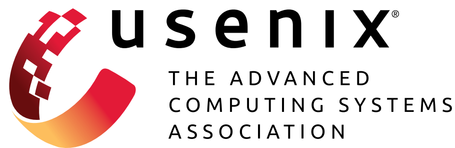


Efficient Memory Overcommitment for I/O Passthrough Enabled VMs via Fine-grained Page Meta-data Management

Yaohui Wang, Ben Luo, and Yibin Shen, *Alibaba Group*

https://www.usenix.org/conference/atc23/presentation/wang-yaohui

This paper is included in the Proceedings of the 2023 USENIX Annual Technical Conference.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-35-9

Open access to the Proceedings of the 2023 USENIX Annual Technical Conference is sponsored by

Efficient Memory Overcommitment for I/O Passthrough Enabled VMs via Fine-grained Page Meta-data Management

Yaohui Wang, Ben Luo, Yibin Shen
Alibaba Group

Abstract

In virtualization systems, guest memory overcommitment helps to improve the utilization of the host memory resource. However, the widely adopted I/O passthrough technique makes this task not intuitive since the hypervisor must avoid DMA (Direct Memory Access) failures when the I/O device accesses the guest memory. There already exist several solutions, for example, IOPF (I/O Page Fault) can fix DMA failures by allowing page fault triggered from the I/O device side, vIOMMU and coIOMMU avoid DMA failures by monitoring the DMA buffers in the guest. However, these solutions all face the performance concerns introduced by the memory backup/restore mechanism, *i.e.*, memory swapping. Some free page based methods (*e.g.*, Ballooning, Free Page Reporting, Hyperupcall) are free from memory swapping, but they either are not DMA-safe or introduce high guest communication overhead. In this paper, we propose V-Probe, a high-efficiency approach to achieve memory overcommitment for I/O passthrough enabled VMs. Using fine-grained page meta-data management, V-Probe allows the hypervisor to inspect and reclaim guest free pages actively and efficiently while guaranteeing DMA-safety. Experiments show that, for both memory reclamation and reallocation, the overhead of V-Probe is in the scale of microseconds, which is faster than Ballooning and IOPF based methods by two orders of magnitude. And our micro-benchmark and macro-benchmark show that V-Probe limits the performance impact of memory overcommitment to a low level.

1 Introduction

In virtualization systems, the total memory size of a VM (Virtual Machine) is commonly constant while it is running, but the working set of memory (*i.e.*, memory accessed actively) inside a VM is usually a subset of the total memory [11, 17, 36, 42, 47]. This results in the inefficiency of memory resource utilization [28, 39]. Memory overcommitment [4, 6, 7, 9, 19, 26, 37, 40, 44] helps to mitigate this problem. It reclaims cold memory (*i.e.*, memory not accessed recently) from a VM, and reallocates the memory to other VMs on demand to increase memory utilization.

However, the widely adopted I/O passthrough technique [3, 5, 12, 15, 26, 27, 41], which significantly reduces the overhead of I/O virtualization [1, 21, 24], requires the hypervisor to keep a fixed memory mapping for the VM during its life cycle to avoid potential DMA failures [4, 26, 40]. This highly limits the ability of memory overcommitment since the memory reclamation will change the memory mapping of a VM dynamically (Section 2).

Previous work tries to solve the contradiction between I/O passthrough and memory overcommitment in two different ways. The first is to introduce the IOPF (I/O Page Fault) mechanism [25, 26, 34]. It allows an I/O device to notify the hypervisor by triggering page faults when the target DMA buffer does not reside in the main memory. After the hypervisor repairs the IOPT (I/O Page Table) entry for the faulted IOVA (I/O Virtual Address), the device can replay the previously failed DMA request. The second is to monitor the guest DMA buffers using the PV (Para-virtualization) technique [4, 10, 11, 40, 44]. Such solutions include vIOMMU [4] and coIOMMU [40]. They use frontend drivers in the guest to inform the hypervisor with DMA buffer alloc/free events. With such knowledge, the hypervisor always keeps the memory mapping of the DMA buffers and thus prevents DMA failures.

Although the above solutions allow memory overcommitment to coexist with the I/O passthrough technique, they face several deficiencies. The first is the compatibility issue, which makes them currently less practical. For example, the IOPF solution requires designated hardware which is not widely supported by hardware manufacturers. And the coIOMMU solution needs changes to the guest OS and is still not supported by off-the-shelf OSes (*e.g.*, the Linux upstream). But even though the compatibility issue can be fixed over time (by evolving the hardware/software), the second issue – performance issue – is unavoidable. When doing memory reclamation/reallocation, these solutions need to backup/restore the

memory content using memory swapping [6, 20, 29, 30, 32]. The overhead it introduces not only slows down the memory reclamation process, but may also cause performance oscillations to the VM and hurts the VM's SLO [6, 49, 50].

While memory swapping is costly, reclaiming guest free pages [44, 45] is a good way to eliminate such overhead, since the content in free pages is meaningless. But because of the semantic gap in virtualization systems, free page reclamation methods usually rely on the communication with the frontend driver running inside the guest to obtain the knowledge of guest free pages. The guest communication introduces extra overhead and increases the response time for the memory reclamation requests (Section 5.1.2). The recently proposed free page inspecting method – Hyperupcall [7] – helps to eliminate such overhead. When doing memory reclamation, Hyperupcall allows the hypervisor to actively invoke the guest injected eBPF functions to inspect guest free pages. However, guaranteeing DMA-safety is a challenging task with this method. In Hyperupcall, since the guest is agnostic to the hypervisor's memory reclamation actions, it may allocate a free page, whose underlying physical page is reclaimed by the hypervisor, as a DMA buffer. Such behavior cannot be perceived by the hypervisor, so it does not have a chance to repair the memory mapping for the DMA buffer, which will further cause DMA failures (Section 2.2.3).

In this paper, we propose V-Probe to address the performance and DMA-safety challenges of memory overcommitment for I/O passthrough enabled VMs. V-Probe targets guest free pages to eliminate the costly overhead of memory swapping. Inspired by Hyperupcall [7], V-Probe uses guest injected helper functions to detect guest free pages actively, which avoids the communication overhead with the guest. But instead of using eBPF, V-Probe uses raw binary helper functions. This avoids the complex eBPF dependencies in the guest OS required by Hyperupcall. Such simplicity makes the deployment of the method easier. V-Probe uses the SFI (Software Fault Isolation) technique [13, 31, 38, 43, 48], which performs strict rule-based instruction-level checks to the injected binary, to prevent malicious code. Using fine-grained page meta-data management, V-Probe not only manages the memory mapping of the reclaimed free pages but also monitors their corresponding page meta-data. This allows V-Probe to react to guest memory allocation events and prevent potential DMA failures. Experiments show that the overhead of V-Probe is in the scale of microseconds. Compared to Ballooning and IOPF, it is faster by two orders of magnitude in both memory reclamation and reallocation. Our micro-benchmark and macro-benchmark show that V-Probe limits the performance impact of memory overcommitment to a low level.

We summarize our contribution as follows:

- We conduct a systematical study of previous VM memory overcommitment methods and characterize these methods in different aspects to provide an overview.
- We propose V-Probe, an efficient memory overcommitment method that targets guest free pages and guarantees DMA-safety for I/O passthrough enabled VMs.
- We evaluate the overhead of V-Probe, and assess its performance in both micro-benchmark and macro-benchmark tests. Results show that V-Probe achieves low overhead and limits the performance impact of memory overcommitment to a low level.

2 Motivation

2.1 I/O Passthrough & DMA-safety

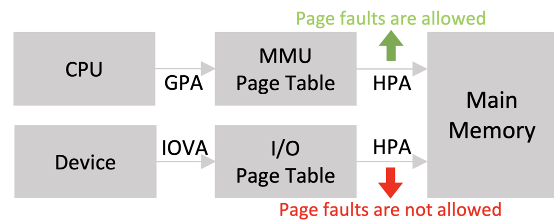


Figure 1: The address translation for CPU and I/O devices in modern hardware architectures. Page faults in the I/O device side will trigger DMA failures.

I/O passthrough, which allows the guest OS to directly interact with the underlying hardware, is widely used in virtualization systems [15, 16]. Using I/O passthrough, a peripheral device can directly access the guest memory through DMA without the intervention of the CPU. It significantly decreases the overhead of guest I/O operations and is also required by the emerging high-performance RDMA (Remote Direct Memory Access) applications [22, 23, 46]. This makes I/O passthrough an irreplaceable part of virtualization systems like the public cloud.

In the same way that the CPU MMU (Memory Management Unit) needs a page table to translate GPA (Guest Physical Address) to HPA (Host Physical Address), devices' DMA requests rely on the IOPT to translate IOVA (I/O Virtual Address, it is usually the same as GPA) to HPA. On the CPU side, if the accessed GPA does not have a valid page table entry, then a page fault will be triggered. The hypervisor will try to fix the memory mapping in the page table, and then resume the guest OS execution from where it triggers the page fault. On the device side, to add IOPF support, the device needs the ability to replay the once failed DMA request after the hypervisor fixes the invalid IOPT entry. But unfortunately, most off-the-shelf devices do not support such mechanism. So they require that all DMA requests must always succeed or they will result in DMA failure and crash of the guest OS (Figure 1). This implies that an I/O passthrough enabled VM has no tolerance to IOPT entry missing [4, 25, 26, 40].

Table 1: The characteristics of the memory reclamation methods from different dimensions.

Methods	DMA-safety	Hardware Compatibility	Guest Compatibility	Main Overhead
IOPF	DMA-safe	Dedicated hardware	No requirements	Memory swapping
vIOMMU	DMA-safe	General hardware	Frontend driver	Memory swapping
coIOMMU	DMA-safe	General hardware	Frontend driver	Memory swapping
Ballooning	DMA-safe	General hardware	Frontend driver	Guest communication
Free Page Reporting	DMA-unsafe	General hardware	Frontend driver	Guest communication
Hyperupcall	DMA-unsafe	General hardware	eBPF tool-chain	Extremely low overhead

In memory overcommitment, the MMU page table entry and the IOPT entry of the reclaimed page will both be invalidated after memory reclamation. As the hypervisor is agnostic to whether a page is used for DMA in the guest, the reclaimed page may be a DMA buffer. When a DMA request targeting the reclaimed DMA buffer arrives, the missing IOPT entry will cause DMA failure. To avoid potential DMA failures, the hypervisor needs to disable memory overcommitment, and statically pin the entire memory of a VM, *i.e.,* keep a fixed memory mapping for the VM during its life cycle.

2.2 Related Work

We discuss existing memory reclamation methods in the aspect of DMA-safety. We also characterize them in other important dimensions like compatibility and performance. We summarize them in Table 1.

2.2.1 I/O Page Fault

IOPF [25, 26, 34] is a hardware feature. When DMA failure occurs, IOPF allows a device to generate a page fault exception to the CPU, and replay the DMA request after the OS repairs the memory mapping. So by using IOPF, memory overcommitment is guaranteed to be DMA-safe.

As IOPF is transparent to the guest, it has no requirements for the guest OSes. However, it faces the issue of poor hardware compatibility as it requires designated hardware devices. Although the PCIe specification [34] has been extended to support IOPF since 2009, with the extensions of ATS (Address Translation Service) and PRI (Page Request Interface), the practice of such standard moves slowly. Currently, few off-the-shelf I/O device supports IOPF. Although manufacturers like AMD and Arm have introduced ATS and PRI to their SoC design [2, 8], the absence of PCIe devices that supports IOPF makes the IOPF call chain incomplete.

Even though the hardware compatibility issue can be fixed over time, the performance issue is unavoidable. Memory overcommitment depending on IOPF needs to backup/restore the memory content when doing memory reclamation/reallocation using memory swapping. Memory swapping introduces I/O overhead since the contents of the reclaimed memory are usually stored in storage media which is much slower than the main memory. Another method of memory swapping is memory compression. It introduces CPU overhead since the compress/decompress process is a heavy computing task. The overhead of memory swapping not only slows down the hypervisor's reaction to memory reclamation requests, but also causes performance oscillation to the VM and degrades the VM's SLO, especially when a VM faces a burst memory pressure and triggers a large number of page faults in a short time.

Meanwhile, the hypervisor's memory reclamation mechanism may also conflict with the memory reclamation inside the guest – The same memory page may be reclaimed twice, once by the guest and once by the hypervisor. In such a case, the guest's access to the memory page will trigger page faults twice, once trapped to the guest kernel and once to the hypervisor. This is the so-called *double paging* anomaly [14, 18, 33, 44] which introduces extra overhead to memory reclamation.

2.2.2 Monitoring DMA Buffers

Another way to avoid DMA failure is to monitor DMA buffer allocations in the guests. It implies two parts. First, when the hypervisor reclaims a guest page, the monitor tells whether it is a DMA buffer. Second, when the guest allocates a DMA buffer whose underlying page is reclaimed, the monitor notifies the hypervisor to fix the memory mapping in time. It is a software solution and does not rely on designated hardware.

vIOMMU [4] is one such solution. It exposes an emulated IOMMU (Input-output Memory Management Unit) to the guest and enables the hypervisor to intercept, monitor, and act upon DMA remapping operations. coIOMMU [40] is another one. It decouples the memory protection and pinning functionality in vIOMMU, which significantly improves the performance. But coIOMMU needs extensive changes to the guest OS, which results in the guest OS compatibility issue. At the same time, although monitoring guest DMA buffers guarantees DMA-safety, it still faces similar performance issues as the IOPF solution because of the need for memory swapping when doing memory reclamation/reallocation.

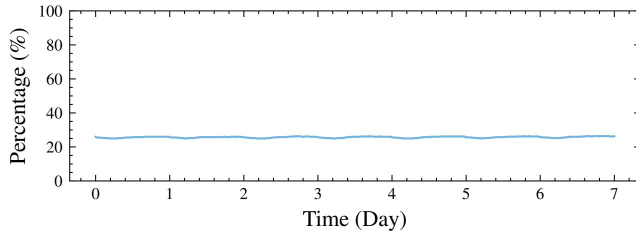


Figure 2: The free memory proportion of 10,000 VMs randomly sampled from the Alibaba cloud in one week period. The X-axis is the time, the Y-axis is the sum of the VMs' free memory size divided by the sum of the VMs' occupied memory size.

2.2.3 Free Page Reclamation

Free page reclamation targets on guest free pages. The quantity of guest free pages is considerable. Figure 2 shows the free memory proportion statistics of 10,000 VMs randomly sampled from the Alibaba cloud in one week period. As it shows, about 25.3% of the VMs' memory is free and this proportion is stable over time. At the same time, reclaiming free pages helps to eliminate the memory swapping overhead, since the contents of free pages are meaningless.

Ballooning [44] is a classical PV method that targets guest free page reclamation. It contains a frontend driver running in the guest, and a backend driver running in the hypervisor. The frontend driver can "inflate" to occupy the guest free pages and report them to the hypervisor, then the backend driver can reclaim those pages. The reclaimed GPA area is not allocatable in the guest until the hypervisor reallocates new pages for it. This can prevent reclaimed GPA areas to be used as DMA buffers, and thus avoids DMA failures.

However, the communication overhead between the frontend and backend drivers (Section 5.1.2) brings performance issues. Similar to the overhead introduced by memory swapping (Section 2.2.1), the communication overhead may also slow down the hypervisor's reaction to memory reclamation request, and cause VM performance oscillation [7, 9, 37].

Except for Ballooning, there are also other methods targeting free page reclamation. Free Page Reporting [45] allows the guest to report its free pages to the hypervisor periodically. And Hyperupcall [7], allows the hypervisor to inspect guest free pages without guest intervention. However, guaranteeing DMA-safety using these methods is a challenging task. As the memory reclamation action in these methods is transparent to the guest, the guest may allocate a free page, whose underlying physical page is reclaimed, as a DMA buffer. But the hypervisor will not be notified by this action, thus it does not have a chance to repair the memory mapping. So following DMA requests addressing this page will fail and cause DMA failures. Figure 3 illustrates the scenario. On the other hand, Free Page Reporting faces the problem of timeliness. Its frontend driver initiates the reporting procedure with a fixed delay (2 seconds in Linux's implementation) after the guest releases a high-order page. So the hypervisor's memory reclamation request may fail even if there are unreclaimed free pages in the guest.

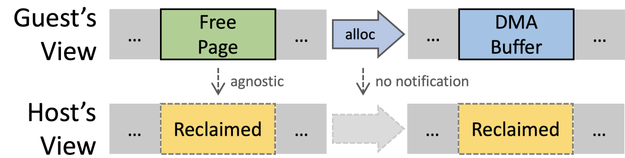


Figure 3: In guest free page reclamation, the hypervisor is agnostic to the DMA buffer allocation in the guest, and following DMA requests will cause DMA failures.

3 Approach

3.1 Design Goals

Based on our analysis in Section 2, we set our design goals as follows:

- **DMA-safety:** I/O passthrough is playing an irreplaceable part in today's virtualization systems since it significantly improves the guest I/O performance. The design should avoid DMA failures to make it safe for I/O passthrough enabled VMs.
- **Hardware Compatibility:** The design should not rely on IOPF as IOPF needs designated hardware. This ensures the solution is available for a large amount of existing hardware.
- **Guest Compatibility:** The design should be compatible with a wide range of commodity guest OSes. This makes sure the solution can be easily deployed to existing software systems.
- **Low Overhead:** The overhead of the solution should be as low as possible. This not only decreases the hypervisor's reaction time for memory requests and enables the hypervisor to take more aggressive memory reclamation decisions, but also limits the performance impact on the VMs and guarantees high VM SLO (Service-level Objective).

According to the summary in Table 1, memory reclamation methods that are unaware of the guest page allocation status (IOPF, vIOMMU, coIOMMU) require memory backup/restore for every reclaimed page, which introduces the overhead of memory swapping. However, obtaining knowledge of guest free pages usually needs communications to the frontend driver in the guest (Ballooning, Free Page Reporting), which introduces communication overhead. Hyperupcall

achieves low overhead, but it faces two challenges. First, as mentioned in Section 2.2.3, Hyperupcall is not DMA-safe. Although discarding guest communication makes the execution of Hyperupcall fast, this also means the guest is agnostic to the reclamation status of its free pages. The guest may allocate a reclaimed free page as a DMA buffer and the hypervisor is not notified by such behavior. It may not have a chance to fix the memory mapping before the DMA request arrives. Second, Hyperupcall relies on the complex toolchain of eBPF and needs modifications to the underlying LLVM compiler. This increases the deployment complexity of Hyperupcall, especially when adapting the toolchain to a variety of guest OSes in the public cloud.

Solving these challenges brings us to the solution of V-Probe. V-Probe is inspired by Hyperupcall. Like Hyperupcall, V-Probe allows the hypervisor to actively reclaim guest free pages without guest intervening. But V-Probe improves the programming model of Hyperupcall to overcome its complexity. More importantly, V-Probe proposes a novel fine-grained page meta-data management technique to guarantee DMA-safety.

3.2 V-Probe Overview

Figure 4 illustrates how V-Probe works. It contains three parts: (1) V-Probe injector, (2) V-Probe data manager, and (3) memory reclamation routine. Next, we will explain each part in detail.

3.2.1 V-Probe Injector

The V-Probe injector is a guest module. It only runs one time just after the guest finishes starting up. During its execution, it accomplishes two tasks: (1) page meta-data layout registration and (2) helper function registration.

In operating systems, like Linux, each physical page has a corresponding piece of page meta-data to record its usage state, *e.g.,* the reference count or the allocation status. And such meta-data is usually organized statically and continuously in ranges of the physical memory. In page meta-data layout registration, the guest first detects the GPA ranges where the page meta-data resides. Then it passes this information to the hypervisor through the registration API that V-Probe provides. As the hypervisor is aware of how GPA is translated to HPA, it can access the guest page meta-data precisely after obtaining the knowledge of guest page meta-data GPA ranges.

But only having access to the guest page meta-data is not enough, the hypervisor needs to understand the meanings of the bytes in it. So we need the helper functions which can parse the page meta-data. In helper function registration, the guest compiles the source codes of helper functions to hardware native binaries and injects them to the hypervisor through the registration API of V-Probe. As the formats of page meta-data can be different among guest OSes, the helper functions are guest-specific and need to be dynamically compiled in the guest.

3.2.2 V-Probe Data Manager

The V-Probe data manager runs in the hypervisor. It provides registration APIs for the V-Probe injector. The registered helper functions and page meta-data layout information are stored here. They help to recognize the status of each page in the guest when performing memory reclamation.

Since the hypervisor will directly call the helper functions via the injected binary code, we need a verifier to guarantee their integrity, *i.e.,* to prevent harmful codes injected from malicious guests. As the use case of the helper functions in V-Probe is limited to parsing guest page meta-data, the logic of the helper functions should be very simple. So the verifier uses strict rule-based restrictions to verify the injected binaries, which protects the hypervisor from security attacks without rejecting honest codes.

The V-Probe data manager also stores the information of the reclaimed memory set. Each record in the set represents a reclaimed guest memory area. Its content contains two dimensions: the reclaimed GFN (Guest Frame Number) range, and the GPA range for the corresponding page meta-data. This information is useful for page refaulting, which will be explained in the next subsection.

3.2.3 Memory Reclamation Routine

The memory reclamation routine is the core logic to reclaim free pages from the guest. Its execution is triggered by system events, like periodic timers, or memory pressure. After the preparation of the previous steps, the hypervisor can scan and parse guest page meta-data to inspect the status of each page.

V-Probe solves the DMA-safety problem via fine-grained page meta-data management. As shown in Figure 5, when reclaiming guest free pages, V-Probe not only invalidates the free pages' mapping in the MMU page table and the I/O page table but also modifies the page meta-data mapping in the MMU page table to read-only. The reclaimed memory set in V-Probe data manager will record the reclaimed GFN range, along with the GPA range of the corresponding page meta-data.

When the guest memory management system is going to allocate a page inside the reclaimed GFN ranges, it will first write to the corresponding page meta-data, *e.g.,* change the free flag. This will trigger a page fault on the CPU side as we have changed the memory mapping of the page meta-data to read-only after reclamation. The page fault notifies the hypervisor. Then the hypervisor looks for the corresponding record in the reclaimed memory set and recovers the memory mappings for both the pages and the page meta-data. This mechanism gives the hypervisor the chance to recover the

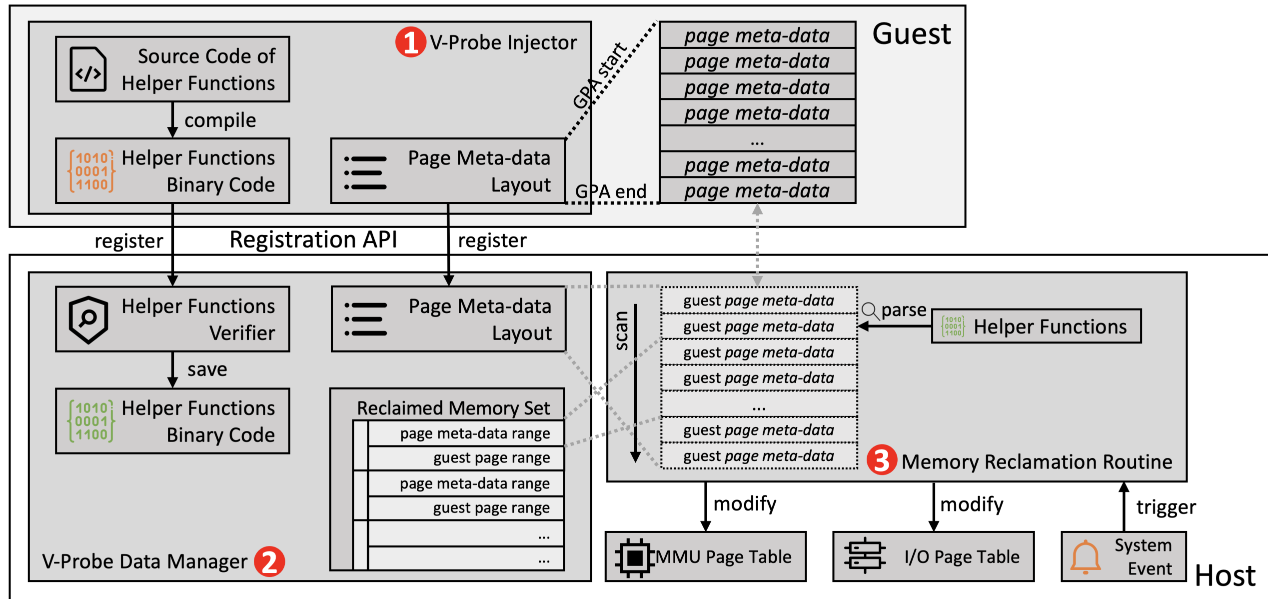
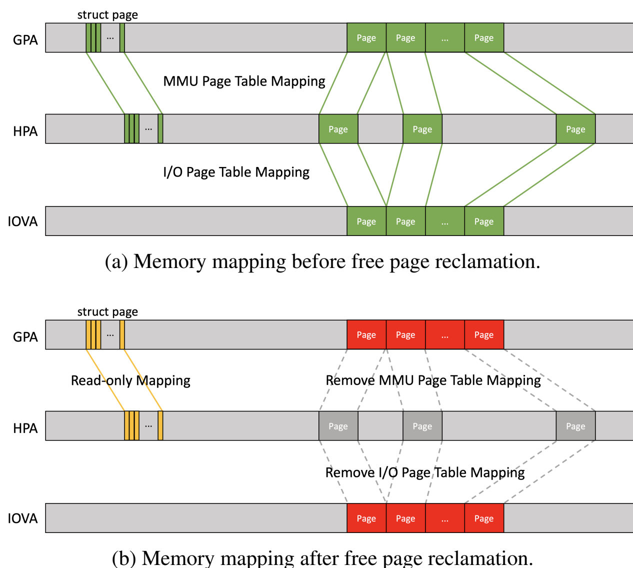


Figure 4: The design overview of V-Probe. It contains three parts: V-Probe injector, V-Probe data manager, and the memory reclamation routine. The V-Probe injector registers helper functions and page page-meta layout information to the V-Probe data manager. The code and data are used by the memory reclamation routine to inspect guest page status and reclaim free pages. The details of the design are explained in Section 3.2.



(a) Memory mapping before free page reclamation.

(b) Memory mapping after free page reclamation.

Figure 5: Memory mapping status before and after free page reclamation. During free page reclamation, V-Probe not only invalidates the page table entries for the reclaimed pages but also modifies the mapping of the page meta-data to read-only. This gives the hypervisor the chance to recover the memory mapping before the guest uses the page.

memory mapping before the guest uses the page, and thus it prevents DMA failures.

4 Implementation

In this section, we introduce the implementation details of V-Probe, which includes four parts: meta-data registration, helper function registration, memory reclamation routine, and page fault handler. We also explain the synchronization problem we face in V-Probe and how we solve this issue.

Our implementation of V-Probe is based on Linux, whose source code can be easily touched and modified. The hardware architecture is based on x86_64. The CPU we use is Intel® Xeon® Platinum 8163 CPU.

4.1 Meta-data Registration

The meta-data of pages in Linux is usually defined as type *struct page*. There are three kinds of *struct page* organization models in Linux, decided by one of the three compile configurations: `CONFIG_FLATMEM`, `CONFIG_DISCONTIGMEM` and `CONFIG_SPARSEMEM_VMEMMAP`. But no matter what the configuration is, the *struct page* data is organized linearly in ranges of continuous memory in the guest physical address space. V-Probe injector detects the *struct page* layout, indicating each *struct page* memory range with two dimensions: (1) the start and end GPA of the *struct page* memory range, and (2) the start and end GFN that are managed by the range

of *struct page*. V-Probe injector registers the information to the hypervisor, then the hypervisor can locate the *struct page* of each guest GFN.

4.2 Helper Functions Registration

```
noinline bool
page_free(struct page *page) {
        return PageBuddy(page);
}

8b 47 30            mov    0x30(%rdi),%eax
25 80 00 00 f0      and    $0xf0000080,%eax
3d 00 00 00 f0      cmp    $0xf0000000,%eax
0f 94 c0            sete   %al
c3                  retq
```

(a) The source code and compiled binary of the helper function `page_free`. This function is used to parse whether a guest page is free. The function it calls (`PageBuddy`) is a Linux built-in function. The size of the compiled binary is only 17 bytes.

```
noinline unsigned long
page_order(struct page *page) {
    return page_private(page);
}

48 8b 47 28         mov 0x28(%rdi),%rax
c3                  retq
```

(b) The source code and compiled binary of the helper function `page_order`. This function is used to parse how many free pages are adjacent to this page. The function it calls (`page_private`) is a Linux built-in function. The size of the compiled binary is only 5 bytes.

Figure 6: The source codes and their compiled binary of the two helper functions in V-Probe.

V-Probe relies on the helper functions to parse guest *struct page* and obtain the status of the pages. They are injected from the guest to the hypervisor after the guest finishes starting up. Since the formats of *struct page* vary from guests OSes, the helper functions need to be dynamically compiled in the guest.

Linux uses the buddy system [35] to manage free pages. It organizes free pages as blocks, and the size of one block is the order of 2. A buddy system block is identified by the leading *struct page* in the block, and it also records the order of the block. By parsing the *struct page*, *i.e.,* testing a specific bit or reading a specific field in it, we can decide whether a *struct page* is the leading one in a block, and get the order of the free page count in this block. In Linux, two kernel built-in functions, `PageBuddy` and `page_private`, help to do the parsing.

The two helper functions used in V-Probe are shown in Figure 6b: (1) `page_free`, which wraps the function `PageBuddy`, and (2) `page_order`, which wraps the function `page_private`. These two helper functions are short and simple, and the sizes of their binary code are also small: 17 bytes for `page_free` and 5 bytes for `page_order`. The definition of the helper functions requires the `noinline` decorator, to ensure they are not compiled to inline functions and can be called by the hypervisor in a function manner.

To avoid harmful codes injected from malicious guests, we need a verifier to guarantee the integrity of the helper function binary. We apply SFI to implement the verifier. As the logic of the helper functions used in V-Probe are very simple, simplified yet strict checking rules in SFI are enough to protect the hypervisor from security attacks without rejecting honest codes. These rules are as follows:

- Register `rdi` is read-only, register `rax` is read-write. Other registers are not allowed to be used.
- The function can only read a limited range of memory, which is usually the size of the guest *struct page*, pointed by the parameter stored in the `rdi` register.
- The function can not write to any main memory.
- No branch instruction or privileged instruction is allowed.
- The last instruction must be `retq`.
- The length of the binary is less than 64 bytes.

The above rules protect the hypervisor from being compromised by the injected code for two reasons. First, they make sure the helper functions obey the convention of a function call, and no branch nor privileged instructions are allowed. This implies the instructions inside the call will be executed sequentially, and the execution will finally return to the caller. Second, during the function call's execution, the accesses to registers and memory are strictly limited to avoid any host memory corruption. But note that these rules are highly related to the Intel X86_64 architecture. They need to be redesigned on other CPU platforms.

4.3 Memory Reclamation Routine

The memory reclamation routine is triggered by host events (like the memory pressure), and its execution is controlled by the hypervisor. Algorithm 1 shows the pseudocode of the procedure. It has two input parameters, *GUEST*, and *MIN_ORD*, which respectively indicate the target guest we try to reclaim memory from and the minimum order we want. The procedure iterates through each GFN in the guest (Line 2) and uses the two helper functions registered before, *GUEST.PAGE_FREE* and *GUEST.PAGE_ORD*, to check whether the guest page is free and what is the order in the buddy system (Lines 6-7). If it meets our requirements (Line 8), then we try to reclaim the corresponding pages (Lines

Algorithm 1 The procedure of memory reclamation

Input: $GUEST$, MIN_ORD
Output: $SUCCESS/FAIL$

1: **procedure** MEMORYRECLAMATION
2: **for** each GFN in GUEST **do**
3: **if** GFN is reclaimed **then**
4: continue
5: $SP \leftarrow$ struct page of GFN in $GUEST$
6: $FREE \leftarrow GUEST.PAGE_FREE(SP)$
7: $ORD \leftarrow GUEST.PAGE_ORD(SP)$
8: **if** $FREE$ and $ORD \geq MIN_ORD$ **then**
9: Lock MUTEX
10: Make the struct page GPA range read-only
11: $SP' \leftarrow$ struct page of GFN in $GUEST$
12: $FREE' \leftarrow GUEST.PAGE_FREE(SP')$
13: $ORD' \leftarrow GUEST.PAGE_ORD(SP')$
14: **if** $FREE'$ and $ORD' \geq MIN_ORD$ **then**
15: $GFN_{st} \leftarrow GFN$
16: $GFN_{en} \leftarrow GFN + (1 << ORD')$
17: $RANGE \leftarrow [GFN_{st}, GFN_{en})$
18: Unmap EPT in $RANGE$
19: Unmap IOMMU in $RANGE$
20: Add $RANGE$ to the reclaimed set
21: Release reclaimed pages to hypervisor
22: Unlock MUTEX
23: **return** $SUCCESS$
24: Make the struct page GPA range read-write
25: Unlock MUTEX
26: **return** FAIL

Algorithm 2 The procedure of page fault handler

Input: $GUEST$, GPA

1: **procedure** PAGEFAULTHANDLER
2: Lock MUTEX
3: $RANGE \leftarrow$ GetReclaimedRange($GUEST$, GPA)
4: $PAGES \leftarrow$ pages reallocated for $RANGE$
5: Map $RANGE$ to $PAGES$ in EPT
6: Map $RANGE$ to $PAGES$ in IOMMU
7: Remove $RANGE$ from the reclaimed set
8: Make the struct page GPA range read-write
9: Unlock MUTEX
10: **return**

Input: $GUEST$, GPA
Output: $RANGE$

11: **procedure** GETRECLAIMEDRANGE
12: **for** each $RANGE$ in the reclaimed set of $GUEST$ **do**
13: $SP_{st} \leftarrow$ the start struct page GPA in $RANGE$
14: $SP_{en} \leftarrow$ the end struct page GPA in $RANGE$
15: **if** $SP_{st} \leq GPA \leq SP_{en}$ **then**
16: **return** $RANGE$

9-25). The core reclamation logic makes the *struct page* GPA range read-only (Line 10), invalidates the memory mapping in EPT (Extended Page Table) and IOMMU page table for the reclaimed GFN range (Lines 15-19), adds this range to the reclaimed set (Line 20), and finally, releases the reclaimed pages to the hypervisor (Line 21). Algorithm 1 uses the mutex (Line 9) and the page status double check logic (Lines 11-14) to avoid synchronization problems, which will be further explained in Section 4.5.

4.4 Page Fault Handler

Since we have modified the EPT mapping of the *struct page* GPA range to **read-only** for the reclaimed GFNs, when the guest tries to allocate these pages, it will **write** to the corresponding *struct page* to modify the bytes of page status flags. Thus a page fault is triggered. The pseudocode of the page fault handler is shown in Algorithm 2. It has two input parameters: $GUEST$ and *GPA*, which respectively indicate the guest and the accessed GPA that triggers the page fault. First, it calls *GetReclaimedRange* to find the reclaimed memory range responsible for *GPA* (Line 3). Then it allocates pages (Line 4), remaps the EPT and IOMMU page table for this range (Lines 5-6), removes the range from the guest's reclaimed memory set (Line 7), and resumes the EPT mapping of the *struct page* GPA range to **read-write** (Line 8).

Function *GetReclaimedRange* is simplified as a loop for ease of demonstration. We use the rbtree (Red-black Tree), which is an implementation of the binary search tree in Linux, to optimize the performance of memory range inserting, deleting, and searching.

4.5 Synchronization Problem

One challenge for the reclamation process is the synchronization problem: the guest may allocate the pages while we are reclaiming them, and the status of the guest *struct page* may change from "free" to "allocated" within the memory reclamation routine, which may cause inconsistency in the system.

Figure 7 show two cases to illustrate how the mutex and double-checking in Algorithm 1 (Lines 9-14) and Algorithm 2 (Line 2) avoids the racing conditions. In reclaim ①, if the guest allocates the free pages after the Hypervisor makes the corresponding *struct page* GPA range read-only, it will trigger page fault and the mutex will prevent the page fault handler to fix the memory mapping before the hypervisor finishes reclamation. This guarantees the atomicity of the memory reclamation process and the page fault handling. In reclaim ②, if the guest allocates the free page before the Hypervisor

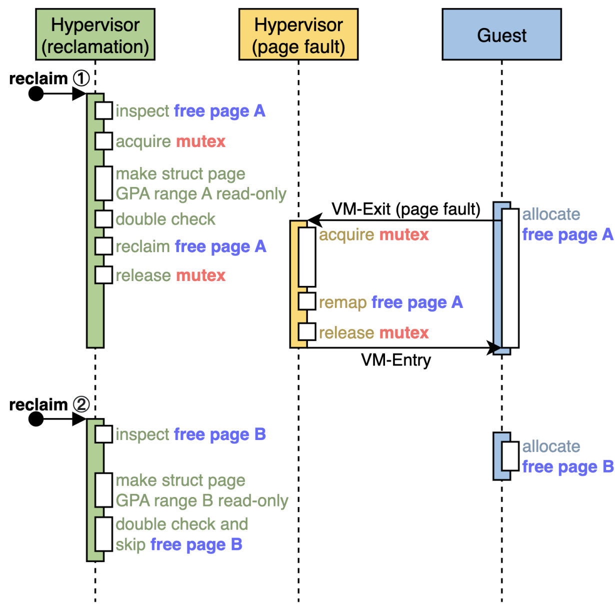


Figure 7: Two cases to illustrate how V-Probe solves the synchronization problem.

makes the corresponding *struct page* GPA range read-only, the double-checking will detect such behavior and skip the page, so as to avoid the inconsistency of the page status knowledge.

5 Evaluation

In this section, we conduct experiments to evaluate V-Probe in different aspects. First, we evaluate V-Probe's overhead in memory reclamation and reallocation tasks. Then we evaluate its impact on workload performance using both the micro-benchmark and macro-benchmark.

The experiments are based on the hardware architecture of Intel® Xeon® Platinum 8163 CPU. The hypervisor is based on QEMU and KVM. And the memory allocator manages the underlying memory of guest VMs in 2MB granularity. The guests in our experiments run Linux OS, each equipped with 2 CPU cores and 4GB main memory.

We use Ballooning and the IOPF based method for comparison. For simplicity, we will call the IOPF based method as IOPF in the following explanation. For memory reclamation and reallocation tasks, we compare V-Probe with both Ballooning and IOPF. For micro-benchmark and macro-benchmark tests, we only use IOPF for comparison. We do not use Ballooning for these benchmarks because the frontend driver of Ballooning will hold the reclaimed free pages and make them unallocatable in the guest. So if the memory usage of the task does not exceed the remaining allocatable memory in the guest, the performance will be the same as the baseline with no memory reclamation. On the other hand, if the memory usage of the task exceeds the remaining allocatable

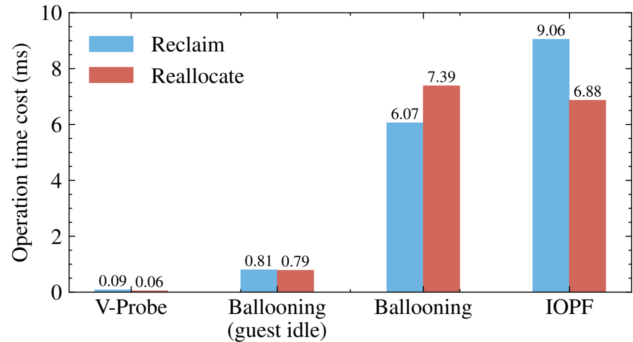


Figure 8: The time cost of V-Probe, Ballooning and IOPF to reclaim (reallocate) 2MB memory from (to) the guest. We run *sysbench* CPU workloads in the guest to emulate the guest's CPU stress.

memory in the guest, the guest will kill the task because of the OOM (Out of memory) error.

We reclaim 30% of the server's memory at the beginning of the each benchmark test. In practice, to avoid host memory insufficiency, the memory reclamation policies need to limit the quantity of memory reclaimed from each guest and the quantity of memory that can be reused by other VMs. 30% of memory reclamation is an extremely high value in practice. In this configuration, the benchmark results reveal the performance impact limit of V-Probe in a real-world environment.

Since IOPF is not commonly supported by off-the-shelf I/O devices, we implement IOPF by ourselves using FPGA-based SmartNIC. The storage media we use for memory swapping is a hard disk drive, with 250MB/s I/O throughput. Notice that other methods relying on memory swapping (vIOMMU, coIOMMU) share similar results with IOPF.

5.1 Overhead

5.1.1 Data Registration

The overall data registration process takes less than 1 second. The main overhead comes from the helper function compilation and page meta-data layout detection within the guest. The overhead of data transferring is small as the size of the registered data is only 232 bytes in our implementation, which includes the binaries of the helper functions, the page meta-data layout information, and the extra API-related fields. Also, as the helper functions are simple and small, the verification cost is negligible. Data registration only runs one time after the guest finishes starting up, and its overhead will not impact the following memory reclamation procedure.

5.1.2 Memory Reclamation and Reallocation

We evaluate the overhead of V-Probe in the memory reclamation (reallocation) task by measuring the time it takes to

reclaim (reallocate) 2MB memory from (to) the guest. For memory reclamation, the cost refers to the time elapsed between the initiation of the memory reclamation request and the moment the hypervisor releases the reclaimed memory back to the host. For memory reallocation, the cost refers to the time elapsed between the initiation of the memory reallocation request and the point at which the hypervisor establishes the new memory mapping.

We use Ballooning and IOPF for comparison. During memory reclamation (reallocation), we run *sysbench* CPU workloads in the guest to emulate the guest's CPU stress. Specifically, we run the command `'sysbench cpu -threads=$(nproc) -time=0 run'` in the guest to make all of the CPUs busy. We also assess the performance of Ballooning when the guest OS is idle, in order to determine the upper limit of its performance. For each measurement, we run the operation 10 times and present the average value as the result.

We will take the memory reclamation task as an example to analyze the experiment results. The memory reallocation task shares a similar analysis. As Figure 8 shows, V-Probe takes only 0.09ms to reclaim 2MB memory on average, while Ballooning and IOPF take 6.07ms and 6.88ms respectively. We analyze the reasons why V-Probe is two orders of magnitude faster than the other two methods as follows:

- Ballooning relies on the communication with the frontend driver running in the guest when reclaiming memory from it. The communication introduces a large delay, especially when the guest is busy with CPU intensive tasks, which reduces the chance for the thread of the frontend driver to gain CPU time slice. The overhead of communication (5.79ms) occupies 95.4% of the total overhead (6.07ms) when using Ballooning to reclaim 2MB memory. Even if the CPU is not busy in the guest, the communication overhead (0.53ms) also occupies 65.4% of the total overhead (0.81ms).

- IOPF relies on memory swapping to avoid guest memory corruption. Memory swapping introduces a large overhead of disk I/O (8.95ms), which occupies 98.8% of the total overhead (9.06ms).

- When triggered by events, V-Probe can use the injected helper functions to inspect the guest free pages actively, without the time-consuming communication with the guest. At the same time, since the reclaimed pages are free memory in the guest and do not contain valuable data, V-Probe does not need memory swapping, which avoids the disk I/O overhead.

The memory overhead of V-Probe's reclamation process is also small. For each reclaimed memory range, we allocate a 64B data structure to maintain the necessary data, which includes the start/end GPA of the *struct page* and the physical page, and other data-structure-specific fields. The overhead is 64B/2MB = 0.031‰ for a 2MB area and 64B/4MB = 0.015‰ for a 4MB area.

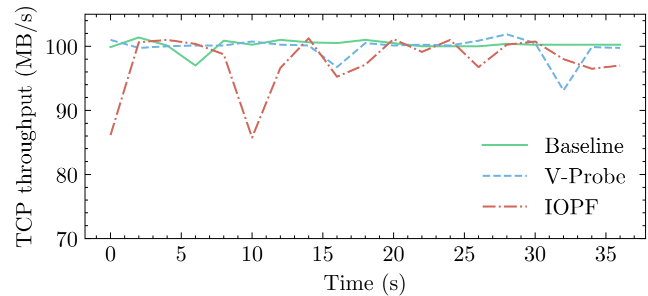


Figure 9: The TCP throughput of the baseline, V-Probe, and IOPF.

5.2 Micro-benchmark

The micro-benchmarks in our evaluation include TCP throughput and UDP latency, as these two metrics are critical for many important applications (*e.g.*, file servers, databases, and key-value stores). The evaluation requires a client-server experiment setting. We measure the performance impact of V-Probe on these metrics when a large amount of memory is reclaimed from the server side and high memory footprint is triggered by the network stream. We compare V-Probe with the baseline (*i.e.,* no memory is reclaimed from the guest) and IOPF.

Although there exist many network performance benchmark tools (*e.g.*, *netperf*), they only focus on the efficiency of the network stack and have a low memory footprint, which means they cannot cover the logic of the memory refault logic (*i.e.*, trigger page fault and reallocate memory). So we design and implement our own micro-benchmarks, which have a high memory footprint while measuring the TCP throughput and UDP latency.

5.2.1 TCP Throughput

To evaluate TCP throughput, we run a client and a server in two separate VMs, both equipped with 2 CPU cores and 4GB main memory. The server serves a 4GB file and the client will download the file when running the experiment. 30% of the server's memory is reclaimed at the beginning of the test. When the client downloads the file, the server will read the file from the disk to the in-memory page cache. As Linux drops the page cache in an on-demand manner, the server will fill the memory with the page cache until the memory is insufficient. This implements a high memory footprint of the experiment and will trigger page refault in the serve. We record the download speed as the TCP throughput.

Figure 9 shows the throughput changes over time in different settings. As the figure shows, the throughput fluctuation

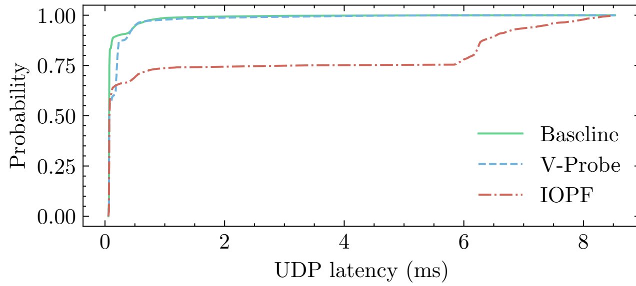


Figure 10: The CDF of the UDP latency when applying the baseline, V-Probe and IOPF.

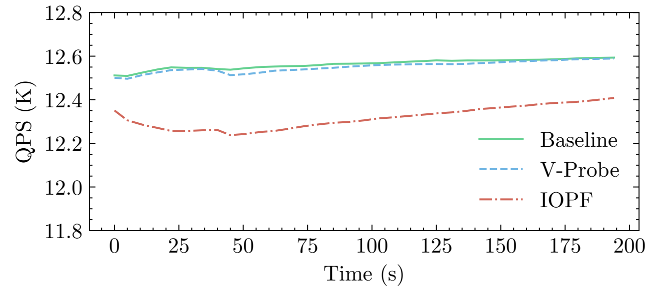


Figure 11: The Redis QPS when applying baseline, V-Probe, and IOPF.

of the baseline and V-Probe are smaller than that of IOPF. The minimum throughput of the baseline, V-Probe and IOPF are 97MB/s, 93.1MB/s, and 85.8MB/s respectively. And the average value is 100.2MB/s, 99.8MB/s, and 97.5MB/s respectively. These results show that V-Probe may degrade the TCP throughput slightly compared with the baseline, but it is much better than IOPF.

5.2.2 UDP Latency

To evaluate UDP latency, we run a client and a server in two separate VMs, both equipped with 2 CPU cores and 4GB main memory. 30% of the server's memory is reclaimed at the beginning of the test. The logic of the UDP server is quite simple: It listens on a port, accepts a UDP connection request, allocates a 2MB buffer and accesses it, then sends a response back to the client. This is to simulate a UDP application with a high memory footprint. Allocating and accessing buffers may cause page faults and page reallocations if the underlying memory is reclaimed by the hypervisor. The UDP client records the latency of each request as the results.

Figure 10 shows the CDF (Cumulative Distribution Function) of different settings. The 90th percentile latency of the baseline, V-Probe and IOPF is 0.23ms, 0.39ms and 6.55ms respectively, and the 99th percentile latency is 1.36ms, 2.43ms and 8.27ms respectively. The overhead of V-Probe is 69.6% and 79.4% in the 90th and 99th percentile latency, respectively, when compared to the baseline. However, it significantly outperforms IOPF, which introduces approximately 27X and 5X overhead in the 90th and 99th percentile latency.

5.3 Macro-benchmark

We use Redis, a widely used in-memory key-value database as the macro-benchmark to evaluate V-Probe. We run a client and a server in two separate VMs, both equipped with 2 CPU cores and 4GB main memory. At the beginning of the experiment, we remove all the data from the Redis server database and reclaim 30% memory from the server. The client runs `redis-benchmark`, the official Redis benchmark tool, to continuously send `SET` commands to the server. The `SET` command will randomly write key-value pairs to the in-memory database. There are 1,000,000 random keys, and the value size of each `SET` operation is 2KB. We record the QPS (queries per second) and the operation latency during the test. We only test the `SET` workload because V-Probe is a free-page-based method, and data insertion is necessary to trigger guest page allocation, and thus necessary to trigger V-Probe's memory reallocation. If we use the `GET` workload to evaluate V-Probe, the data in Redis will always stay in the main memory, with no page fault or memory reallocation operations will not utilize the functionality of V-Probe.

Figure 11 shows the QPS over time in the different settings. As it shows, the result of V-Probe (12.55K) is close to the baseline (12.56K). And the result of IOPF (12.31K) is about 2% lower than the other two. For the tail latency of each operation (not shown in the figure), the baseline and V-Probe exhibit a 99.99th percentile latency of 1ms, while the IOPF exhibits a noticeably higher 99.99th percentile latency of 7ms (the precision of the latency data is limited to the `redis-benchmark` tool). This indicates the Redis performance while applying V-Probe for memory reclamation is close to the case with no memory reclamation and better than applying IOPF.

6 Discussion

6.1 Compatibility

In this paper, we present a Linux-based implementation of V-Probe which is highly straightforward for two reasons. First, the V-Probe injector, including the helper functions, is lightweight (140 LOC (Lines of Code)), and it is dynamically compiled after the startup of the guest. This means that the V-Probe injector does not need to change for different releases of the Linux kernel. Second, V-Probe does not hack any API in the guest OS. This implies that V-Probe does not enforce any changes on the guest OS and retains full compatibility for guest OSes.

V-Probe relies on the continuous arrangement of the guest page meta-data. Accordingly, while in this paper we examine

a Linux-based implementation of V-Probe, the adoption of V-Probe is similar in Unix-like OSes that use a similar memory management mechanism to Linux. For example, FreeBSD uses ranges of *struct vm_page* (like *struct page* in Linux) to arrange the page meta-data and uses the buddy system to manage them.

V-Probe introduces about 1,300 LOC to the hypervisor, as the logic of V-Probe on the hypervisor side is much more complex. But this does not impact the practice of V-Probe as most of the cloud providers use customized hypervisors, and V-Probe can be easily integrated into these software systems.

V-Probe introduces high hardware compatibility as it only relies on the basic virtualization features of the CPU. These features are widely supported by popular CPU platforms such as Intel, AMD, Arm, and RISC-V. Furthermore, V-Probe does not rely on IOPF which is not commonly supported by off-the-shelf I/O devices.

6.2 Fine-grained Page Meta-data Management

Using fine-grained page meta-data management, V-Probe can effectively avoid DMA failures in I/O passthrough enables VMs. At the same time, the idea of fine-grained page meta-data management can also be combined with other free page reclamation methods to avoid DMA failures. For example, in Free Page Reporting, except for the free pages, the guest can also report the page meta-data to the hypervisor. So by managing the access mode of the guest page meta-data's GPA range, Free Page Reporting can avoid DMA failures in the same way V-Probe does. But unlike V-Probe, which uses helper functions to inspect guest free pages without notifying the guest, these methods rely on the frontend drivers to obtain the guest page status, which means the guest communication overhead is unavoidable.

6.3 Threats to Validity

There are two threats to the validity of our work. First, this paper defines and examines an example implementation of the V-Probe design that is specific to Intel X86_64 architecture and Linux-based OSes. Therefore, architecture-based and OS-based assumptions of this example implantation would need to be adjusted when moving to other architectures and OS types. However, since V-Probe relies on the continuous arrangement of the guest page meta-data, V-Probe may not apply to the OSes that use different ways to arrange the page meta-data.

Second, when reclaiming free pages, V-Probe needs to modify the memory mapping for the page meta-data GPA range. But since memory mapping is by the granularity of pages, the meta-data should reside on the same physical page. So V-Probe can only reclaim continuous free pages in batches. As a result, guest free page fragmentation may weaken the memory reclamation effect of V-Probe. But free page fragmentation in the guests is usually caused by high memory pressure, which suggests the system not to reclaim memory from them. Also, this problem can be mitigated by the memory compaction functionality in the guest OS.

7 Conclusion

In this paper, we conduct a systematic survey of previous VM memory reclamation methods and analyze their relationship with the widely deployed I/O passthrough technique. Based on the analysis, We propose V-Probe, a non-intrusion and efficient approach to achieve memory overcommitment for I/O passthrough enabled VMs. Using fine-grained page meta-data management, V-Probe enables the hypervisor to actively inspect and reclaim the guest free pages while guaranteeing DMA-safety. We implement V-Probe and evaluate its efficiency in different aspects. Experiment results show that the overhead of V-Probe is on the micro-second scale and it has low performance impact on the guest workload. It also has high compatibility with different hardware platforms and a wide range of Linux kernel releases, which simplifies the deployment.

References

[1] Abdullah Aljumah and Mohammed Altaf Ahmed. Design of high speed data transfer direct memory access controller for system on chip based embedded products. *Journal of Applied Sciences*, 15(3):576–581, 2015.

[2] AMD. Amd i/o virtualization technology (iommu) specification, 2021. https://developer.amd.com/wp-content/resources/48882_IOMMU_3.05_PUB.pdf.

[3] Ardalan Amiri Sani, Kevin Boos, Shaopu Qin, and Lin Zhong. I/o paravirtualization at the device file boundary. *ACM SIGARCH Computer Architecture News*, 42(1):319–332, 2014.

[4] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, Assaf Schuster, et al. viommu: efficient iommu emulation. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 73–86, 2011.

[5] Nadav Amit, Abel Gordon, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. Bare-metal performance for virtual machines with exit-less interrupts. *Communications of the ACM*, 59(1):108–116, 2015.

[6] Nadav Amit, Dan Tsafrir, and Assaf Schuster. Vswapper: A memory swapper for virtualized environments. *Acm Sigplan Notices*, 49(4):349–366, 2014.

[7] Nadav Amit and Michael Wei. The design and implementation of hyperupcalls. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 97–112, 2018.

[8] Arm. Arm system memory management unit architecture specification, 2021. https://developer.arm.com/documentation/ihi0070/latest.

[9] Kapil Arya, Yury Baskakov, and Alex Garthwaite. Tesseract: reconciling guest i/o and hypervisor swapping in a vm. *Acm Sigplan Notices*, 49(7):15–28, 2014.

[10] S Anish Babu, MJ Hareesh, John Paul Martin, Sijo Cherian, and Yedhu Sastri. System performance evaluation of para virtualization, container virtualization, and full virtualization using xen, openvz, and xenserver. In *2014 fourth international conference on advances in computing and communications*, pages 247–250. IEEE, 2014.

[11] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[12] Muli Ben-Yehuda, Michael D Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The turtles project: Design and implementation of nested virtualization. In *USENIX Symposium on Operating System Design and Implementation (USENIX OSDI)*, volume 10, pages 423–436, 2010.

[13] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 45–58, 2009.

[14] Khien-mien Chew and Avi Silberschatz. On the avoidance of the double paging anomaly in virtual memory systems. 1992.

[15] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. High performance network virtualization with sr-iov. *Journal of Parallel and Distributed Computing*, 72(11):1471–1480, 2012.

[16] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking:{SmartNICs} in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, 2018.

[17] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 193–206, 2003.

[18] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular disco: Resource management using virtual clusters on shared-memory multiprocessors. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 154–169, 1999.

[19] Fei Guo, Seongbeom Kim, Yury Baskakov, and Ishan Banerjee. Proactively breaking large pages to improve memory overcommitment performance in vmware esxi. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 39–51, 2015.

[20] Chien-Chin Huang, Gu Jin, and Jinyang Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1341–1355, 2020.

[21] Tai-Yi Huang, JW-S Liu, and Jen-Yao Chung. Allowing cycle-stealing direct memory access i/o concurrent with hard-real-time programs. In *International Conference on Parallel and Distributed Systems (ICPADS)*, pages 422–429. IEEE, 1996.

[22] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 295–306, 2014.

[23] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design guidelines for high performance {RDMA} systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, 2016.

[24] Donghyuk Lee, Lavanya Subramanian, Rachata Ausavarungnirun, Jongmoo Choi, and Onur Mutlu. Decoupled direct memory access: Isolating cpu and io traffic by leveraging a dual-data-port dram. In *International Conference on Parallel Architecture and Compilation (PACT)*, pages 174–187. IEEE, 2015.

[25] Ilya Lesokhin, Haggai Eran, Shachar Raindel, Guy Shapiro, Sagi Grimberg, Liran Liss, Muli Ben-Yehuda, Nadav Amit, and Dan Tsafrir. Page fault support for network controllers. *ACM SIGARCH Computer Architecture News*, 45(1):449–466, 2017.

[26] Ilya Lesokhin and Dan Tsafrir. *I/O Page Faults*. PhD thesis, Computer Science Department, Technion, 2015.

[27] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *USENIX Symposium on Operating System Design and Implementation (USENIX OSDI)*, volume 4, pages 17–30, 2004.

[28] Jiaxin Li, Dongsheng Li, Yuming Ye, and Xicheng Lu. Efficient multi-tenant virtual machine allocation in cloud data centers. *Tsinghua Science and Technology*, 20(1):81–89, 2015.

[29] Shuang Liang, Ranjit Noronha, and Dhabaleswar K Panda. Swapping to remote memory over infiniband: An approach using a high performance network block device. In *2005 IEEE International Conference on Cluster Computing*, pages 1–10. IEEE, 2005.

[30] Duo Liu, Kan Zhong, Xiao Zhu, Yang Li, Lingbo Long, and Zili Shao. Non-volatile memory based page swapping for building high-performance mobile devices. *IEEE Transactions on Computers*, 66(11):1918–1931, 2017.

[31] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. Software fault isolation with api integrity and multi-principal modules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 115–128, 2011.

[32] Antonio Marsico, Roberto Doriguzzi-Corin, and Domenico Siracusa. An effective swapping mechanism to overcome the memory limitation of sdn devices. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 247–254. IEEE, 2017.

[33] KAZUHIKO OHMACHI, THORU NISHIGAKI, and SHIGEO TAKASAKI. Analysis of pawp/vms: Paging algorithm to prevent double paging anomaly in virtual machine systems. *J. Inform. Processing*, 4(2):55–60, 1981.

[34] PCI-SIG. Address translation services revision 1.1, 2009. http://www.pcisig.com/specifications/iov/ats/.

[35] James L Peterson and Theodore A Norman. Buddy systems. *Communications of the ACM*, 20(6):421–431, 1977.

[36] Mendel Rosenblum and Tal Garfinkel. Virtual machine monitors: Current technology and future trends. *Computer*, 38(5):39–47, 2005.

[37] Assaf Schuster, Nadav Amit, and Dan Tsafrir. Memory swapper for virtualized environments, November 7 2017. US Patent 9,811,268.

[38] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary {CPU} architectures. In *19th USENIX Security Symposium (USENIX Security 10)*, 2010.

[39] SK Shravan, J Lakshmi, and Neeraj Bisht. Towards improving data center utilisation by reducing fragmentation. In *IEEE International Conference on Cloud Computing (CLOUD)*, pages 941–945. IEEE, 2018.

[40] Kun Tian, Yu Zhang, Luwei Kang, Yan Zhao, and Yaozu Dong. coiommu: A virtual iommu with cooperative dma buffer tracking for efficient memory management in direct i/o. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 479–492, 2020.

[41] Cheng-Chun Tu, Michael Ferdman, Chao-tung Lee, and Tzi-cker Chiueh. A comprehensive implementation and evaluation of direct interrupt delivery. *Acm Sigplan Notices*, 50(7):1–15, 2015.

[42] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L Santoni, Fernando CM Martins, Andrew V Anderson, Steven M Bennett, Alain Kagi, Felix H Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.

[43] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 203–216, 1993.

[44] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *USENIX Symposium on Operating System Design and Implementation (USENIX OSDI)*, pages 181–194, 2002.

[45] Wang Wei. Provide support for free page reporting, 2020. https://lwn.net/Articles/808807/.

[46] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 87–104, 2015.

[47] Zhen Xiao, Weijia Song, and Qi Chen. Dynamic resource allocation using virtual machines for cloud computing environment. *IEEE Transactions on Parallel and Distributed Systems*, 24(6):1107–1117, 2012.

[48] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy (SP)*, pages 79–93. IEEE, 2009.

[49] Pengfei Zhang, Xi Li, Rui Chu, and Huaimin Wang. Hybridswap: A scalable and synthetic framework for guest swapping on virtualization platform. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 864–872. IEEE, 2015.

[50] Qi Zhang and Ling Liu. Shared memory optimization in virtualized cloud. In *2015 IEEE 8th International Conference on Cloud Computing*, pages 261–268. IEEE, 2015.