Out of Hypervisor (OoH): Efficient Dirty Page Tracking in Userspace Using Hardware Virtualization Features

Stella Bitchebe
Univ. Côte d'Azur & ENS Lyon
Nice & Lyon, France
bitchebe@i3s.unice.fr

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

***Abstract*—This paper introduces Out of Hypervisor (OoH), a new virtualization research axis. Instead of emulating full virtual hardware inside a VM to support a hypervisor, the OoH principle is to individually expose current hypervisor-oriented hardware virtualization features to the guest OS. This way, guest's processes could also take benefit from those features. We illustrate OoH with Intel PML (Page Modification Logging), a feature that allows efficient dirty page tracking to improve VM live migration. Because dirty page tracking is at the heart of many essential tasks including process checkpointing (e.g., CRIU) and concurrent garbage collection (e.g, Boehm GC), OoH exposes PML to accelerate these tasks in the guest. We present two OoH solutions namely Shadow PML (SPML) and Extended PML (EPML) that we integrated into CRIU and Boehm GC. Evaluation results showed that EPML speeds up CRIU checkpointing by about 13× and Boehm garbage collection by up to 6× compared to SPML, `/proc`, and userfaultfd while reducing their overhead on monitored applications by about 16×.**

***Index Terms*—virtualization, page modification logging, checkpointing, garbage collection**

I. Introduction

The attractive costs and the efficiency of Cloud computing have led to its massive expansion, including to the HPC domain. In the cloud, applications run inside virtual machines (VM). In this context, dirty page tracking is an important need for both the guest kernel and its userspace processes. The former tracks dirty pages to know if a file-backed memory page should be copied to disk when swapped out. In userspace, dirty page tracking is used by several applications such as garbage collectors (GC) [**?**], container or process checkpoint/restore systems [**? ?**], use-after-free vulnerability mitigation systems [**?**], etc. For illustration, dirty page tracking is used by concurrent GCs like Boehm [**?**] to reduce application pause time during the construction of reachable objects. Existing dirty page tracking solutions rely on *userfaultfd* (hereafter *ufd*) or */proc/<PID>/pagemap* (hereafter */proc*), two interfaces offered by Linux. As assessed in Section III, these two interfaces are extremely expensive since they are based on page write protection, which induces a lot of page faults and world transitioning (kernel space to userspace and vice versa). We measured an overhead of up to $15\times$ with *ufd* and $4\times$ with */proc*.

Intel Page Modification Logging (PML) [**?**] is a **hardware virtualization feature** originally introduced to track dirty pages *at the scale of a whole VM by the hypervisor* to accelerate VM live migration [**?**]. In this paper, we study how PML can be diverted to be used by the guest kernel and its applications to accelerate their execution. We identify three main challenges to achieving that. (C_1) PML can be exploited uniquely by the hypervisor. In other words, only software (the hypervisor) running in *VMX root ring 0* CPU mode can use PML. Yet, the guest kernel and its applications run in CPU *VMX non-root ring* ≥ 0. (C_2) PML operates in coarse-grained that is, it concerns the entire VM. We want to use PML at the granularity of a process within the VM while allowing the hypervisor to continue using it at the scale of the VM. (C_3) PML only logs guest physical addresses (GPAs) while userspace processes need guest virtual addresses (GVAs).

This paper presents Out Of Hypervisor (OoH), a principle that allows the utilization of hardaware virtualization features from inside the VM, while preserving the underlaying security of hypervisor and VMs. The paper particularly focuses on Intel PML[1]. We describe two solutions of OoH for PML and present two systems that can use them. These systems are Boehm (a GC) and CRIU (a process checkpoint/restore system). The first OoH solution called Shadow PML (noted SPML) requires no hardware modification. In contrast, the second OoH solution called Extended PML (noted EPML) slightly extends the hardware implementation of PML to avoid the limitations of SPML. We investigated SPML to point out the need for a hardware extension of PML. SPML relies on hypercalls (VM $\rightarrow$ hypervisor) and virtual interrupts (hypervisor $\rightarrow$ VM) to respectively enable/disable PML and to periodically copy GPAs (logged by the CPU to a PML buffer in the hypervisor) to a ring buffer shared between the hypervisor and the guest OS. It is then up to the guest OS to perform GPA to GVA reverse mapping. EPML on its side hijacks two hardware virtualization features (VMCS

[1]Future work will present OoH for Intel SPP.

SC22, November 13-18, 2022, Dallas, Texas, USA
978-1-6654-5444-5/22/$31.00

shadowing [**?**] and posted-interrupt [**?**]), and slightly extends PML to avoid the intervention of the hypervisor on the critical path. EPML is able to log the address of a dirty page into two buffers simultaneously. The first buffer is exclusively managed by the guest OS while the second buffer is managed by the hypervisor. Finally, EPML logs GVA to the guest level buffer and logs GPA to the hypervisor level one.

To facilitate the utilization of OoH, we provide a generic library that we implemented following the UIO driver principle [**?**]. The library has two parts: a kernel module and a userspace template code. The former does not need to be managed by the application developers, who simply needs to integrate the template code into their applications. We prototyped and evaluated SPML on a DELL Intel Core i7-8565U processor machine that supports PML, using the Xen hypervisor (a popular hypervisor used by Amazon). Regarding EPML, we rely on BOCHS, the only emulator (to the best of our knowledge) that implements PML. In both designs, the guest OS is Linux. We use both micro- and macro-benchmarks for evaluations. For the latter, we use all the five in-memory database engines from tkrzw [**?**] and six applications from Phoenix [**?**] (a MapReduce application set). To evaluate our prototypes, we set up a rigorous methodology, especially for EPML which extends the hardware. To this end, we build a mathematical formula that accurately approximates the overhead or improvement of each solution. We systematically compare SPML and EPML with `/proc` and `ufd`. We are interested in the impact of each solution on the tracked application (e.g., Phoenix) and the tracking system (e.g., CRIU). The solutions are classified as follows, in decreasing order of the overhead they induce: SPML, `ufd`, `/proc`, and EPML. `/proc`, which is the default solution implemented in both CRIU and Boehm, incurs an overhead of up to 102% with CRIU on the Phoenix `pca` application, and up to 232% with Boehm on the Phoenix `string-match` application. The overhead of SPML is up to 114% with CRIU and 273% with Boehm on the same applications. EPML leads to the lowest overhead, which is about 7% with CRIU on `pca` and 24% with Boehm on `string-match`. Hence, EPML can improve existing systems by up to 62%. Concerning the tracking system and compared to `/proc`, SPML induces up to 5× slowdown on CRIU and 3× slowdown on Boehm GC. EPML brings up to 4× speedup compared to `/proc` and 13× speedup compared to SPML for CRIU. Finally for Boehm, EPML brings up to 2× speedup compared to `/proc` and up to 6× speedup compared to SPML.

In summary, we make the following contributions:

- (Empirical contribution) We finely quantify the impact of page fault-based dirty page tracking.
- (Conceptual contribution) We present OoH and two solutions namely SPML and EPML.
- (Technical contribution) We prototype SPML and EPML in real and emulated environments (BOCHS) using popular system software (Xen hypervisor and Linux guest OS).
- (Technical contribution) We integrate PML into two popular tracker systems namely CRIU and Boehm GC.
- (Empirical contribution) We rigourously evaluate our designs using micro- and macro-benchmarks.

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, and discusses potential security issue consideration. Section VI presents the evaluation results. Section VII presents the state of the art. Section VIII concludes the paper.

II. Hardware assisted virtualization

This section provides 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 mode cpl 0` 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 with 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 into two types: ordinary VMCS and shadow VMCS. A shadow VMCS is a VMCS that is pointed to 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 of guests. It relies on Extended Page Table (EPT) and requires specific changes to VMCS. When PML is

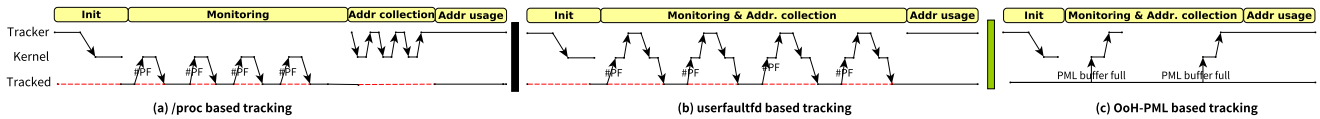

Figure 1: Impact of `/proc`, `ufd` and OoH-PML methods on Tracked and Tracker. The two former methods lead to several suspensions (red dashed lines) of Tracked, due to #PF -Page Faults- and context switches. `ufd` induces the longest suspension time (#PF are handled in userspace). However, dirty page address collection takes much more time with `/proc` (due to the parsing of `/proc/PID/pagemap`), thus impacting Tracker. OoH has the benefits of both world and does not require the suspension of Tracked.

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 GPAs. 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 over. The handler of that vmexit 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 running inside a VM may also need to monitor dirty pages for garbage collection or checkpointing. We call Tracker the monitoring thread and Tracked 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 `ufd` and `/proc`. Fig. 1 summarizes their functioning compared to a OoH-PML-based solution. `ufd` 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 checkpointing).

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 suspendend during the initialization phase. The ideal execution time of Tracked is when it runs without beeing tracked. The ideal execution time of Tracker is the ideal execution time of Tracked. As one can deduce from Fig. 1, 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.

A. The cost of `ufd`

Fig. 1.b summarizes the functioning of a `ufd`-based dirty page monitoring solution. To use `ufd`, Tracker first registers the memory region it wants to monitor. After the registration, it will be notified by the kernel each time a page fault concerning the registered region occurs. `ufd` 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, Tracked is suspended until the fault is resolved. In the case of `write_protect`, the Tracker should write-unprotect the faulted page in order to unpause Tracked. One can see that, with `ufd`, the collection of dirty page addresses can be done during the monitoring phase.

We assess the overhead of `ufd` using as Tracked a synthetic program (presented in Section VI) that just parses and writes to an array of buffers. The size of each buffer is 4KB, allocated at page boundaries. We are interested in monitoring the entire array. Table I second and fifth rows show the overhead of `ufd` 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 15× and 14× for 1GB on Tracked and Tracker respectively. We break down 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 3,3383ms for 1GB). The total suspension time of Tracked represents on average about 93% of its execution time.

On Tracked	1MB	10MB	50MB	100MB	250MB	500MB	1GB
`ufd`	195	272	583	1,050	1,266	1,462	1,463
`/proc`	104	55	114	208	302	307	335

On Tracker	1MB	10MB	50MB	100MB	250MB	500MB	1GB
`ufd`	93	169	477	940	1,269	1,153	1,349
`/proc`	47	43	58	148	151	143	147

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

B. The cost of `/proc`

Fig. 1.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 Tracked'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 ufd. After this, once Tracked tries to modify a monitored page, a fault occurs. The handler of that fault sets in Tracked's PTE the soft-dirty bit of the faulted page (this operation costs about $33.5\mu s$). At the end of the monitoring period, Tracker reads the soft-dirty bits (bit 55) from `/proc/PID/pagemap` 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 more than $4\times$ for 1GB of memory. This overhead is lower than the one induced by `/ufd` as shown above. Concerning Tracker, see Table I bottom, the overhead is up to $2\times$ (147%). When compared to `/ufd`, although `/proc` increases the address collection phase, its cost is compensated by the smaller suspension it induces during the monitoring phase.

C. Alternative

A way to use PML for a process is to dedicate a VM to the latter, thus exploiting PML as is only by the hypervisor. This approach works for some use cases such as checkpointing. In fact, to checkpoint the process the user would checkpoint the corresponding VM.

However, this approach would not be effective for use cases where PML is required at runtime. It is the case for garbage collectors that we study in this paper. The GC runs inside the guest, so it needs to access PML from there. It is not possible, at the hypervisor level, to perform garbage collection for processes running inside a guest (which is a black box for the hypervisor).

Another important drawback of this approach is that it is contrary to the current trends of colocating tasks within VMs to save resources (thus money) and reduce Inter-Process Communication costs (to which HPC applications are very sensitive). This practice is common in FaaS platforms, where functions of the same client are co-located within the same VM [**?**]. Besides, several HPC FaaS platforms are initiatives developing [**? ?**].

D. Generalization

We can organize hardware virtualization features in two groups. (G_1) Features that facilitate resource multiplexing, such as EPT (Extended Page Table), SRIOV (Single Root I/O Virtualization), APICv (Intel's Advanced Programmable Interrupt Controller virtualization), etc. These features are not relevant for exposure to the guest OS in the spirit of OoH. (G_2) Features that are intended to facilitate and improve VM management tasks realized by the hypervisor, such as Intel PML, CAT (Cache Allocation Technology), SPP (Sub Page write Permission), PT (Processor Tracing), and so on. Considering a VM as a process, it makes sense to expose features of this G_2 category to the guest, thus allowing it to improve its management tasks. Although this paper focuses on illustrating OoH with Intel PML, notice that we are also studying the application of OoH principles to Intel SPP for improving secure heap memory allocators that run from inside VMs. In fact, to mitigate buffer overflows, several heap memory allocators [**? ? ?**] use memory guard pages as they allow synchronous overflow detection. Nevertheless, guard pages lead to significant memory waste. By relying on Intel SPP, we intend to reduce that overhead by a factor of 32 according to the number of sub-pages allowed by Intel SPP within a memory page.

IV. Out of Hypervisor

The goal of OoH is to make some hardware virtualization features usable from inside the guest OS. We do this with as minimum changes as possible in the hardware, the hypervisor, and the guest kernel. In addition, we want to propose a simple utilization interface to application developers. §IV-A presents the methodology that OoH designers can follow when applying this principle to a hardware virtualization feature. Then §IV-B-IV-E illustrate that methodology to the exposure of Intel PML.

A. Methodology

OoH argues for software and hardware approaches. Obviously, the latter should be envisioned only when the former is not efficient. When followed, the latter should try to re-use as maximum as possible existing functionalities before thinking of hardware changes. To expose a given hardware virtualization feature, the OoH principle is as follows. To facilitate the exploitation of the exposed feature, OoH designers should provide userspace applications with a library. The latter should rely on a guest kernel module that preserves the privilege of the kernel on multiplexing the exposed feature. OoH designers use hypercalls and event channels as communication mechanisms between the hypervisor and the guest. Hypercalls allow the guest (OoH kernel module) to instruct the hypervisor (feature initialization for instance) while event channels allow the hypervisor to send a specific signal to the guest. VMCS shadowing, a feature that has been invented for improving nested virtualization, can be leveraged to implement OoH. It allows to reduce the number of interventions of the hypervisor, thus improving performance. Finally, and only when unavoidably necessary, some hardware changes can be made such as ISA extension or VMCS data structure modifications.

B. Overview

We present two solutions of OoH for PML, namely Shadow PML (noted SPML) and Extended PML (noted EPML). SPML requires no hardware modification, while EPML slightly extends the hardware for better performance. Fig. 2 presents the architecture of the two solutions (detailed in §IV-C and §IV-D). In the guest, we provide OoH as a userspace I/O

(UIO) driver composed of a kernel module (OoH Module) and a userspace library (OoH Lib). At load time, the former does a set of initialization operations, including ring buffer allocation that is shared with userspace (and the hypervisor in SMPL only). Tracker uses OoH Lib to register the PID of Tracked with OoH Module. From there on, the processor can log dirty pages' addresses to a 512KB PML buffer, which is copied to the ring buffer once full. Relying on OoH Lib, Tracker can periodically fetch the collected addresses to achieve its goal (e.g., checkpointing).

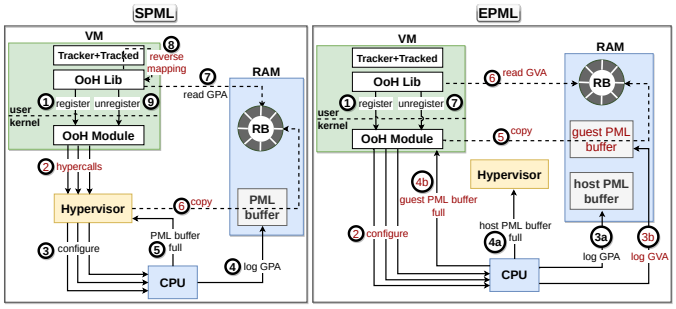

Figure 2: OoH: SPML and EPML architectures. Sofware/Hardware changes are highlighted in red.

EPML differs from SPML in two ways: (1) With EPML, the processor also logs GVAs, thus avoiding costly reverse mapping in OoH Lib; (2) With EPML, the guest kernel can directly deal with the processor, thus avoiding costly hypercalls.

C. Shadow PML (SPML)

The basic idea behind SPML is to make the hypervisor emulate PML for guest's processes. Indeed, in this solution, the hypervisor is the only component able to perform PML's instructions to the processor. Fig. 2 left summarizes SPML functioning, with three main features.

(1) To ensure that the processor only logs GPAs for tracked processes, we introduce two new hypercalls `disable_logging` and `enable_logging`. The former is called by OoH Module every time a tracked process is scheduled-out. This hypercall copies the content of the PML buffer to the ring buffer and instructs the processor to stop logging. `enable_logging` is the inverse operation, invoked by OoH Module when a tracked process is scheduled-in.

(2) In SPML, the processor logs GPAs while Tracker needs GVAs. To fill this gap, OoH Lib *reverse maps* GPA to GVA by parsing the page table of Tracked using the `/proc` interface.

(3) Recall that the hypervisor can also use PML for its own purposes (e.g., VM live migration). To coordinate the two levels, we introduce two flags (`enabled_by_guest` and `enable_by_hyp`) that indicate which level has enabled PML. When the PML buffer is full, the hypervisor does not fill the ring buffer unless `enabled_by_guest` is set. And similarly, if `enabled_by_hyp` is not set, the hypervisor bypasses the operations corresponding to its use of PML, thus avoiding unnecessary additional CPU time consumption. If the hypervisor wants to deactivate PML, it first checks that `enabled_by_guest` is not set and vice versa.

The main limitation of SPML lies in its performance overhead caused by the high number of hypercalls and reverse mapping operations that it generates, justifying EPML.

D. Extended PML (EPML)

Fig. 2 right summarizes EPML. The basic idea behind it is to provide a second level of PML directly controlled by the guest OS (OoH Module). Every tracked process is associated with a guest-level PML buffer by OoH Module, exactly as the hypervisor manages a PML buffer per vCPU.

To minimize hardware changes, EPML leverages the existing VMCS shadowing (see Section II) feature which allows a guest to perform `vmread` and `vmwrite` instructions without vmexit to the hypervisor. At load time, OoH Module calls the hypervisor to enable and configure VMCS shadowing. This is the only hypercall performed in EPML. Therefore, when a tracked process is scheduled-in or out, OoH Module accordingly enables or disables address logging using `vmwrite` instruction. Contrary to SPML, EPML does not interfere with the hypervisor's needs.

From a hardware point of view, EPML makes the following small changes. We introduce in the VMCS a new field (called `Guest PML Address`) that represents the address of the guest-level PML buffer. Because the guest (OoH Module here) only sees GPA, the value that it sets to the Guest PML Address should be translated to a host physical address (HPA) so that the processor can log to the right location in RAM. To tackle this challenge, we extend the VMX ISA so that if a `vmwrite` instruction to the `Guest PML Address` field is performed when the processor is in guest mode, it first translates the address to a HPA (using EPT) before writing it to the shadow VMCS.

Another improvement brought by EPML is the capability to log GVAs, thus avoiding reverse mapping by OoH Lib. We modify the page walk circuit to make the processor log the GVA to the guest-level PML buffer and the GPA to the hypervisor-level PML buffer.

The last hardware extension is to handle guest-level PML buffer full events. We modify the hardware so that when the guest-level PML buffer is full, the processor raises a virtual *self-IPI (Inter-Processor Interrupt)* which is handled in the guest by OoH Module. Notice that, this is a very small modification that leverages an existing feature called Posted-Interrupts. With the latter, the processor is able to directly deliver interrupts to the guest OS when running in the guest mode without vmexits.

E. Implementation

We implemented OoH in Xen 4.10.0 hypervisor, Linux 4.15.0 guest OS, and BOCHS 2.6.11 Intel x86 emulator (only for EPML, which extends the hardware). We use CRIU and Boehm GC as Trackers.

Xen: It is a popular hypervisor used by Amazon EC2. The main changes we made concern the introduction of the two hypercalls `disable_logging` and `enable_logging`, and the copy of the content of the PML buffer to RB.

Linux Core: We essentially modified the interrupt table to handle the virtual *self-IPI* raised by the processor when the guest-level PML buffer is full in EPML. Notice that OoH Module is a kernel module, thus it is not part of the Linux core.

BOCHS: It is a very popular Intel machine emulator, which yet implements PML. We modified it to implement EPML, mainly by extending: the `vmwrite` instruction, the page table walk circuit to log GVAs, and the PML logging process to raise *self-IPI* on buffer full.

CRIU: It is a popular checkpoint/restore tool integrated into many well-known projects such as OpenVZ [**?**], Podman [**?**], or Docker [**?**]. CRIU relies on `/proc`. To integrate OoH with it, we mainly patched two steps: (1) Initialization: OoH avoids pausing (`echo 4 > /proc/PID/clear_refs`) Tracked at the initialization phase because the activation of PML is immediate and does not interfere with the execution of Tracked, as illustrated in Fig. 1.c. (2) Address collection: OoH avoids parsing the `/proc/<PID>/pagemap` file to retrieve dirty pages' addresses. The ring buffer is read instead.

Boehm GC: Boehm GC [**?**] is a popular C and C++ garbage collector that is included in many well-known projects such as Mozilla [**?**], GNU Java compiler [**?**], or Inkscape [**?**]. Boehm GC provides incremental and generational collection based on dirty page tracking. In its current implementation, Boehm relies on `/proc`. To integrate OoH, we mainly patched the *mark phase*, corresponding to where the GC checks for modified pages. As with CRIU, OoH avoids reading from `/proc/PID/pagemap` and resetting dirty bits via `/proc/PID/clear_refs`.

Table **??** summarizes for each system, the number of LOC and modified files that the implementation required.

System		Xen	Linux	Bochs	CRIU	Boehm
#LOC	SPML	182	6	N/A	251	254
	EPML	120	14	44	140	144
#files	SPML	13	2	N/A	9	4
	EPML	9	9	6	9	4

Table II: Number of LOC and modified files in OoH implementation.

V. Security and Isolation

OoH empowers the guest processes by sharing some hypervisor-oriented hardware virtualization features (PML in this paper) with guest OSes. One may legitimately see this as a source of potential threats. In this section, we show that neither SPML nor EPML increases the vulnerability of the hypervisor (against guest OSes), the guest OS (against others), and processes (against others within the same guest VM). Our trust model is the same as that of `/proc` and `ufd` that is, the hypervisor does not trust guest OSes, which in turn do not trust their processes. We elaborate below on the security of the hypervisor, VMs, and processes.

Concerning the hypervisor, (1) only SPML requires its modification, which represents only 194 LOC. The latter is negligible compared to the hypervisor code size (about 900K+ LOC for Xen). Accordingly, we are pretty confident that our added code is at least as safe as existing hypercalls. (2) In both SPML and EPML, the guest OS does not see hardware physical addresses (HPA). Remember that SPML logs GPA and EPML logs GVA, both being virtual addresses. These address types are traditionally seen and managed by the guest OS, including when `/proc` and `ufd` are used. In OoH, the hypervisor remains the sole layer that sees and manages HPA. (3) The ring buffer that the hypervisor shares with each guest OS is allocated within the guest's address space and not that of the hypervisor. Thus, a guest can not leverage it to corrupt the hypervisor.

Concerning guest OSes, neither SPML nor EPML reduces their isolation level. In SPML, a dedicated ring buffer is used per guest. Therefore, a guest can only see logged addresses that belong to its address space.

In the previous version of our prototype, all tracked processes of the same guest could see the same SPML/EPML logging buffer. That implementation could potentially lead to side-channel attacks as a tracked process could learn the memory access pattern of tenant tracked processes. (Thanks to reviewer feedback, we have (easily) updated that implementation to dedicate a per-process ring buffer and restrict its access to tracker processes only. This was an implementation detail.)

VI. Evaluations

This section presents the evaluation results. We want to answer the following questions: (1) what is the potential overhead or improvement of SPML and EPML compared to existing solutions (`/proc` and `ufd`)? (2) what is the scalability of SPML and EPML? (3) to what extent SPML and EPML are able to efficiently capture all dirty pages?

A. Experimental environment

a) Machines and Systems: We carried out the experiments on a machine with 8 Intel Core i7-8565U and 16 GB of RAM. Especially to evaluate `ufd` we use Linux 5.11 because version 4.15.0 does not support the `write_protect` mode. To emulate EPML we used BOCHS. In the emulated environment, the VM has 1 vCPU and 1GB of memory (due to BOCHS constraints). In all experiments on the real machine, the VM has 1 dedicated CPU (to meet the emulated setup) with 5GB of memory.

b) Benchmarks: We used both micro- and macro-benchmarks. The former is composed of two applications: an array parser (shown in Listing 1) and GCBench [**?**], a popular micro-benchmark to evaluate GCs. For macro-benchmarks, we used two benchmark suites: tkrzw [**?**] a suite of key value data processing engines, and Phoenix [**?**] a

Application	Config. small ⋆⋆ Memory Cons.	Config. medium ⋆⋆ Memory Cons.	Config. large ⋆⋆ Memory Cons.
Phoenix & GCBench			
GCbench	500K 16 18 ⋆⋆ 15.07MB	650K 18 20 ⋆⋆ 67.76MB	750K 20 22 ⋆⋆ 223.41MB
histogram	0.1GB-datafile ⋆⋆ 102.27MB	0.5GB-datafile ⋆⋆ 441.28MB	1.5GB-datafile ⋆⋆ 1.49GB
kmeans	-d 500 -c 500 -p 500 -p 100 ⋆⋆ 4.26MB	-d 1K -c 1K -p 1K -s 100 ⋆⋆ 16.41MB	-d 5K -c 5K -p 5K -s 100 ⋆⋆ 195.64MB
matrix-multiply	500 500 ⋆⋆ 5.56MB	1K 1K ⋆⋆ 16.21MB	2K 2K ⋆⋆ 47.33MB
pca	-r 1K -c 1K -s 200 ⋆⋆ 8.12MB	-r 5K -c 5K -s 200 ⋆⋆ 97.85MB	-r 10K -c 10K -s 200 ⋆⋆ 195.50MB
string-match	50MB-datafile ⋆⋆ 56.40MB	100MB-datafile ⋆⋆ 106.14MB	200MB-datafile ⋆⋆ 212.09MB
word-count	50MB-datafile ⋆⋆ 100.65MB	100MB-datafile ⋆⋆ 143.99	200MB-datafile ⋆⋆ 205.88MB
tkrzw			
baby	–iter 3M –threads 3 ⋆⋆ 253.64MB	–iter 5M –threads 3 ⋆⋆ 421.48MB	–iter 10M –threads 3 ⋆⋆ 848.56MB
cache	–iter 3M –cap_rec_num 3M –threads 5 ⋆⋆ 218.21MB	–iter 5M –cap_rec_num 5M –threads 5 ⋆⋆ 361.91MB	–iter 10M –cap_rec_num 10M –threads 5 ⋆⋆ 721.46MB
stdhash	–iter 3M –buckets 100K –record_comp zlib –threads 2 ⋆⋆ 358.64MB	–iter 5M –buckets 100K –record_comp zlib –threads 2 ⋆⋆ 595.80MB	–iter 10M –buckets 100K –record_comp zlib –threads 2 ⋆⋆ 1.18GB
stdtree	–iter 3M –threads 2 ⋆⋆ 415.12MB	–iter 5M –threads 2 ⋆⋆ 694.07MB	–iter 10M –threads 2 ⋆⋆ 1.38GB
tiny	–iter 5M –buckets 30M –threads 3 ⋆⋆ 681.35MB	–iter 5M –buckets 30M –threads 5 ⋆⋆ 977.66MB	–iter 5M –buckets 30M –threads 7 ⋆⋆ 1.27GB

Table III: Configuration setup and memory consumption for each tkrzw (bottom), Phoenix and GCbench (top) applications. For `GCBench`, the parameters are the array size, the lived tree depth and, the stretch tree. `histogram`, `string-match`, and `word-count` use datafile as input parameter.

shared-memory implementation of Google's MapReduce data processing model. Concerning tkrzw, we focused on the five in-memory engines and we injected `set` requests. For each of the macro-benchmarks, we consider three configuration sizes namely *Small*, *Medium*, and *Large*. Table III presents the per-application memory size for each configuration. All results presented in this section are mean of 5 runs.

```
1  ...
2  #define PAGE_SIZE sysconf(_SC_PAGE_SIZE)
3  #define num_pg xx //memory size=xx*PAGE_SIZE
4
5  void main(void)
6  {
7   unsigned long *region=malloc(num_pg*PAGE_SIZE);
8   /*
9    * Pin all the pages in-memory to be sure that
10   * they are not swapped out
11   */
12  mlockall(MCL_CURRENT|MCL_FUTURE|MCL_ONFAULT);
13  for( ; ; )
14   for(unsigned long i=0;i<num_pg;i++)
15    region[(i*PAGE_SIZE)/sizeof(unsigned long)]=i;
16 }
```

Listing 1: Micro-benchmark code.

B. Methodology

Since we do not have a real machine equipped with EPML, we build a formula to estimate its impact compared to other techniques. We first present a generic formula that captures the functioning of all techniques. Then we specialize the formula for each technique. Finally, we demonstrate the accuracy of each formula using metric values collected during real experiments. For ease of understanding, we consider in the rest of this section that Tracked is a single-threaded application. Let us recall that Tracker executes in the same thread with Tracked, so each time Tracker runs, it disrupts the execution flow of Tracked.

Let x be a tracking technique. x is either `/proc`, ufd, SPML, EPML or *oracle*. The latter represents a hypothetical technique able to provide all dirty pages with no additional cost. Tracker's code (noted C_{tker}) can be organized into two parts: the tracking technique (noted C_x) and the tracking routine (noted C_p). The latter is the part of Tracker that implements the tracking goal, e.g., writing to disk during checkpointing. In Tracker's execution flow, C_x and C_p alternate. We note C_{tked} the original code of Tracked (i.e., without being monitored by any Tracker). We are interested in the overhead of the tracking technique x on the execution time of Tracker and Tracked.

a) Overhead of x on Tracker: The execution time of Tracker when it implements x can be computed using Formula 1:

$$E(C_{tker}) = E(C_x) + E(C_p) + I(C_x, C_p) \quad (1)$$

where $E(C)$ is the execution time of code C (with $E(C_{oracle}) = 0$) and $I(C1, C2)$ is the impact of C_1 on C_2. This impact mainly consists of cache pollution. We experimentally observed that $I(C_x, C_p)$ is negligible. Therefore, the overhead of x on Tracker is reduced to $E(C_x)$. Formula 1 can be developed for each technique as follows:

$$\begin{aligned}
E(C_{/proc}) &= E(C_{echo\ 4 > /proc/PID/clear_refs}) \\
&\quad + E(C_{page\ table\ walk\ in\ userspace}) \\
E(C_{UFD}) &= E(C_{ioctl\ write_protect}) \\
&\quad + E(C_{ioctl\ register}) \\
&\quad + E(C_{ioctl\ write_unprotect}) \\
E(C_{SPML}) &= E(C_{ring\ buffer\ copy}) \\
&\quad + E(C_{reverse\ mapping}) \\
&\quad + E(C_{enable/disable\ PML}) \\
E(C_{EPML}) &= E(C_{ring\ buffer\ copy}) \\
&\quad + E(C_{enable/disable\ PML})
\end{aligned} \quad (2)$$

Table Va and Table Vb present the measured costs for all events involved in Formula 2.

b) Overhead of x on Tracked: The execution time of Tracked when it is monitored by a tracker using the technique x can be expressed by Formula 3:

$$E(C_{tked\_tker}) = E(C_{tked}) + E(C_{tker}) + I(C_x, C_{tked}) \quad (3)$$

where $I(C_x, C_{tked})$ consists of page faults, vmexits, etc., which are not negligible. Thus, the overhead of x on Tracked is $E(C_{tker}) + I(C_x, C_{tked})$. Formula 4 develops $I(C_x, C_{tked})$

for each technique:

$$
\begin{aligned}
I(C_{/proc}, C_{tked}) &= E(C_{PFH\ in\ kernelspace}) \\
&\quad + E(C_{context\ switch}) \\
I(C_{UFD}, C_{tked}) &= E(C_{PFH\ in\ userspace}) \\
&\quad + E(C_{context\ switch}) \\
I(C_{SPML}, C_{tked}) &= E(C_{vmexits}) \\
&\quad + N \times E(C_{vmread/vmwrite}) \\
I(C_{EPML}, C_{tked}) &= N \times E(C_{vmread/vmwrite})
\end{aligned}
\tag{4}
$$

where N is the number of context switches during PML execution, and PFH stands for *Page Fault Handling*.

*c) **Validation of the formulas**:* To validate our formulas, we executed Tracker and Tracked for each technique and we collected the following metrics: the execution time, and the number of occurrences of each event related to the tracking technique. Using these values, we compute $E(C_{tker})$ and $E(C_{tked\_tker})$. Then, we compare the obtained results with real measurements, except for EPML. Notice that by validating `/proc`, ufd, and SPML formulas, we are also validating by construction the formula for EPML. For illustration, Table IVa and Table IVb present results respectively for SPML and `/proc` when Tracker is CRIU and Tracked is `baby` (from tkrzw benchmark suite). We can see that Formula 2 and Formula 4 estimate $E(C_{tker})$ and $E(C_{tked\_tker})$ with an average accuracy of 96.34% and 99% respectively. We can then consider that the formula that estimates the impact of EPML is relevant.

Metric	Time (ms)
$E(C_{tker})$ **measured**	**5503.79**
$E(C_{tked\_tker})$ **measured**	**135255.35**
$E(C_p)$	251.35
$E(C_{copy\_rb})$	0.49
$E(C_{disable\ pml})$	2.06
$E(C_{rev.\ mapping})$	5419
$E(C_{tker})$ **estimated**	**5672.9**
$E(C_{vmexits})$	18000
N	39
$E(C_{vmread,vmwrite})$	1.73×10^{-3}
$E(C_{tked\_tker})$ **estimated**	**136919.85**

(a) SPML

Metric	Time (ms)
$E(C_{tker})$ **measured**	**1097.99**
$E(C_{tked\_tker})$ **measured**	**115283.98**
$E(C_p)$	251.35
$E(C_{clear\_refs})$	1.409
$E(C_{PTwalk})$	0.89
$E(C_{tker})$ **estimated**	**1116.09**
$E(C_{PFHuser})$	0.27
$E(C_{tked\_tker})$ **estimated**	**114418.58**

(b) `/proc`

Table IV: Metrics collected to estimate $E(C_{tker})$ and $E(C_{tked\_tker})$ for techniques SPML and `/proc` using Formulas 1, 2, 3, and 4.

About the estimation of $I(C_{EPML}, C_{tked})$, we rely on SPML to obtain N, the number of context switches required to compute $I(C_{EPML}, C_{tked})$. Indeed, by construction, this number is the same in both SPML and EPML. We validated this by repeating the previous experiments with EPML and SPML techniques in the emulated environment provided by BOCHS. We collected N with a percentage difference of 2%.

C. Basic costs

We present in this section the cost of all internal metrics that allow us to understand the higher-level performance metrics. The values of these metrics also tell us about the scalability of each tracking technique. The first column of Table Va lists the metrics, organized into nine categories. For each metric, we indicate whether or not its value depends on Tracked memory size (second column of Table Va). For metrics that are agnostic to the size of Tracked memory, their basic costs are presented in the third column of Table Va. For the other metrics we report their basic costs in Table Vb while varying the Tracked memory size. We also indicate in column four of Table Va which tracking methods the metric is associated with.

Metric	**Depend on mem.**	**Cost (μs)**	**Technique**
$M1$. context switch (from user to kernel space)	No	0.315	All
ioctl syscalls			
$M2$. write_protect	Yes	-	ufd
$M3$. init. PML	No	5,651	SPML & EPML
$M4$. deactivate PML	No	2,816	SPML & EPML
page fault handling			
$M5$. in kernel space	Yes	-	/proc, ufd
$M6$. in userspace	Yes	-	ufd
vmx operations			
$M7$. vmread	No	0.936	EPML
$M8$. vmwrite	No	0.801	EPML
hypercalls			
$M9$. init. PML	No	5,495	SPML
M_{10}. + init. VMCS shadowing	No	5,878	EPML
M_{11}. PML deact.	No	2,060	SPML
M_{12}. + VMCS shadowing deact.	No	2,755	EPML
M_{13}. enable PML logging	No	0.3	SPML
M_{14}. disable PML logging	Yes	-	SPML
M_{15}. `clear_refs`	Yes	-	/proc
M_{16}. page table walk (in userspace)	Yes	-	/proc & SPML
M_{17}. reverse mapping	Yes	-	SPML
M_{18}. ring buffer copy	Yes	-	EPML & SPML

(a) Metrics that are agnostic to Tracked memory size. `clear_refs` is the shortcut for `echo 4 > /proc/PID/clear_refs`.

	1MB	10MB	50MB	100MB	250MB	500MB	1GB
M_{15}	0.032	0.0912	0.174	0.288	0.613	1.153	2.234
M_{16}	1.912	14.479	41.832	82.289	161.973	307.109	594.187
M_5	0.003	0.3	1.68	3.34	8.39	16.79	33.58
M_6	2.5	27.3	152.3	347.1	882.8	1,585	3,483
M_{14}	0.042	0.047	0.138	0.156	0.189	0.203	0.208
M_{18}	0.003	0.01	0.03	0.048	0.109	0.383	0.671
M_{17}	6.183	24.653	85.117	255.437	1,211	4,123	15,738

(b) Metrics that depend on Tracked memory size. Times are given in milliseconds (ms).

Table V: Cost of internal metrics.

Table VI summarizes our analysis of the values reported in Table Va and Table Vb:

- `/proc`: it involves 4 metrics, among which 3 depend on Tracked memory size. The most costly metric is page table walk in userspace (M_{16}) which can take up to $594ms$. Page fault handling in kernel space (M_5) is the second most costly metric. It can take up to $33ms$, which is quite significant since it may be involved frequently during the monitoring phase. This is why it dramatically impacts `/proc` scalability (from both Tracked and Tracker perspectives).

	/proc	ufd	SPML	EPML
associated metrics	M_1, M_5, M_{15}, M_{16}	M_1, M_2, M_5, M_6	M_1, M_3, M_4, M_9, M_{11}, M_{13}, M_{14}, M_{16}, M_{17}, M_{18}	M_1, M_3, M_4, M_7, M_8, M_{10}, M_{12}, M_{18}
metrics depending on Tracked mem. size	3 (M_5, M_{15}, M_{16})	3 (M_2, M_5, M_6)	4 (M_{14}, M_{16}, M_{17}, M_{18})	1 (M_{18})
metrics involved in the monitoring phase	1 (M_5)	2 (M_5, M_6)	2 (M_{13}, M_{14})	2 (M_7, M_8)
the two most costly metrics	M_5, M_{16}	M_5, M_6	M_{16}, M_{17}	M_{10}, M_{12}
metrics which impact scalability from Tracker point of view	**3** (M_5, M_{15}, M_{16})	**3** (M_2, M_5, M_6)	**4** (M_{14}, M_{16}, M_{17}, M_{18})	**1** (M_{18})
metrics which impact scalability from Tracked point of view	**3** (M_5, M_{15}, M_{16})	**2** (M_5, M_6)	**2** (M_{13}, M_{14})	**2** (M_7, M_8)

Table VI: Influence of /proc, ufd, SPML and EPML on internal metrics. $\{M_i\}$ are defined in Table Va.

- ufd: it involves 3 metrics, among which 2 depend on Tracked memory size. The most costly metric is page fault handling in userspace (M_6), which costs up to $3,483ms$ when Tracked memory size is 1GB. This metric is further involved during the monitoring phase, thus impacting the scalability of ufd (from both Tracked and Tracker perspectives).
- SPML: it involves 10 metrics, among which 4 depend on Tracked memory size. The most costly metric is reverse mapping (M_{17}) which takes up to $15s$ for 1GB Tracked memory size. This metric will only impact the scalability of Tracker because it is not involved in the monitoring phase. Figure 3 presents the proportion of time taken by each step of the collection phase in SPML. We can observe that reverse mapping is the bottleneck of SPML and represents on average more than 68% of the total collection time.
- EPML: it involves 8 metrics, among which only one depends on Tracked memory size. The most costly metric is PML initialization (M_{10}) which also includes VMCS shadowing initialization. It costs about $5,878ms$. 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 vmread (M_7) and vmwrite (M_8) whose costs are very low (less than $1\mu s$). This makes EPML scalable.

D. Micro-benchmark Results

This section evaluates the impact of each dirty page tracking technique on Tracked where it is a micro-benchmark application. We vary the memory size of Tracked and we measure its execution time with and without the tracking technique. Figure 4 plots the slowdown incurred by each technique. We can observe that except EPML, the overhead on Tracked increases with its memory size. For almost all memory sizes, SPML incurs the greatest slowdown, up to 66×. This high overhead is due to reverse mapping that can take up to $15s$ for 1GB of memory (see Figure 3). Figure 4 reveals that ufd also highly increases the execution time of Tracked (by up to 15×). When the memory size is less than 250MB, ufd appears to be the worst technique. This is explained by the fact that page fault handling in userspace, which is the bottleneck of ufd technique (see Table Vb), is more costly than reverse mapping for a working set less than 250MB. The overhead of EPML is negligible regardless of the memory size of Tracked, confirming its scalability. With a maximum overhead of about 0.6%, EPML appears to be the best technique.

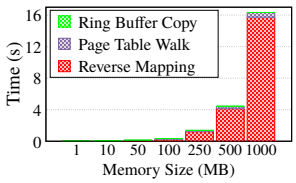

Figure 3: Time of metrics *Reverse mapping*, *PT walk*, and *Ring buffer copy* when using SPML technique. Reverse mapping appears to be the bottleneck of SPML.

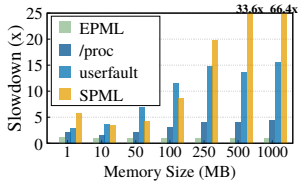

Figure 4: Overhead of each tracking technique on the micro-benchmark.

E. Boehm Results

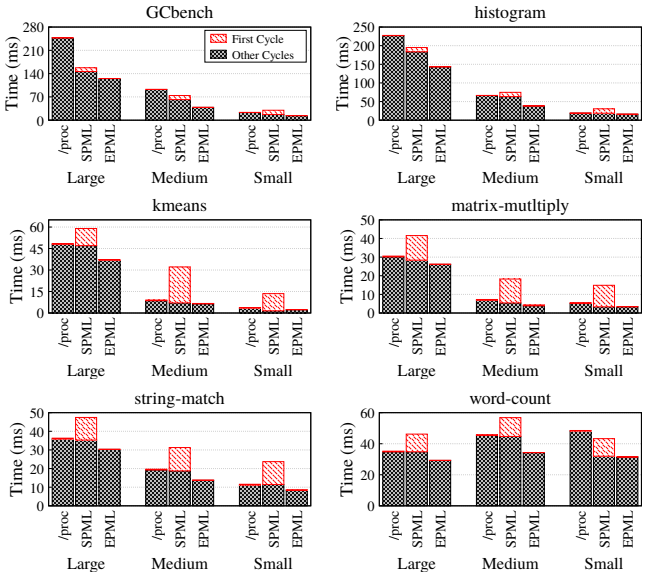

Figure 5: Execution time of Boehm GC when implemented with /proc, SPML, and EPML. The figure highlights the first garbage collection cycle during which Boehm performs the reverse mapping with SPML, the reason why its execution time is higher for SPML compared to the two other techniques.

In this section, we evaluate the impact of /proc, SPML, and EPML on Boehm GC using GCBench and Phoenix macro-benchmarks. We do not evaluate Boehm using the tkrzw benchmark because the former only works properly for

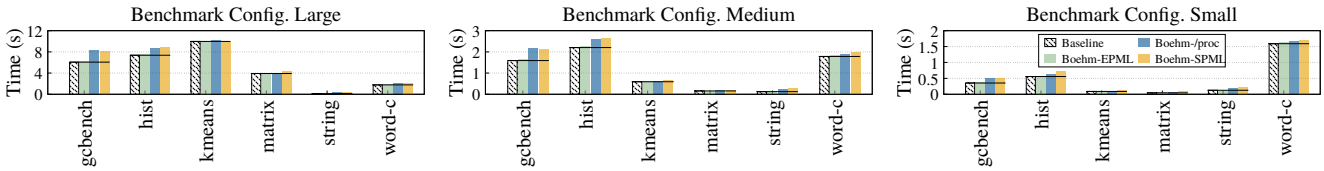

Figure 6: Impact of Boehm GC on Tracked execution time when using `/proc`, SPML and EPML techniques. The baseline is the ideal execution time of the application, i.e. when not tracked.

C applications [**?**]. We implemented SPML and EPML in Boehm. We evaluate both the impact on Tracker and Tracked.

a) Impact on Tracker (Boehm GC): Figure 5 presents the impact of each technique on Boehm GC. Due to page limitations, we do not present the results for all applications. Boehm GC can perform from 2 (e.g., for `histogram` config. *Small*) to 23 (e.g., for `GCBench` config. *Large*) cycles of garbage collection depending on the allocation intensity of the workload. Figure 5 plots the garbage collection time during the execution of each application. We emphasize the first cycle as it highlights the overhead of SPML on Boehm which performs the reverse mapping during this cycle [2]. We can observe in Figure 5 that if we ignore the first cycle, SPML outperforms `/proc`. This explains why EPML is the best solution, as it avoids reverse mapping (see Figure 3). Nonetheless, SPML outperforms `/proc` by up to 36% for applications `histogram` (configuration *Large*), `word-count` (configuration *Medium*), and `GCBench` (configurations *Large* and *Medium*). EPML outperforms both `/proc` and SPML respectively by up to 58% and 47% (with `GCBench` config. *Medium*).

b) Impact on Tracked: The execution time of applications (Tracked) that use Boehm GC will be impacted at least by the duration of the GC (see Figure 5). Figure 6 assesses the level of this impact according to the technique that Boehm uses. We can see that compared to `/proc`, SPML increases the overhead of Boehm GC on most applications. But for `GCBench` configuration *Medium*, SPML reduces the overhead of Boehm compared to `/proc` by about 1.7%. About `matrix-multiply` that runs in 51ms, SPML increases by about 63% the overhead of Boehm compared to `/proc` . This increase is only 2% for `GCBench` configuration *Medium* which runs in 6s and 0.5% for `histogram` configuration *Large* which runs in 7.3s. EPML significantly reduces the overhead of Boehm compared to `/proc` and SPML for all applications, by about 62% with `string-match` application.

F. CRIU Results

In this section, we evaluate the impact of `/proc`, SPML, and EPML on CRIU using macro-benchmarks. For Phoenix applications, we use the *Large* configuration and we increase the number of map-reduce tasks to make the applications have long execution times for checkpointing.

[2]During the following cycles, Boehm just reuses the addresses collected during the first cycle.

a) Impact on Tracker (CRIU): We evaluated each stage of the CRIU checkpointing algorithm, which includes among others the memory write (MW) phase and the memory dump (MD) phase. CRIU collects the dirty pages to be dumped during the MD phase and writes them to the disk during the MW phase. Depending on the tracking technique implemented by CRIU, these two phases can be done sequentially or simultaneously. When CRIU implements the `/proc` technique it walks the process page table to get dirty pages and writes them to the disk as it finds them. While with SPML and EPML it first collects all dirty pages from the ring buffer and then writes them to the disk. This is the reason why SPML and EPML significantly improve the MW phase compared to `/proc`, as shown in Figure 7. We measured an improvement of up to 26× with application `tiny` configuration *Large*. In addition, MW time is almost constant with SPML and EPML while with `/proc` it can take up to 5.7s.

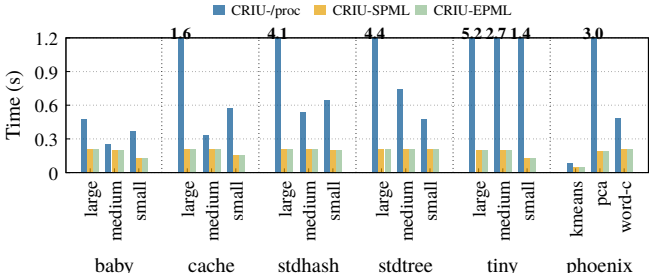

Figure 7: Memory write time during CRIU checkpointing.

Figure 8 presents the complete checkpointing time, highlighting the MD phase. When using SPML, CRIU performs reverse mapping during the MD phase, which drastically increases the checkpointing time. This leads to a non-negligible overhead as we can see in Figure 8. Indeed, complete checkpointing is up to 5× slower with SPML compared to `/proc` for both tkrzw (`baby` configuration *Large*) and `kmeans` configuration *Large*). Since reverse mapping represents on average more than 66% of MD time with SPML, avoiding it may lead to better performance compared to `/proc`. This is why EPML allows CRIU to execute faster compared to `/proc` and SPML. Indeed, EPML brings up to 4× speedup to CRIU checkpointing compared to `/proc` and up to 13× speedup compared to SPML, respectively with `tiny` and `baby` configuration *Large* (in Figure 8).

b) Impact on Tracked: Applications are paused during checkpointing. Therefore, CRIU increases the execution time of the checkpointed application. Figure 9 presents the overhead

of CRIU. The default implementation of CRIU (that uses `/proc`) can significantly impact the checkpointed application, by up to ∼102% for `pca` application. Concerning SPML, it incurs a higher overhead on the application execution time compared to `/proc`. Figure 9 shows that this overhead can vary from ∼1% (with `kmeans`) to ∼114% (with `pca`). EPML leads to the best results. Its overhead does not exceed 14%, with an average of only 3%.

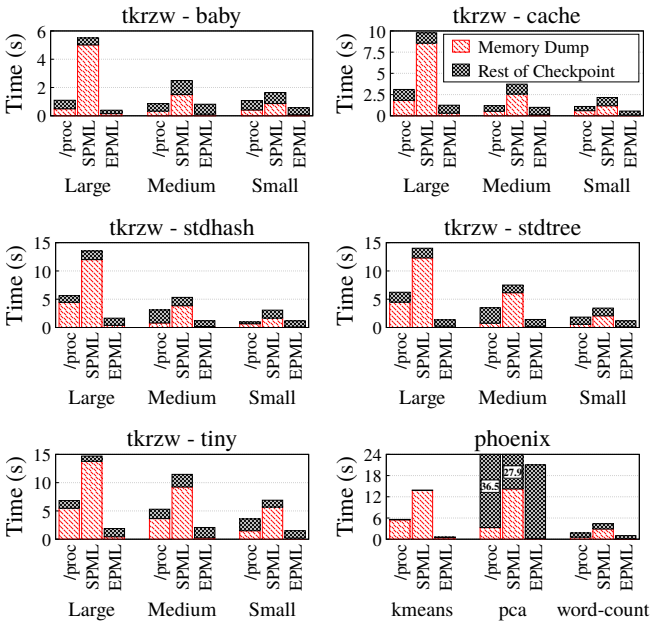

Figure 8: Execution time of CRIU when implemented with `/proc`, SPML, and EPML. The figure MD phase during which CRIU performs the reverse mapping with SPML.

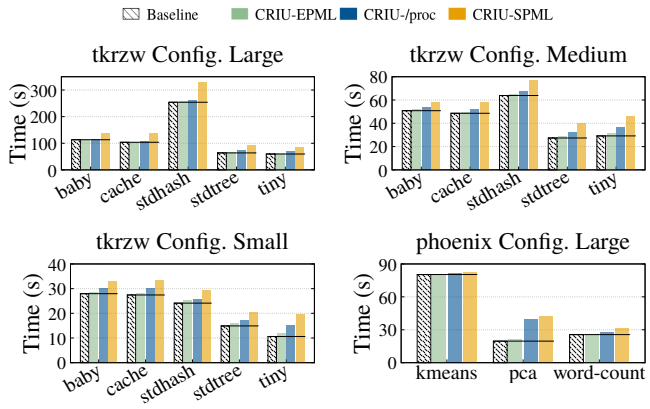

Figure 9: Impact of CRIU on Tracked execution time when using `/proc`, SPML and EPML techniques. The baseline is the ideal execution time of the application, i.e. when not tracked.

G. Scalability

In Sections VI-E and VI-F we have evaluated the scalability of SPML and EPML with different working set sizes. In the present section, we vary the number of tenant VMs from one up to 5. The evaluation scenario is the same as in the previous sections. We use Boehm and the Phoenix-`histogram` application (Config. *Large*). Results are presented in Fig. 10 and 11. We can observe that, the performance impact of both SPML and EPML on Tracker (Fig. 10) and Tracked (Fig. 11) is the same as what we obtained with one VM (Fig. 5 and Fig. 6 config. Large). In addition, this performance remains quite constant when the number of VMs increases.

VII. Related work

a) Nested virtualization: The main source of performance degradation in virtualized environments is VM traps. The latter lead to VM execution suspension and also to cache pollution [**?**] due to context switches. The number of VM traps increases at least by a factor of two in nested virtualized environments [**?**]. The reduction of VM traps is a hot topic in both non-nested [**? ? ?**] and nested virtualized systems [**? ? ?**]. Device passthrough is a simple approach for improving I/O performance in nested and non-nested virtualized environments by providing direct access to the VM. However, it dedicates the entire device to a single VM, resulting in 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 [**?**], that EPML leverages, has been introduced by Intel to reduce traps when a nested hypervisor accesses some VMCS fields. SV_T, by Vilanova et al. [**?**], 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 that 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. [**?**], proposed that the host hypervisor provides virtual devices directly to nested VMs without the intervention of intermediate hypervisors. The intermediate hypervisors only intervene 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 that compose 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 processes 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 faults interception. To minimize the overhead of this approach, some alternatives have been proposed. For the hypervisor, Intel introduced PML, the hardware

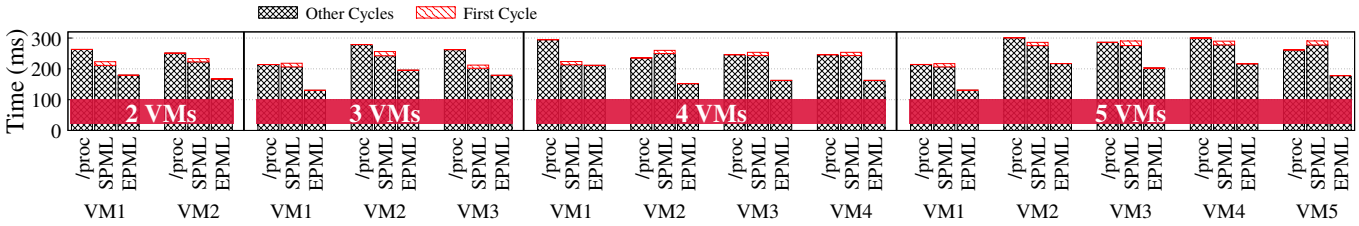

Figure 10: Impact of each tracking technique on Tracker when varying the number of VMs.

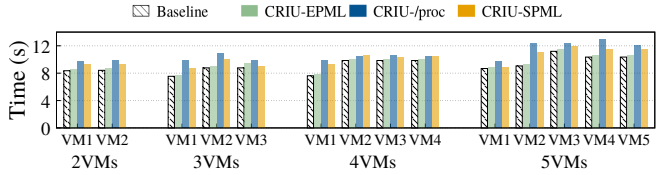

Figure 11: Impact of each tracking technique on Tracked when varying the number of VMs.

feature that we study in the present paper. Bitchebe et al. [**?**] showed that PML can decrease both VM live migration and checkpointing duration. The authors also extended PML to log read pages in order to efficiently estimate VM working set size. In non-virtualized environments, Lu et al. [**?**] built a memory allocator which maps several objects to the same physical page, thus reducing the number of tracked pages. Nevertheless, this solution does not avoid frequent interruption of the tracked process due to page faults.

VIII. Conclusion

This paper introduces Out of Hypervisor (OoH), a new research axis that advocates the exposure of individually current hypervisor-oriented hardware virtualization features to the guest OS so that its processes can also take benefit from those features. We illustrated OoH with Intel PML, a feature that allows efficient dirty page tracking. We prototyped OoH following two solutions namely Shadow (SPML) and Extended PML (EPML). The former requires no hardware changes but incurs significant performance overhead. It is not the case of EPML that extends the original PML to avoid SPML limitations. We evaluated and compared SPML and EPML with two popular dirty page tracking techniques namely `/proc` and `ufd`. We implemented OoH using the Xen hypervisor, Linux OS, and the Bochs emulator. We considered CRIU and Boehm GC as the use cases, where the target applications are Phoenix (a shared-memory data processing model) and tkrzw (a key value data processing model). The evaluation results showed that the different techniques can be classified as follows, from the most to the least costly: SPML, `ufd`, `/proc`, and EPML. Indeed, EPML brings up to 13× speedup to systems using the other techniques, while reducing their overhead on applications by up to 16×.

Acknowledgements

This work was funded by the ANR project Scalevisor ANR-18-CE25-0016, and with the support of the NEC Research Fellowship Program 2021 and the Google Scholarship for Women in Computer Science 2022. We would also give a special thank to Chiraz Benamor who made our work easier throught great administrative support.