes; in other words, cloning should behave similarly to calling `fork()`.
- **I/O cloning**: cloning should go beyond duplicating address spaces and copy-on-write to enable disk and network I/O to function seamlessly after cloning.
- **Inter-VM communication**: in many cases, related processes communicate via IPC such as pipes or shared memory. The same primitives should be supported transparently for cloned unikernels.

To preserve overall system security, the new cloning interfaces exposed by the hypervisor are kept to the absolutely minimum required. The virtualization toolstack must assist in carrying out security checks when creating the parent domain to keep the behaviour of cloning domains under control.

Previous work, ranging from solutions targeting classical multiprocess VM cloning to unikernel replication, addressed some of the properties stated above, but none addressed them all. Solutions that addressed introducing process abstraction back to unikernels either broke the unikernel definition by adding multiprocess support inside the VM [40] or, when using multiple unikernel instances, needed coordination between the toolstack and the guest requesting replication [34, 64], thus breaking the transparency property. Moreover, the approach based on coordination takes an unavoidable major toll on performance when the child domain is also involved [64]. The challenging matter of I/O cloning was either not addressed at all [40, 64] or addressed for a specific use case (e.g. networking for load balancing [34]).

The VM fork abstraction was first introduced by Snowflock [38] which addressed multiprocess HVM guests on Xen. The replication is triggered by the toolstack which also assists in replicating virtual CPU and memory states. This approach brings its own performance hits, but on the other hand it has the advantage of bringing little requirements to the hypervisor which only adds support for memory sharing. This feature is used by the toolstack to change the ownership of pages previously owned by the parent guest and to mark those pages for COW. It was recently revived for the use case of Linux-based VMs fuzzing [39] and we extended further for memory cloning support in Nephele.

Just like in the case of unikernel-based VM cloning, I/O cloning is also a tough challenge for multiprocess VM cloning. Previous solutions either need some rewriting of internal I/O-related guest state (e.g. network interface addresses [38]) or they target very specific use cases [49, 52].

We focus our solution on the Xen hypervisor [13], but implementation would be similar for KVM or other virtualization solutions relying on paravirtualization. We now provide sufficient background for the Xen hypervisor and then go into detail in the following subsections discussing the overall architecture of our solution.

3 Xen Background

The Xen[13] hypervisor has pioneered paravirtualization where guest virtual machines are aware they are running in a virtualized environments and can use efficient hypercalls (the equivalent of system calls in a virtualized context) to implement functionality otherwise very expensive to emulate, for instance access to I/O devices.

To understand how cloning can be efficiently supported in a paravirtualized system such as Xen, we provide an overview of guest instantiation in Xen

VM instantiation on Xen is made of different steps that are managed by the toolstack. The default Xen toolstack comprises the `xl` command line interface and its libraries: `libxl`, a library that includes most of the toolstack functionality and which provides the API that needs to be used by all Xen toolstack implementations, and `libxc`, an essential library for accessing hypervisor services in an OS agnostic manner.

The toolstack resides inside a privileged VM called *Domain 0* (Dom0) which is automatically instantiated on system boot; all other domains (called DomU) have to be explicitly created by the toolstack. The hypervisor manages only the minimum critical set of resources, namely CPU, memory, timers and interrupts, while the control of the access to other hardware devices is delegated to Dom0, which already contains the required device drivers to support those devices.

This approach enables Xen to leverage the full hardware compatibility provided by Dom0 and to make use of it by means of multiplexing services. In the case of paravirtualized devices, Xen introduces the split-device model: each device is split into a frontend residing in the guest and a backend device typically residing in Dom0. The drivers communicate for both initialization and data transmission by using the primitives provided by Xen: *grants* for shared memory and *events channels* as notification mechanisms. Hardware devices are multiplexed for different backends by using software switches (e.g. bridges for network devices) in Dom0.

On VM instantiation, the toolstack initiates requests to hypervisor to allocate guest virtual CPUs and memory. I/O devices are created by leveraging the Xenstore key-value store which is used as a device registry with notifications support for updates. During the first stage of device initialization, backend and frontend drivers negotiate the means they will use for communication (i.e. grants and event channels) and the device capabilities. If the negotiation succeeds, both drivers are marked as connected and a virtual device is created on each end, making way for the second stage which mainly includes the Dom0 user space operations that are required to enable device multiplexing (e.g. adding the newly created virtual network interface to a software bridge).

4 Nephele: cloning unikernels efficiently

Nephele implements cloning support for the Xen hypervisor; at a high-level, Nephele adds copy-semantics for each of the components involved in the instantiation process and performs most of the work in the virtualization platform, keeping guest changes to a minimum. Nephele adds a new hypercall to the hypervisor and implements the other needed operations in Dom0 via changes to the virtualization toolstack.

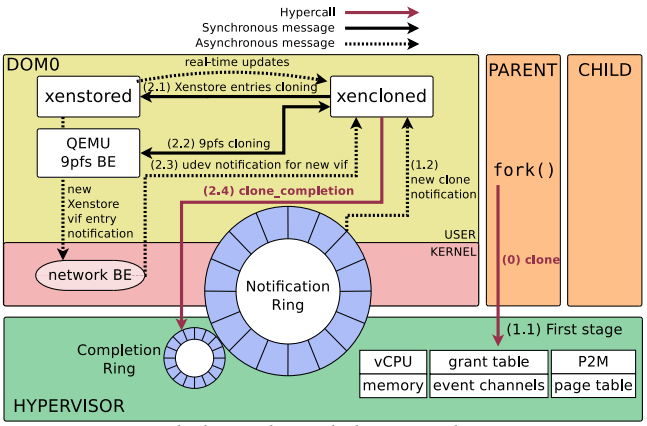

Figure 1. Nephele: unikernel cloning solution overview

The operation of Nephele is shown in Figure 1. When a unikernel VM requests cloning, Nephele creates one or more address spaces which have the same contents as the calling VM; each address space is mapped to an (almost) identical unikernel VM. This is the first stage of the cloning process, and is executed by the hypervisor.

Besides cloning CPU and memory, Nephele handles cloning the I/O state as well, in a seamless manner, without affecting performance. This is handled in the second stage of the cloning process by `xencloned`, a new toolstack component running in Dom0 and that performs the userspace operations that are required for completing the state of a clone.

Using the cloning hypercall from a VM is as easy as calling `fork()` from a process: guests do not get involved in extra coordination for completing the cloning operation, they only wait for Nephele to take care of everything.

While implementing Nephele we add copy-semantics to the virtualization components that handle domain creation (booting), and this enables us to reuse much of the code involved in booting. However, the copy operations are optimized and bypass the steps needed only in booting.

Our evaluation shows that the performance of cloning is significantly better than booting (up to 8x faster, see §5), showing that cloning is superior to VM booting.

In the next sections we describe in detail the operation and components of Nephele, starting with an overview of the two stages in the cloning process.

Nephele Operation Overview. The hypervisor runs the first stage of the cloning process which begins with initializing the internal structures of the child domain (e.g. page table) by copying and editing the data structures of the parent. Next, it clones the vCPUs states, memory, event channels and the grant table; these steps are detailed in §4.2.

Once this stage is completed, the hypervisor will fill a new entry in the notification ring buffer and will send a notification via a virtual interrupt (i.e. VIRQ_CLONED) to the xencloned daemon. The completion of stage 2 is signaled back by xencloned to the hypervisor via the CLONEOP hypercall. The notification can also act as backpressure, slowing down the first stage of the cloning process when the notification ring is full.

The xencloned daemon is vital for cloning unikernels because it coordinates all the userspace operations needed in the second stage. Once a new clone is created, xencloned first introduces it to xenstored daemon. Next it writes all the Xenstore entries of the clone based on the parent entries: backend and frontend entries for devices (e.g. console, network, 9pfs filesystem), grant reference and event port for communication with the Xenstore daemon, etc.

A part of the backend drivers is subscribed to Xenstore changes, so they will receive notifications once a new update is made. For example, the network backend will be notified once a new virtual network interface (vif) is written in the Xenstore registry and it will proceed with creating the actual interface. On interface creation, udev [35] events will be generated and sent to userspace where they will be handled by xencloned which in turn will carry on with the userspace operations required for the completion of the device initializations (e.g. adding the newly created vif to a bridge).

The rest of the backend drivers are instantiated explicitly by the toolstack. For example, on booting, the 9pfs filesystem backend is launched as a process by xl for any new guest that uses 9pfs filesystems. On cloning, xencloned sends a request to the 9pfs backend process (which was launched previously on the creation of the parent domain) to clone the parent 9pfs state.

The parent domain is paused until the completion of stage 2 in order to keep its state consistent for all its clones. On completion xencloned notifies the hypervisor and the parent is resumed. The child domains are either resumed or left in paused state, depending on how they are configured.

4.1 New interfaces for cloning

Following our goal of keeping the new interfaces to a minimum, we introduce a single hypercall called CLONEOP for all the cloning operations exposed by the hypervisor to both toolstack and guests, each operation being implemented as a subcommand of the hypercall. Our current implementation includes subcommands that enable cloning globally/for each guest, initiate cloning and notify the hypervisor of the completion of I/O cloning.

Guest-hypervisor interface. The first stage of cloning starts when a guest calls the clone subcommand of the CLONEOP hypercall to clone itself. The call specifies the machine address of the start_info page (an essential Xen specific page which acts as a directory for critical information like page table base address, special I/O frame numbers, etc.), the number of clones to be created and the address of an array which will be filled by the hypervisor with the domain IDs of the clones. The guest is paused until the clone operation finishes for all the required children. Similarly to calling fork(), the clone operation is transparent and on its completion both the parent and child domains can continue their execution without any additional steps. The clone subcommand can also be called from Dom0 when cloning is triggered from outside the VM (e.g. for VM fuzzing); in this case the domain ID of the guest being cloned is explicitly passed as a parameter of the hypercall.

We added support for inter-domain communication between parent domain and its clones by leveraging the primitives provided by Xen for sharing memory between domains and for sending notifications, namely grant references and event channels, respectively. We extended the grant references API with the DOMID_CHILD wildcard value for indicating the domain being granted with memory access because, unlike regular granting which needs the domain being granted to be explicitly specified, sharing memory to clones can be initiated before knowing the domain ID of any of them. Likewise, when creating event channels to be used in inter-domain communication between parent and clones, the same DOMID_CHILD wildcard value is used instead of setting an actual domain ID to specify the domain at the other end of the channel.

Toolstack-hypervisor interface. A guest can be cloned only if its xl configuration file specifies a non-zero value for the maximum number of clones. We extended the domain control interface (domctl) to enable and disable cloning for a given domain and to also pass the configured maximum number of clones to the hypervisor. Enabling cloning operations on a global level is initiated by the xencloned daemon on its start. When enabling cloning globally, xencloned also submits the memory address of the shared ring that will be used to receive cloning notifications from the hypervisor. A notification contains only the minimum required information for xencloned to proceed with the second stage, e.g. the domain IDs and the frame numbers of start_info pages of both parent and child.

We also added a new virtual interrupt, VIRQ_CLONED, for notifying cloning events. After adding a new entry to the cloning notifications ring, the hypervisor triggers a VIRQ_CLONED interrupt to wake the xencloned daemon.

On stage 2 completion, xencloned calls the clone_completion operation of the CLONEOP hypercall to notify the hypervisor that all the required userspace operations are completed. Completion events occur asynchronously and out-of-order considering that guests running on the same machine can have different I/O configurations, with different types of devices.

4.2 Breakdown analysis of the cloning process

Given the complexity of the cloning process, in this section we detail the steps involved, how abstractions are cloned and how resources are reused. Since a clone can also be the parent

of another clone, we define a *family* to be the set of domains that are related, i.e. two domains are members of the same family if and only if they do have some common ancestor domain or one of them is the ancestor of the other.

vCPUs. The vCPUs states of the child domain replicate the states of parent vCPUs. The CPU affinity remains the same for the child domain as well. The user registers are also replicated, excepting the `rax` register which saves the hypercall return value; on success it will be zero for the parent and one for any child. On return from the `CLONEOP` hypercall, the parent can access the children domain IDs using the array it provided when invoking the hypercall and which is filled by the hypervisor. The child domain can retrieve its own domain ID using the standard ways (e.g. via Xenstore).

Memory. Similarly to the `fork()` syscall, memory cloning for unikernels involves sharing the same physical memory between parent and children, with a number of exceptions. Memory pages that are writable prior to cloning are marked as read-only and will be copied-on-write (COWed) when one of the family instances will attempt to write data into it.

We reused and extended the page sharing mechanism, which was introduced by Snowflock [38] for HVM guests, to support paravirtualized guests as well. On sharing a page, its ownership is transferred from their original domain owner to a special domain called `dom_cow`, while its reference counter is incremented for each domain that uses it. Once the reference counter reaches one, meaning a single owner, on the next page fault the ownership will be transferred from `dom_cow` to the domain generating the fault, which may be different from the original owner domain.

Memory regions that are domain specific are excepted from sharing and need to be measured before starting the cloning process. We will call such memory as *private memory* and we will describe the types of private memory that we identified for our current implementation in the next paragraphs. Depending on how they are used, the private memory pages are either duplicated or rewritten. Examples of data that needs to be rewritten includes the parent domain ID references and frames references of private memory.

Our first example of private memory is represented by the memory pages used for building the child page table, which will map, as in the case of the parent, guest virtual addresses directly to machine physical addresses, an approach known in Xen terminology as *direct-paging* [13]. This is a Xen optimization which combines two stages of translation, from guest virtual to guest physical addresses and from guest physical to machine physical addresses, directly into a single stage. However, if we want to migrate the VM to a different machine then we also need a physical-to-machine (`p2m`) mapping for guest physical addresses to machine physical addresses. It is used on the target machine of the migration to rebuild the domain page table and is updated with the new machine frame numbers. The `p2m` mapping is another type of private memory and is also used and updated on cloning, similarly to migration, when building the child page table.

Another type of private memory is represented by I/O-related memory, a first example being the shared rings that are used for communication between the frontend and backend drivers of paravirtualized devices. In particular, the paravirtualized network devices use a one page ring-buffer which keeps references to up to 256 pages, each page used for saving a single network packet. An RX ring is always full, occupying a total of 257 pages that are duplicated on cloning, while a TX ring contains references only to packets that are not yet transmitted and therefore not freed. Special I/O pages, such as the console page, the Xenstore interface page and the `start_info` page are other examples of private memory.

The ratio shared-to-private memory affects the memory density on the same physical machine, the higher the ratio the higher the density. Statically linked unikernels tend to have high binary sizes, with a significant proportion of the memory containing text sections, making them great candidates for increasing the memory density by means of cloning.

As proved by prior work [65], child instantiation is dominated by the cloning of the page table when the total memory size of the parent VM is at least in the order of tens of megabytes. The same conclusion applies for cloning, as our microbenchmarks show in section 5.2.

4.2.1 Cloning I/O devices

Finding a suitable solution to support I/O seamlessly for clones is challenging: we need a generic and easy to extend approach in order to support an extensible list of device types, but the behavior and state information may differ significantly from one device type to another.

To achieve this, changes are needed to each of the components of the split-driver model (e.g. backend and frontend drivers, shared rings), together with userspace operations that finalize the I/O setup, as described next.

Xen uses Xenstore as a registry for device discovery, therefore extending it for cloning is vital to support I/O. Whenever a new domain is created, `xl` introduces it to the Xenstore daemon. With cloning, the domain introduction is initiated by `xencloned`, the introduction request being augmented with an additional parameter indicating the parent ID. Next, `xencloned` proceeds with initiating the cloning of each parent device by writing the new Xenstore entries for the frontend and backend drivers. On writing the entries, the backend drivers are notified and in turn create internal states for the new devices. On guest booting, successful initialization of a paravirtualized device implies a negotiation between the frontend and backend drivers, with each of the two drivers going through several states until the two ends agree on getting connected. With cloning, the negotiation is skipped and the two ends are created connected.

Both backend and frontend driver information is identified in the Xenstore registry by a unique key which includes

```
bool xs_clone(struct xs_handle *h, xs_transaction_t t, unsigned int parent_domid,
              unsigned int child_domid, enum xs_clone_op op,
              const char *parent_path, const char *child_path);
```

Figure 2. The Xenstore client library API was extended with the `xs_clone` function which clones the entries in `parent_path` to the new `child_path` directory. Depending on `op` parameter, it will either do a regular in-depth directory copy or it will use heuristics to adapt the entries if they described a device.

`xs_clone_op_basic`	Normal copying
`xs_clone_op_dev_console`	Console device cloning
`xs_clone_op_dev_vif`	Network device cloning
`xs_clone_op_dev_9pfs`	9pfs device cloning

Figure 3. `xs_clone_op` enumeration values

the owning guest ID. For clones, such keys (and values referencing them) must be rewritten to include the new clone ID. With most device types, this is the only kind of Xenstore information that has to be rewritten on cloning. Based on this observation, we introduced a new type of Xenstore request, `xs_clone`, which aims to optimize the cloning of Xenstore directories that contain device information. The backend and frontend keys are rewritten direcly by the Xenstore daemon; as a result the number of write requests to Xenstore is significantly reduced and, consequently, the I/O cloning duration much improved as we will show in section 5.1.

Frontend drivers exchange information with their backend counterparts using shared rings. To clone the rings we have two options: either the same ring is shared between the frontends of parent and clones or a new ring is created for each clone frontend. Sharing the ring is not a viable solution in this case because the states of frontends evolve independently and they would need extra coordination for concurrent access. Nevertheless, creating a new ring for a cloned device is not a straight-forward solution either because consideration must be taken regarding how rings are used in the case of each device type. For example, in the case of network devices we chose to copy the rings because their contents are tightly linked to the internal states of the guests: packets in the TX ring are created based on some pending requests that need to be serviced in both parent and child domains, while the RX packets might contain the waited responses. Another reason backing the copying of the entire RX rings comes from the fact that the entries in the RX ring are preallocated by the guest and may contain allocator metadata, as happens with the netfront implementation of the Unikraft unikernel. However, for the console device, we decided to not copy the ring because duplicating the parent console output for the child would hinder debugging.

Extending I/O cloning support with new device types requires changes only in the implementations of `xencloned` and of the backend driver. While `xencloned` needs to be extended with new functionality for cloning Xenstore entries, the changes for the backend driver should focus on reusing as much code as possible. For example, Nephele enables cloning support in the netback driver by only adding 14 lines of code which shortcut the control flow to directly connect the backend and the frontend.

Besides the Xenstore console device, which is critical for logging and debugging, Nephele adds support for cloning network and 9pfs devices, with each of the device types bringing its particular challenges. We continue by detailing the solutions for each of the supported device types.

Console devices Cloning console devices involves only creating the Xenstore entries for the child domain console. The Qemu process that manages the console backends is notified by Xenstore on the writing of the new entries and internally creates the state associated with the clone, without needing any changes in its code base.

Network devices Our design goals for network devices are (1) to clone transparently and seamlessly and (2) to keep the performance of each clone device on par with the original device. Our first goal implies that the clone devices have the same MAC and IP addresses as the original device.

We explore two off-the-shelf solutions to achieve our goals. First solution is based on Linux bond [18], an interface aggregation mechanism that is provided by the Linux kernel and that features load balancing for virtual interfaces with identical MAC and IP addresses. We configure load balancing with the XOR policy as running mode to enable choosing clone interfaces by hashing IP and port values. Therefore, this approach does not keep any state regarding the aggregated interfaces and the only overhead it brings is originated in computing the hashing when selecting a clone interface.

Second solution uses Open vSwitch (OVS) groups [5] and addresses the scenarios when more information is needed to select the clone interfaces for forwarding. Although the vanilla version of OVS selects clone interfaces by hashing, it can be easily extended for more complex selection criterias that can leverage the state information it keeps regarding the incoming flows.

The network device cloning process starts with the writing of the clone Xenstore entries. Consequently, the network backend is notified and proceeds with creating the clone device state, bypasses the negotiation with the frontend by marking the backend state as connected and triggers the `udev` events delivered to `xencloned` which will continue with the userspace operations (e.g. adding the clone network interface either to a Linux bond or to an OVS group).

9pfs filesystem The 9pfs filesystem [2] is an NFS-like remote filesystem which allows multiple guests to use the same root filesystem and for which Unikraft [36] showed lower latency in comparison to Linux. It also brings a new challenge: unlike the network backend driver that runs in the kernel, the 9pfs backend runs as a Qemu process in Dom0. The 9pfs backend keeps a table with file descriptors, called file IDs (`fids`), associated to all open files, similarly to the file descriptors table maintained by the Linux kernel for each process. For cloning the `fids` table we had two options. The first option launches a new backend process for each clone and propagates the `fids` table of the parent to the child backend process. This approach stresses the limits of the host system when reaching a high density of clones, turning Dom0 in a major bottleneck. The second option, which is adopted by Nephele, uses the same backend process for the parent and all its clones. We also extended QEMU Machine Protocol (QMP), a protocol used for management messages, to support transmitting cloning requests from the `xencloned` daemon.

4.3 Inter-process communication vs Inter-domain communication

After successfully calling `fork()` on multiprocess OSes, the parent process must be able to communicate with the child process by means of inter-process communication (IPC). Likewise, we introduce inter-domain communication (IDC) as the suite of primitives and mechanisms used for communication between the parent and child domains. We show that IPC mechanisms can be replicated as IDC based on the primitives provided by the virtualization platform. On Xen, memory can be shared between different domains using the grant reference primitive: the domain owning a memory region will grant another domain access to it. For notifications we use event channels: virtual channels maintained by the hypervisor for delivering events from one domain to another, similarly to interrupts delivery.

The applications we target for our evaluation use anonymous pipes and socket pairs for IPC. We added an API in Unikraft that provides creation of memory areas shared between parent and clones based on grant references; we use the `DOMID_CHILD` wildcard to specify the domain being granted, instead of setting an explicit domain ID. A clone is granted permission to all the IDC pages that originally belonged to its parent domain; the ownership of these pages is transferred to `dom_cow`, just like for any shared page.

The API is also used for managing IDC notifications; similarly to marking file descriptors as ready whenever an ongoing I/O operation is completed on Linux, we leverage event channels to notify the completion of such operations to the domain waiting at the other end. Again, the parent domain sets the `DOMID_CHILD` wildcard on IDC event channels to specify they are connected to child domains. On creation, a clone is implicitly bound to all the IDC event channels of its parent.

Our solution does not impose any restrictions on how the IDC mechanisms are designed or implemented on guest unikernels. Unlike multiprocess VMs where all the processes have to share the grant references and event channels that are assigned by the hypervisor, on unikernels the same number of resources is entirely used by a single "process", unlocking a wider space of possible solutions for implementing IDC mechanisms.

5 Microbenchmarks

In this section we evaluate the performance of Nephele by running microbenchmarks. Our implementation is based on Xen 4.16 and has 17.5 KLOC, with 5670 lines added to the hypervisor, 9200 added to `xencloned` and 2700 to the rest of the Xen toolstack. We run experiments on a server with an Intel Xeon E5-1620 v2 CPU at 3.70 GHz with 4 cores and 16 GB of DDR3 RAM. Dom0 runs Alpine Linux 3.13.0 and Linux kernel 5.9.0, with the root filesystem entirely stored on a ramdisk to reduce the overheads related to the storage medium.

5.1 Instantiation

To evaluate instantiation times we use a simple UDP server running on top of Mini-OS [3] and we iteratively start new VMs, measuring the completion time of each instantiation (same methodology as LightVM [42]);Each VM is configured to use only 4 MB of memory and a single vif interface. Once the UDP server is ready it sends a UDP packet to notify the host.

Figure 4 shows the results where the VM instantiation baseline time ranges from 160 ms up to 300 ms in the case of the 1000th booted VM. Figure 4 also shows the time needed to restore a VM from a previously saved image; the experiment runs for 1000 iterations - on each iteration, a new instance is created, saved to a new image and next restored from the image. The values shown in figure 4 represent the durations between launching the restore command and the time when the UDP server running on Mini-OS is ready. The values are slightly higher than in the case of booting, ranging from 180 ms to 330 ms, the difference occurring because the entire allocated VM memory is copied back from the image into the machine memory, regardless of the amount of memory that is actually used by the VM.

To evaluate cloning, we boot a single instance of the same UDP server which then clones itself 1000 times by calling `fork()` immediately after sending the UDP boot notification to the host. We use a stateless switching solution to connect the clones to the host; each clone has an identical MAC and IP address pair. We configured a Linux bond interface with `balance-xor` load balancing policy and `layer3+4` transmit hash policy, i.e. the slave virtual interfaces are picked by the bond driver by using a hash of IP addresses and port values. To avoid collisions we had to pick a specific IP address and assign a unique port number to each UDP server running on clones so that there were no two different <address, port> tuples mapping to the same slave interface.

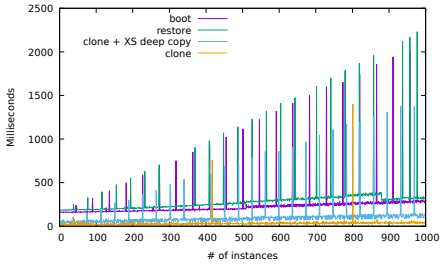

Figure 4. Instantiation times for Mini-OS UDP server.

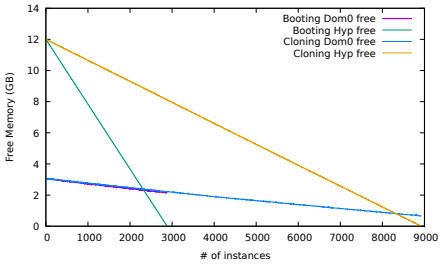

Figure 5. Memory consumption for booting vs. cloning

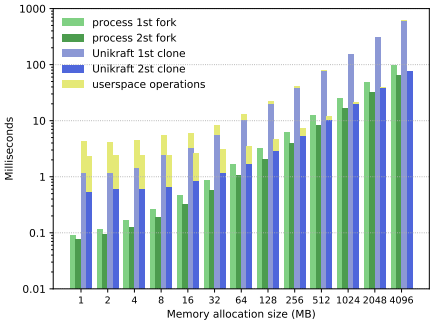

Figure 6. Fork and cloning duration depending on used memory size.

Figure 4 shows the durations of the `fork()` call, from the start time that is saved by the parent VM before calling the function to the completion time that is saved by each child VM. The values show how long it takes for each instance to reach the same state as the parent using the cloning support as alternative to regular booting.

On average, cloning takes from 20 ms to 30 ms, with a linear growth much lower than in the case of booting because of fewer interactions with the Xenstore daemon, an effect of using the `xs_clone` request. In order to illustrate this, we also ran the experiment without using `xs_clone`: `xencloned` uses a deep-copy approach for cloning the Xenstore key-value entries by sending a writing request for each entry, similarly to how the Xenstore entries are created on regular instantiation. In this case, the values range from 40 ms up to 130 ms.

The spikes in the figure are generated by the rotation of access logs files in which Xenstore logs every incoming request, just as reported by LightVM. Disabling the access logging has no effect whatsoever on the value ranges for each experiment. In our experiment, when using `xs_clone` requests, access logging also drops signficantly and the number of spikes drops to only 2.

The cloning duration is dominated by the second stage of the cloning process, while on average the first stage which runs entirely inside the hypervisor takes only 1 ms for the application used in this experiment. Increasing the configured memory size will also increase the first stage duration as we show in Section 5.2.

When launching a new VM, vanilla `xl` checks whether the name provided in the configuration file is unique in the system by iterating through the names of all the running VMs, this behaviour generating a superlinear growth of the instantiation time with the number of instances. On cloning, the name of a clone is generated and set by `xencloned`, therefore the uniqueness of the name is guaranteed by the toolstack and no check is required. In order to make a fair comparison, we disabled the name validation performed by `xl` for our baseline boot numbers, given that VM names were automatically generated by us and thus unique. Otherwise we would have had the same superlinear growth shown by LightVM [42].

5.2 Memory cloning

One advantage of cloning is the efficient usage of memory: memory pages that are only read by clones are not duplicated anymore. However, there are some concerns regarding the costs of these benefits. Sharing memory that has hitherto been private brings an overhead on clone instantiation times, while creating copies of memory pages on write operations generate an overhead on the operations themselves. In this subsection we analyze both the gains and losses regarding the memory usage when cloning unikernel-based applications.

Memory density. The COW mechanisms allow running more VM instances on the same physical machine, similarly to creating processes on traditional OSes. Memory regions such as text pages, read-only pages or writable pages that are written only at the application initialization will be shared between all related VMs, i.e. parent and clones. Therefore, a higher percentage of memory, out of the total, that is shared yields a higher density of clones that can be created.

When running thousands of instances, special care should be taken with how the total machine memory is split between Dom0 and the rest of the VMs, a special consideration being required also about the memory that is needed by the hypervisor for its own internal bookkeeping. The Dom0 memory should accommodate all the memory needs of the driver backends and of the Xenstore daemon for its key-value store (e.g. on our setup `oxenstored` needed up to 350 MB of resident memory). For our experiment, we split the total memory of 16 GB into 4 GB for Dom0 and 12 GB for the hypervisor.

For the results shown in figure 5, we use the same image we used for evaluation of instantiation times. With regular booting we get 2800 instances, while with cloning we get 8900 instances. The 3x increase of instances number shows that each clone consumes 1.6 MB of memory (out of which 1 MB is used for the RX network ring alone), in contrast to the 4 MB consumed by each booted instance. In the end, we saved a total of 21 GB, much more than the total machine memory. As expected, the available memory on Dom0 decreases with the same rate for both booting and cloning given that the Xenstore entries and the backends data are duplicated for each clone.

Effects on instantiation time. Previous work [65] proved that process forking duration is dominated by the copying of the page tables when the used memory size starts reaching hundreds of megabytes. In this experiment, we focus on how allocated memory size determines the fork and cloning durations and compare the results for the same application, built for Linux and run as a process and built for Unikraft [36] and run as a VM, respectively. The application allocates a chunk of memory that must be resident. On Unikraft we use the tinyalloc [50] memory allocator which yields the best results from all the supported allocators [36]. Once the required memory is allocated, the application starts a simple TCP server that receives requests for forking/cloning. For the cloning numbers, we skip cloning the I/O devices and keep only the mandatory operations of the second stage, e.g. introducing the child VM to the Xen toolstack.

Given that on the first call of `fork()` of a process, the entire address space is marked as COW, the second call will take less time than the first; the same behaviour applies to cloning as well. Unlike Linux, on Xen, each page has an owner domain; for this reason sharing a page also involves changing the owner of that page from the parent domain to a special Xen domain called `dom_cow`. Figure 6 shows the results for the first and second fork/clone duration of each application variant. For each allocation size we show the average numbers we get after running 10 repetitions each. The cloning duration is constant for small allocation sizes because of the mandatory limit of minimum 4 MB of memory that Xen imposes on any domain. The gap between second fork and second clone numbers decreases from 5757% (0.07 msec for process and 4.1 msec for unikernel) to 21% (65.2 msec for process and 79.2 msec for unikernel).

The numbers for cloning also include the time spent by Dom0 userspace operations, such as introducing the new domain to the Xenstored daemon and setting the domain name and console. The userspace operations take on average 3 ms for the first cloning and 1.9 ms for the second cloning, regardless the memory allocation size. The difference occurs because on first cloning the parent Xenstore information is read and cached by `xencloned` to speed up future invocations by skipping several Xenstore read operations.

6 Use Cases

We now explore real-world use cases that outline the necessity of unikernel cloning support when porting applications to unikernels. Our first two use cases show that cloning support helps in closing the gap towards full POSIX compatibility by running NGINX and Redis with cloning. Next we show how cloning is the only practical way of fuzzing unikernels. Finally, we show the gains that cloning brings to Function as a Service frameworks, enabling superior security and similar or better performance to using containers, which is the state of the art.

6.1 Towards full POSIX Compatiblity

Unikernels proved their potential as a middle ground solution that brings together stronger isolation than containers and better performance than multiprocess VMs. However, porting POSIX applications on top of unikernels raises a new challenge: how should we deal with the cases when applications make use of the POSIX `fork()` primitive?

Many such applications depend on `fork()` to enhance their performance or for complementary functionality, but in the case of unikernels these requirements have not been met yet. We target NGINX and Redis, two of the most popular cloud applications, and highlight their behaviour when running with VM cloning. In both use cases, we build the same application source code to create a Linux binary or a Unikraft VM and compare the results.

NGINX HTTP throughput. Our first use focuses on NGINX, a popular web browser for cloud applications [58]. NGINX uses `fork()` to launch worker processes for scaling up request throughput. One recommended configuration [23] runs each worker on a different CPU core to increase the throughput linearly with the number of workers. On Linux, every worker process uses the same IP address and port combination when listening for new incoming TCP connections, a feature known as *socket sharding* [30], which is configured by setting the `SO_REUSEPORT` option on the listening socket. The load balancing of the incoming connections across the listeners is then carried out by the kernel [32]. When using unikernel clones as workers, the virtual interfaces of the parent and the clones are aggregated using a Linux bond interface, thus the load balancing of new incoming connections is carried out outside the VMs, by the Linux bond driver in Dom0, removing the need to support socket sharding inside the unikernel.

Figure 7 shows the HTTP thoughput for NGINX workers running as Linux processes and as Unikraft clones, respectively, confirming the expected linear growth with the number of workers. To measure request throughput we use `wrk` [24], an HTTP benchmarking tool. In our experiment, `wrk` keeps 400 open HTTP connections with each worker in order to saturate the virtual link for a total duration of 5 seconds, and we repeat the test 30 times. The results show that using Unikraft clones yields higher and less variable throughput because each CPU core is used exclusively by its pinned worker clone and because it avoids switches between user and kernel space.

Redis in-memory database serialization. Redis relies on `fork()` to create processes for saving the in-memory database to storage. `fork()` in this case is used to snapshot the process memory, a behavior that cannot be replicated by replacing processes with threads. Therefore having `fork()` support is mandatory for running Redis in production.

A common usage for Redis is to initiate the database saving periodically and/or when some preconfigured number of database updates is reached, or when the client tool sends a save command. In this experiment we measure how the

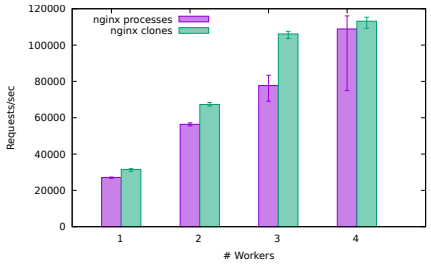

Figure 7. NGINX HTTP requests throughput

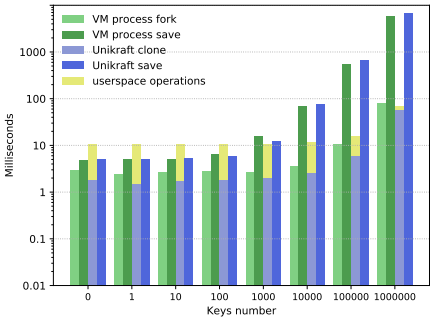

Figure 8. Redis database saving times

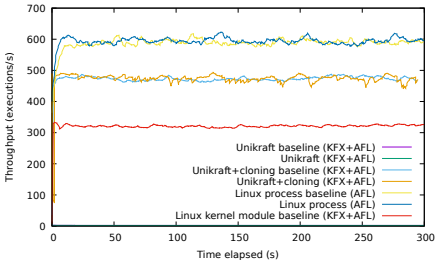

Figure 9. Fuzzing

number of database updates affects the `fork()` call durations and we compare the database saving times to check for the overheads that may arise after cloning the required I/O state. As Unikraft supports only 9pfs at the moment [36], for the baseline we run Redis as process inside an Alpine Linux VM and save the database to a 9pfs share mounted in the VM. We use a ramdisk for the entire Dom0 root filesystem.

When we issue two consecutive calls of `fork()` right after the initialization of the parent process, the first call will always take longer because the entire process memory is marked for COW. In figure 8 we thus report the values of the second `fork()` call (and how that is affected by the number of database updates) together with the time needed to save the snapshot. We compare the performance when running a Redis process in a Linux VM versus a Unikraft instance. Once Redis starts we send a save command which will in turn trigger the first `fork()` call. Next we use mass insertion to populate the in-memory database, after which we send a second database saving command.

Figure 8 breaks down the total time into time spent for cloning the I/O state of Unikraft clones, which includes toolstack introduction and 9pfs cloning. The figure shows that the constant cost of I/O cloning is amortized for larger database updates, leading to save times that are comparable for VM cloning and process fork. In our experiment, the I/O cloning is optimized to include to only clone devices actually needed by the clones. In particular, we skip cloning network devices because the Redis clones do not need any network support.

6.2 VM fuzzing

Fuzzing is a powerful tool for bug finding in both applications and OS kernels [26, 51, 56]; it is particularly useful in experimental OSes, such as unikernels, which aim to provide support for as many applications as possible.

Kernel Fuzzer for Xen Project (KFX) [39] is a tool using the AFL fuzzer [62] which leverages VM forks to fuzz HVM guests running on Xen. VM forks were initially introduced in Xen for Snowflock [38] and later revisited and revived for KFX.

We extended KFX to support fuzzing paravirtualized guests and to use the API provided by the cloning support as a replacement to legacy VM forks. KFX does coverage-guided fuzzing and therefore it needs to instrument the VM code in order to step through the binary code of targeted guest. To achieve this, KFX clones the targetted VM and does the actual instrumentation on the clone domain. Instrumentation involves breakpoint insertion for the instructions that change the control flow (e.g. branch instructions). We extended the `CLONEOP` hypercall with the `clone_cow` operation to trigger COW explicitly for the pages where the breakpoints need to be inserted by KFX in the code regions of clones.

A fuzzing session is split into iterations. On each iteration the clone domain receives an input that is generated by AFL. At the end of the iteration, KFX restores the instrumented memory of the clone so that each iteration starts with the same memory state. For paravirtualized guests, the memory restore is performed by the `clone_reset` operation of the hypercall.

For our experiment we chose to fuzz the system call support in Unikraft and we measure the fuzzing throughput as reported by AFL, in executions/second. We devised an application that runs an adapter which interprets the input generated by AFL as system calls and which is built and run both as a native Linux process and as a Unikraft application. The fuzzing of the native Linux process is performed only with AFL (i.e. without KFX) and it shows a superior baseline given that there is no code coverage involved.

For Unikraft we study whether cloning improves the fuzzing throughput when compared with fuzzing without cloning support. Without cloning support, we start a new VM instance for each AFL input because it is the only way of reaching the same state at the beginning of each iteration.

Figure 9 shows the fuzzing throughput for each of our experiment runs. Fuzzing Unikraft without cloning averages at 2 executions/sec because the VM is recreated for each input. The syscall subsystem is not fully supported for the Unikraft tree version we used in the experiment and this can generate considerable variations in fuzzing throughput, therefore we also picked a simple syscall, namely `getppid`, to show the baseline throughput when a fully supported syscall is fuzzed, both for Unikraft (with and without cloning) and for the Linux process as well. Enabling cloning support raises the throughput average at 470 executions/sec, lower only by 18.6% than the 590 executions/sec average for fuzzing the native Linux process.

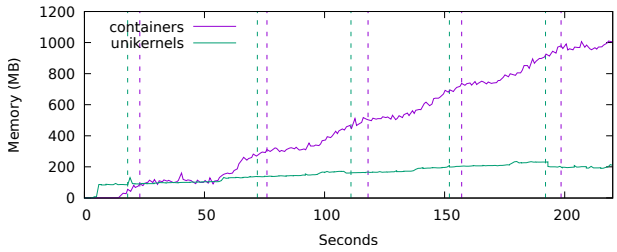

Figure 10. Comparison of memory consumption in OpenFaaS when using containers vs. unikernels. The dashed vertical lines indicate the times when instances are reported as ready by Kubernetes.

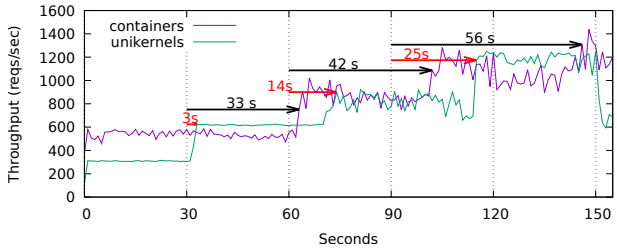

Figure 11. Reaction of containers vs. unikernels in OpenFaaS at increasing function call demand.

We also compare the results we got for Unikraft and cloning support with the results for fuzzing a Linux VM kernel module. We followed the sample KFX demo and used a simple self-contained code snippet that does not make any library calls. The throughput we got averages at 320 executions/sec, which is 31.9% lower than Unikraft and cloning support. Although the fuzzed module code changes less data, the memory reset for the Linux VM takes on average 250 usec, double than for Unikraft memory reset, having a higher state to restore and more dirty pages (a consistent average of 8 pages for Linux in comparison to an average of 3 pages for Unikraft).

6.3 Function as a Service

Function as a Service (FaaS) platforms rely on containers to launch high-level language interpreters which in turn run the functions provided by users. Prior work [42] showed that using containers to run a single application is expensive and unikernels are more suitable for such use cases. We push this conclusion even further and show that resources can be even more efficiently used when adding cloning support to unikernels as a replacement of containers in FaaS frameworks, while benefitting from the added security provided by unikernels.

To cope with variable load without increasing request latency, FaaS frameworks can be configured to keep a number of instances ready for warm starts. However, when these instances do not suffice, new instances are created until the latencies get back to the accepted values. We expect that in both cases, using unikernels with cloning should result in lower resources consumption and lower latency.

We chose OpenFaaS 0.23.0 [8], an open-source framework with auto-scaling support, designed to run functions in multiple programming languages. To scale functions, OpenFaaS runs queries periodically to check the load per instance, with new instances being launched whenever the load value gets above a preconfigured threshold. For the autoscaling configuration, we kept the default query interval of 30 seconds, we set the requests-per-second (RPS) scaling mode (i.e. the load is the RPS value for each instance) with the default threshold of 10 RPS [9] and we configured to launch a single new instance whenever the threshold is exceeded.

For our experiments we used two setups: a vanilla setup with OpenFaaS instantiating containers for baseline numbers and our modified setup running unikernels instead of containers. In both cases, we deployed a simple Python function returning a "Hello World" string. To deploy functions on top of unikernels, we used KubeKraft [31], a closed-source solution for packaging, instantiating and inspecting Unikraft-based VM images as containers for orchestration frameworks like Kubernetes. The resulting Docker image wraps a 6 MB binary image linking together Unikraft with the Python 3.7.4 interpreter and its library dependencies (e.g. newlib [4] libc implementation and lwip network stack [21]). The Python runtime is shared between all unikernel instances via a 9pfs root file system. We used the Apache Benchmark 2.3 (`ab`) tool [6] to generate requests and to measure the requests throughput, for each session running 8 workers to generate a total of 500K requests. For each of our experiments we used two identical machines, each running Intel(R) Xeon(R) CPU E5-2650L v2 at 1.70 GHz CPUs with 10 cores, one machine for running the setup and another for generating the requests. On the unikernel setup we allocated 4 of the cores for the VMs and the rest for Dom0.

Our first experiment measures the memory used in each of the two setups. For the vanilla setup we used `free` to measure the memory used in total by both containers and services, while in the case of the unikernel-based setup we used `free` to measure the memory used by the services in Dom0 (Xen toolchain, Kubernetes and OpenFaaS services) and `xl info` to measure the memory consumed by the VMs. Figure 10 shows the memory consumption numbers and the times when a new instance was ready, as reported by Kubernetes, with unikernel clones being ready on average 5 seconds sooner than the containers. Although the occupied memory sizes for the first instances are similar in both setups, with 85 MB for the first unikernel (out of which 64 MB are consumed by the VM, the rest being spent by the services in Dom0) and 90 MB for the first container, the evolution of occupied memory sizes of the consequent instances favors the unikernels which require tens of megabytes (35 MB on average) as opposed to hundreds of megabytes (220 MB on average) for containers.

The second experiment shows in figure 11 how fast each of the environments reacts to increasing demand. Even though the native Linux stack yields better requests throughput in

comparison to Unikraft's `lwip` stack (600 requests/sec as opposite to 300 requests/sec), the newly instantiated unikernel clones get to service the incoming requests much faster, being able to track request load closely.

7 Related work

Nephele is the first complete solution for unikernel cloning that offers POSIX compatibility, I/O cloning and performance. Nephele builds upon extensive prior work in the area of cloning virtual machines and processes, as we discuss below.

Unikernel cloning. There are several recent solutions for unikernel cloning, each of them addressing only a subset of our goals. Kylinx [64] is a Xen based solution that adds support for calling `fork()`, but handles only CPU and memory cloning and no I/O support. The `fork()` call is based on a coordination between the parent domain, the child domain and the toolstack which adds significant overhead from all the required domain crossings. The IPC subsystem is initialized asynchronously after the `fork()` call returns in both parent and child domains, while with our solution IPC is already established when the call ends. Fractal [34] is a self-scaling solution that uses multi-host replication. Although it only presents the evaluation of a self-scaling website and focuses only on replicating the networking state, the approach based on Open vSwitch groups was a starting point to provide cloning support for networking in our solution.

Iso-UniK [40] breaks the unikernel definition by adding multiprocess support inside the OSv unikernel [33] and bringing back the separation between user and kernel modes. While it does advocate for generality, it omits addressing I/O cloning.

Unikernels and processes. Making unikernels similar to processes and adding support for related abstractions was also the subject of prior work [37, 40, 59]. [59] argues that unikernels can be run as processes, without any underlying virtual hardware abstraction, while still keeping their isolation properties. Lupine [37] states that some applications have a non-unikernel character, meaning that they are multiprocess, and proposes a solution that tailors Linux towards achieving a unikernel behaviour through specialization and system call elimination by running applications in kernel mode.

VM Replication. VM replication is mainly used for live migration of VMs, typically across hosts [16, 63], with techniques such as pre-copy [17, 44], post-copy [27, 28] or a combination of both [29], replicating either a single instance or multiple instances [11] at once, and addressing either generic applications or specialized ones ranging from virtual middleboxes to HPC systems, for achieving elasticity [49] or for improving failure recovery [43, 52]. Multi-host replication is out of scope for our solution since moving unikernel clones on different machines would break the page sharing potential and would add complexity to the application logic. We believe that previous solutions for transparently migrating processes [19, 47, 53] can be adapted to clone unikernels to other hosts.

Fork function support. For all the wide adoption through its long history [45], the `fork()` function was also subject to critics [14]. While it was argued [14] that in combination with `exec()` it makes a cumbersome approach for creating processes from scratch, previous research showed that `fork()` is suited for creating clones, including virtualization environments. Snowflock [38], a Xen based solution that stood the test of time [1], introduced the VM fork abstraction for replicating generic HVM guests across multiple hosts. It was also shown that the fork abstraction can be used to speed up instantiation in serverless computing systems [10, 20]. Such optimizations can be enhanced even more following recent findings. ON-DEMAND-FORK [65] proved that the `fork()` call duration is dominated by the page table cloning when the process memory size starts taking hundreds of megabytes and proposed the solution of applying COW for pages of the page tables as well.

Memory deduplication. Research addressing duplicated memory in virtualization environments covered two main directions. First one leverages COW and was adapted from traditional OSes starting with Potemkin [55] for single host page sharing and was later extended by Snowflock [38] for multi-host setups. Second option uses content-based page sharing and was introduced by VMware ESX [57] for same host memory deduplication and later extended for multi-host setups [60] and subpage sharing [25, 60]. The Linux kernel implementation, called KSM [12] and initially designed for KVM, is currently used for bare-metal Linux as well. Content-based page sharing was shown to introduce vulnerabilities for either inter-process [15] or inter-VM [54, 61] memory sharing when an attacker process or VM, respectively, is allowed on the same host. Nephele prevents this vulnerability by controlling the set of VMs that can share memory to family-related unikernel-based guests belonging to the same tenant.

8 Conclusions

We have presented Nephele, a complete cloning solution for unikernel VMs that offers memory and I/O device cloning. Our implementation in Xen adds a single hypercall to the hypervisor and does most of the heavy lifting in Dom0 via `xencloned`, supporting cloning for paravirtualized guests.

Evaluation shows that cloning a VM is 8 times faster than booting it, and that it helps seamlessly support fork-dependent applications such as NGINX and Redis with performance comparable to process forking. We have further showed that cloning enables more efficient fuzzing and Function as a Service frameworks, hinting that unikernel cloning should become a viable option in future virtualization systems.

In future work we intend to port Nephele to KVM and to understand how it is best to configure host environments for the high density workloads enabled by unikernel cloning. Cloning can also be used to side-step other limitations of existing unikernels, for instance lack of SMP support can be mitigated by running clones on different CPUs.

References

[1] Eurosys test-of-time award. https://www.eurosys.org/awards/test-of-time-award. Accessed: 2021-10-11.

[2] Introduction to the plan 9 file protocol. http://man.cat-v.org/plan_9/5/intro. Accessed: 2022-07-10.

[3] Mini-os. https://wiki.xenproject.org/wiki/Mini-OS. Accessed: 2022-07-10.

[4] Newlib. https://sourceware.org/newlib. Accessed: 2022-07-10.

[5] Open vswitch manual. http://www.openvswitch.org/support/dist-docs/ovs-ofctl.8.txt. Accessed: 2022-07-10.

[6] Apache benchmark. https://httpd.apache.org/docs/2.4/programs/ab.html, 1996. Accessed: 2022-07-10.

[7] Transparent page sharing (tps) in hardware mmu systems. https://kb.vmware.com/s/article/1021095, 2014. Accessed: 2022-04-29.

[8] Openfaas. https://www.openfaas.com, 2016. Accessed: 2022-07-10.

[9] Openfaas source code. https://github.com/openfaas/faas, 2016. Accessed: 2022-07-10.

[10] Akkus, I. E., Chen, R., Rimac, I., Stein, M., Satzke, K., Beck, A., Aditya, P., and Hilt, V. {SAND}: Towards {High-Performance} serverless computing. In *2018 Usenix Annual Technical Conference (USENIX ATC 18)* (2018), pp. 923–935.

[11] Al-Kiswany, S., Subhraveti, D., Sarkar, P., and Ripeanu, M. Vmflock: Virtual machine co-migration for the cloud. In *Proceedings of the 20th international symposium on High performance distributed computing* (2011), pp. 159–170.

[12] Arcangeli, A., Eidus, I., and Wright, C. Increasing memory density by using ksm. In *Proceedings of the linux symposium* (2009), Citeseer, pp. 19–28.

[13] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. *ACM SIGOPS operating systems review 37*, 5 (2003), 164–177.

[14] Baumann, A., Appavoo, J., Krieger, O., and Roscoe, T. A fork() in the road. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (2019), pp. 14–22.

[15] Bosman, E., Razavi, K., Bos, H., and Giuffrida, C. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *2016 IEEE symposium on security and privacy (SP)* (2016), IEEE, pp. 987–1004.

[16] Choudhary, A., Govil, M. C., Singh, G., Awasthi, L. K., Pilli, E. S., and Kapil, D. A critical survey of live virtual machine migration techniques. *Journal of Cloud Computing 6*, 1 (2017), 1–41.

[17] Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2* (2005), pp. 273–286.

[18] Davis, T. Linux ethernet bonding driver howto. https://www.kernel.org/doc/Documentation/networking/bonding.txt, 2011. Accessed: 2021-10-11.

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

[20] Du, D., Yu, T., Xia, Y., Zang, B., Yan, G., Qin, C., Wu, Q., and Chen, H. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (2020), pp. 467–481.

[21] Dunkels, A. Design and implementation of the lwip tcp/ip stack. *Swedish Institute of Computer Science 2*, 77 (2001).

[22] Eisenbud, D. E., Yi, C., Contavalli, C., Smith, C., Kononov, R., Mann-Hielscher, E., Cilingiroglu, A., Cheyney, B., Shang, W., and Hosein, J. D. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, Mar. 2016), USENIX Association, pp. 523–535.

[23] Garrett, O. Inside nginx: How we designed for performance & scale. https://www.nginx.com/blog/inside-nginx-how-we-designed-for-performance-scale, 2015. Accessed: 2021-10-08.

[24] Glozer, W. wrk - a http benchmarking tool. https://github.com/wg/wrk, 2022. Accessed: 2022-07-10.

[25] Gupta, D., Lee, S., Vrable, M., Savage, S., Snoeren, A. C., Varghese, G., Voelker, G. M., and Vahdat, A. Difference engine: Harnessing memory redundancy in virtual machines. *Communications of the ACM 53*, 10 (2010), 85–93.

[26] Hertz, J. Triforceafl. https://github.com/nccgroup/TriforceAFL, 2015. Accessed: 2022-04-29.

[27] Hines, M. R., and Gopalan, K. 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* (2009), pp. 51–60.

[28] Hirofuchi, T., Nakada, H., Itoh, S., and Sekiguchi, S. Enabling instantaneous relocation of virtual machines with a lightweight vmm extension. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing* (2010), IEEE, pp. 73–83.

[29] Hu, L., Zhao, J., Xu, G., Ding, Y., and Chu, J. Hmdc: Live virtual machine migration based on hybrid memory copy and delta compression. *Appl. Math 7*, 2L (2013), 639–646.

[30] Hutchings, A. Socket sharding in nginx release 1.9.1. https://www.nginx.com/blog/socket-sharding-nginx-release-1-9-1, 2015. Accessed: 2021-03-31.

[31] Jung, A. Deploying unikernels in production with kubernetes. https://www.youtube.com/watch?v=cV-xawN9_cg, 2021. Accessed: 2022-09-10.

[32] Kerrisk, M. The so_reuseport socket option. https://lwn.net/Articles/542629, 2013. Accessed: 2022-07-10.

[33] Kivity, A., Laor, D., Costa, G., Enberg, P., Har'El, N., Marti, D., and Zolotarov, V. OSv—Optimizing the operating system for virtual machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (Philadelphia, PA, June 2014), USENIX Association, pp. 61–72.

[34] Koleini, M., Oviedo, C., McAuley, D., Rotsos, C., Madhavapeddy, A., Gazagnaire, T., Skejgstad, M., and Mortier, R. Fractal: Automated application scaling. arXiv:1902.09636, 2019.

[35] Kroah-Hartman, G. udev man page. https://www.freedesktop.org/software/systemd/man/udev.html. Accessed: 2021-10-11.

[36] Kuenzer, S., Bădoiu, V.-A., Lefeuvre, H., Santhanam, S., Jung, A., Gain, G., Soldani, C., Lupu, C., Teodorescu, Ş., Răducanu, C., et al. Unikraft: fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems* (2021), pp. 376–394.

[37] Kuo, H.-C., Williams, D., Koller, R., and Mohan, S. A linux in unikernel clothing. In *Proceedings of the Fifteenth European Conference on Computer Systems* (2020), pp. 1–15.

[38] Lagar-Cavilla, H. A., Whitney, J. A., Scannell, A. M., Patchin, P., Rumble, S. M., De Lara, E., Brudno, M., and Satyanarayanan, M. Snowflock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European conference on Computer systems* (2009), pp. 1–12.

[39] Lengyel, T. K. Kernel fuzzer for xen project. https://github.com/intel/kernel-fuzzer-for-xen-project, 2020. Accessed: 2022-04-29.

[40] Li, G., Du, D., and Xia, Y. Iso-unik: lightweight multi-process unikernel through memory protection keys. *Cybersecurity 3* (2020), 1–14.

[41] Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., and Crowcroft, J. Unikernels: Library operating systems for the cloud. *ACM SIGARCH Computer Architecture News 41*, 1 (2013), 461–472.

[42] Manco, F., Lupu, C., Schmidt, F., Mendes, J., Kuenzer, S., Sati, S., Yasukata, K., Raiciu, C., and Huici, F. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), pp. 218–233.

[43] Nagarajan, A. B., Mueller, F., Engelmann, C., and Scott, S. L. Proactive fault tolerance for hpc with xen virtualization. In *Proceedings of the 21st annual international conference on Supercomputing* (2007), pp. 23–32.

[44] Nelson, M., Lim, B.-H., Hutchins, G., et al. Fast transparent migration for virtual machines. In *USENIX Annual technical conference, general track* (2005), pp. 391–394.

[45] Nyman, L., and Laakso, M. Notes on the history of fork and join. *IEEE Annals of the History of Computing 38*, 3 (2016), 84–87.

[46] Olteanu, V., Agache, A., Voinescu, A., and Raiciu, C. Stateless datacenter load-balancing with beamer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (Renton, WA, Apr. 2018), USENIX Association, pp. 125–139.

[47] Osman, S., Subhraveti, D., Su, G., and Nieh, J. The design and implementation of zap: A system for migrating computing environments. *ACM SIGOPS Operating Systems Review 36*, SI (2002), 361–376.

[48] Patel, P., Bansal, D., Yuan, L., Murthy, A., Greenberg, A., Maltz, D. A., Kern, R., Kumar, H., Zikos, M., Wu, H., Kim, C., and Karri, N. Ananta: Cloud scale load balancing. In *SIGCOMM* (2013).

[49] Rajagopalan, S., Williams, D., Jamjoom, H., and Warfield, A. Split/merge: System support for elastic execution in virtual middleboxes. In *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)* (2013), pp. 227–240.

[50] Schmidt, K. malloc, free replacement for unmanaged, linear memory situations. https://github.com/thi-ng/tinyalloc, 2011. Accessed: 2022-07-10.

[51] Schumilo, S., Aschermann, C., Gawlik, R., Schinzel, S., and Holz, T. {kAFL}:{Hardware-Assisted} feedback fuzzing for {OS} kernels. In *26th USENIX Security Symposium (USENIX Security 17)* (2017), pp. 167–182.

[52] Sherry, J., Gao, P. X., Basu, S., Panda, A., Krishnamurthy, A., Maciocco, C., Manesh, M., Martins, J., Ratnasamy, S., Rizzo, L., et al. Rollback-recovery for middleboxes. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), pp. 227–240.

[53] Smith, J. M., and Ioannidis, J. *Implementing remote fork () with checkpoint/restart.* Department of Computer Science, Columbia Univ., 1987.

[54] Suzaki, K., Iijima, K., Yagi, T., and Artho, C. Memory deduplication as a threat to the guest os. In *Proceedings of the Fourth European Workshop on System Security* (2011), pp. 1–6.

[55] Vrable, M., Ma, J., Chen, J., Moore, D., Vandekieft, E., Snoeren, A. C., Voelker, G. M., and Savage, S. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *Proceedings of the twentieth ACM symposium on Operating systems principles* (2005), pp. 148–162.

[56] Vyukov, D. Syzkaller. https://github.com/google/syzkaller, 2015. Accessed: 2022-04-29.

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

[58] Walker, A. The best docker repositories, apps, and images in docker hub in 2019. https://learn.g2.com/best-docker-containers-repository, 2019. Accessed: 2022-07-10.

[59] Williams, D., Koller, R., Lucina, M., and Prakash, N. Unikernels as processes. In *Proceedings of the ACM Symposium on Cloud Computing* (2018), pp. 199–211.

[60] Wood, T., Tarasuk-Levin, G., Shenoy, P., Desnoyers, P., Cecchet, E., and Corner, M. D. Memory buddies: exploiting page sharing for smart colocation in virtualized data centers. *ACM SIGOPS Operating Systems Review 43*, 3 (2009), 27–36.

[61] Xiao, J., Xu, Z., Huang, H., and Wang, H. Security implications of memory deduplication in a virtualized environment. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2013), IEEE, pp. 1–12.

[62] Zalewski, M. American fuzzy lop (afl). https://lcamtuf.coredump.cx/afl/, 2017. Accessed: 2022-04-29.

[63] Zhang, F., Liu, G., Fu, X., and Yahyapour, R. A survey on virtual machine migration: Challenges, techniques, and open issues. *IEEE Communications Surveys & Tutorials 20*, 2 (2018), 1206–1243.

[64] Zhang, Y., Crowcroft, J., Li, D., Zhang, C., Li, H., Wang, Y., Yu, K., Xiong, Y., and Chen, G. Kylinx: a dynamic library operating system for simplified and efficient cloud virtualization. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)* (2018), pp. 173–186.

[65] Zhao, K., Gong, S., and Fonseca, P. On-demand-fork: a microsecond fork for memory-intensive and latency-sensitive applications. In *Proceedings of the Sixteenth European Conference on Computer Systems* (2021), pp. 540–555.