Out of Hypervisor (OoH): When Nested Virtualization Becomes Practical

Stella Bitchebe  
Université Côte d’Azur, ENS Lyon  
France  
bittebe@i3s.unice.fr

Alain Tchana  
ENS Lyon  
France  
alain.tchana@ens-lyon.fr

Abstract—This paper introduces Out of Hypervisor (OoH), a new research axis close to nested virtualization. Instead of emulating a full virtual hardware inside a VM to support a hypervisor, OoH principle is to individually expose current hypervisor-oriented hardware virtualization features to the guest OS so that its processes could also take benefit from those features. In fact, several hardware virtualization features such as Intel PML, SPP, CAT and EPT which currently can only be used by the hypervisor also be beneficial for processes which run inside the VM. We illustrate OoH with Intel PML (Page Modification Logging), a feature which allows efficient dirty page tracking for improving VM live migration. According to the fact that dirty page tracking is at the heart of process checkpointing (CRIU) and concurrent garbage collection (Boehm), we present two OoH PML designs namely Shadow PML (SPML) and Extended PML (EPML). The former requires no hardware changes but incurs significant overhead, justifying EPML which extends PML. We evaluated and compared SPML and EPML with /proc and userfaultfd, two default solutions in Linux. We do this using a key value store database as the benchmark. The results show that EPML reduces CRIU checkpointing time by about 14% while leading to a negligible overhead (of about 0.5%) compared to SPML, /proc and userfaultfd.

I. INTRODUCTION

Nested virtualization is the stacking of multiple hypervisors [14, 15, 19, 22, 23, 28, 29], see Fig. 1b. It induces a significant overhead due to the huge number of VM traps (at least ×2 [28]) compared to non-nested virtualized systems, see Fig. 1a. Vilanova et al. [28] measured that the execution of cpuid instruction is degraded by about 73% in a nested virtualized system. Despite the efforts of both the industry and academia to reduce this overhead [15, 22, 23, 28], nested virtualization is recommended by cloud providers such as Microsoft Azure [6] for testing, development, and demo. Even if there is some niche utilization of nested virtualization (to realize rootkits [18]), its adoption in production is not currently envisioned by cloud users. Indeed, when cloud customers need isolation and deployability within VMs, they prefer containers (e.g., Docker [3]), which are better in terms of performance and have a rich ecosystem (e.g., Kubernetes [5]). Even the ideal VM-based nested virtualization solution would lead to a higher overhead than container-based nested virtualization as containers naturally outperform VMs.

In this paper we introduce Out of Hypervisor (OoH), a new research axis which advocates that instead of putting a lot of research efforts to emulate a full virtual hardware inside the VM for supporting a real hypervisor, we can focus on exposing current hypervisor-oriented hardware virtualization features – such as Intel Page Modification Logging (PML), Sub-Page write Protection (SPP), Cache Allocation Technology (CAT), Extended Page Table (EPT) – to the guest OS so that its processes (thus containers) can also take benefit from those virtualization features, see Fig. 1d. We claim that many hypervisor-oriented hardware features, which currently can only be used by the bare metal hypervisor for managing VMs, could also be beneficial for processes which run inside VMs. As an example, the exposition to the guest OS of Intel Page Modification Logging (PML), which is currently only used by the hypervisor for dirty page tracking for accelerating VM live migration and checkpointing, would allow improving process/container checkpointing (e.g., with CRIU [2]) or garbage collection (e.g., with Boehm [4]). Another example is Intel Sub-Page write Protection [4] that would allow reducing memory waste in guard page-based secured heap memory allocators [24, 26, 27].

One may legitimately ask what is the difference between OoH and Dune [13]. The difference is that Dune leverages hypervisor-oriented hardware virtualization features (such as Extended Page Table) to make privileged instructions, which are generally only accessible in kernel mode (ring 0), available to processes, see Fig. 1c. OoH makes hypervisor-oriented hardware virtualization features accessible to processes in virtualized machines. Dune processes may take benefit from OoH.

To demonstrate OoH we show in this paper how Intel PML can be exposed to guest’s processes. We call this OoH-PML. Then we illustrate the power of OoH-PML by using it to speedup CRIU and Boehm GC. Traditional solutions for dirty page tracking are usually costly as they are based on write protection, which induces a lot of page faults. The popular solution under Linux is based on dirty bit and present bit invalidation using the /proc interface. As shown in Section III this solution is extremely expensive and can increase execution time by up to 208.47%. Prior to PML, hypervisors also use such approaches to track dirty pages

1The hypervisor which directly runs when the machine is powered-on.
2Intel SPP allows memory protection at the granularity of 128B instead of 4KB.
during VM live migration and checkpointing. To accelerate these operations, thanks to their importance in the cloud, Intel in collaboration with VMware introduced PML, a hardware virtualization feature that allows the processor to log in RAM guest physical addresses (GPAs) of pages for which the dirty bit has been set during page table walk. Xen and KVM, two popular open-source hypervisors, currently support PML when the machine provides it. Bitchebe et al. [16] empirically validated the benefits of PML in accelerating VM live migration and checkpointing.

PML is currently a hypervisor-oriented feature, meaning that it can only be used by a hypervisor. However the need for tracking dirty pages is also present within the VM (e.g., process checkpointing or garbage collection). The application of OoH to PML raises three main challenges: (C₁) PML can only be managed by the hypervisor. We want userspace processes to be able to control PML and to access the PML buffer. These should be done without hindering the hypervisor functioning. (C₂) PML now works at coarse-grained, that is it concerns the entire VM. We want to use PML at the granularity of a process inside the VM, while allowing its use by the hypervisor at the scale of the entire VM. (C₃) PML only logs GPA. Userspace processes manipulate guest virtual addresses (GVA).

In this paper, we first explore two designs of OoH-PML. Second, we show how they can be used by CRIU and Boehm GC. The first design, that we call Shadow PML (noted SPML), requires no hardware changes as it is based on the current PML implementation. The second design, that we call Extended PML (noted EPML) extends PML to avoid all the limitations of SPML. We present SPML in order to justify the necessity of EPML because we know that hardware changes, even minor as in EPML, are not easy to actually implement. SPML makes the hypervisor emulate PML in software for the guest OS. Relying on hypercalls and virtual interrupt injection inside the guest, SPML regularly reports logged GPA to a ring buffer shared between the hypervisor and the guest. It is then up to the guest level to perform reverse mapping from GPA to GVA using the tracked process’s page table. SPML also manages to coordinate hypervisor and guest usages of PML. Concerning EPML, it hijacks the hardware virtualization features namely VMCS shadowing [11] and posted-interrupt [12] and slightly extends the original PML to avoid the intervention of the hypervisor when it is used by the guest OS. VMCS shadowing and posted-interrupt were introduced to allow guest OS directly manipulating virtual machine control structures (VMCS) and receiving interrupt respectively. We extend PML in two ways. First, an EPML capable CPU is able to log to two PML buffers simultaneously. The first PML buffer, that we call guest-level PML buffer, is configured and exclusively controlled by the guest OS. The second PML buffer, that we call hypervisor-level PML, is configured and exclusively controlled by the hypervisor. Each tracked thread is associated a guest-level PML buffer that is placed by the guest OS into the processor every time it is scheduled-in. Second, we extend the CPU to log the GVA (instead of GPA) of the modified page to the guest-level PML buffer. Note that the GVA is already known by the processor at logging time.

We also present in this paper a library for the utilization of OoH-PML. That library is independent from the chosen OoH-PML design. We implement it following the UIO driver principle [11], that is it is composed of two parts: a kernel module and a userspace template code. The former does not need to be manipulated by the application developer while she should integrate the latter with her application. We do not rely on the /proc interface, although it would have required no application modification since several applications (e.g., CRIU) already rely on, because it would have required kernel modification. Also, as /proc is a file system, it brings performance degradation.

We have prototyped and evaluated SPML on a DELL Intel Core i7-8565U processor machine that supports PML, using Xen hypervisor. Concerning EPML, we rely on BOCHS, the only emulator, to the best of our knowledge, that implements PML. In both cases, the guest OS is Linux. We use CRIU and Boehm GC as usecases. To respect the page limit, we do not discuss Boehm GC results. The integration of the library requires 251LOC for CRIU. We use both micro- and macro-benchmarks. Regarding the latter, we use tkzw [9], which is a set of database managers. To evaluate our prototypes, we set...
up a rigorous methodology, especially for EPML, because of the lack of a machine that supports it. Therefore, we build a formula which quite accurately approximates its performance.

We systematically compare SPML and EPML with two existing dirty page tracking solutions namely /proc and userfaultfd. In terms of performance impact on the tracked process, the solutions are classified as follows, in decreasing order of the overhead they induce: SPML, userfaultfd, /proc, and EPML. SPML induces an overhead of up to 6546% on the micro-benchmark and 58% on tkrzw Tiny in-memory database engine. EPML leads to the lowest overhead, which is about 0.5% on the micro-benchmark and 0.47% on tkrzw. Concerning CRIU, SPML lengthens the checkpointing time by about 903% while EPML reduces it by about 14%.

In summary, we make the following contributions:

- (Principle) We introduce OoH, a new research axis for nested virtualization.
- (Empirical contribution) We finely quantify the impact of page fault-based dirty page tracking.
- (Conceptual contribution) We present two designs of OoH-PML namely SPML and EPML.
- (Technical contribution) We prototype SPML and EPML in real and emulated environments (BOCHS) using popular software systems (Xen hypervisor and Linux guest OS).
- (Technical contribution) We integrate OoH-PML into two popular systems namely CRIU and Boehm garbage collection.
- (Empirical contribution) We rigorously evaluate our designs using micro- and macro-benchmarks (tkrw).

The rest of the paper is organized as follows. Section II presents the background. Section III motivates our work. Sections IV presents SPML and EPML. Sections V presents the integration into usecases. Section VI presents the evaluation results. Section VII presents the state of the art. Section VIII presents the conclusion.

II. BACKGROUND

This section presents the necessary background to understand our contribution.

A. Virtual Machine Control Structure (VMCS)

Intel VT-x (Virtualization Technology extensions) is a hardware virtualization technology for Intel processors. Its architecture is based on a central design decision that is to not change the semantics of individual instructions of the ISA. With VT-x, a virtualized environment may allow the guest OS to execute exactly the same instructions as on bare metal. Intel VT-x introduces two new CPU execution modes namely vmx root and vmx non-root, that are orthogonal to the traditional 4 cpl levels. The hypervisor runs in vmx root cpl 0 mode whereas the guest OS runs in vmx non-root mode cpl 0, thus avoiding the necessity to rewrite the guest OS. Interactions and transitions between vmx root and vmx non-root modes are automatically triggered by the CPU using the virtual machine control structure (VMCS) associated to the running vCPU. In addition to control structures, a VMCS also stores the guest state (when transitioning from guest to hypervisor) and the hypervisor state (when transitioning from the hypervisor to the guest).

VMCS is manipulated using new instructions (e.g., vmwrite, vmread, vmlaunch, etc.). In its early version, VMCS was only manipulated in vmx root mode. To improve nested virtualization, Intel recently introduced VMCS shadowing which organizes VMCS in two types: ordinary and shadow VMCS. A shadow VMCS is a VMCS that is pointed by another one, which becomes an ordinary VMCS. The latter is only manipulated by the hypervisor while the former can be directly accessed by the guest OS. Thus in vmx non-root mode, vmread and vmwrite instructions can be executed by the guest OS on shadow VMCS. Despite the introduction of shadow VMCS, nested virtualization of an entire VM is still inefficient because other hardware features are still unavailable in vmx non-root mode for nested hypervisors. Our work exploits VMCS shadowing to design an efficient OoH use case.

B. Intel Page Modification Logging (PML)

PML is a hardware virtualization technology that has been introduced to allow the hypervisor to efficiently monitor dirty memory pages. It relies on Extended Page Table (EPT) and requires specific changes of VMCS. When PML is enabled, each write instruction that sets the dirty flag in EPT during page table walk triggers the logging of the GPA at its origin. To this end, a new 64-bit VM-execution control field called PML Address is added to VMCS. The PML address points to a 4KB memory page, called PML Buffer, that can hold up to 512 logged GPA. A new 16-bit guest-state field, called PML Index, is also introduced to indicate the index of the next logging entry in PML buffer. PML index starts at 511. Whenever the PML buffer is full, the CPU triggers a VMExit and the hypervisor takes the hand. The handler of that VMExit then copies the content of the PML buffer to a larger buffer (managed by the hypervisor) and resets the PML index to 511. In Xen and KVM, which are two popular open source hypervisors (used by Amazon EC2), the content of the larger buffer is used to know which pages should be resent during the VM live migration pre-copy phase. It is important to note that in its current implementation, the activation of PML is machine (concerns all CPUs) and VM (concerns all vCPUs) wide. In this paper we are trying to make PML efficiently usable from inside a VM, at the granularity of a process.

III. MOTIVATIONS

Dirty page tracking, thus PML, is not only essential for hypervisors. A thread which runs inside a VM may also need to monitor dirty pages for garbage collection or checkpointing. We call Tracker the monitoring thread and Tracking the thread whose memory is monitored. The traditional approach used by Tracker is the invalidation of dirty and present bits from
Tracked’s page table entries (PTE). Linux offers two interfaces that Tracker can leverage. These are `userfaultfd` and `/proc`. Fig. 2 summarizes their functioning compared to an OoH-PML-based solution. `userfaultfd` and `/proc` are introduced in this section while OoH-PML is presented in Section IV. The activity of Tracker can generally be organized in four main phases: the initialization of the tracking method, the monitoring, the collection of dirty page addresses and the exploitation of the latter (e.g., for checkpoiniting).

We consider in this section that the fourth phase is empty as its duration is agnostic to the tracking method, in comparison with the three other phases. We launch Tracker and Tracked at the same time but the latter is suspended during the initialization phase. The ideal execution time of Tracked is when it runs without being tracked. The ideal execution time of Tracker is the ideal execution time of Tracked. As one can deduce from Fig. 2 the choice of the tracking method can impact both Tracker and Tracked. We can see that OoH is the only method which theoretically leads both systems to their ideal execution time.

| Size   | 1MB | 10MB | 50MB | 100MB | 250MB | 500MB | 1GB |
|--------|-----|------|------|-------|-------|-------|-----|
| Tracked |     |      |      |       |       |       |     |
| userfaultfd | 155 | 224  | 588  | 1,201 | 1,256 | 1,403 |     |
| /proc | 104 | 55  | 119  | 208   | 302   | 307   | 255 |
| Tracked | 153 | 109 | 477  | 940   | 1,269 | 1,153 | 1,349 |
| userfaultfd | 47 | 45 | 58 | 148 | 151 | 143 | 147 |
| /proc | 47 | 45 | 58 | 148 | 151 | 143 | 147 |

Table I: Overhead (in %) of `userfaultfd`- and `/proc`-based dirty page tracking methods.

A. The cost of `userfaultfd`

Fig. 2.b summarizes the functioning of a `userfaultfd`-based dirty page monitoring solution. To use `userfaultfd`, Tracker first registers the memory region that it wants to monitor. After the registration, it will be notified by the kernel each time a page fault which concerns the registered region occurs. `userfaultfd` supports two monitoring modes: `miss` and `write_protect`. For `miss`, a notification is sent to Tracker when Tracked accesses a monitored page for the first time. Concerning `write_protect`, a notification is sent when Tracked attempts to modify a monitored page. In both modes, Tracker is suspended until the fault is resolved. In the case of `write_protect`, the Tracker should write-protect the faulted page in order to unpause Tracked. One can see that, with `userfaultfd`, the collection of dirty page addresses can be done during the monitoring phase.

We assess the overhead of `userfaultfd` using as Tracked a synthetic program (presented in VII) that just parses and writes to an array of buffers. The size of each buffer is 4K, allocated at page boundaries. We are interested in monitoring the entire array. Table II second and fifth rows show the overhead of `userfaultfd` while we vary the array size. We can see that the overhead linearly increases with the array size. We measured an overhead of up to 1.462% and 1.349% for 1GB on Tracked and Tracker respectively. We breakdown the page fault handling time into two components: the time spent inside the kernel (about 33.6ms for 1GB) and the time spent in Tracker (about 3,338ms for 1GB). The total suspension time of Tracked represents in average about 93% of its execution time.

B. The cost of `/proc`

Fig. 2.a summarizes the functioning of a `/proc`-based dirty page monitoring solution. Tracker first instructs the kernel to clear soft-dirty bits of Tracked’s PTEs. This is done by writing 4 to `/proc/PID/clear_refs` file, where PID is the process identifier of Tracked. This operation is dominated by the time taken by the kernel to parse Tracker’s PTEs and to flush the TLB (about 2.234ms when the monitored memory is 1GB). All of this lengthens the initialization step compared to `userfaultfd`. After this, once Tracker tries to modify a monitored page, a fault occurs. The handler of that fault sets in Tracker’s PTE the soft-dirty bit of the faulted page (this operation costs about 33.5µs). At the end of the monitoring period, Tracker reads the soft-dirty bits (bit 55) from `/proc/PID/pagemap` in order to determine all dirty page addresses (this costs about 594.187ms when the monitored memory is 1GB). The total suspension time represents about 73% of the total execution time of Tracked.

As shown in Table I top, the impact of `/proc` on Tracked varies with the memory size. We measured an overhead of about 335.24% for 1GB of memory. This overhead is lower than the one induced by `/userfaultfd` as shown above. Concerning Tracker, see Table I bottom, the overhead varies from 47% up to 147%. When compared to `/userfaultfd`, although `/proc` increases the address collection phase, its cost is compensated by the smaller suspension it induces during the monitoring phase.

IV. OoH-PML: DESIGN AND IMPLEMENTATION

Although our design is generic, we rely on a Xen virtualized system to describe it. The guest OS is Linux. The goal of OoH-PML is to make PML usable from inside the guest OS. We do this with as minimum as possible changes in the hardware, the hypervisor, the guest kernel. In addition, we want to propose a simple utilization interface to application developers.

A. General overview

We study two designs namely Shadow PML (noted SPML) and Extended PML (EPML). SPML requires no hardware modification while EPML introduces a slight extension of the original PML feature. SPML exploits all existing mechanisms to make an OoH version of PML that works as efficiently as possible with existing machines. Despite these efforts, we measured a very significant overhead due to the current architecture of PML, thus justifying EPML. The latter leverages existing nested virtualization features (such as shadow VMCS and Posted-Interrupts) and requires minor hardware changes in order to provide the ideal OoH-PML performance.

The utilization of OoH-PML requires a small modification of Tracker. In fact, we provide OoH-PML to userspace following the UIO driver principle, that is the driver is organized
in two components: a kernel module and a userspace code. This organization brings flexibility in our design that is the kernel and the implementation of OoH-PML can evolve independently. The userspace code that we provide is a template that the Tracker developer should integration. This template includes the following steps. First the code registers with the UIO kernel module the PID of Tracked. This allows the kernel to mark Tracked as to be monitored using OoH-PML. After that, the template code mmap a ring buffer (see below its initialization) and loops on it. This buffer is the place where the addresses of dirty pages are stored by OoH-PML. Contrary to the original PML which logs GPA, OoH-PML logs GVA. At the end of the loop, Tracker can unregister Tracked with the kernel to stop its monitoring.

The UIO kernel module is at the heart of the software part of OoH-PML. At load time, it does some initialization operations such as: hardware initialization of PML, allocation and initialization of the ring buffer which is shares with the hypervisor (for SPML only) and future Trackers. When a Tracker registers with the module for monitoring, the latter installs with Linux scheduler two callbacks leveraging the preempt-notifier interface: one for schedule-in events and the second for schedule-out events. This way, the module can know when to enable or disable PML.

Although the utilization of the UIO driver principles requires a slight modification of Tracker, we choose this approach rather than /proc which is likely already taken into account several Tracker applications (such as CRIU) for two main reasons: performance and code scalability. First /proc is a file system, thus its access requires to walk through the file system stack which is known to degrade performance. Second, to integrate /proc with OoH-PML, we should modify the core of the guest kernel (Linux) for two purposes: (1) implementation of a new version of echo 4 > /proc/clear_refs; (2) for each PML buffer full event, the handler should walk through Tracking’s PTE to set the dirty bit for the corresponding entries. We showed in Section III that theses operations are costly.

B. Shadow PML (SPML)

The basic idea behind SPML is to make the hypervisor emulates PML for the guest.

1) PML initialization and logging: The actual initialization of PML is done by the hypervisor when loading the guest loads the UIO kernel module. It has no effect if PML was already initialized by another VM. It consists in initializing VMCS structures, allocating a region in RAM for the PML buffer and initializing the PML index. As the initialization of PML concerns all vCPUs of the VM, processors may log to the same buffer GPA from any process running inside the VM. This raises the first challenge \((C_1)\) of SPML as, we aim to log only the addresses of processes that the user wants to track in the guest. Indeed, if the guest kernel schedules-in a process that does not need to be tracked, the processor will log to the PML buffer the addresses generated by that process, which are not those needed by Tracker. To alleviate this, we introduce two new operations for HYPERVISOR_vcpu_op hypercall: enable_logging and disable_logging. We use these hypercalls to control the processor logging process at low cost as follows. When a tracked process is scheduled-out, the guest kernel performs a disable_logging hypercall and the hypervisor retrieves from the PML buffer all the addresses that have been logged and sets the PML index to 512 to prevent logging. When a tracked process is scheduled-in, the guest kernel performs a enable_logging hypercall and the hypervisor resets to 511 the PML index to make the CPU restart its logging activity.

2) PML buffer handling: When the PML buffer is full (at the hypervisor level), the processor stops logging and triggers a vmexit so that the hypervisor can retrieve the addresses and reset the PML index. In SPML we set up a ring buffer to which the hypervisor copies the logs. When the ring buffer itself is full, the hypervisor triggers a virtual interrupt at the destination of the guest OS, which is handled by our UIO kernel module. The latter in turn notifies Tracker in userspace. No more address is filled in the ring buffer until it is flushed. Notice that, Tracker can manage to regularly collect addresses from the ring buffer in order to avoid this situation. This is what we do in our experiments.

Because there is a unique PML buffer per vCPU, when many processes are tracked simultaneously we need to ensure the homogeneity of logs while filling the ring buffer. This is the second challenge \((C_2)\) of SPML: how to identify logged addresses per process? To answer this question, we organize the ring buffer in blocks in which the two first entries are respectively the PID of a tracked process (filled in by the UIO driver module when scheduling the process in) and the current number of addresses logged for this process (incremented by...
the hypervisor when filling the ring buffer and decremented by Tracker when collecting logs).

3) Coexistence with the hypervisor: SPML must not interfere with the utilization of PML by the hypervisor. This is the third challenge (C₃). Let us imagine that the guest asks for deactivating PML while the hypervisor is still using it (e.g., to perform live migration) or vice versa. We coordinate their intervention as follows. (1) We add two flags enable_by_vmm and enable_by_guest per vCPU to indicate which level has initialized PML. (2) We add some control checks in the PML buffer shuching code path inside the hypervisor so that: if enable_by_vmm is not set, the hypervisor does not execute the original portion of code corresponding to its own use (this may incur unnecessary additional CPU time utilization as the hypervisor is not actually using PML); if enable_by_guest is not set, the hypervisor does not fill the ring buffer. (3) When the guest asks to deactivate PML, a control check first verifies that enable_by_vmm is not set (meaning that the guest was the only level to use PML) and vice versa. If the hypervisor or the guest asks to disable PML while the other is still using it, we just unset the corresponding flag without effectively disabling PML on the processor. (4) On schedule-out of a tracked process, as stated in §IV-B1, the guest instructs the hypervisor to stop logging addresses for its use. In this case, the hypervisor first flushes the PML buffer so that each of the hypervisor and the guest may collect addresses that have been logged till this moment, and if enable_by_vmm is set, we just unset another flag sched_in that tells the hypervisor if a tracked process is scheduled-in (and thus if it must fill the ring buffer or not); otherwise (i.e., the hypervisor is not using PML), we set the PML index to 512. Doing so, the CPU will stop logging, which avoids unnecessary CPU time waste and additional vmexits as neither the hypervisor nor the guest needs logs at that moment. (5) On schedule-in of a tracked process, the guest also instructs the hypervisor to restart collecting addresses and filling the shared ring buffer. In this case, we need to flush the PML buffer to avoid collecting addresses that do not belong to the current tracked process. But before doing this, the hypervisor checks the flag enable_by_vmm: if set, the hypervisor first copies the addresses that are in the PML buffer and resets the PML index.

To verify that this coordination protocol effectively works, we performed five iterations of live migrations on a VM using SPML. We observed no crash of the VM and no interruption of live migration.

4) Address collection in userspace: With PML, the processor logs the GPA of the modified pages while userspace needs GVA. This is the fourth challenge (C₄) of SPML. To address this challenge, the template code released with SPML includes a portion of code which performs a reverse mapping operation to translate each GPA to GVA every time it reads the ring buffer in the loop. To do so, it walks through the tracked process’s page table relying on /proc interface.

5) SPML limitations: SPML leads to two limitation types: performance overhead and address lost. Performance overhead is due to two factors: a potential huge number of hypercalls (at each process schedule-in and out event) and the reverse mapping operation to translate GPA to GVA. The former factor impacts both Tracked and Tracker while the latter factor only impact Tracker. Concerning address lost, if a GPA is no longer present in the page table (because of an unallocation) of the tracked process at reverse mappin time or has been assigned to another GVA, the corresponding GVA would either be nil or inaccurate. All these limitations motivate Extended PML.

C. Extended PML (EPML)

The basic idea behind EPML is to provide a second level of PML that is entirely controlled by the guest OS.

1) PML initialization and logging: From a software point of view, the differences between EPML and SPML are as follows (see Fig. 3): (1) the ring buffer is no longer shared with the hypervisor; (2) the flags enable_by_vmm and enable_by_guest are no more used. To minimize hardware changes, EPML leverages the existing VMCS shadowing feature. At kernel module load time, the hypervisor is called to initialize the shadow VMCS pointed to by each of ordinary VMCS maintained for the VM. This is the only hypercall performed in EPML. This initialization consists of allocating a memory region for the shadow VMCS and filling its address in the VMCS link pointer field of the ordinary VMCS, and initializing the fields VMREAD_BITMAP and VMWRITE_BITMAP of the shadow VMCS. These bitsmaps allow the hypervisor to specify which fields of the shadow VMCS the guest is allowed to read or write without trapping. When a tracked process is scheduled-in or out, the kernel accordingly enables or disables the logging process by setting the PML index to 512 in the shadow VMCS using a vmwrite instruction.

![Figure 3: Hardware access difference between EPML and SPML.](image)

From a hardware point of view, EPML makes the following changes. We introduce in the VM-Execution Control area of VMCS a new field (called Guest PML Address) that represents the address of the memory region where the processor will log the addresses for the use of the guest OS. EPML allows the guest to configure a guest-level PML buffer per tracked process, exactly as the hypervisor manages a PML buffer per vCPU. Subsequently, when a new process is registered to be tracked, the guest kernel allocates a memory region for its guest-level PML buffer. Further, when this process is scheduled-in, before enabling the logging process, the guest kernel loads its corresponding guest-level PML buffer by filling its address in the Guest PML Address field of
the shadow VMCS using a `vmwrite` instruction. Because the
guest only sees GPA, the value that it sets to the Guest PML
Address field should be translated to a host physical address
(HPA) so that the processor can log to the right location in
RAM. To tackle this, we extend the VMX ISA so that if a
`vmwrite` instruction is performed (to fill the Guest PML
Address field) when the processor is in VMX non-root mode,
it first translates the address to a HPA (using EPT or TLB)
before writing it to the shadow VMCS. We claim that this
translation is very less expensive than performing a hypercall
as in SPML.

Another improvement brought by EPML is the capability of
the processor to log GVA in order to avoid reverse mapping
of GPA as SPML requires. We modify the page walk process
so that the processor can log the GVA to the guest-level PML
buffer and the GPA to the hypervisor-level PML buffer.

2) Buffer full event handling: Fig. 4 summarizes the differ-
ence between EPML and SPML. To handle guest-level PML
buffer full events, EPML leverages Posted-Interrupts. With the
latter the processor is able to deliver a virtual interrupts in vmx
non-root mode without vmexits. Using this functionality, we
modify the hardware behavior so that when a guest-level PML
buffer is full, the processor raises a virtual `self-IPI` (Inter-
Processor Interrupt). This interrupt is handled by the UIO
kernel module. As the latter services a `virtual` device, it cannot
directly receive an interrupt from the processor. So we define
a new interrupt vector in Linux that the processor uses for the
virtual IPI. To redirect the handling of that virtual IPI to the
UIO module, we rely on the Linux-SoftIRQ principle. We define
a new softirq and divide the handling of the virtual interrupt
into two parts: the top-half in the core kernel and the bottom
half in the UIO module. The top half just raises the softirq
while the bottom half copies the content of the guest-level
PML buffer to the ring buffer.

![Figure 4: Buffer full handling difference between EPML and
SPML.](image)

3) Coexistence with the Hypervisor: VMCS shadowing
was introduced for the purpose of nested virtualization. Once
initialized by the hypervisor, the latter no longer interferes
with its utilization. Anyway, as the VM and the hypervisor
can be both using PML simultaneously we must just pay little
attention on PML index. Along with the guest-level PML
buffer address, we also introduce in the VMCS structure a
new field for its PML Index.

We study the utilization of OoH by CRIU and Boehm GC,
which only consists in integrating the userspace template code
with these systems. Due to page limit, we only present CRIU
in the present paper.

a) CRIU overview: CRIU is a Linux software that can
checkpoint to disk the state of a process or a container. It
is currently integrated into many well-known projects such
as OpenVZ [7], Podman [8] or Docker [3]. To checkpoint a
process, CRIU saves the state of the latter as a collection of
files: we talk about dumping the application. CRIU supports
two dumping modes namely incremental and non-incremental.
Non-incremental dumping consists in saving the entire mem-
ory state of the process while in incremental mode CRIU only
dumps dirty pages since the previous dump. Thus, OoH could
be beneficial for incremental dumping. In respect with our
nomenclature, CRIU is a Tracker. In its current implementation,
it relies on `/proc`. To integrate our solution, we mainly
patched two CRIU steps (the reader should remember the four
steps of Tracker introduced in III and Fig. 3):

- Initialization phase: In respect with the utilization
  `/proc`, CRIU first pauses the tracked tasks before initial-
  ization (echo 4 > /proc/PID/clear_refs), as
  illustrated in Fig. 2a. With OoH, this pause is no longer
  necessary as the activation of PML is immediate and does
  not interfere with the execution of the tracked thread, as
  illustrated in Fig. 2c.
- Address collection: We discard from CRIU the portion of
code which parses `/proc/PID/pagemap` for collecting
addresses as with OoH addresses are collected during the
monitoring phase.

The patch adds 251 LOC to CRIU in 9 files.

VI. EVALUATIONS

The goal of the evaluation is to answer the following
questions: (1) what is the potential overhead or improvement
of SPML and EPML compared to existing solutions (`/proc`
and `userfaultfd`)? (2) what is the scalability of SPML and
EPML? (3) to what extend the utilization of PML from inside
the guest impacts its utilization by the hypervisor? (4) to what
extend SPML and EPML are able to capture the entire dirty
page working set?

A. Experimental environment

a) Machine and Systems: We carried out our experiments
on a machine with 8 Intel Core i7-8565U and 16 GB of RAM.
The virtualization system we use is Xen 4.10.0 and each VM
runs Ubuntu 18.04 with Linux 4.15.0. Especially to evaluate
`userfaultfd` we use Linux 5.11 because version 4.15.0
does not support the `write Protect` mode. Concerning the
emulation of EPML, we use BOCHS 2.6.11. In all experiments
on the real machine, the VM has 4 dedicated vCPUS with 5GB
of memory. In the emulated environment, the VM has 1 vCPU
and 1GB of memory due to BOCHS constraints.
b) Benchmarks: We use a micro- and a macro-benchmark. The former is a synthetic algorithm which parses an array of 4KB buffers, see listing 1. Its performance metric is the execution time. For the macro-benchmark, we use tkrzw [9], which is a suite of database managers. We evaluated all five in-memory engines that tkrzw provides using a load injector which performs set requests. Table II shows for each manager the configuration setup.

| Parameter          | Baby           | Cache          | Stdhash        | Stdthree       | Tiny           |
|--------------------|----------------|----------------|----------------|----------------|----------------|
| #threads           | 3              | 5              | 4              | 7              | 5              |
| #iterations        | 10M            | 10M            | 10M            | 100M           | 10M            |
| #max records       | -              | 10M            | -              | 100M           | 10M            |
| compression mode   | -              | -              | zlib           | -              | -              |
| #buckets for hashing| -              | -              | 10M            | -              | 100M           |

Table II: The configuration set-up for each tkrzw engines. (M means million)

Because of the lack of an EPML capable machine, we perform experiments on the real machine. To explain that equation, let us reuse the same nomenclature (Tracker and Tracked) introduced in Section III. Notice that we are interested in evaluating the performance of Tracker and Tracked. We use the term entity to refer to one or the other. Given \( P_{ideal} \), the ideal execution time of an entity, \( C_{vmread} \) and \( C_{vmwrite} \) respectively the costs of vmread and vmwrite, \( N \) the sum of the number of scheduled-in and scheduled-out events which involves Tracked, and \( C_{copyrb} \), the cost of copying the ring buffer content, then the performance of the entity under EPML (noted \( P_{EPML} \)) is estimated using this equation:

\[
P_{EPML} = P_{vanilla} + N \times (3 \times C_{vmurite} + C_{vmread}) + C_{copyrb}
\]  

All the results presented in this section are mean of 5 runs. We present in this section the cost of all internal metrics which allow to understand higher level performance metrics. The value of these metrics also tell us about the scalability of each tracking technique. The first column of Table III lists the metrics, organized in nine categories. For each metric, we indicate if its value is influenced or not by Tracked memory size (second column of table III). For metrics which are agnostic to Tracked memory size, their basic cost is reported in the third column of table III. For the other metrics, we reported their basic costs in table IV while varying Tracked memory size. We also indicate in column four of table III which tracking methods influence the metric.

Table V summarizes our analysis of the values reported in table III and table IV.

- **/proc**: it influences 4 metrics, including 3 which depend on Tracked memory size. Among them, the most costly metric is page table walk in userspace \( (M_{36}) \) that can take up to 594 ms. Page fault handling in kernel space \( (M_{70}) \) is the second most costly metric, that can take up to 233 ms, which is quite significant in respect with the fact that it may involve frequently during the monitoring phase. This is why it dramatically impacts /proc scalability (from both Tracked and Tracker points of view).
- **userfault**: it influences 3 metrics, including 2 which depend on Tracked memory size. The most costly metric is page fault handling in userspace \( (M_{60}) \), which costs up to 3,483 ms when Tracked memory size is 1GB. This metric, in addition, is involved during the monitoring phase, thus impacting the scalability of userfault for (from both Tracked and Tracker points of view).
- **SPML**: it influences 10 metrics, including 4 which depend on Tracked memory size. The most costly metric is reverse mapping \( (M_{17}) \) which takes up to 15Kms for 1GB Tracked memory size. This metric will only impact the scalability of Tracker because it is not involved in the monitoring phase. Fig. 5 presents the proportion of time taken by each step of the collection phase in SPML. We can observe that reverse mapping is definitely the bottleneck of SPML as it represents in average more than 60% of the total collection time.
- **EPML**: it influences 8 metrics, including only one metric depends on Tracked memory size. The most costly metric is PML initialization \( (M_{10}) \) which also includes VMCS
shadowing initialization. It costs about 5,878\,\mu s. Because this metric does not depend on Tracked memory, it does not impact the scalability of Tracker. The metrics that impact the scalability of Tracked are \texttt{vmread} ($M_7$) and \texttt{vmwrite} ($M_8$) whose costs are very low (less than 1\,\mu s). This makes EPML scalable.

C. Impact on Tracked

This section presents the impact of each dirty page tracking technique on Tracked. For these evaluations, Tracked is the micro-benchmark. Fig. 6 presents the results. We observe that, for almost all memory sizes, SPML incurs the greatest overhead compared to the other techniques. The overhead of EPML is almost null regardless Tracked memory size. This makes EPML the best technique amongst all and confirms its scalability.

D. CRIU

We compare three prototypes: CRIU \texttt{/proc}, CRIU SPML and, CRIU EPML. We did not consider it useful to implement

![Figure 6: Overhead of each tracking technique on the micro-benchmark. The values for EPML shown as they are too low.](image-url)
a userfaultfd-based CRIU version because our main goal is to outperform CRIU /proc, which is the default implementation of CRIU. Notice that, according to our nomenclature, CRIU is Tracker.

![Graph](image)

(a) During the initialization phase

(b) During the collection and exploitation.

Figure 7: CPU usage generated by CRIU /proc and CRIU SPML

**a) Impact on CRIU:** We use the micro-benchmark (Tracked) for this evaluation. We evaluate the impact on each CRIU phase. First, Fig. 7 shows CPU consumption values for CRIU /proc and CRIU SPML for the initialization (Fig. 7a), collection+exploitation (Fig. 7b) phases. In CRIU, collection+exploitation is the actual checkpointing phase, which is the most critical one. Compared to /proc, which requires to clear dirty bits at initialization time, we can observe that SPML reduces the CPU usage during by about 14.3% in average. Also, the CPU usage with CRIU SPML is almost constant while it increases with Tracked memory size in CRIU /proc. We did not collect CPU consumption values for CRIU EPML according to the fact that it is emulated. Nevermind, for the initialization phase we think that it would lead to the same results as CRIU SPML. Concerning the collection+exploitation phase, its CPU consumption would be lower than both CRIU SPML and CRIU /proc. It would correspond to CPU usage to write dirty pages to disk.

Table VI shows the checkpointing time. We can see that CRIU EPML is the best prototype. It outperforms CRIU /proc and CRIU SPML by about 61% and 1,515% respectively, when Tracked memory size is 1GB.

| Size      | 1MB  | 10MB | 50MB | 100MB | 250MB | 500MB | 1GB  |
|-----------|------|------|------|-------|-------|-------|------|
| /proc     | 107.17 | 132.45 | 280.35 | 399.26 | 577.97 | 888.47 | 1,627 |
| SPML      | 112.81 | 148.66 | 327.42 | 756.09 | 2,123 | 5,520 | 16,326 |
| EPML      | 105.33 | 119.36 | 237.74 | 327.96 | 405.24 | 607.00 | 1,011 |

**b) Impact on Tracked:** For this evaluation Tracked is a tkrzw engine. Table VII shows the amount of memory consumed by each engine during the experiment. We use these memory consumption values to compute, in the case of EPML, the cost of the internal metrics that depend on Tracked memory size. Fig. 8 presents the overhead of each solution. We can observe that SPML significantly slowdowns the tracked application. An overhead of up to 58% is observed for Tiny. The performance difference between CRIU SPML and CRIU /proc for Tiny is 114%. Concerning CRIU EPML, it leads to the lowest overhead. It ameliorates CRIU /proc by about 11% on Stdhash.

![Graph](image)

Figure 8: Overhead of CRIU /proc, CRIU SPML, CRIU EPML on five tkrzw engines. Lower is best.

**E. Potential Missed Addresses under SPML**

As stated in §IV-B5, SPML may miss some dirty page addresses during reverse mapping. This section evaluates the proportion of these addresses while varying the dirty page working set size of Tracked. The latter is the micro-benchmark. Fig. 9 shows the results. We can observe that the proportion decreases with the working set size: from 76.7%
down to 2.7%. Therefore, applications with a small dirty page working set size will suffer a lot from this SPML limitation. Anyway, losing even one dirty page could be problematic for the system to which dirty page tracking are destined. 

For example, the fault recovery system which relies on the latest checkpoint file will try to restart the application with an inconsistent state, which may crash. This may compromise the high availability goal that checkpoint-restore was supposed to bring. By design, EPML does not suffer from this problem.

F. Coexistence with the Hypervisor

The purpose of this section is to validate that both the VM and the hypervisor can use PML simultaneously. To this end, we realize the following experiment. We start two VMs noted VM$_1$ and VM$_2$. In VM$_1$, we launch our micro-benchmark along with a tracking system. At the same time, we launch the live migration of VM$_2$. Recall that Xen live migration uses PML when it is supported by the machine, which is our case. The results are as follows. First, we observed that the hypervisor was able to complete live migration without crashing. Second, we observed that the migration time increases by about 45% compared to when it is performed alone. This is explained by the multiplication of the number of vmexit events due to PML buffer full. Notice that the same buffer is shared among the hypervisor and VMs. By design EPML does not include this overhead as it allows the hypervisor and the guest evolve independently on their dedicated PML buffer.

VII. RELATED WORK

a) Nested virtualization: The main source of performance degradation in virtualized environments are VM traps. The latter lead to the suspension of the execution of the VM and also to cache pollution [21] due to context switch. The number of VM traps increases at least by a factor of two in nested virtualized environments [28]. The reduction of VM traps is a hot topic in both non-nested [17, 20, 21] and nested virtualized systems [22, 23, 28].

Device passthrough is a simple approach for improving I/O performance in nested and non-nested virtualized environments by providing a direct access to the VM. However, it dedicates the entire device to a single VM, resulting to sub-optimal resource utilization. In addition, device passthrough does not permit VM live migration, which is an important operation for cloud providers as it is used for maintenance. VMCS shadowing [11], that EPML leverages, has been introduced by Intel to reduce traps when a nested hypervisor uses some VMCS fields. SV$_T$, by Vilanova et al. [28], exploited simultaneous multithreading (SMT) processors to minimize VM traps. SV$_T$ runs every nested virtualization level on a separate SMT thread and it replaces VM trap and VM resume to avoid context switches between nested hypervisors and the host hypervisor (the one which directly runs atop the hardware). In SV$_T$, only one SMT thread can run at a given time leading to core waste.

DVH, by Lim et al. [23], proposed that the host hypervisor to provide virtual devices directly to nested VMs without the intervention of intermediate hypervisors. The intermediate hypervisors only interves at virtual device initialization time to make it visible and directly accessible to the nested VM. The authors illustrated DVH with four devices: virtual-IO, virtual timer, virtual IPI and virtual idle. Although DVH is promising, its application to all devices which composed a full hardware is unpractical. With OoH, we are advocating for exposing only hardware virtualization features that could help applications, which is tractable.

b) Dirty page tracking: This activity is necessary for both hypervisors and processes. The hypervisor relies on it to perform pre-copy based live migration and also checkpointing. Dirty page monitoring is at the heart of concurrent garbage collectors and other userspace processes such as CRIU for container and process checkpointing, or Redis for dumping the database. So far, the main approach used for monitoring dirty pages is two steps: invalidation of PTE dirty bit and present bit, and page-fault interception. To minimize the overhead of this approach, some alternatives have been proposed. For the hypervisor, Intel introduced PML, the hardware feature that we study in the present paper. Bitchebe et al. [16] showed that PML can decrease both VM live migration and checkpointing duration. The authors also extended PML to also log read pages in order to efficiently estimate VM working set size. In non-virtualized environments, Lu et al. [25] built a memory allocator which maps several objects to the same physical page, thus reducing the number of tracked pages. Although this solution does not avoid frequent interruption of the tracked process due to page faults.

VIII. CONCLUSION

In this paper we introduced Out of Hypervisor (OoH), a new research axis that advocates the exposition of individually current hypervisor-oriented hardware virtualization features to the guest OS so that its processes could also take benefit from those features. We illustrated OoH with Intel PML (noted OoH-PML), a feature which allows efficient dirty page tracking. We prototyped OoH-PML following two designs namely Shadow (SPM)L and Extended PML (EPML). The former requires no hardware changes but incurs significant performance overhead. It is not the case of EPML which extends the original PML to avoid SPML limitations. We evaluated and compared SPML and EPML with two popular dirty page tracking techniques namely userfaultfd and userfaultfd. We considered CRIU as the use case where the target application is tkrw, a database engine. The evaluation results showed that the different techniques can be classified as follows: SPML, userfaultfd, userfaultfd, and EPML. We are currently applying OoH to Intel SPP in order to improve secured heap memory allocators.

REFERENCES

[1] “4th generation intel core vpro processors with intel vmces shadowing,” https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-vmces-shadowing-paper.pdf, accessed: 2021-11-21.
[2] “Criu,” [https://www.criu.org/Main_Page](https://www.criu.org/Main_Page), accessed: 2021-11-21.

[3] “Docker,” [https://www.docker.com/](https://www.docker.com/), accessed: 2021-11-21.

[4] “A garbage collector for c and c++,” [https://www.hboehm.info/gc/](https://www.hboehm.info/gc/), accessed: 2021-11-21.

[5] “Kubernetes,” [https://kubernetes.io/](https://kubernetes.io/), accessed: 2021-11-21.

[6] “Nested virtualization in azure,” [https://azure.microsoft.com/en-us/blog/nested-virtualization-in-azure/](https://azure.microsoft.com/en-us/blog/nested-virtualization-in-azure/), accessed: 2021-11-21.

[7] “Openvz,” [https://openvz.org/](https://openvz.org/), accessed: 2021-11-21.

[8] “podman,” [https://podman.io/](https://podman.io/), accessed: 2021-11-21.

[9] “Tkrzw: a set of implementations of dbm,” [https://dbmx.net/tkrzw/](https://dbmx.net/tkrzw/), accessed: 2021-11-21.

[10] “userfaultfd(2) - linux manual page,” [https://man7.org/linux/man-pages/man2/userfaultfd.2.html](https://man7.org/linux/man-pages/man2/userfaultfd.2.html), accessed: 2021-11-21.

[11] “The userspace i/o howto,” [https://www.kernel.org/doc/html/v4.12/driver-api/uio-howto.html](https://www.kernel.org/doc/html/v4.12/driver-api/uio-howto.html), accessed: 2021-11-21.

[12] “Vt-d posted interrupts.” [https://events.static.linuxfound.org/sites/events/files/slides/VT-d%20Posted%20Interrupts-final%20.pdf](https://events.static.linuxfound.org/sites/events/files/slides/VT-d%20Posted%20Interrupts-final%20.pdf), accessed: 2021-11-21.

[13] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazieres, and C. Kozyrakis, “Dune: Safe user-level access to privileged CPU features,” in 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12). Hollywood, CA: USENIX Association, Oct. 2012, pp. 335–348. [Online]. Available: [https://www.usenix.org/conference/osdi12/technical-sessions/presentation/belay](https://www.usenix.org/conference/osdi12/technical-sessions/presentation/belay)

[14] G. Belpaire and N.-T. Hsu, “Hardware architecture for recursive virtual machines,” in Proceedings of the 1975 Annual Conference, ser. ACM ’75. New York, NY, USA: Association for Computing Machinery, 1975, p. 14–18. [Online]. Available: [https://doi.org/10.1145/800181.810258](https://doi.org/10.1145/800181.810258)

[15] 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 Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, ser. OSDI’10. USA: USENIX Association, 2010, p. 423–436.

[16] S. Bitchebe, D. Mvondo, L. Réveillère, N. de Palma, and A. Tchana, “Extending intel pml for hardware-assisted working set size estimation of vms,” in Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, ser. VEE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 111–124. [Online]. Available: [https://doi.org/10.1145/3453933.3454018](https://doi.org/10.1145/3453933.3454018)

[17] K. Cheng, S. Doddamani, T.-C. Chiueh, Y. Li, and K. Gopalan, “Directvisor: Virtualization for bare-metal cloud,” in Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, ser. VEE ’17. New York, NY, USA: ACM, 2017, pp. 201–213. [Online]. Available: [https://doi.org/10.1145/3050748.3050751](https://doi.org/10.1145/3050748.3050751)

[18] J. Connelly, T. Roberts, X. Gao, J. Xiao, H. Wang, and A. Stavrou, “Cloudskulk: A nested virtual machine based rootkit and its detection,” in 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2021, pp. 350–362.

[19] R. P. Goldberg, “Survey of virtual machine research,” Computer, vol. 7, no. 6, pp. 34–45, 1974.

[20] A. Gordon, N. Amit, N. Har’El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir, “Eli: Bare-metal performance for i/o virtualization,” in Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS XVII. New York, NY, USA: Association for Computing Machinery, 2012, p. 411–422. [Online]. Available: [https://doi.org/10.1145/2150976.2151020](https://doi.org/10.1145/2150976.2151020)

[21] A. Landau, M. Ben-Yehuda, and A. Gordon, “Splitx: Split guest/hypervisor execution on multi-core,” in 3rd Workshop on I/O Virtualization (WIOV 11). Portland, OR: USENIX Association, Jun. 2011. [Online]. Available: [https://www.usenix.org/conference/wiov11/splitx-split-guesthypervisor-execution-multi-core](https://www.usenix.org/conference/wiov11/splitx-split-guesthypervisor-execution-multi-core)

[22] J. T. Lim, C. Dall, S.-W. Li, J. Nieh, and M. Zygier, “Neve: Nested virtualization extensions for arm,” ser. SOSP ’17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: [https://doi.org/10.1145/3132747.3132754](https://doi.org/10.1145/3132747.3132754)

[23] J. T. Lim and J. Nieh, “Optimizing nested virtualization performance using direct virtual hardware,” in Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 557–574. [Online]. Available: [https://doi.org/10.1145/3373376.3378467](https://doi.org/10.1145/3373376.3378467)

[24] B. Liu, P. Olivier, and B. Ravindran, “Slimguard: A secure and memory-efficient heap allocator,” in Proceedings of the 20th International Middleware Conference, ser. Middleware ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1–13. [Online]. Available: [https://doi.org/10.1145/3361525.3361532](https://doi.org/10.1145/3361525.3361532)

[25] K. Lu, W. Zhang, X. Wang, M. Luján, and A. Nisbet, “Flexible page-level memory access monitoring based on virtualization hardware,” in Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, ser. VEE ’17. New York, NY, USA: ACM, 2017, pp. 201–213. [Online]. Available: [http://doi.acm.org/10.1145/3050748.3050751](http://doi.acm.org/10.1145/3050748.3050751)

[26] S. Silvestro, H. Liu, C. Crosser, Z. Lin, and T. Liu, “Freeguard: A faster secure heap allocator,” in
[27] S. Silvestro, H. Liu, T. Liu, Z. Lin, and T. Liu, “Guarder: A tunable secure allocator,” in Proceedings of the 27th USENIX Conference on Security Symposium, ser. SEC’18. USA: USENIX Association, 2018, p. 117–133.

[28] L. Vilanova, N. Amit, and Y. Etsion, “Using smt to accelerate nested virtualization,” in Proceedings of the 46th International Symposium on Computer Architecture, ser. ISCA ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 750–761. [Online]. Available: https://doi.org/10.1145/3307650.3322261

[29] F. Zhang, J. Chen, H. Chen, and B. Zang, “Cloudvisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization,” in Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, ser. SOSP ’11. New York, NY, USA: Association for Computing Machinery, 2011, p. 203–216. [Online]. Available: https://doi.org/10.1145/2043556.2043576