Imperial College London

BEng Interim Report

Imperial College London

Department of Computing

Automated Performance Optimisation in Virtual Machines

Author:
Venkata Sai Sri Varshitha Pathapati

Supervisor:
Prof. Lluís Vilanova

Second Marker:

January 23, 2025

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Virtual Machines and Virtualization techniques are used for isolation, security and better utilisation of resources. However, they have a overhead of memory address translation for many workloads. Current techniques for address translation in VMs include nested paging and shadow paging which either require greater memory accesses or high hypervisor intervention. Extensive research is being done in this area to minimise the overhead of address translation in VMs for them to have native performance.
One such example of research, is the Translation Pass Through (TPT) which aims to achieve near-native address translation from the guest virtual address to the host physical address by using new hardware support of physical frame tagging in commodity CPUs. However, TPT at the current implementation, if TPT is enabled for the guest, each guest process needs to be enabled manually at the start of the process, for which we need to know if an application would benefit from using TPT address translation before its execution. The aim of this project is to automate the process by identifying such applications through profiling using linux perf and other APIs and dynamically enabling TPT to achieve near-native performance for an application/process in the guest OS.

Chapter 2

Background

In this chapter, we outline some key concepts related to virtualization, VMs, hypervisors, and address translation to set the background for understanding the concerns in achieving near-native memory address translation in Virtual Machines (VMs). We begin by reviewing basic concepts of virtualization, virtual machines, address translation mechanisms in virtual machines, and profiling/performance monitoring. Throughout this section, we will introduce key terminology that will be used in the rest of the report.

2.1 Virtualization and Virtual Machines (VMs)

Virtualization is the process of creating a software-based or "virtual" version of a computer, with some allocated resources (CPU, memory, storage, network) provided by the **host system**, which could be a personal computer or a remote server [7]. This represents an abstraction or layering principle, whereby the virtual exposed resource is made identical to the underlying physical resource being virtualized [8].
A **Virtual Machine (VM)** is a complete computing environment that is isolated from the rest of the system. It has its own processing capabilities, memory, and communication channels [8]. Each VM operates as a separate environment and often runs a different operating system (OS). VMs are partitioned from the host system to ensure that applications within a VM cannot interfere with the host or other VMs. VMs are sometimes referred to as **guests** or **guest systems**, and in this report, these terms are used interchangeably. General use cases for VMs include building and deploying applications to the cloud, experimenting with new OS features, accessing virus-infected data, and much more.
Since VMs are independent of each other and the host, VMs have numerous advantages which are [7, 8, 9]:

- **Cost Efficiency:** VMs eliminate the need for purchasing new physical servers to run different OSes or configurations. Multiple VMs can be created on a single server, enabling resource sharing and maximizing the utilization of available hardware.
- **Isolation and Security:** Each VM operates in its own isolated environment, preventing malware or failures in one VM from affecting other VMs or the host machine. This isolation leads to fault tolerance and increased security.
- **Portability:** VMs are cross-platform portable, meaning they can run on various host machines. The migration process between hosts is relatively easy to achieve. As such, applications can be deployed across different environments without requiring modifications.

2.1.1 Hypervisor

A **hypervisor** or a **Virtual Machine Monitor (VMM)** is a software that creates and runs VMs. A hypervisor allows one host to support multiple guests by virtually sharing resources such as memory and CPU [10]. Sometimes in literature *hypervisor* is used to refer to the software that manages and runs the VMs and *VMMs* are used to refer to the portion of the Hypervisor that focus on CPU and Memory virtualization [8]. However, this this report, these terms are used

interchangeably. When multiples VMs are being run on the host at the same time, the hypervisor multiplexes (ie. allocates and schedules) the physical resources among VMs fairly.

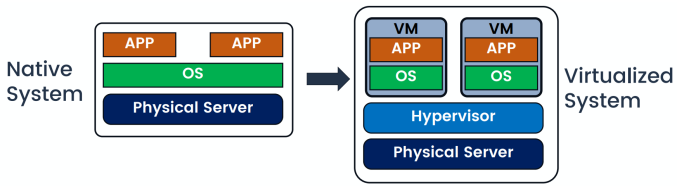

Figure 2.1: Comparison of a Native system and a Virtualized system. Virtualized systems run multiple guests

There are two main types of hypervisors as discussed below.

Type-1 Hypervisor

Type-1 hypervisors act like light-weight OS that run on the host machine directly [10]. Type-1 Hypervisor is in direct control of all resources of the host since it runs on the **bare machine**. Hence, such hypervisors are also responsible for resource allocation and scheduling as well as virtualization. **Kernel-based Virtual Machine (KVM)** is an open-source virtualization technology for Linux, and is a part of Linux Kernel, this effectively converts the host OS to a Type-1 Hypervisor [11].

Type-2 Hypervisor

Type-2 hypervisors runs as a software layer on an OS, like any other computer program/application. Unlike Type-1 hypervisors, type-2 hypervisors are not concerned with resource allocation and scheduling, they are handled by the OS, hence the hypervisor is only responsible for the virtualization of the resources allocated. Examples of such hypervisors are Oracle VirutalBox, VMWare Workstation [11].

2.2 Address Translation in Non-Virtualized x86-64

Virtual Memory is an abstraction of the physical memory. Paging is when we eliminate the need to allocate contiguous memory locations, instead the physical memory is divided into equal sized (4KiB) *frames* and the applications virtual memory is also divided into equal sized (4KiB) *pages*. When a process requests memory, OS allocates some frames to the process which are mapped to the processes virtual address space. Each application/program addresses memory using the **Virtual Address (VA)**. In x86-64, the translation between **Virtual Address (VA)** to **Physical Address (PA)** is performed by using hierarchical address-translation tables referred to as **page table**, usually containing 4 levels, as in 2.2. Page Table is maintained by the OS and is traversed by the Page Table Walker (PTW) unit in the Memory Management Unit (MMU) to translate the **VA** to **PA**.
The base address of the page table is stored in register **CR3** [6]. A **page walk** is an iterative process performed by the hardware (Page Table Walker) to translate the VA to PA, this requires 4 memory accesses, one for each level of the radix tree traversal. Since, it is expensive to access memory multiple times because each memory access takes several clock cycles. The MMU caches the VA to PA translation in **Translation Lookaside Buffer (TLB)** which is used for fast look up of the mapping. On a TLB miss, a page-walk is performed to translate the VA to PA.

2.3 Address Translation in Virtualized x86-64

In Virtualized environments (ie. for applications running on the VM), there are 2 levels of address translation [3].

- **gVA to gPA(hVA) :** guest Virtual Address to guest Physical Address which is the same as the Host Virtual Address. This mapping is held in the **guest Page Table (gPT)** maintained by the VM.

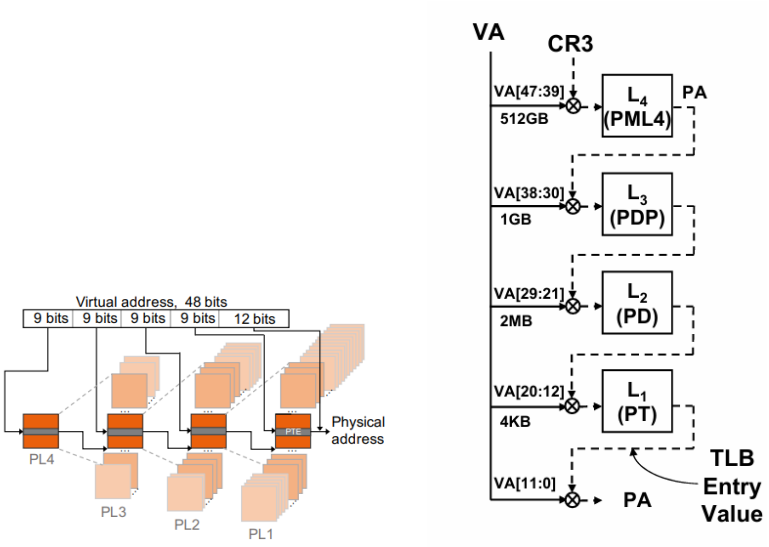

Figure 2.2: Page table as 4-level radix tree in x86-64 [1]

- **gPA(hVA) to hPA :** guest Physical Address which is the same as host Virtual address to host Physical Address. This mapping is maintained by the **host Page Table (hPT)** which is maintained by the hypervisor.

Current VMs use one of the three machanisms which are Nested Paging, Shadow Paging and Paravirtualization. These are discussed in the following sections.

2.3.1 Nested Paging

Nested Paging is supported by Intel EPT or AMD nPT [2], these terms could be used interchangeability. This is a very common hardware accelerated approach that performs gPA to hPA using two hierarchies of page tables, both the gPT and hPT mentioned above. Every gPA in the gPT requires a gPA to hPA translation by the MMU. This process is called a **2-D (two dimensional) Page Walk** [3, 2, 6].

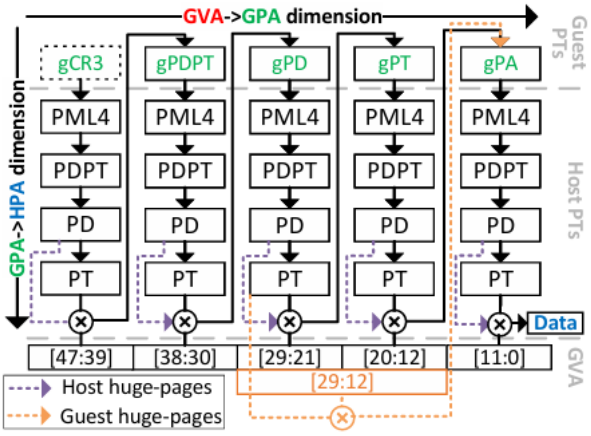

Figure 2.3: 2D page table walk for Nested Paging [2]

2-D page walk, for a single address translation with m levels of gPT and n levels of hPT, each level of the gPT traverses all levels of the hPT (mn accesses in total), and $n + m$ accesses to the contents of the respective page table, hence requiring a total of $mn + n + m$ memory accesses. In x86-64, with 4 level page table with 4KB pages, that is 24 memory accesses on a TLB miss [8].

With an increase in the working set of applications and the depth of the radix tree used for page tables, this could lead to a quadratic increase in memory virtualisation overhead [2].
Despite the large overhead for the translation, Nested Paging is still a popular approach because it enables guests to perform the page table updates without the intervention of the hypervisor which is the approach used in Shadow Paging (discussed next).

2.3.2 Shadow Paging

This is a lesser used technique for address translation in VMs. With Shadow Paging the hypervisor creates a **Shadow Page Table (sPT)** on demand by merging the gPT and hPT which holds the complete translation from the gVA to hPA [3]. On a TLB miss, the PTW (Page Table Walker) performs a 1D page walk on the sPT. The native page table pointer register points to the sPT. Thus, on a TLB miss for x86-64, requires 4 memory accesses similar to a non-virtualised/native address translation.

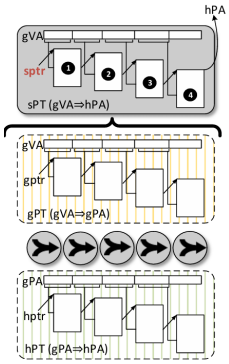

Figure 2.4: sPT created by merging gPT and hPT [3]

However, this technique doesn't allow updates to the page tables (gPT and hPT) since the sPT should be kept consistent. The hypervisor write-protects the gPT such that every write to the gPT is traps (VMTraps) into the hypervisor to update the sPT. These hypervisor intervention causes significant overhead for applications. In intervention of the hypervisor occurs on every update to the gPT and also privileged guest instructions such as TLB flushes [2]. This carries a significant overhead for maintenance of the sPT despite achieving near native performance on a TLB miss.

2.3.3 Paravirtualization of MMU

This is no longer a commonly used technique for address translation in VMs. It predates modern techniques such as Nested Paging and Shadow Paging. In a paravirtualized system like Xen-PV [2], the guest OS interacts with the hypervisor via hypercalls to request changes in the gVA to hPA page table managed by the hypervisor. These hypercalls have a significant overhead similar to shadow paging VMTraps.

2.4 Performance Monitoring

Most processors provide a set of performance counters to measure key micro-architecture events such as TLB misses, branch mis-predictions, cache hits, cache misses and many others [12]. These events are crucial to analyse the performance of any software/hardware performance analysis. We would like to carry out performance monitoring to investigate the address translation overhead for virtualized applications and activate some new hardware acceleration extensions that was developed in the recent research called TPT (Translation Pass Though) [2], refer Chapter 3.5 for further details.

2.4.1 Performance Monitoring Unit (PMU)

The performance counters provided by the processor are implemented in silicon by a hardware unit called Performance Monitoring Unit (PMU). PMUs are found in processor cores, cache controllers,

memory controllers, graphic cards and IO devices which provide crucial data on how hardware resources are being used by software and VMs. Interpretation of such data can help us identify bottlenecks and hint on how to eliminate them, thus improving the performance. [12]

2.4.2 Linux Perf

Linux Perf is a profiling tool for the Linux kernel also called Performance Counter for Linux (PCL) or perf_events [4]. perf is part of the Linux Kernel which can be used to monitor performance with some commands. To view all available performance events using the Linux Perf tool, you can run the command `perf list` in the terminal. There are commands for counting (`perf stat`), profiling (`perf record`), static and dynamic tracing (`perf probe`), reporting and many others from different sources such as such as hardware counters, kernel tracepoints, and software-defined events, referenced in 2.5.

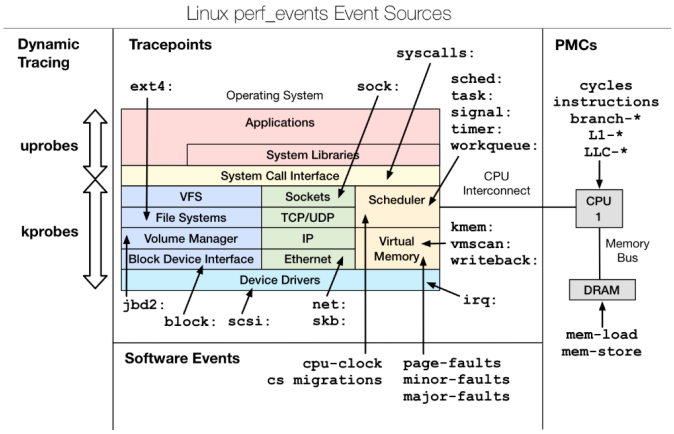

Figure 2.5: perf_events sources [4]

2.4.3 Intel PCM

Intel Performance Counter Monitor is an API to monitor various statistics for Intel processors through command line for real-time monitoring such as CPU utilisation and cycles, cache performance, energy consumption and others. PCM uses the hardware performance counters available in Intel processors to gather information about system performance. For more information, refer to the Intel Performance Counter Monitor (PCM) repository [13].

2.4.4 Other Performance Monitoring APIs

There are several other high-level performance monitoring tools available on linux. Some examples include `htop` or `top` for monitoring resource utilisation including CPU, memory, swaps. Another example is to use `vmstat` to monitor page faults or other memory related performance counters. This is not an exclusive list of Kernel API but just a selection of few commonly used.

2.5 Conclusion

In conclusion, memory address translation in VMs introduces non-trivial performance overheads. These overheads are expected to grow as applications move to larger working sets as they will have higher TLB miss rate and architectures using deeper radix trees for page table design to support more physical memory [2]. For address translation, both management of page tables and the number of memory accesses is needed performance. However, with the 2 major current implementations, we have one at the expense of another.
We have presented a brief overview about profiling in general and some tools that could be used, these would be used and discussed further in the later sections in the report. This section provides an overview about performance counters.

Chapter 3

Related Work

The aim of this chapter is to conduct a extensive review of the academic literature on the topic of Address translation in VMs from the gVA to the hVA. In order, we will cover the current-state-of-the-art research to improve the performance of address translation in traditional nested paging and shadow paging discussed in the earlier chapter. We will focus on multiple implementations that lead to an improvement in performance of address translation namely Huge Pages, Agile Paging, DVMT, caching and most importantly TPT (3.5).

3.1 Huge Pages

Huge Pages is a relatively simple idea of reducing the overhead of address translation for nested paging by making the page sizes larger for memory intensive applications. Instead of the traditional 4KiB page size for most x86_64 systems, we have relatively larger pages such as 2MiB or 1GiB which are commonly used [14]. Having larger pages for the guest VM doesn't eliminate the overhead of nested paging but reduces it. This is because of fewer page-walk cycles due to the reduced levels in the radix tree for the page-walk as there are fewer pages to manage. Lager page sizes also reduce the TLB misses for a process with a large working set, this is because the page table size is fixed, hence, the only way to reduce TLB miss is to increase the page size. Huge Pages can be used to improve the address translation in VM by enabling the support of huge pages for both the host and the guest OS and configured as needed. **Transparent Huge Pages (THP)** attempt to automate the management of huge pages without application/process knowledge but has limitations such as latency issues for processes [15].

3.2 Agile Paging

This technique of Agile Paging is inspired from the past work of *Selective Hardware software Paging (SHSP)* by *Wang et al.* in the paper "Selective hardware/software memory virtualization" [16]. The proposed change for SHSP is to entirely switch between shadow paging and nested paging dynamically for a guest process by monitoring the TLB miss rate and guest page faults. However, switching entirely adds latency to the process of switching to shadow paging because it requires high hypervisor intervention each time to re-build the sPT which is expensive and affects performance.

Agile Paging combines the techniques of nested paging and shadow paging (2.3.1 and 2.3.2) and exceeds the benefits of both by combining them. A virtualized page walk on a TLB miss starts with shadow paging and optionally switches to shadow paging in the same page-walk when frequent updates to the gPT causes high hypervisor intervention to update the sPT (gVA to hPA mapping) to be kept consistent. The observation is that shadow paging has lower overhead than nested paging except when the gPT updates. Thus the key intuition is that the page tables are ***not updated uniformly***. The lower-levels of the page table radix tree (leaves) have are updated more often than the upper-level nodes. Hence, the authors of the paper "Agile Paging" [3] *Gandhi et al.* propose using shadow paging with sPT for the TLB misses where the upper levels of the guest page table usually remain static, thus not needing hypervisor intervention. Then, switching

to nested paging for fast in-place updates for the lower-levels of the guest page table since they would be updated frequently (needing high hypervisor intervention which affects performance if shadow paging was continued).

The paper describes the design of agile paging as 3 registers for the pointers for the page tables for sPT, gPT and hPT to support both shadow paging and nested paging. If agile paging is enabled, the virtualised page walk starts of with shadow paging and switches to nested paging. For a fine-grained switching between shadow paging and nested paging on any level of the gPT, the authors propose a change to the presentation of each page table entry (PTE) by adding a **switching bit**. When the swithching bit is set, this notifies the page table walker (PTW) in the MMU to switch from shadow paging to nested paging. In that case, the sPT entry holds the hPA of the next gPT for nested paging.

The paper then goes to describe the different degrees of nesting for virtualized address translation in agile paging: (i) full shadow paging, (ii) full nested paging and (iii) four degrees of shadow to nested paging switching at each level of the sPT. From evaluation, the paper concludes that Agile Paging on average has a maximum of 4 or 5 memory accesses on a TLB miss, which is close to achieving native performance for address translation. The paper then discusses the policy to use which degree of nested paging. To proposed policy for the switch is that the hypervisor detects changes to the gPT and makes a switch to reduce the overhead. It is notices that the updates to the page table are binomial at a time interval of 1 second (ie no updates or many updates). Based on the observation, the authors have proposed a policy of switching from shadow paging to nested paging if at least 2-writes to any level of the gPT are detected by the hypervisor in a second or other fixed time interval. The paper then discusses a more complex and effective policy that uses dirty bits to detect changes to the page table without the monitoring the hypervisor during the time interval, this is for the switch from nested paging to shadow paging. This is because the current implementation also allows a switch from nested paging to shadow paging but hasn't shown performance improvements compared to the switch from shadow paging to nested paging.

The paper then discusses methods of evaluation to measure the improvement of address translation in agile paging over nested/shadow paging. The authors use a novel approach for the evaluation, they initially generate a trace of the page table updates from the hypervisor, and then use the trace with *BadgerTrap* to calcuate performance using a linear model. A *BadgerTrap* is a tool that converts all x86_64 TLB misses into VMTraps (a hypervisor intervention), this allows the authors to analyse the TLB misses while having a native performance for TLB hits. The idea is to measure the overhead of a TLB miss which could be done using the *BadgerTrap*.

This evaluation techniques were used on a wide variety of workloads such as compute- and memory-intensive single-threaded workloads from SPEC2006, multi-threaded workloads from PARSEC, bioinformatics tasks from BioBench, and large-scale graph-processing tasks (Big-Memory) which mainly have a high TLB miss rate overhead. Then the paper discusses the performance analysis results of the evaluation of agile paging, the key result statistic is that agile paging improves the performance over nested paging and shadow paging by 12% and performed 4% slower than native. The key idea that contributes towards these results is by converting the changing portion of the gPT to nested mode, agile paging is able to prevent most of the VMexits/VMTraps that makes shadow paging slower, thus improving the performance.

3.3 DVMT-Do-it Yourself Memory address Translation

Do-It-Yourself Virtual Memory Translation (DVMT) is a technique for applications/processes to customize their virtual-to-physical address translation. This is described in the paper "Do-It-Yourself Virtual Memory Translation" by *Alam et al.* [5]. This is implemented by decoupling memory protection validation from virtual to physical address mapping. This decoupling allows applications to manage their own custom mappings for address translation. Though, DVMT can be used for native applications, it authors describe its potential for virtualized environments. Using DVMT, it has been shown for virtualized environments to achieve a 1.2-2x speedup compared to the traditional techniques of nested and shadow paging.

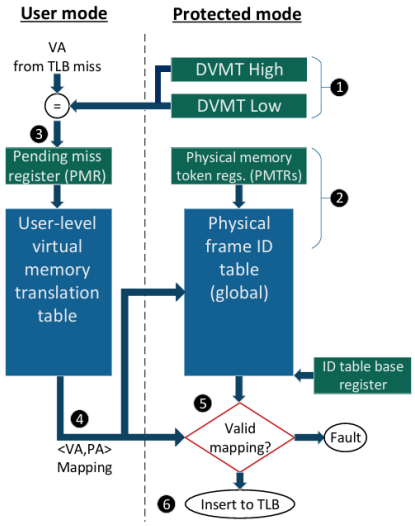

Figure 3.1: Design overview of DVMT [5]

Initially, the paper describes how DVMT works for native/non-virtualized applications as shown in fig 3.1. The overall flow of DVMT is that, when an application in initialised it requests a contiguous range of virtual addresses by setting the DVMT High and low values maintained by the OS to determine if DVMT is to be used on a TLB miss. Also during initialisation, the process also requests frames to be allocated in the the way required by DVMT. The OS then creates permission tokens for check if the frames assigned to the process are only accessed. These are stored in the running processes **physical memory token registers (PMTRs)**. On a TLB miss, if the VA is in the DVMT high/low range, the hardware triggers the address translation sequence by updating a **Pending Miss Register (PMR)** and wakes up a *helper thread* which performed the address translation and updates the TLB if the permission check passes. Since, address translation is decoupled from memory protection, isolation between processes and VMs enforced by permission tokens. Each frame is allocated a permission token, this idea is more generalised in TPT discussed in 3.5. All of the permission tokens are maintained by the OS in the *physical frame ID table*. If the permission check fails, an exception is raised to the OS (page fault). This check also enforces inter-VM isolation for virtualized environments. Since, the permission check is decoupled from the address translation, the authors points out a advantages which is that they can be performed in parallel with the address translation hiding latency, this idea is also generalised in TPT 3.5.

The paper also discusses the 3 ways the perform custom address translation for the application/process: flat, hash, and DS. Flat is a 1D array indexed by the VA, similar to a 1D page table. Hash scheme uses a hash table that functions as a software TLB using vector SIMD instructions. DS scheme is implemented using segments and addition, similar to the technique of segmentation used by the OS for native address translation instead of paging. However, this technique has a lot of limitations such as requiring contiguous physical memory and also variable size segments unlike paging where a page is of a fixed size. Custom implementations for address translation are specific to the application to have fast address translation if needed.

Later, the paper extends this implementation to Virtualised environments, since, the ideas of custom memory translation from gVA to hPA by the guest application would be more performant than nested or shadow paging. The key ideas described above do not change for virtualized environments, but they are implemented in the hypervisor instead of the OS. There are 2 ways DVMT could be implemented in virtualised environments. Firstly, hypervisor-only DVMT, where the gPT is unchanged and used, and the hypervisor maintain the custom page table for the gPA(hVA) to hPA address translation. Alternatively, DVMT is implemented in both the guest and the hypervisor, where in addition to the hypervisor having custom address translation from gPA to hPA, the guest also manages gVA-to-gPA translations using its own DVMT-enabled hardware state. Both of these methods improve the address translation form gVA to hPA performance than nested/shadow paging. Finally, the paper describes the evaluation techniques used to evaluate the performance

improvement using DVMT, with the the different implement ion of the custom page table (flat, hash and DS) . This is done by measuring the speed up using DVMT for various workloads in virtualized environments such as big-memory applications (e.g.,PowerGraph,Memcached,Redis). The paper concludes that there is ~1.5-2x speedup by uisng DVMT than nested paging used by x86_64 for virtualized address translation.

3.4 Prioritising Caches

Page Walk Caches (PWC) and Nested TLB

The paper "Accelerating Two-Dimensional Page Walks for Virtualized Systems" by *Bhargava et al.* [6] focuses on discusses the AMD Opteron **Page Walk Cache (PWC)** which is designed to reduce the translation latency of a page-walk across multiple levels of the radix tress by caching intermediate results of a page walk.The author then extending the PWC with **Nested TLB** for 2D page walks improves the address translation for virtualised processes by 46% than with no caching. However, since the page tables are not modified uniformly as discussed in agile paging (3.2), they have poor cacheability in the PWC. However, the paper proposes that this could be improved by increasing the page size as discussed in 3.1.

Page Walk cache is is a small, fast, fully associative cache.A PWC hit prevents a traversal through the radix tree. Since, each time we check if we can have a PWC entry until we have the hPA. Else, when we traverse the page-table on a PWC miss, we add the entry to the PWC for a faster access next time. The could PWC stores entries from all levels of the page table other than the leaf, this is because it is the same as the TLB entry. The paper then describes 3 types of PWC, 1D PWC (a in 3.2), 2D PWC (b in 3.2) and 2D PWC with nested translation (c in 3.2). For virtualized processes, in 1D PWC, we only cache entries in the gPT dimension. This means the nested entries (circles) as shown in 3.2 (a) do not benefit form the PWC. With a 2D PWC, this is the extension of the 1D PWC caching all 24 page-table references of a 2D Walk. Finally, the paper describes the 2D PWC + NT where we would have the 2D PWC with Nested TLB (NTLB). The aim of the NTLB is to reduce the number of page table references that take in a page walk. On each TLB hit, we skip all of the nested translations as shown in 3.2 (c) significantly improving performance by reducing the no of references we make with the 2D implementation.

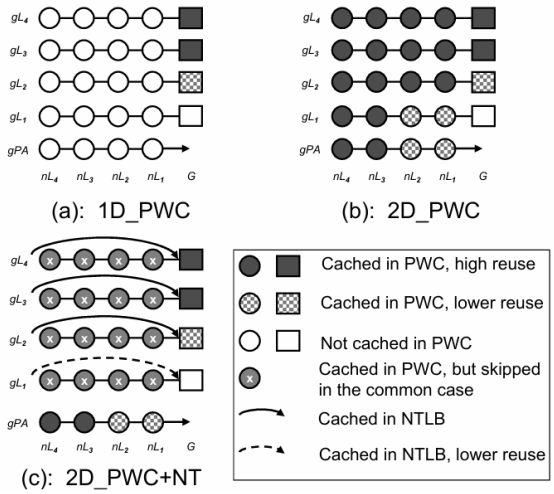

Figure 3.2: PWC [6]

The paper then evaluates the the performance of PWC and NTLB on numerous workloads such as the MiscServer (database back-end with SAP; stock market analysis), WebServer, JavaServer, and other CPU benchmarks. The results of the performance analysis for virtualized environments are 2D PWC significantly outperforms 1D PWC, achieving 38% improvement for the JavaServer workload. 2D PWC + NTLB improves the performance further, with additional improvements

of 7%. When 2d PWC + NTLB are combined with Huge Pages (2MiB), virtualized address translation achieves 93% of the native performance, achieving near-native address translation since 2D page walk is only performed on a cache miss.

3.5 TPT - Translation Pass-Through

This section describes the current implementation of *Translation Pass-Through* with technical details taken directly from the paper *"Translation Pass-Through for Near-Native Paging Performance in VMs"* [2]. This is a new memory virtualization mechanism that achieves near-native performance for address translation in VMs. The aim of this project is to extend the functionality of the TPT implementation discussed in section 4.

3.5.1 Overview

TPT enables guest VMs to directly control the address translation from gVA to hPA for the physical memory assigned to them using 1D page tables. It enforces inter-VM isolation by the hosts exploiting new hardware support for physical memory tagging in commodity CPUs (eg.AMD SEV-SNP).The paper mentions to achieve speed-up of 2.4x over nested page table and 1.4x over shadow page table using TPT. The performance improvement in TPT for the address translation comes from using 1D page table in the guest VMs combined with hardware memory protection checks using tags. An optimisation is that tag check can be hidden by performing them in parallel with the memory accesses and traversal of the page table. In contrast, nested page tables and shadow page table require sequential operations, making them harder to optimise or parallelize.

The guest OS could enable TPT dynamically at run-time for those applications that would benefit from it while maintaining the traditional techniques for the other applications (kernel-space and other user-space applications). To enable TPT for a process, the guest user writes a `procfs` entry, which triggers the construction of the page tables and can activated/used when the process is rescheduled. This is implemented by having **TPT-page table** which holds the mapping of gVA to hPA in the guest in addition to a **Non-TPT page table** for nested paging, both together called (**dual page table**) and are kept synchronised using `pv_ops` with low overhead and transparency.

The hypervisor still has control over the frames it has assigned to the quests, it can reclaim frames from VM if needed, and VM migration is still supported, since dual page tables are maintained for each TPT-enabled application. Hence, in such a case, it would revert back to using Non-TPT page table with nested paging by injecting `tpt_status` exception to the hypervisor. TPT page-tables can be disabled by the hypervisor by setting the `VMCS TPT-tag` to zero for the entire VM.

Memory Protection Tags

Physical memory protection tags are recently introduced in commodity CPUs are utilised for memory encryption, this could be used for inter-VM isolation for the purpose of address translation. In these systems, the MMU checks the host physical memory accesses against a **Host Frame Permission Table (RMP)** which is used to verify which VM can access the memory location. RMP is basically an 1D Array of physically contiguous locations that contain an entry for host frame, which is an identifier for the VM that the frame is assigned to by the hypervisor. Since, every access to the hPA is checked in the RMP on a TLB-miss or cached RMP data (cache lines are extented to hold RMP data or are held in separate caches), it ensures inter-VM isolation.

Performance benefits of TPT

The main performance benefit of TPT is for user-space applications with large working sets where traditional address translation techniques such as nested paging or shadow paging would be of a bottleneck. Or for designs that implement deeper radix trees to support more physical memory. For example, a 4-level page walk in TPT incurs upto 9 memory accesses (4 for page table, 1 for actual data and the rest (4) for keeping the dual page tables in sync using `pv_ops`) where as nested paging incurs 24 memory accesses. As the depth increases to 5, TPT goes from 9 to 11 whereas nested increases from 24 to 35 memory accesses.

3.5.2 Design

Initially every process has a single non-TPT page table (traditional page table maintained by the guest OS used with nested paging or shadow paging, we assume nested paging is used). When an application requests TPT page table, at this point the guest OS maintains dual page tables for that process, keeping them in sync with each other. To construct the TPT page table, the guest OS retrives the gVA to hPA mappings from read-only **Guest Address Map** which is maintained by the hypervisor for each VM. This map is exposed as a virtual PCIe device to VMs (`TPT_addrmap`). We also have 2 registers **CR3** and **TPT-CR3** which hold the base addresses of the page tables. For TPT-enabled processes, when the MMU traverses the TPT page table, the MMU checks the Host frame Permission Table (RMP) for a tag match for the hPA, if the match fails then an exception is raised into the hypervisor since that particular VM is doesn't have access to the particular frame (This suggests and incorrect/invalid entry in the page table). The **TPT Memory Manager (TPT-MM)** in the hypervisor keeps track the host frames allocated to update both the Guest Address Map and the host frame permission table (RMP).

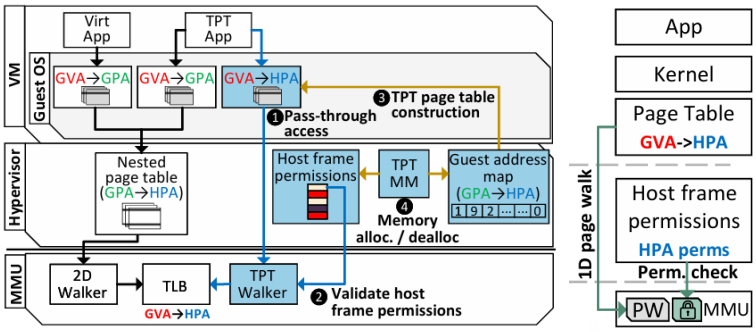

Figure 3.3: TPT design prototype [2]

3.5.3 Evaluation

In the paper, it has been shown that TPT has improved performance than traditional paging. Since TPT requires frame tage checks in hardware, they have been simulated in software to evaluate the performance of TPT since the current hardware support for them is limited and not implemeted for virtualized environments. Since, TPT-CR3 register is required in hardware to hold the TPT page table location, this has also been emulated in hardware for the time being. The performance of TPT was evaluated mainly in 2 ways : TPT-naive and TPT-opt. The naive implementation performs the page-walk and the frame tag checks in sequence, whereas the opt implementation performed them in parallel. The paper then describes for each application benchmark (data center workloads such as kcbench, XSBench, Canneal) evaluating TPT (where applicable) with 2 different access patterns (random and sequential) accorss differnt page sizes for both guest and the host. It has been concluded that TPT-opt out-performs traditional page tables in all cases. Overall, the performance of TPT matches Native exhibiting 2.4x over paging and 1.4x over shadow paging respectivly on average (random and sequential accesses included).

3.6 Conclusion

There exists a significant body of literature on address translation in virtualized environments, with various techniques aimed at achieving near-native address translation for guest VMs. This chapter highlighted a few key approaches, but it is important to note that there are numerous other software and hardware advancements in this field not covered here that also contribute to addressing the challenges of memory virtualization.

The main focus of this project is to extend the functionality of TPT as described in Chapter 4.

Chapter 4

Project Overview

A feature that could be added is automatically enabling TPT (3.5) for application where traditional address translation techniques such as nested paging and shadow paging are of a bottleneck. This project will be implemented by assuming *nested paging* is used in x86_64. An example of such applications are those with **data centre workloads** run in the VM such as graph processing (Page Rank algorithm, BFS algorithm), optimisation code for chip design, database management systems or kernel compilation. These processes generally have large working set sizes (RSS) and and high TLB miss rates. The guest OS could enable TPT on a per-process basis, for user-space code only, where TPT would improve performance. This is mainly for applications that have a large working set and would benefit from translation from gVA to hPA with native performance (4 memory accesses). TPT can be enabled for a process based on an explicit user request to create the dual page tables if it is enabled for the VM. The aim of this project is to extend the current implementation with automated runtime policies by monitoring different performance metrics at **fixed intervals** discussed in 4.1 and enable TPT per-process to have native performance. The policy aims to provide a small threshold like those in branch predictors for switching modes [3].

Performance Metric	**Reason**
TLB miss rate	Indicates how frequently address translations miss in the TLB (MMU), high miss rate indicates that the process must undergo a page-walk for each address translation for a TLB miss. Thus, a process with high TLB miss rate would benifit from using TPT page - tables to reduce miss-penalty.
TLB miss-penalty	Measures cycles spent on TLB misses, highlighting the overhead of address translation. In the case of Nested Paging (4-level radix tree) it is 24 memory accesses for the 2D Page Walk. In case of Shadow Paging it is the number of cycles spent spent traversing the the sPT which is 4 memory accesses.
RSS Values (Resident Set Size)	Shows the amount of physical memory a process requires, indicating processes with large working sets.
Page Walk cycles	Tracks the time/cycles spent traversing page tables, correlating with TLB miss rates and performance issues.
Memory access patterns	Analyzes access patterns to assess if the process performs random or sequential memory operations. Random access may increase TLB misses and require more frequent page table walks, thus would benefit if TPT is enabled resulting in less cycles spent on page table walks than for nested paging because TPT requires fewer memory accesses.
No. of Hypervisor interventions/ No. cycles for each hypervisor intervention	Highlights the impact of the hypervisor on performance, specifically with shadow paging.
Total execution cycles	Captures the overall runtime cost, allowing comparison of performance before and after TPT enablement.
Repeated for different host/guest page sizes	Evaluates the impact of various page sizes for both the guest and host (1 GiB Huge pages).

Table 4.1: Performance metrics

Chapter 5

Plan

5.1 Project Plan

In this section, we will briefly discuss the project timeline and the tasks to focus on during each time block.

Duration	**Main Goal**	**Tasks/TODO**	**Notes**
Term 2 Week 2, 3 (13th Jan – 24th Jan)	Report Writing	• Focus on Background/Related Work section of the report • Set up installation on Kea06 and run benchmarks • Include benchmarks in the report (if possible)	Interim report due on 23rd Jan
Term 2 Week 4 (27th Jan – 31st Jan)	Setup project	• Set up and run benchmarks in the paper (if not done by the interim report)	
Term 2 Week 5, 6, 7 (3rd Feb – 21st Feb)	Basic Implementation - Technical	• Determine performance characteristics for automating runtime policies (e.g., TLB miss rates, RSS values, memory access patterns, etc.) • Learn how to enable TPT • Enable TPT for selected metric evaluations	
Term 2 Week 8, 9 (24th Feb – 7th Mar)	Evaluation	• Evaluate performance boost with and without TPT enabled • Write a draft report	Fallback position with basic implementation and evaluation for at least one performance measure

Duration	Main Goal	Tasks/TODO	Notes
Term 2 Week 10 (10th Mar – 14th Mar)	Revision Week		
Term 2 Week 11 (17th Mar – 21st Mar)	Exam Week		
Spring Break Week 1 (24th Mar – 25th Apr)	Fallback position report writing, Implemenation	• Investigate additional performance characteristics for TPT • Explore techniques for optimizing TPT-enabled applications • Implement and evaluate performance improvements • Draft implementation and evaluation chapters	Break - if not done with implementation - overlap with evaluation and Report writing
Term 3 Week 1, 2 (28th Apr – 16th May)	Evaluation	• Finalize performance characteristics • Gather statistics on performance boost • Finalize code changes	
Term 3 Week 4 (19th May – 23rd May)	Report Writing	• Finalize the implementation section • Update code or performance metrics if needed	
Term 3 Week 5 (26th May – 30th May)	Report Writing	• Finalize the evaluation section using gathered statistics	
Term 3 Week 6 (2nd June – 6th June)	Report Writing	• Work on the Conclusion section • Finalize the Introduction section	
Term 3 Week 7 (9th June – 13th June)	Proof-Reading / Checking	• Proof-read the report and repository • Finalize the report	Report due on 13th June

Duration	**Main Goal**	**Tasks/TODO**	**Notes**
Term 3 Week 8 (16th June – 20th June)	Presentation	• Work on presentation slides	
Term 3 Week 9 (23rd June, 24th June)	Presentation	• Submit presentation slides	Slides due on 24th June

Table 5.1: Project Plan Table

5.2 Evaluation Plan

In this section, we will briefly discuss how the success of the project would be evaluated. This is to ensure that we have met the main goal of the project which is to automate the address translation to use TPT in guess application where necessary.
The miro-benchmarks that we would be using to evaluate the project are taken from the original TPT paper a combination of sequential and random memory access patterns [2] which are as follows:

Workload	**Description**	**RSS**
kcbench	Kernel compilation benchmark (v4.19)	1 GiB
XSBench	Monte Carlo neutron transport algorithm	99 GiB
Canneal	Optimization for chip design (PARSEC)	109 GiB
GUPS	Random integer updates in memory (HPCC)	129 GiB
PR	Page Rank (GAPBS) on kron graph	72 GiB
BFS	BFS Algorithm (GAPBS) on kron graph	70 GiB
CC	CC Algorithm (GAPBS) on kron graph	70 GiB
Memcached	Facebook ETC (3×10^8 keys; mut. client)	108 GiB

Table 5.2: Benchmark workloads and their RSS (Resident Set Size) [2].

For each of these benchmarks, we aim to run under different configurations of page sizes [2]:

- 4KiB guest and host page sizes
- 4Kib guest and 2MiB host page sizes (huge page support)
- 2MiB guest and host page sizes (huge page support)

For each configuration, to evaluate the success of the project we measure the following:

1. **The overhead caused by profiling tool**
   (a) measure the number of cycles spent on profiling
   (b) number of cycles spent for TPT to be enabled dynamically

   This allows us to analyse if the overhead of profiling is minimal compared to continuing with the current technique of address translation instead of switching to TPT. If the overhead of profiling becomes a bottleneck, then we could evaluate how the profiling tool could be improved.

2. **The improvement in the address translation performance before and after dynamically enabling TPT for a guest process.** This can be measured by comparing the following
   (a) no. of memory access on page-walks
   (b) cycles spent on page-walk
   (c) total execution cycles

   Taking these measurements when the application uses default nested paging and after enabling TPT allows us to see an improvement in performance of address translation for an application after enabling TPT. This could be measured assuming the overhead of the profiling tool could be ignored.

By measuring the above metrics we can evaluate the implemented feature to automate enabling TPT.

References

[1] Margaritov A, Ustiugov D, Bugnion E, Grot B. Prefetched Address Translation; 2019. Accessed: 2025-01-12. https://www.pure.ed.ac.uk/ws/files/111444962/Prefetched_Address_Translation_MARGARITOV_DoA260719_AFV.pdf.

[2] Bergman S, Silberstein M, Pietzuch P, Shinagawa T, Vilanova L. Translation Pass-Through for Near-Native Paging Performance in VMs. 2023. Available from: https://www.doc.ic.ac.uk/~lvilanov/publications/files/atc23_tpt.pdf.

[3] Gandhi J, Hill MD, Swift MM. Agile Paging: Exceeding the Best of Nested and Shadow Paging. Department of Computer Sciences, University of Wisconsin-Madison. 2025. Available from: https://ieeexplore.ieee.org/document/7551434.

[4] Gregg B. perf: Linux Profiling with Performance Counters; 2025. Accessed: 2025-01-12. https://www.brendangregg.com/perf.html.

[5] Alam H, Zhang T, Erez M, Etsion Y. Do-It-Yourself Virtual Memory Translation. In: Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA). Toronto, ON, Canada: IEEE/ACM; 2017. Available from: https://lph.ece.utexas.edu/merez/uploads/MattanErez/isca2017_dvmt.pdf.

[6] Bhargava R, Serebrin B, Spadini F, Manne S. Accelerating Two-Dimensional Page Walks for Virtualized Systems. In: ASPLOS'08: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems. Seattle, Washington, USA: ACM; 2008. Available from: https://dl.acm.org/doi/10.1145/1346281.1346286.

[7] Microsoft. What is a Virtual Machine?; 2024. Available from: https://azure.microsoft.com/en-gb/resources/cloud-computing-dictionary/what-is-a-virtual-machine/.

[8] Edouard Bugnion JN, Tsafrir D. Hardware and Software Support for Virtualization. Morgan & Claypool Publishers; 2017.

[9] Google. What is a Virtual Machine?;. Available from: https://cloud.google.com/learn/what-is-a-virtual-machine.

[10] VMware. What is a Hypervisor?; 2025. Accessed: 2025-01-12. Available from: https://www.vmware.com/topics/hypervisor.

[11] contributors W. Hypervisor; 2025. Accessed: 2025-01-12. Available from: https://en.wikipedia.org/wiki/Hypervisor.

[12] Dimakopoulou M, Eranian S, Koziris N, Bambos N. Reliable and Efficient Performance Monitoring in Linux. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC16). Salt Lake City, Utah, USA: IEEE; 2016. Available from: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7877112.

[13] Corporation I. Intel Performance Counter Monitor (PCM); 2025. Accessed: 2025-01-12. https://github.com/intel/pcm.

[14] Wikipedia contributors. Page (computer memory) — Wikipedia, the free encyclopedia; 2025. [Accessed: 18-Jan-2025]. Available from: https://en.wikipedia.org/wiki/Page_(computer_memory)#:~:text=Starting%20with%20the%20Pentium%20Pro,use%201%20GiB%20pages%20in.

[15] Red Hat, Inc . Using Huge Pages with Virtual Machines; 2025. [Accessed: 18-Jan-2025]. Available from: https://docs.openshift.com/container-platform/4.9/virt/virtual_machines/advanced_vm_management/virt-using-huge-pages-with-vms.html.

[16] Wang X, Zang J, Wang Z, Luo Y, Li X. Selective Hardware/Software Memory Virtualization. In: Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. New York, NY, USA: ACM; 2011. Available from: https://dl.acm.org/doi/10.1145/1952682.1952710.

[17] Linux Kernel Organization I. License Rules — The Linux Kernel documentation; 2019. Accessed: 2025-01-16. Available from: https://www.kernel.org/doc/html/v5.1/process/license-rules.html.

Declarations

D.1 Ethical Considerations

D.1.1 Copyright and License compliance

This project involves profiling the Linux Kernel Code using performance counters and also modifying some parts of the code base if needed. Hence, the main ethical concern is the copyright and license compliance.
The Linux Kernel is provided under the terms of the GNU General Public License version 2 only (GPL-2.0), as provided in `LICENSES/preferred/GPL-2.0` [17]. GPLv2 is a **copyleft** license, meaning that any modified versions of the kernel distributed publicly must also be released under the same license. This ensures that the freedom to use, modify, and distribute the software is preserved for all users.
In this project, any modifications made to the Linux Kernel code will adhere strictly to the terms of GPLv2.

D.1.2 Security Considerations

The initial implementation of TPT has security considerations mainly for side-channel attack for inter-VM isolation by speculatively executing page table walks and permission checks in parallel [2]. This project, that extends TPT still has these security considerations. However, the new feature to automatically enable TPT likely doesn't introduce any new security considerations.