Enabling Efficient Hypervisor-as-a-Service Clouds with Ephemeral Virtualization

Dan Williams[†] Yaohui Hu[‡] Umesh Deshpande[*] Piush K Sinha[‡] Nilton Bila[†]
Kartik Gopalan[‡] Hani Jamjoom[†]

[†]IBM T.J. Watson Research Center [‡]Binghamton University [*]IBM Almaden Research Center

Abstract

When considering a hypervisor, cloud providers must balance conflicting requirements for simple, secure code bases with more complex, feature-filled offerings. This paper introduces *Dichotomy*, a new two-layer cloud architecture in which the roles of the hypervisor are split. The cloud provider runs a lean *hyperplexor* that has the sole task of multiplexing hardware and running more substantial hypervisors (called *featurevisors*) that implement features. Cloud users choose featurevisors from a selection of lightly-modified hypervisors potentially offered by third-parties in an "as-a-service" model for each VM. Rather than running the featurevisor directly on the hyperplexor using nested virtualization, Dichotomy uses a new virtualization technique called *ephemeral virtualization* which efficiently (and repeatedly) transfers control of a VM between the hyperplexor and featurevisor using memory mapping techniques. Nesting overhead is only incurred when the VM is accessed by the featurevisor. We have implemented Dichotomy in KVM/QEMU and demonstrate average switching times of 80 ms, two to three orders of magnitude faster than live VM migration. We show that, for the featurevisor applications we evaluated, VMs hosted in Dichotomy deliver up to 12% better performance than those hosted on nested hypervisors, and continue to show benefit even when the featurevisor applications run as often as every 2.5 seconds.

1. Introduction

Modern commodity hypervisors increasingly implement complex *hypervisor-level services*, including rootkit detection [37], live patching [7], intrusion detection [13], high availability services [11], and a plethora of VM introspection-enabled applications [12, 15, 24, 32, 34, 36]. As a result, when considering a hypervisor, cloud providers must balance desires for small, secure, and stable code bases with limited functionality (e.g., that simply multiplex hardware) against rich service offerings with increased complexity and potentially less security and stability. In this paper, we explore a step towards a *hypervisor-as-a-service* model, in which cloud providers can focus on maintaining a simple, secure and stable code base while simultaneously encouraging the further (potentially third-party) development of hypervisor-level services.

We propose *Dichotomy*, a new cloud architecture in which cloud providers have the best of both worlds. Dichotomy cleanly splits the role of the hypervisor into two parts: the *hyperplexor* and *featurevisor*. The hyperplexor, run by the cloud provider, is a small, secure, and stable hypervisor designed solely to multiplex physical hardware and support featurevisors. A featurevisor is a lightly-modified, full-fledged commodity hypervisor that runs on top of the hyperplexor, implements rich services, and performs management of a guest VM. A cloud user can potentially choose a different featurevisor for each VM depending on the services it requires. Furthermore, featurevisors may be developed by

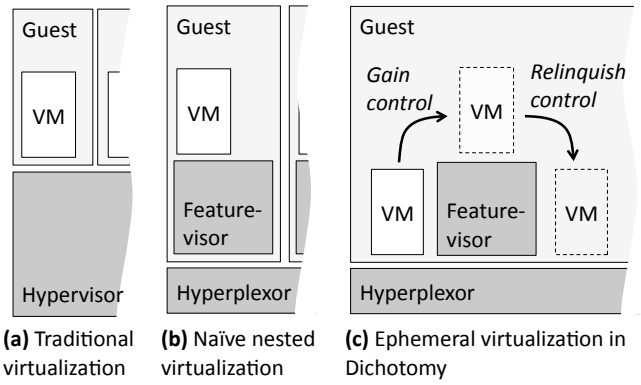


Figure 1: Splitting the role of the hypervisor with ephemeral virtualization

VEE '16 April 2–3, 2016, Atlanta, GA, USA

ACM 978-1-4503-3947-6/16/04…$15.00
DOI: http://dx.doi.org/10.1145/2892242.2892254

many different (e.g., third-party) vendors and made available in an *as-a-service* model (e.g., hypervisor-as-a-service).

Dichotomy achieves the division of responsibilities described above through a new type of virtualization called *ephemeral virtualization*. Ephemeral virtualization, depicted in Figure 1, is similar to nested virtualization [5, 14, 16, 31, 47] in that it logically involves one hypervisor (the hyperplexor) managing a second hypervisor (the featurevisor), which manages the guest VM. However, ephemeral virtualization avoids the overhead of nested virtualization most of the time by enabling the second-layer hypervisor (the featurevisor) to voluntarily (and temporarily) relinquish control and management responsibilities of the guest VM to the hyperplexor. At these times, performance becomes indistinguishable from a single layer of virtualization. The featurevisor registers *triggers* with the hyperplexor in order to indicate when it next needs control over the VM. There are many featurevisor applications that do not need continuous control over the guest and therefore can benefit from ephemeral virtualization, including rootkit detection, guest patching, and sample-based logging, profiling, monitoring, or analysis.

The relative performance of a VM workload running on a featurevisor in Dichotomy (when compared to a traditional cloud or a hypervisor-as-a-service cloud using standard nested virtualization) is determined by four factors: (1) the amount of time the featurevisor needs to be in control of the VM to perform an action, (2) the frequency the featurevisor needs to regain control of the VM to repeat an action, (3) the overhead of nesting, and (4) the overhead of switching the VM between running directly on the hyperplexor to running under the control of the featurevisor and vice versa. The first two factors define the *duty cycle* for the featurevisor and are featurevisor dependent. The third factor is featurevisor and VM workload dependent. The design of Dichotomy centers around minimizing the fourth factor, switching overhead, therefore augmenting the circumstances in which ephemeral virtualization results in performance superior to standard nested virtualization.

Dichotomy achieves low switching overhead by ensuring that both the hyperplexor and featurevisor share an up-to-date view of most of the guest VM state. Importantly, Dichotomy avoids memory copies by mapping guest memory into both the hyperplexor and featurevisor. Remaining state (e.g., VCPU and I/O) are migrated similarly to VM migration. We have implemented Dichotomy using KVM/QEMU as the basis for both the featurevisor and hyperplexor, using timer-based triggers to switch between the two. We describe how to calculate—given a VM workload, featurevisor application, and switching overhead—when ephemeral virtualization outperforms nested virtualization. Furthermore, we have experimentally evaluated the performance of guest VMs running several benchmark workloads under Dichotomy, using featurevisors that perform rootkit detection and sample-based network monitoring. We found that Dichotomy delivers up to 12% better VM performance than nested virtualization. Dichotomy provides an improvement even for featurevisor applications that seize control of VMs as frequently as every 2.5 seconds. Switching VM control between the hyperplexor and the featurevisor is fast, averaging 80 milliseconds, because minimal amount of state is copied between hypervisors. Furthermore, in some realistic circumstances, the performance of VMs running on Dichotomy nears that of VMs on single-layer virtualization, unlike VMs on nested virtualization.

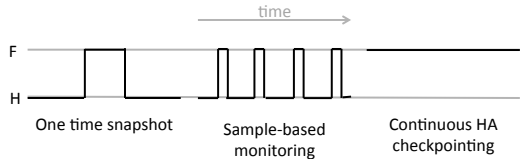


Figure 2: Duty cycles during ephemeral virtualization showing VM execution time on the featurevisor (F) vs. the hyperplexor (H)

2. Featurevisors and the Duty Cycle

There are many hypervisor-level features that already exist in commodity hypervisors and many more that may appear in the future. In this section, we define the *duty cycle* [3] in the context of ephemeral virtualization and categorize specific hypervisor-level applications implemented in featurevisors in terms of their duty cycle.

A featurevisor can be thought of as any hypervisor-level service that exists in one or more commodity hypervisor (or may exist in the future). Featurevisor services may run in the control domain (e.g., Xen's Domain 0) as user applications or in the hypervisor proper. For example, featurevisors may be created to perform tasks as commonplace as VM snapshots, to more unusual services such as root-kit detection [37] or live guest OS patching [7].

We define a featurevisor's duty cycle as the fraction of one period during which the featurevisor has control of the guest. The *period* is the duration of time from one occurrence of a featurevisor gaining control over a VM to the next occurrence, including the time that it voluntarily relinquishes control. A higher duty cycle implies that a featurevisor spends more time exerting control over the VM.

Ignoring switching overhead for a moment, the duty cycle of a featurevisor provides an indication of the performance implications of ephemeral virtualization on a VM, as well as the lower and upper bounds on performance. In the best case, with a duty cycle of 0%, the VM always runs directly on the hyperplexor; performance will match traditional single-layer virtualization. With a duty cycle of 100%, the VM always runs nested on the hypervisor and the hyperplexor; performance will match nested virtualization.

Figure 2 shows the duty cycle for three example featurevisors. A one time VM snapshot featurevisor has a low duty

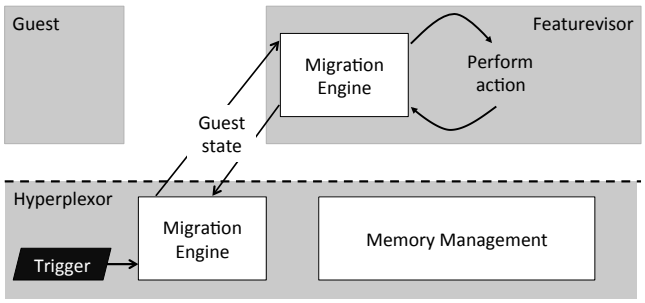


Figure 3: Design Overview of Dichotomy

cycle because, despite the length of time the featurevisor manages the VM to compute the snapshot, it will run directly on the hyperplexor for the majority of the time. Other, more interesting featurevisors with similar duty cycles characterized by infrequent actions include live guest OS patching [7], VM management [33, 42, 48] tools, and specialized virtual devices that are irregularly accessed.

The second duty cycle shown in Figure 2 depicts sample-based monitoring, in which a very inexpensive operation (e.g., reading performance counters or statistics) occurs at a regular interval. The performance of these featurevisors depends highly on the period (or the frequency of the event). Other example featurevisors that will exhibit this type of pattern include sample-based event logging [25], root-kit detection [37], near field monitoring [36], and other VM introspection services [39, 43].

The final duty cycle shown in Figure 2 depicts a continuous snapshot mechanism for high availability, such as Remus [11]. In this case, the featurevisor maintains full control of the VM at all times to track memory access, buffer output and pause the VM during backup. The duty cycle is 100%, meaning that ephemeral virtualization in this case is equivalent to nested virtualization. Other example featurevisors of this type include memory deduplication [2, 17, 44] and intrusion detection with interrupt logging [13].

We target featurevisors with a duty cycle less than 100%. The goal of Dichotomy is to improve performance over pure nested virtualization by switching control of a VM back and forth between the hyperplexor and featurevisor. In the next section, we describe how we design Dichotomy with a low switching cost, which should be amortized away for realistic applications by the time the VM spends on the hyperplexor.

3. Design

The overall architecture of Dichotomy is shown in Figure 3. A guest initially runs directly on the hyperplexor. Dichotomy's *memory management* maps guest memory into the featurevisor so that switching control over the guest from hyperplexor to featurevisor (and vice versa) does not require expensive migration of guest memory contents. It also ensures that modifications to the memory map are synchronized as control over the guest changes.

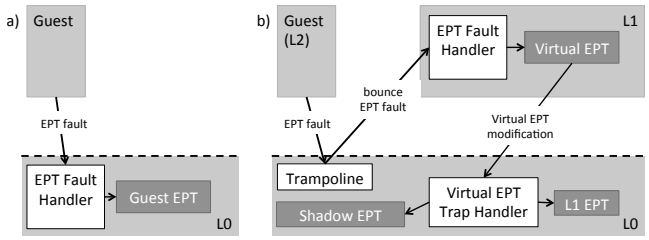


Figure 4: Memory management in a) single layer virtualization and b) standard nested virtualization

Control over the guest is switched between hyperplexor and featurevisor via Dichotomy's *migration engine*. The featurevisor registers *triggers* – a list of events upon which it should gain control of the guest – with the hyperplexor. Upon a trigger event, the hyperplexor initiates migration of remaining guest state to the featurevisor. The featurevisor performs an action potentially scheduling or modifying the guest using its own data structures and virtual hardware. It then migrates the guest back to the hyperplexor.

In the simplest case, a hypervisor represents a guest VM as a set of physical memory pages, a memory map that translates guest-physical memory addresses to the addresses of these physical memory pages, CPU context (e.g., registers and control structures) and I/O information. We do not assume that the hyperplexor and the featurevisor maintain guest state in the same format, but that it is possible to translate between formats.

In the rest of this section, we describe in detail (1) how Dichotomy shares and maintains guest memory between the hyperplexor and featurevisor, and (2) how Dichotomy efficiently migrates control of the guest between them.

3.1 Memory management

In Dichotomy, the guest may run on either the hyperplexor or the featurevisor. To avoid memory copying while the guest transitions from one to the other, Dichotomy maps guest memory into both the featurevisor and the hyperplexor. The key challenge in memory management is ensuring that the shared guest memory and memory maps remain consistent as the guest transitions from the hyperplexor to featurevisor (or vice versa), even if guest pages are added to, removed from, or remapped in the guest memory map.

Background. Modern x86 hypervisors manage the physical memory resources a guest can access using a virtualization feature called *extended page tables* (EPT) [20]. As shown in Figure 4(a), in a standard (one-level) virtualization environment, the hypervisor manages a data structure that contains the memory map information about the guest, which is backed by an EPT (*guest EPT*). The EPT indicates to the hardware what the mapping should be between guest-physical pages and machine-physical pages. Whenever the guest attempts to access a guest-physical memory address that is not present (or not allowed) based on an EPT entry,

the hardware generates an *EPT fault* and traps into the hypervisor. Using this mechanism, the hypervisor can implement demand paging (among other things) by interpreting EPT faults and updating the guest map and guest EPT accordingly.

Figure 4(b) depicts memory management in a nested environment. In a nested environment, guest pages are mapped into the second-layer hypervisor (L1). L1 manages memory for the guest as if it has direct control over the virtualization features of the hardware (i.e., single-layer virtualization). As such, it maintains a guest EPT. However, the guest EPT is virtualized; manipulations to the *virtual EPT* by L1 trigger a trap to L0. L0, in turn, will interpret the trap and maintain a *shadow EPT* that the guest actually runs with, which directly maps guest-physical addresses to the appropriate machine-physical addresses.[1] If the guest causes an EPT fault, L0 bounces the fault to L1 using a *trampoline*.

Sharing Memory. Figure 5 depicts how Dichotomy shares and manages guest memory between the hyperplexor and featurevisor. The hyperplexor has access to all of the physical memory on the machine and can assign some of it to a guest running directly on the hyperplexor in the normal way, maintaining an EPT for the guest called the *dual guest/shadow EPT* (Figure 5). Similarly, the hyperplexor assigns some physical memory to the featurevisor and maintains an EPT for it, depicted *F-EPT* in Figure 5. By including the physical pages corresponding to the guest (e.g., the targets specified in the dual guest/shadow EPT) in the F-EPT, the hyperplexor can share the guest memory with the featurevisor.

To avoid conflicts with the hyperplexor, the featurevisor must only use the shared guest memory for guest pages. As such, the hyperplexor and featurevisor agree upon a region of the featurevisor's physical memory that will be reserved for guest pages. This can be done through convention (e.g., a well-known memory region as in BIOS/OS interaction) or through an explicit handshake protocol using hypercalls. In Dichotomy, featurevisors allocate a reserved region for guest pages and register them with the hyperplexor upon initialization.

Managing the Guest Memory Map. As the guest runs on either the hyperplexor or featurevisor, the guest memory map may change. For example, a new page may be added to the guest memory map if the guest accesses part of its physical-memory space for the first time. Guest EPT faults are routed along either the hyperplexor path (denoted by "H" in Figure 5) or the featurevisor path (denoted by "F")

[1] This style of nested page table management is also known as *multi-dimensional paging* [29, 46]. Another approach to nested page table management, called *shadow-on-EPT*, creates a shadow page table in L1 for the guest's standard page tables. We do not discuss shadow-on-EPT further due to its performance overhead, caused by the more frequent standard-paging-related guest traps.

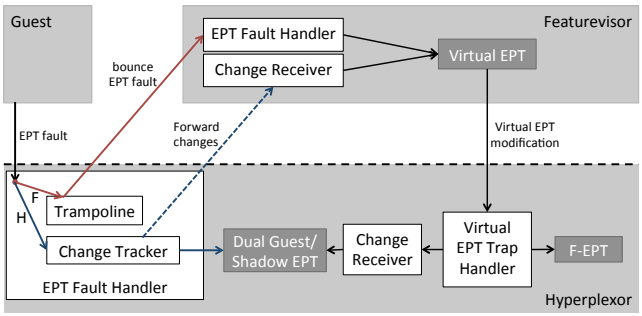


Figure 5: Memory management in Dichotomy

depending on whether the guest happens to be running on the hyperplexor or the featurevisor.

When the guest is running on the hyperplexor, memory management proceeds in a similar fashion to single-layer virtualization (Figure 4(a)). For the guest, the hyperplexor maintains the dual guest/shadow EPT in its "guest" role (its "shadow" role is described in the next paragraph in the context of the featurevisor). If the guest triggers an EPT fault (e.g., to trigger demand paging), the hyperplexor receives a trap. However, before updating the EPT, the EPT fault handler passes the fault through a *change tracker*. The change tracker records EPT modifications, which will be transferred to the featurevisor at some later time, before it begins running the guest. The featurevisor contains a *change receiver*, which interprets a list of recorded EPT modifications from the change tracker and updates the featurevisor's virtual EPT accordingly.

When the guest is running on the featurevisor, memory management proceeds in a similar fashion to standard nested virtualization (Figure 4(b)). The featurevisor does not need to implement a change tracker, because all EPT faults and updates already necessarily pass through the hyperplexor. When the guest is running on the featurevisor, guest EPT faults still enter the hyperplexor directly; the hyperplexor simply bounces the faults to the featurevisor via the trampoline. Then, the featurevisor handles the fault. If the featurevisor updates the virtual EPT due to the fault (or for any other reason), the hyperplexor will receive a trap. The hyperplexor passes the trap to its change receiver. The change receiver interprets the virtual EPT trap and updates the dual guest/shadow EPT (in its "shadow role" now). At this point, the hyperplexor dual guest/shadow EPT is synchronized with the virtual EPT in the featurevisor.

Encoding EPT Changes. When the hyperplexor makes a change to an EPT entry in the dual guest/shadow EPT, the change tracker encodes it in a format that will be later interpreted by the featurevisor's change receiver. In Dichotomy, the change tracker constructs a new EPT entry that specifies the mapping between a guest physical page and a featurevisor physical page. For an EPT entry in the dual guest/shadow EPT referring to the mapping $(x_{guest} \rightarrow y_{hyperplexor})$, the

```
forever:
  execute guest while waiting for a trigger

on_trigger:
  relinquish_guest
  wait_for_guest
```

Figure 6: Hyperplexor behavior

```
forever:
  wait_for_guest
  do action
  relinquish_guest
  finish action
```

Figure 7: Featurevisor behavior

change tracker first performs a reverse lookup in the featurevisor EPT to obtain the mapping $(z_{featurevisor} \rightarrow y_{hyperplexor})$. With this information, the change tracker can construct an entry that the featurevisor can directly interpret: $(x_{guest} \rightarrow z_{featurevisor})$. The change tracker also copies the flags (e.g., write protections, etc.) so that those in the new EPT entry match those in the dual guest/shadow EPT.

3.2 Guest Switching

Control of the guest switches between hyperplexor and featurevisor via a VM migration-like procedure, in which guest state is transferred between the migration engines in the hyperplexor and featurevisor. Specifically, one migration engine pauses the guest and transfers the guest VCPU state, I/O state, and any unsynchronized page table mappings to the other migration engine. As described above (and evaluated in Section 6), this transfer is efficient because only the EPT modifications need to be sent, not the page contents. The remaining state (CPU, I/O, etc.) is relatively small (about 16 KB).

The migration engines expose a simple interface to other programs (at the same level): `wait_for_guest`, which blocks until the migration engine at the other end issues a `relinquish_guest` call, which initiates the migration procedure.

The hyperplexor, which has control of the guest at the start of its execution, runs the guest until a *trigger* occurs. The hyperplexor conceptually runs the program specified in Figure 6. The featurevisor, on the other hand, conceptually runs the program specified in Figure 7. The action performed by the featurevisor is split into two segments to allow the featurevisor to relinquish the guest before completing its operation. For example, if the featurevisor is performing a snapshot-like activity, the featurevisor may compute an in-memory snapshot, relinquish the guest, then complete the action by writing the snapshot to disk.

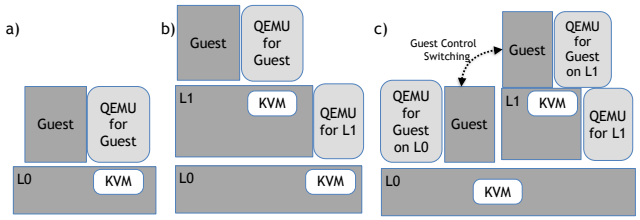


Figure 8: The position of KVM hypervisor as a Linux kernel module and QEMU as a user-level process, for a) single layer virtualization, b) standard nested virtualization, and c) Dichotomy.

Triggers. Each featurevisor in Dichotomy registers itself with the hyperplexor. As part of the registration process, the featurevisor specifies when it should be invoked from a set of hyperplexor *triggers*. The most trivial trigger is time based: for example, a featurevisor can specify a time interval detailing the frequency it should be invoked. To date, Dichotomy supports such a time-based trigger; further triggers, such as network events, or guest execution of a particular protected instruction or access to a particular part of memory, are discussed in Section 5.

4. Implementation

We have implemented Dichotomy using the KVM/QEMU virtualization platform in the Linux operating system. We begin with an overview of the roles of KVM and QEMU components, followed by implementation-specific details of guest memory sharing, fault handling, and control transfer.

Background. The KVM/QEMU [21] virtualization platform in Linux consists of two major components (a) KVM, which implements the core hypervisor functionalities as a Linux kernel module, and (b) QEMU which is a user-level management process, one per guest, that manages the life cycle of a guest, communicates with the KVM kernel module on behalf of the guest, and controls its I/O operations. Figure 8 shows the position of the KVM kernel module and the QEMU management process for non-nested, nested, and Dichotomy VMs. Figure 8(a) shows a non-nested guest running on the L0 KVM hypervisor and being managed by an associated QEMU process. When spawning a guest, the corresponding QEMU process designates a portion of its virtual memory address space as belonging to the guest, configures the guest's memory, virtual CPU and virtual I/O devices with the KVM kernel module, and launches the guest. Privileged operations by the guest result in `vmexits` or traps to the KVM hypervisor, which either handles the trap itself (such as for page faults) or forwards the trapping event to the QEMU process for further processing (such as I/O operations). Figure 8(b) shows a nested guest running on a L1 KVM hypervisor and being managed by an associated QEMU process which also runs on L1. The L1 hypervisor itself runs as a VM on the L0 hypervisor and has its own as-

sociated QEMU process on L0. `vmexits` by the guest are forwarded by the L0 KVM hypervisor to its L1 counterpart. Figure 8(c) shows a Dichotomy guest whose control switches back and forth between the L0 (hyperplexor) and L1 (featurevisor). At both levels, there is an associated persistent QEMU process that manages the Dichotomy guest's execution. These two QEMU processes coordinate with each other to exchange the guest execution state upon events that trigger the transfer of guest control. Depending on where the guest executes at any given instant, `vmexits` are handled as in standard non-nested or nested modes.

Memory Management. The guest's memory is shared between the featurevisor and hyperplexor dynamically during runtime as each page of the guest is accessed. To enable this dynamic sharing, however, the guest initialization operation must set up a sharing mechanism in the hyperplexor. Specifically, during initialization, the guest QEMU process on the hyperplexor registers the guest's memory map, i.e. QEMU's virtual memory address range associated with the guest, with its respective KVM. Similarly, the guest QEMU process on the featurevisor registers the guest's memory map with the KVM in the featurevisor, which in turn pre-allocates and registers a set of featurevisor physical addresses for the guest with the KVM in the hyperplexor. Since the guest memory is assigned from the virtual address space of the QEMU process, no physical memory is reserved in advance. Therefore, EPTs corresponding to the QEMU processes are empty at initialization.

The allocation and sharing of guest memory pages occurs when guest page faults are processed as described in Section 3.1 under the heading "Managing the Guest Memory Map". Our implementation differs from the design in the way we realize the *dual guest/shadow EPT*. To realize the ideal design – namely a single EPT that tracks guest memory map changes on both the hyperplexor and featurevisor – would have required extensive changes to the core memory management implementation in Linux and KVM. Instead, the hyperplexor maintains two EPTs for the guest – a guest EPT and a shadow EPT. The former is used when the guest executes over the hyperplexor and the latter is used when the guest executes over the featurevisor.

The guest EPT and shadow EPT are kept synchronized by the hyperplexor when handling guest EPT faults. Specifically, the EPT fault handling process described in Section 3.1 differs in the implementation in the following manner. Upon a shadow EPT fault (when the guest executes on the featurevisor), before allocating a new physical page to resolve an incomplete shadow EPT entry, the hyperplexor first checks if the faulting guest physical address has been already allocated in the guest EPT (as a result of the guest running previously on the hyperplexor). If the guest EPT mapping exists, then it is used to update the shadow EPT and F-EPT, else a new physical page is allocated. Conversely, upon a guest EPT fault (when the guest runs on the hyperplexor), the shadow EPT is first consulted before a new physical page is allocated.

Guest Switching. The QEMU process includes a pre-copy [9] based live migration mechanism wherein the entire VM state is transferred from a source hypervisor to a destination over a TCP connection. We implemented guest switching in Dichotomy by modifying this migration mechanism in the QEMU of both the hyperplexor and featurevisor. Besides the primary modification of replacing guest memory transfer with memory sharing (described above), we also implemented an interface for triggering the guest switching. When the trigger is invoked, since the memory is shared, only the CPU and I/O states are transferred over the TCP connection between the two QEMU processes. After each switch, the featurevisor and hyperplexor QEMUs change their roles, from source to target or vice-versa. The QEMU which relinquished the guest now waits for the guest to return; while waiting, it maintains all the user and kernel data structures representing the guest state so they can be updated and reused once the guest returns.

Interestingly, we found that the use of certain virtualization features trigger the reinitialization of VM structures. For example, during migration, on the destination side, the load of VAPIC state will reinitialize the EPT. We found this had the side effect of imposing a "cold-start" penalty on the VM workload performance immediately after a guest switch due to the need to repopulate the virtual and shadow EPT entries from scratch. In our implementation, we presently disable usage of the VAPIC to avoid EPT reinitialization so that this cold-start penalty is avoided. In future work, we will investigate ways to avoid EPT reinitialization without disabling VAPIC.

5. Discussion and Future Directions

In this section, we discuss future work needed for a broader applicability of ephemeral virtualization. We focus the discussion on triggers, hyperplexor services, featurevisor composition, and featurevisor support.

Triggers. Dichotomy currently supports time-based triggers, for which featurevisors specify how often they should run. This is useful for sample-based featurevisor services, such as monitoring. However, a richer set of triggers in the hyperplexor may enable more interesting featurevisors.

For example, featurevisors could provide interesting network functionality in a similar style to software-defined networking (SDN) if they could be triggered every time a new network flow arrives. The featurevisor could then decide what to do with the flow, for example, it could block flows to act like a firewall, or record them for a network tomography application. Such a trigger could be implemented in the hyperplexor by augmenting or replacing the existing mechanism to call the SDN controller in open vSwitch.

As another example, featurevisors could provide interesting debugging or logging facilities if they could be triggered

every time a guest accesses a particular part of memory. For example, the featurevisor could track changes to an important data structure in order to trigger replication of the data. This type of trigger could be implemented in the hyperplexor by modifying the permission bits on EPT entries and modifying the EPT fault handler to trigger a switch to the featurevisor.

Similarly, featurevisors could provide interesting debugging or logging facilities if they could be triggered every time a particular guest function is executed. Upon gaining control, the featurevisor could step through the function, emulating each instruction. One way to implement such a "guest program counter" trigger in the hyperplexor could be by temporarily rewriting guest code at load time to ensure that the guest traps at a specific program counter value.

Hyperplexor Services. There may be a set of common "featurevisor utilities" that are independently implemented in many featurevisors and can be implemented as services in the hyperplexor. For example, tracking guest memory writes (dirty page tracking) is useful for featurevisors performing a wide range of memory-related activities, including working set estimation and guest checkpointing. Gaining control on every guest write would result in a high duty cycle (near 100%) for these applications. Furthermore, switching so often would be unnecessary if the hyperplexor maintained and delivered a summary of guest pages written in the current epoch to the featurevisor. In other words, implementing a dirty page tracking utility in the hyperplexor may enable a class of featurevisors to become significantly more efficient. As the set of featurevisors grow, there is an opportunity to identify such utilities across featurevisors and explore their implementation in the hyperplexor.

Featurevisor Composition. To this point, we have described Dichotomy in terms of a one-to-one mapping between guest VMs and featurevisors. It is possible that this relationship be generalized in either direction.

First, a VM could be associated with multiple featurevisors. For example, the VM could infrequently switch to a featurevisor performing a backup service and more frequently switch to a featurevisor performing sample-based network monitoring. The main issues to consider when associating multiple featurevisors with a VM relate to how the featurevisors interact. For example, which featurevisor takes precedence if both are triggered by the same event? How can the hyperplexor ensure that triggers for one featurevisor are not lost when executing on a different featurevisor? Even with cooperative featurevisors, the mechanisms needed to enable a single VM to be associated with multiple featurevisors is a subject of future work.

Second, multiple VMs could be associated with a single featurevisor. For example, multiple VMs could temporarily use a high-speed shared memory communication channel [27, 45] implemented in the featurevisor. Conceptually, this requires straightforward changes to the guest memory registration area and the communication channel (to identify which guest VM each message relates to).

	Host	Featurevisor	Guest
Single	4 CPUs, 10 GB	N/A	2 VCPUs, 1-8 GB
Nested	12 CPUs, 128 GB	4 VCPUs, 10 GB	2 VCPUs, 1-8 GB
Dichotomy	12 CPUs, 128 GB	4 VCPUs, 10 GB	2 VCPUs, 1-8 GB

Table 1: System configuration (CPU, Memory) of single-layer, nested, and Dichotomy virtualization.

Featurevisor Support. Finally, our implementation of both the hyperplexor and featurevisor is based on the KVM/QEMU hypervisor. It is conceptually straightforward to allow featurevisors to be based on different hypervisors (e.g., Xen) or built from scratch. The key design of sharing memory instead of migrating it can be maintained despite heterogeneous hypervisors; however, the remaining VM state and messages through the communication channel must be sent in a "canonicalized" form, with "drivers" at each featurevisor to translate them into their featurevisor-native structures.

6. Evaluation

In this section, we compare the performance of Dichotomy against alternative approaches for implementing hypervisor-level services, and use experimental results to show that Dichotomy is best suited for the task. Our results show that Dichotomy delivers low performance overheads due to fast switching times. The specific goals of our experiments are as follows:

- Investigate the operating region where the overheads of running VMs on Dichotomy are low and an improvement over running VMs on nested hypervisors full time.
- Demonstrate two featurevisors that implement hypervisor-level services–VM introspection and network monitoring–to manage unmodified guest VMs.
- Demonstrate that the switching times between the hyperplexor and the featurevisor are small and the associated penalty is minimal.

Our evaluation setup consists of a server containing six dual-core Intel Xeon 2.10 GHz CPUs and 128 GB memory. The L0 and L1 hypervisors run the 3.14.2 Linux kernel, KVM 3.14.2, and QEMU 1.2.0.

We compared the performance of a VM running on Dichotomy against a VM running on a single-layer hypervisor and a nested hypervisor. Table 1 shows the system configurations for the three approaches. The guest VM (column 4) is assigned 2 VCPUs and 1 to 8 GB memory in all three configurations. The featurevisor (column 3) is assigned 4 VCPUs and 10 GB memory in the nested and Dichotomy configurations. The physical host (column 2) is restricted in the single-layer virtualization configuration to use 4 physical CPUs and 10 GB memory, to match the featurevisor in the

	idle(s)	kernbench(s)	netperf(s)
volatility	3.42±0.15	3.43±0.25	3.34±0.10
netmon	1.08±0.05	1.079±0.006	1.084±0.009

Table 2: Service times t_f of featurevisor applications in seconds.

other two configurations; in nested and Dichotomy configurations, the host uses all available CPUs (12) and memory (128 GB).

6.1 Workloads

The guest VMs used in our experiments were either idle or ran one of the three benchmarks below. We perform 5 iterations for each of the tests and report the average.

- **Quicksort** is the simplest of our benchmarks as it only stresses CPU and memory, but does not perform I/O after initialization. The `quicksort` benchmark consists of two phases: *initialization*, in which the benchmark allocates 800MB of memory and populates it with random data; and *sorting*, in which the benchmark sorts the data using quicksort. We measure the time taken to complete the sorting.
- **Kernbench** [22] is a multi-threaded benchmark that measures the time taken for repeatedly compiling the Linux kernel. `kernbench` stresses memory, CPU and I/O. It reads the kernel source code files from an external disk, compiles the code and writes the binary output files back to the disk. We used the default setting, in which `kernbench` uses two threads to compile the kernel and performs three iterations to measure the average compilation time.
- **Netperf** [30] is a single-threaded network benchmarking tool. It is primarily I/O bound. We use `netperf` to measure network throughput. We run a `netperf` client inside the guest and a `netperf` server on an external host. The machine hosting the guest and the external host are connected to the same switch with 1Gbps Ethernet links. During each test, the `netperf` client sends a TCP stream to the `netperf` server. We measure the average throughput of the TCP stream over 100 seconds.

For the experiments with Dichotomy and the nested hypervisor, we used three featurevisor configurations.

- The **no-op** featurevisor provides only standard VM management functions offered by KVM/QEMU. It immediately relinquishes control back to the hyperplexor.
- The **volatility** featurevisor implements a VM introspection application. `volatility` [43] is an introspection tool that saves a memory dump of a VM, then performs analysis on it. For our experiments, the output of the analysis is an accurate list of all processes running inside the VM. We configured `volatility` to save the VM's memory dump in a memory-based filesystem (*tmpfs*) to avoid disk I/O overheads.
- The **netmon** featurevisor implements a network monitoring application using the `tcpdump` tool to capture packets traversing through the virtual network interface of the VM. To approximate sample-based monitoring, the featurevisor only runs `tcpdump` for 1 second before relinquishing the guest.

quicksort	Runtime(s)	Slowdown (α)	CPU(%)
base	60.6±0.547	1.0	100
no-op	62.8±0.44	0.96 (β)	99
volatility	63.4±0.89	0.95	99
netmon	62.8±0.44	0.96	99
kernbench	**Runtime(s)**	**Slowdown (α)**	**CPU(%)**
base	48.37±3.63	1.0	92
no-op	54.6±0.42	0.88 (β)	100
volatility	56±2.77	0.86	100
netmon	63.3±13.35	0.76	100
netperf	**Mbps**	**Slowdown (α)**	**CPU(%)**
base	941.1±0.014	1.0	4
no-op	853.0±14.7	0.91 (β)	40
volatility	725.5±5.58	0.77	32
netmon	830.5±16.34	0.88	40

Table 3: Workload performance, slowdown (α), and CPU usage. The slowdown for `no-op` is equivalent to standard nesting overhead (β).

The guest VM is migrated between the hyperplexor and featurevisor at a given sampling rate. For the nested virtualization configuration, the featurevisor functionality is implemented in the L1 hypervisor, while in the single-layer virtualization configuration, it is implemented directly in the L0 hypervisor.

6.2 Application Characterization

We begin with a characterization of the featurevisor applications in terms of both their service times and their impact on the performance of workloads running within the guest VM. *Service time* is the length of time needed by the featurevisor to perform a task on the guest VM.

Table 2 shows the service times for the `volatility` and `netmon` applications, representing the theoretical minimum switching period for a Dichotomy VM with 2GB memory. Notice that the service times remain largely unaffected by the VM's workload because the featurevisor has sufficient vCPU and memory resources (as shown in Table 1) to run its application and, if necessary, to prioritize its execution over the VM.

Table 3 shows the slowdown of the `quicksort`, `kernbench` and `netperf` workloads when the guest runs on top of a featurevisor (with no switching). The slowdown

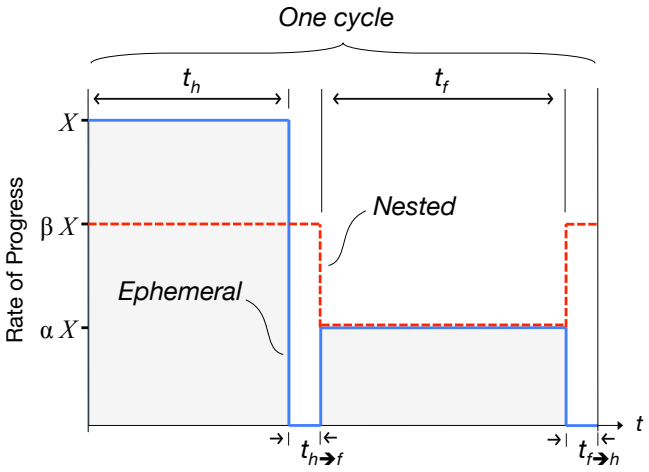


Figure 9: Analysis of one duty cycle

factor α represents the normalized performance of the workload compared to the guest running directly in an unmodified KVM L0 hypervisor (the row labeled 'base'). For featurevisors, the workload performance is measured with either an idle featurevisor (`no-op`) or one that continually executes the `volatility` or `netmon` tasks. The slowdown on the `no-op` featurevisor represents the minimum overhead of running a guest on a nested hypervisor (L1). We observe the most slowdown with `kernbench` and `netperf` due to I/O contention.

6.3 Expected Results

We do not expect Dichotomy to outperform nested virtualization for all featurevisor applications and all workloads. For instance, featurevisors that require a high duty cycle (especially with high switching frequencies) will incur prohibitive switching overhead. To build intuition and increase confidence in our experimental results, we first examine the operating region analytically.

Without loss of generalization, we only consider a single cycle (Figure 9). We focus on quantifying the progress that a guest VM workload makes during a full cycle, both in full nested mode and when switching back and forth between the hyperplexor and featurevisor. Let t_h be the time a VM spends on the hyperplexor. Let t_f be the time a VM spends on the featurevisor. During t_h, let X represent the expected rate at which the VM makes progress in $Units/t$. When a VM runs on top of a featurevisor, it can be slowed down in two ways. The first slowdown is purely due to nesting overhead. This is captured by β, resulting in a reduced rate of progress βX. The second slowdown is when a featurevisor application is running. There, the VM will be slowed down by rate α (i.e., due to nested virtualization overhead and contention for resources utilized by the featurevisor application). Thus, we expect the VM to make αX progress.

As depicted by the solid blue line in Figure 9, in ephemeral mode, the VM's rate of progress will switch between X and αX. There is, however, a non-zero switching time in which the VM is paused (i.e., makes no progress). This time is captured by $t_{h\rightarrow f}$ and $t_{f\rightarrow h}$, which represent the time to move the VM between the hyperplexor and the featurevisor, and vice versa. In contrast, Figure 9 also shows the VM's rate of progress in full nested mode (the red dotted line) which switches between αX and βX with no switching time.

A full cycle, t, can thus be expressed as $t_h + t_{h\rightarrow f} + t_f + t_{f\rightarrow h}$. The progress that a guest VM workload makes with ephemeral virtualization ($P_{ephemeral}$) is:

$$P_{ephemeral} = Xt_h + (0 \times t_{h\rightarrow f}) + \alpha X t_f + (0 \times t_{f\rightarrow h}) \quad (1)$$

In contrast, the progress made by the VM in full nested mode (P_{nested}) is expressed as:

$$P_{nested} = \alpha X t_f + \beta X (t_h + t_{h\rightarrow f} + t_{f\rightarrow h}) \quad (2)$$

Now, we calculate the range where the VM makes more progress during ephemeral mode when compared to nested mode (i.e., when $P_{ephemeral} > P_{nested}$). Performing direct substitution of the above equations and solving for t_h, we get:

$$t_h > \frac{\beta}{1-\beta}(t_{h\rightarrow f} + t_{f\rightarrow h}) \quad (3)$$

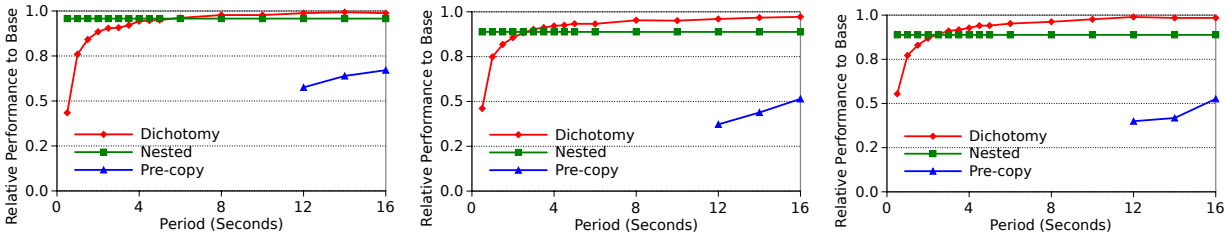
The results confirm intuition: *it is better to switch back and forth as long as the VM stays on the hyperplexor long enough to amortize the switching cost.* In the next subsection, we experimentally find that ephemeral mode delivers lower performance overhead than full nested mode for periods as small as 2.5 seconds.

6.4 Macro Benchmarks

Figures 10, 11, and 12 show the relative performance of each VM workload (`quicksort`, `kernbench`, and `netperf`) compared to the the single level virtualization (`Base`), under each featurevisor (`no-op`, `netmon`, and `volatility`). Each graph shows the results when running the VM and featurevisor in Dichotomy, a standard nested virtualization environment, and a naïve ephemeral virtualization implementation using standard pre-copy live migration as the switching mechanism (denoted `Pre-copy`).

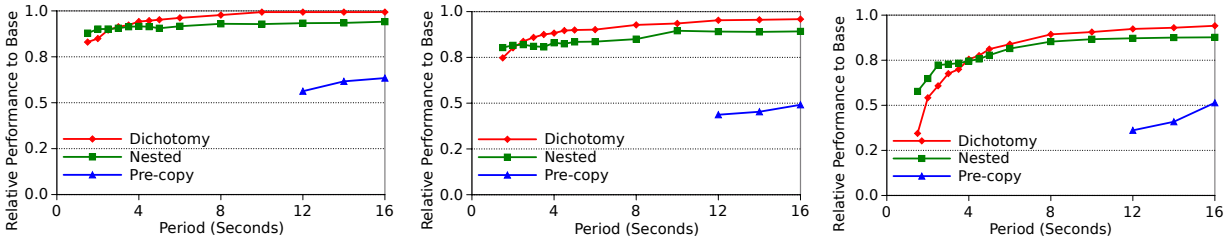
For these experiments, we vary the period length, or the frequency at which we run the featurevisor. We determine the actual period lengths at which ephemeral virtualization outperforms nested virtualization in a practical setting. Moreover, we are interested in the following general trends: 1) Dichotomy outperforms nested for sufficiently large period length, and 2) a fast switching mechanism is important for ephemeral virtualization.

All of the results confirm these general trends. In general, guest performance in Dichotomy converges more quickly to a better value than nested virtualization, up to 12% improvement. Regardless of the featurevisor application, as period



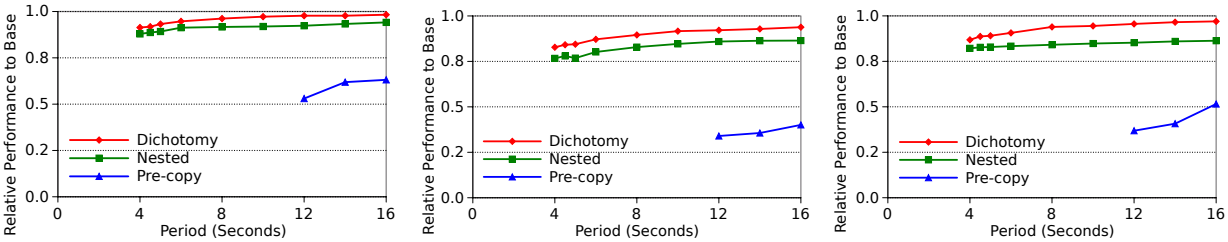
(a) `quicksort` (b) `kernbench` (c) `netperf`

Figure 10: Workload runtimes when switching between hyperplexor and the `no-op` featurevisor.



(a) `quicksort` (b) `kernbench` (c) `netperf`

Figure 11: Workload runtimes when switching between hyperplexor and a featurevisor running the `netmon` task.



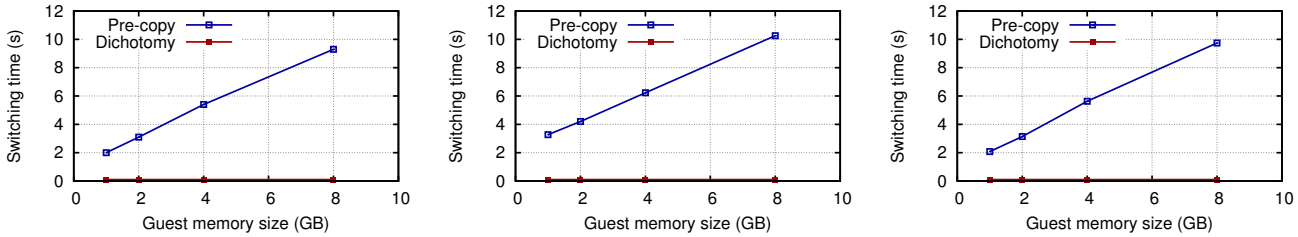
(a) `quicksort` (b) `kernbench` (c) `netperf`

Figure 12: Workload runtimes when switching between hyperplexor and a featurevisor running the `volatility` task.

length increases, VM performance on Dichotomy converges to single-level virtualization (X from the analysis in Section 6.3), whereas VM performance on nested converges to base nested performance with no featurevisor running (βX from the analysis in Section 6.3). The importance of a fast switching time is dramatically emphasized by the poor performance of pre-copy live VM migration in all cases.

To explain the results in more detail, we first examine Figure 10. The `no-op` featurevisor does no work, but immediately yields the guest back to the hyperplexor. It does, however, incur switching overhead. The smallest period we test is about 160 ms, due to the fact that switching overhead (about 80 ms each direction) dominates at periods this small. As expected, Figure 10 shows that the performance of the workloads with Dichotomy suffers at short periods due to the switching overhead, but the performance improves with longer periods. For comparison, pre-copy requires about 4s to switch the guest's execution. Therefore the workload performance can only be demonstrated for period durations greater than 10s ($>$ *2 * switch time + featurevisor service time*). Furthermore, the iterative memory copying during the workload execution adversely impacts the workload performance with pre-copy.

The performance of `kernbench` and `netperf` with Dichotomy matches the performance with the nested mode at 3 and 2.5 second periods respectively, while the performance of `quicksort` matches its performance with nested only after a 6 second period. Since `quicksort` does not stress I/O, as opposed to `netperf` and `kernbench`, it has lower nesting overhead in comparison, (i.e. βX from the analysis in Sec-



(a) idle (b) `kernbench` (c) `netperf`

Figure 13: Switching times of guest VMs with varying memory sizes between hyperplexor to featurevisor.

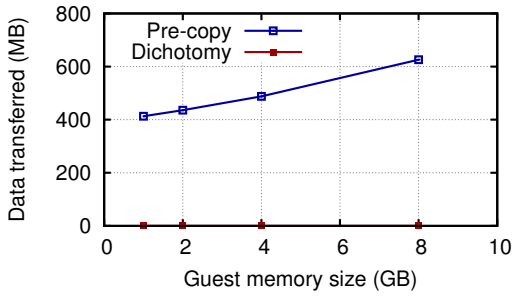


Figure 14: Data copied for switching

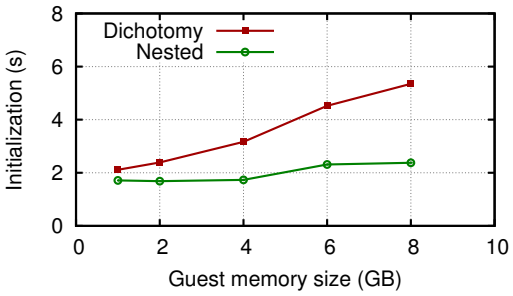


Figure 15: One-time initialization cost

tion 6.3 for `quicksort` is higher than that with `kernbench` and `netperf`). Therefore, for `quicksort`, Dichotomy can only amortize the switching overhead for longer period durations (i.e., when t_h is greater).

Figure 11 shows the performance of the all three workloads with `netmon`. In each period, the guest executes on the featurevisor for 1s before switching back to the hyperplexor. All three workloads show a similar pattern as with the `no-op` featurevisor, with an amortization point of 4 seconds or less. The performance of the VMs on Dichotomy remains fairly stable, except for the `netperf` workload. The `netperf` workload is most affected by the `netmon` featurevisor because it is a network-bound workloads.

Figure 12 shows the workload performance with the `volatility` featurevisor application. For each period, the guest executes on the featurevisor until the VM introspection is complete. Table 2 shows the service time of `volatility` for different workloads. Since the service time with `volatility` is longer than with `netmon`, we can demonstrate the workload performance for period lengths 4s or greater and as a result the crossing point of Dichotomy and nested is not visible. This result is important, as it indicates that for realistic featurevisors with service times of a few seconds, Dichotomy will always outperform standard nesting, even if they run frequently.

6.5 Micro Benchmarks

In the previous subsection, we showed that Dichotomy's low switching time played an important role in the system's performance. In this subsection, we quantify Dichotomy's switching times and its VM initialization overheads. Figure 13 compares VM switching times between the hyperplexor and a `no-op` featurevisor using Dichotomy and pre-copy live VM migration. Switching times to and from the hyperplexor—$t_{h\to f}$ and $t_{f\to h}$ —were observed to be similar, hence they are not distinguished here. The figure shows that, as the guest memory size is increased from 1 GB to 8 GB, the measured switching time for pre-copy also increases (ranging from 2 to 10 seconds), mainly due to memory copying overhead. In contrast, the switching time using Dichotomy is fairly constant around 80 ms, since the guest memory is shared in advance during initialization by the hyperplexor and the featurevisor.

We also measured the downtime experienced by a guest running `kernbench` when switching between the hyperplexor and the featurevisor when using pre-copy versus dichotomy. With pre-copy, the downtime remains within the range of 300ms to 350ms, whereas with Dichotomy the downtime is around 80 ms.

Figure 14 shows the number of data bytes copied as the VM size is increased; pre-copy shows an increase from 400MB to 625MB; Dichotomy transfers a constant amount of 15.8KB consisting of VCPU and I/O state.

Figure 15 compares the one-time initialization overhead for a nested guest against the Dichotomy guest. The nested guest initialization time increases slightly from 1.7 seconds to 2.3 seconds. However, Dichotomy guest initialization time increases from 2.1 seconds to 5.3 seconds since the initialization involves sharing the Dichotomy guest's memory between the hyperplexor and the featurevisor. In our current implementation, this requires the featurevisor to register the guest's memory address space with the hyperplexor through multiple hypercalls, the cost of which increases with increasing guest memory size. This cost can be potentially reduced by batching the registration operations into fewer hypercalls.

7. Related Work

Several lines of research have considered dynamic switching between execution environments: virtualized, bare-metal, and emulated. In this section, we discuss such switching approaches, and related work on hypervisor feature confinement and nested virtualization.

Ho et al. [19] use an on-demand approach to switch between single-layer guest VM execution (on Xen) and a QEMU-based emulator. Their system is tailored to a single application: performing taint tracking and the enforcement of a taint policy to disallow execution of tainted data. With Dichotomy, we are interested in a general approach to support a range of applications of which taint-based policy enforcement is an interesting use-case.

There has also been work examining dynamic switching between a "bare-metal" operating system and running the OS in a virtual machine. The on-the-fly introduction of a VMM underneath an operating system has been explored by on-demand virtualization [23]. It relies on OS hibernation mechanisms and the conversion results in about 90 seconds of downtime. Also, Mercury [8] proposes "self-virtualization", in which a VMM is dynamically attached or detached beneath an operating system only when needed with about 0.2 ms switching time. VMware offers products to convert between physical and virtual machines and vice versa [40, 41], but they do not target running systems. By targeting nested virtualization, Dichotomy leverages existing mechanisms to deal with the complexity of physical hardware, interrupt mapping, etc.

We propose a hypervisor-as-a-service environment in which many different hypervisors provide specialized features. In related work, there have been approaches that reduce the hypervisor's functionality to its essentials for a particular problem domain. A "microvisor" [26] does not virtualize all resources and is only applicable for the problem of online server maintenance. Similarly, CloudVisor [49] uses nested virtualization on a slim, trusted base hypervisor. Dichotomy's support for introducing and removing layers complements this work.

A related line of research relates to disaggregating the large administrative domain [6, 10, 28, 35] typically associated with a hypervisor, such as Domain 0 in Xen. The goal of these efforts to is replace a single large administrative domain with several small sub-domains (akin to privileged service-VMs) that are more resilient to attacks and failures, better isolated from others, and can be customized on a per-VM basis. Thus a VM could pick and choose the services of specific sub-domains which run at the same level as the VM atop the common hypervisor. Dichotomy's nested architecture essentially reduces the privilege of the featurevisor in addition to splitting them from the hyperplexor.

Dichotomy leverages existing work on nested virtualization and live VM migration. Nested virtualization was originally proposed and refined in the 1970s [4], has been studied in a microkernel environment [14], and has now become mainstream due to new implementations [5, 16] leveraging hardware support for virtualization on the x86 architecture [1, 38]. Dichotomy uses nested support in KVM [21]. Live VM migration [9, 18] enables VMs to migrate from one hypervisor to another with minimal downtime. Dichotomy both transfers control and completes migration faster than existing techniques by eliminating memory copying.

8. Conclusion

We presented Dichotomy, a new two-layer cloud architecture that splits the role of hypervisors between hyperplexors and featurevisors. This split enables cloud providers to focus on the security and stability, while at the same time allowing a third-party featurevisor ecosystem to grow in a hypervisor-as-a-service idiom. Experiments with our prototype show that, through ephemeral virtualization, Dichotomy delivers better VM performance than nested virtualization, even for featurevisor applications that access the VM as often as every 2.5 seconds. We attribute Dichotomy's performance to low VM switching times between hyperplexor and featurevisor, averaging 80 ms. In the future, we look forward to exploring the interaction and agility of featurevisors in a hypervisor-as-a-service model.

Acknowledgement

This work is supported in part by the National Science Foundation through grants 1527338, 1320689, and 0845832, and by the Air Force Rome Labs through grant CA01160915BINGU.

References

[1] AMD Virtualization (AMD-V). http://www.amd.com/us/solutions/servers/virtualization.

[2] A. Arcangeli, I. Eidus, and C. Wright. Increasing memory density by using ksm. In *Proc. of Linux Symposium, Ottawa, Canada*, July 2009.

[3] S. F. Barrett and D. J. Pack. *Microcontrollers Fundamentals for Engineers and Scientists*, chapter 4, pages 51–64. Morgan & Claypool Publishers, San Rafael, CA, July 2006.

[4] G. Belpaire and N.-T. Hsu. Formal properties of recursive virtual machine architectures. In *Proc. of ACM SOSP, Austin, TX*, pages 89–96, Nov. 1975.

[5] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The turtles project: Design and implementation of nested virtualization. In *Proc. of USENIX OSDI*, Vancouver, Canada, Oct. 2010.

[6] S. Butt, H. A. Lagar-Cavilla, A. Srivastava, and V. Ganapathy. Self-service cloud computing. In *Proc. of ACM CCS, Raleigh, NC*, pages 253–264, Oct. 2012.

[7] H. Chen, R. Chen, F. Zhang, B. Zang, and P. Yew. Live updating operating systems using virtualization. In *Proc. of ACM VEE*, Ottawa, Canada, June 2006.

[8] H. Chen, R. Chen, F. Zhang, B. Zang, and P.-C. Yew. Mercury: Combining performance with dependability using self-virtualization. In *Proc. of IEEE ICPP*, Xi'an, China, Sept. 2007.

[9] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proc. of USENIX NSDI*, Boston, MA, May 2005.

[10] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield. Breaking up is hard to do: Security and functionality in a commodity hypervisor. In *Proc. of ACM SOSP*, Cascais, Portugal, Oct. 2011.

[11] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: high availability via asynchronous virtual machine replication. In *Proc. of USENIX NSDI*, San Francisco, CA, Apr. 2008.

[12] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *Proc. of ACM CCS*, pages 51–62, 2008.

[13] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proc. of USENIX OSDI*, Boston, MA, Dec. 2002.

[14] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proc. of USENIX OSDI*, Seattle, WA, Oct. 1996.

[15] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. of NDSS Symposium, San Diego, CA*, Feb. 2003.

[16] A. Graf and J. Roedel. Nesting the virtualized world. In *Linux Plumbers Conference, Portland, OR*, Sept. 2009.

[17] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. In *Proc. of USENIX OSDI*, San Diego, CA, Dec. 2008.

[18] M. Hines and K. Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proc. of ACM VEE*, Washington, DC, Mar. 2009.

[19] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. In *Proc. of ACM EuroSys*, Leuven, Belgium, Apr. 2006.

[20] Intel 64 and IA-32 Architectures. Software Developers Manual, Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B, 3C and 3D. `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf`.

[21] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the linux virtual machine monitor. In *Proc. of Linux Symposium, Ottawa, Canada*, June 2007.

[22] C. Kolivas. Kernbench: http://ck.kolivas.org/apps/kernbench/kernbench-0.50/.

[23] T. Kooburat and M. Swift. The best of both worlds with on-demand virtualization. In *Proc. of USENIX HOTOS*, Napa, CA, May 2011.

[24] K. Kourai and S. Chiba. HyperSpector: Virtual Distributed Monitoring Environments for Secure Intrusion Detection. In *Proc. of ACM VEE*, Chicago, IL, June 2005.

[25] J. Levon. *OProfile: System-wide profiler for Linux systems, http://oprofile.sourceforge.net/about/*.

[26] D. E. Lowell, Y. Saito, and E. J. Samberg. Devirtualizable virtual machines enabling general, single-node, online maintenance. In *Proc. of ACM ASPLOS*, Boston, MA, Oct. 2004.

[27] A. C. Macdonell. *Shared-memory optimizations for virtual machines*. PhD thesis, University of Alberta, Edmonton, Canada, 2011.

[28] D. G. Murray, G. Milos, and S. Hand. Improving xen security through disaggregation. In *Proc. of ACM VEE*, Seattle, WA, Mar. 2008.

[29] G. Natapov. Nested EPT to make nested VMX faster. In *KVM Forum, Edinburgh, UK*, Oct. 2013.

[30] Netperf. http://www.netperf.org/netperf/.

[31] D. L. Osisek, K. M. Jackson, and P. H. Gum. ESA/390 interpretive-execution architecture, foundation for VM/ESA. *IBM Systems Journal*, 30(1):34–51, Feb. 1991.

[32] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *IEEE Symposium on Security and Privacy, Oakland, CA*, pages 233 – 247, May 2008.

[33] RedHat CloudForms. *http://www.redhat.com/en/technologies/cloud-computing/cloudforms*.

[34] R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing. In *Recent Advances in Intrusion Detection, Boston, MA*, pages 1–20, Sept. 2008.

[35] U. Steinberg and B. Kauer. Nova: A microhypervisor-based secure virtualization architecture. In *Proc. of EuroSys, Paris, France*, pages 209–222, 2010.

[36] S. Suneja, C. Isci, V. Bala, E. de Lara, and T. Mummert. Non-intrusive, out-of-band and out-of-the-box systems monitoring in the cloud. In *SIGMETRICS'14, Austin, TX*, 2014.

[37] J. Toldinas, D. Rudzika, V. Štuikys, and G. Ziberkas. Rootkit detection experiment within a virtual environment. *Electronics and Electrical Engineering–Kaunas: Technologija*, (8): 104, 2009.

[38] R. Uhlig, G. Neiger, D. Rodgers, A. Santoni, F. Martins, A. Anderson, S. Bennett, A. Kagi, F. Leung, and L. Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.

[39] vmitools. *https://code.google.com/p/vmitools/*.

[40] VMware, Inc. Virtual Machine to Physical Machine Migration. `http://www.vmware.com/support/v2p/doc/V2P_TechNote.pdf`, 2004.

[41] VMware, Inc. VMware Converter Users Manual. `http://www.vmware.com/pdf/VMware_Converter_manual.pdf`, 2006.

[42] VMWare vRealize. *https://www.vmware.com/products/vrealize-suite*.

[43] Volatility Framework. http://code.google.com/p/volatility/.

[44] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proc. of USENIX OSDI*, Boston, MA, Dec. 2002.

[45] J. Wang, K.-L. Wright, and K. Gopalan. XenLoop: a transparent high performance inter-VM network loopback. In *Proc. of ACM HPDC, Boston, MA*, pages 109–118, June 2008.

[46] O. Wasserman. Nested Virtualization: Shadow Turtles. In *KVM Forum, Edinburgh, UK*, Oct. 2013.

[47] D. Williams, H. Jamjoom, and H. Weatherspoon. The Xen-Blanket: Virtualize once, run everywhere. In *EuroSys, Bern, Switzerland*, Apr. 2012.

[48] Xen Cloud Platform. *http://wiki.xenproject.org/wiki/XCP_Overview*.

[49] F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proc. of ACM SOSP*, Cascais, Portugal, Oct. 2011.