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

Stella Bitchebe - (bitchebe@i3s.unice.fr)
Advisor: Pr Alain Tchana - (alain.tchana@ens-lyon.fr)
ENS Lyon

Rennes - April, 29

1. Dirty Page Tracking

2. OoH for PML

Dirty Page Tracking in Userspace

Purpose

- WSS (working set size) estimation
- Live migration
- Checkpointing
- Garbage collection

Current approach

- Page write protection
- Two main solutions
  - Linux /proc interface
  - Linux userfaultfd (ufd) interface

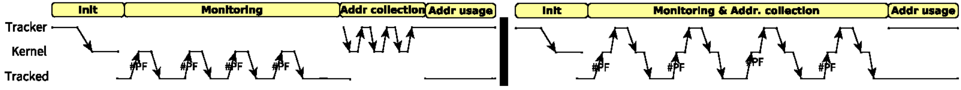
Dirty Page Tracking in Userspace

Nomenclature

- Tracker: the monitoring thread (e.g., CRIU, Boehm GC)
- Tracked: the thread whose memory is monitored (any application)

Overall Functioning

- Tracker's activity can be organized in four phases:
  - the initialization of the tracking method,
  - the monitoring
  - the collection of dirty page addresses
  - the exploitation of the latter (e.g., for checkpointing)



(a) /proc based tracking

(b) userfaultfd based tracking

Page write protection

Overhead

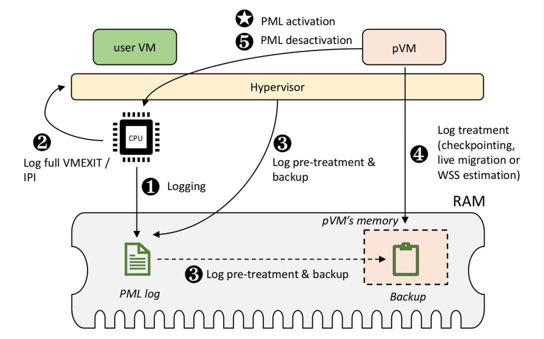
- ufd
  - We measured up to 15.6× and 14.5× slowdown for 1GB on Tracked and Tracker respectively
  - Due to page fault handling and context switches
- /proc
  - We measured about 2.234ms to parse the PT and flush the TLB (in the kernel), and up to 4.3× and 2.5× slowdown for 1GB on Tracked and Tracker respectively
  - We measured about 594.187ms to parse the PT in userspace (/proc/PID/pagemap) for 1GB

1. Dirty Page Tracking

2. OoH for PML

Intel PML

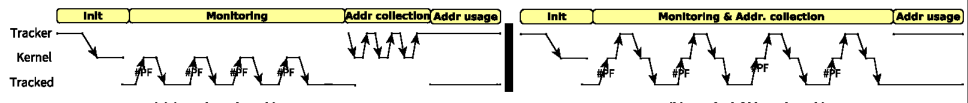
Functioning



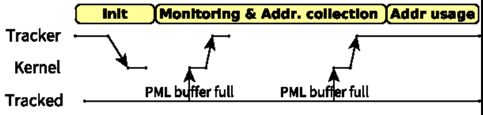
Intel PML in the OoH Context

- To accelerate CRIU checkpointing and Boehm garbage collection

Dirty page tracking in userspace (with PML)



(a) /proc based tracking
(b) userfaultfd based tracking



(c) OoH-PML based tracking

OoH for PML

Challenges

- (C_1) PML can only be managed by the hypervisor
- (C_2) PML works at coarse-grained, that is it concerns the entire VM
- (C_3) PML only logs GPAs

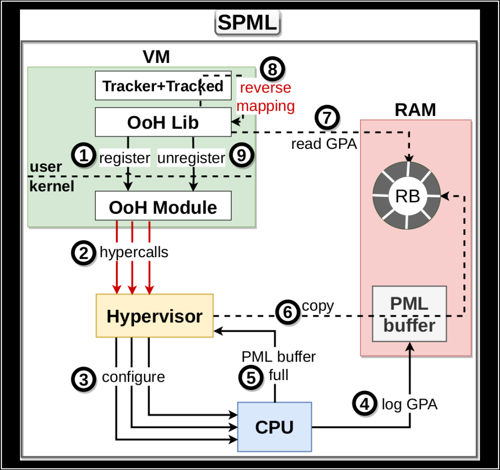
OoH for PML

Two Solutions

- Extended PML (EPML): small hardware changes
- Shadow PML (SPML): no hardware modification
  - Its huge overhead justifies EPML

OoH for PML

SPML Design



OoH for PML

SPML limitations

- Costly reverse mapping (takes about 15.739 *s* for 1GB working set)

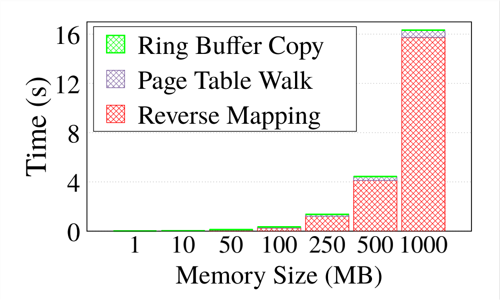


Figure: *Reverse mapping appears to be the bottleneck of SPML.*

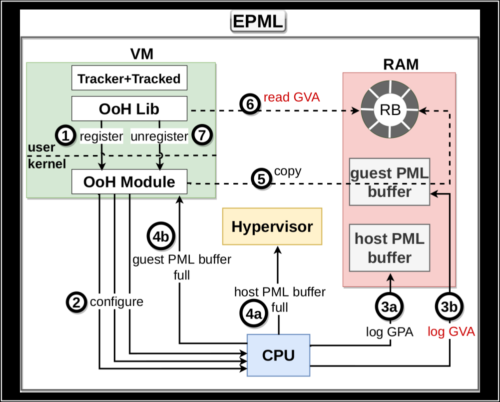
OoH for PML

SPML limitations

- Costly reverse mapping (takes about 15.739 s for 1GB working set)
- Costly hypercalls (4.49μs to perform an empty hypercall)

OoH for PML

EPML Design



OoH for PML

EPML

- We leverage VMCS shadowing to allow the guest to perform `vmread` and `vmwrite` instructions
- At load time, OoH Module calls the hypervisor to enable and configure VMCS shadowing

OoH for PML

Hardware changes

- We introduce in VMCS's VM-Execution Control area a new field (called `Guest PML Address`)
- We extend the PT walk to make the processor log GVA to the guest-level PML buffer and the GPA to the hypervisor-level PML buffer.
- We modify the hardware so that when the guest-level PML buffer is full, the processor raises a virtual self-IPI (Inter-Processor Interrupt)

OoH for PML

Implementation

- We implemented EPML's hardware changes in BOCHS
- We used Xen as the hypervisor and Linux as the guest OS
- We integrated OoH Lib with CRIU and Boehm GC

OoH for PML

Benchmarks

- Macro-benchmarks: tkrzw **tkrzw** applications (key value store) and Phoenix applications (MapReduce)
- We considered three working set sizes (Small, Medium and Large)

OoH for PML

Evaluations

- What is the potential overhead or improvement of SPML and EPML compared to `/proc` and `ufd`?
- What is the scalability of SPML and EPML?
- How to accurately evaluate EPML?
- Approach
  - build a formula
  - show the accuracy of that formula on other techniques, which are measurable
  - by construction, the formula is accurate for EPML

OoH for PML

Evaluations

- The execution time of Tracker when it implements technique x

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

where $E(C)$ is the execution time of code C and $I(C_1, C_2)$ is the impact of C_1 on C_2.

Evaluations

Impact on Tracker

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

Evaluations

Evaluations

- The execution time of Tracked when it is monitored by a tracker using the technique x:

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

where $I(C_x, C_{tked})$ consists of page faults, vmexits, etc.. Thus, the overhead of x on Tracked is $E(C_{tker}) + I(C_x, C_{tked})$.

Evaluations

Impact on Tracked

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

For $I(C_{EPML}, C_{tked})$, we use SPML's N (the number of context switches) as it is the same (validated by running SPML and EPML under BOCHS).

Formula validation

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*

Evaluations

Formula validation

An accuracy of about 96.34% and 99% respectively for SPML and /proc formulas

Evaluations

Impact on Boehm

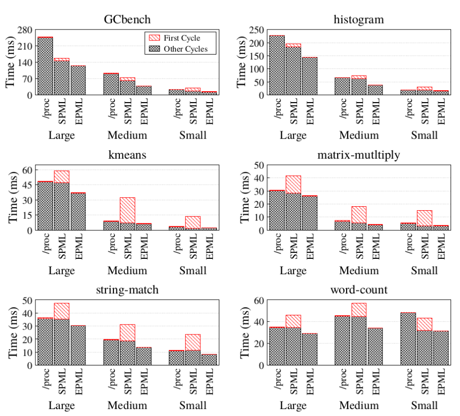


Figure: *We highlight the first GC cycle during which Boehm performs reverse mapping with SPML.*

Evaluations

Impact on applications

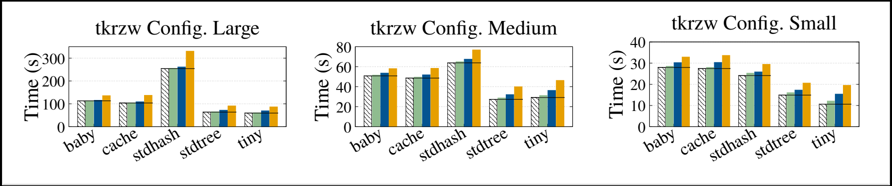


Figure: *Impact of Boehm GC.*

Evaluations

Impact on CRIU

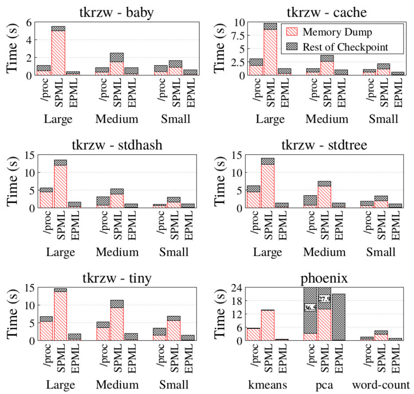
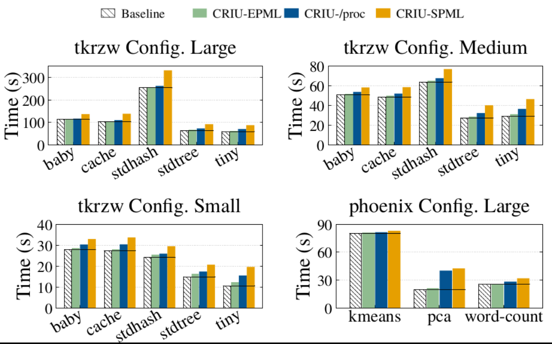


Figure: *We highlight MD phase during which CRIU performs reverse mapping with SPML.*

Evaluations

Impact on the checkpointed app



Evaluations

Summary

- Impact on the tracker
  - 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; up to 2× speedup compared to `/proc` and up to 6× speedup compared to SPML.

Evaluations

Summary

- Impact on the application performance
  - `/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 and 24% with Boehm.

Summary

Dirty Page Tracking and OoH for Intel PML

- For wss estimation, live migration, checkpointing, GC, ...
- Induce high overhead on applications
- For improving process/container checkpointing, concurrent GCs
- Excellent results in both reducing overhead on tracked applications and improving the tracking application

Thake Away

- Existing sofware-based tools can be improved using hardware virtualization features
- When thinking hypervisor-oriented hardware virtualization features, we must think how they can also be used by processes from inside VMs, this may need few efforts