Honey, I Shrunk the Guests

Page Access Tracking using a Minimal Virtualisation Layer

Dustin Tien Nguyen
Friedrich-Alexander-Universität Erlangen-Nürnberg
Erlangen, Germany
nguyen@cs.fau.de

Sebastian Rußer
Friedrich-Alexander-Universität Erlangen-Nürnberg
Erlangen, Germany
sebastian.russer@fau.de

Maximilian Ott
Friedrich-Alexander-Universität Erlangen-Nürnberg
Erlangen, Germany
ott@cs.fau.de

Rüdiger Kapitza
Friedrich-Alexander-Universität Erlangen-Nürnberg
Erlangen, Germany
ruediger.kapitza@fau.de

Wolfgang Schröder-Preikschat
Friedrich-Alexander-Universität Erlangen-Nürnberg
Erlangen, Germany
wosch@cs.fau.de

Jörg Nolte
Brandenburgische Technische Universität Cottbus-Senftenberg
Cottbus, Germany
joerg.nolte@b-tu.de

Abstract

Advances in memory-intensive areas such as machine learning and graph processing drive demand for suited memory technologies. Part of the answer lies in multi-memory architectures with fast and high-capacity memory tiers. Heterogeneous systems can be very diverse, ranging from NUMA to persistent memory and CXL-attached memory. Efficient usage requires support from the operating system (OS) with knowledge of memory performance characteristics and detailed insight into the page access behaviour of processes. Gathering statistics on the *read/write*-ratio of accesses and (huge) page utilisation is expensive with common methods.

In this paper, we propose *vmmload*, a minimal virtualisation layer to monitor user space programs using a virtualisation-based page modification log (PML) for efficiently gathering runtime statistics. While PML is a hardware extension that was originally designed to track virtual machine memory access, *vmmload* enables observation of individual processes. *vmmload* monitored user space processes run with PML-enabled virtualisation, but are still able to interface with the host OS. We explore the viability of using PML without a conventional virtual machine and providing the host OS with the ability to use these extensions directly.

CCS Concepts

• **Software and its engineering → Memory management**; **Virtual machines**.

Keywords

Operating System, Virtualisation, Intel PML, Hypervisor

ACM Reference Format:
Dustin Tien Nguyen, Sebastian Rußer, Maximilian Ott, Rüdiger Kapitza, Wolfgang Schröder-Preikschat, and Jörg Nolte. 2025. Honey, I Shrunk the Guests: Page Access Tracking using a Minimal Virtualisation Layer. In *4th Workshop on Heterogeneous Composable and Disaggregated Systems (HCDS '25), March 30, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3723851.3723852

1 Introduction

Workloads like graph processing and machine learning are memory intensive and benefit from large capacity and low latency [13, 17], and satisfying their needs is an ongoing quest for more suitable hardware solutions. The trend manifests in the effort to adopt alternative memory technologies and disaggregated memory. Available DRAM contenders are persistent memory (PMEM) with very high capacity [10] or high bandwidth memory (HBM) [8]. In addition, there is CXL.mem, which allows a host system to access disaggregated memory, supplied by a remote host via Compute Express Link (CXL), directly with load and store operations [4]. Consequently, systems using CXL.mem and PMEM have to manage large quantities of memory efficiently. Even though the vast amount of memory provided by these two technologies is easily accessible for both the operating system (OS) and user space programs via *load*/*store* instructions, efficient usage is difficult. Systems equipped with HBM, PMEM, or CXL.mem have to deal with differences in non-functional properties, such as the variance in access latency and bandwidth compared to the resident DRAM. These properties may even change based on factors such as access frequency, the ratio of *load*/*stores*, and parallel access [21, 27]. Thus, on top of managing a significant amount of memory, these tiered-memory systems have to deal with non-uniform access (NUMA) problems [12].

The amount of fast-tier memory (usually DRAM) is limited by either hardware or financial constraints. Even though all processes would benefit from fast-tier memory, the OS has to arbitrate a mix of scarce fast-tier and plentiful capacity-tier memory. Hence, it is vital to find a good balance of memory for each process by handing out sufficiently fast-tier memory to not slow down progress, while retaining enough for other processes.

Thus, the complexity increases, as the OS has to manage memory per application and consider the memory type as well. The OS has to make a good estimate on the memory access behaviour of each process through observation, so that their working set can be placed in the most appropriate memory. Typically, frequently written pages should reside in the fast-tier memory, while the capacity-tier memory is suited for sporadically used pages. However, these access

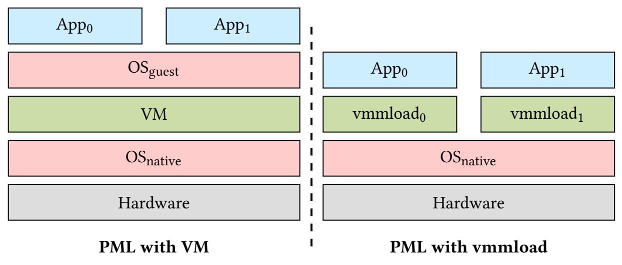


Figure 1: *vmmload* omits the guest OS in its design and allows processes to communicate with the host OS.

patterns are not static and may change over the lifetime of processes. Therefore, it is necessary to constantly monitor a process's memory access behaviour and to adapt placement decisions by migrating pages from one memory to another. Due to the performance costs associated with the page migration, the placement decisions must be good and should not be contradicted in the near future [7, 11].

The problem is further aggravated by the use of huge pages, which are actually a measure to improve performance. Huge pages reduce the translation overhead of virtual address spaces by mapping huge continuous memory regions.

Naturally, a huge page requires consecutive physical pages, which would be served from a single memory tier. This may lead to non-optimal memory usage if a whole huge page is placed in scarce fast-tier memory, but only a small fraction is frequently accessed. In these cases, it might be advisable to split huge pages into smaller ones and place the fractions in the appropriate memory tier [11]. Hence, tiered memory systems require a way to detect accesses at sub-huge-page granularity.

Thus, for tiered memory systems, we consider the monitoring of the following as essential 1) frequency of memory accesses, 2) sub-huge-page accesses, 3) *read* and *write*.

In this paper, we propose using the page modification log (PML) virtualisation extension to solve the problems mentioned above. It is originally designed to support virtual-machine monitors (VMMs) to observe their guests' memory usage [9]. Typically, a virtual machine (VM) is used to host a guest OS, which in turn manages processes, so the PML information is gathered for an entire VM instead of a single, targeted application. *vmmload* omits the VM and starts individual applications directly within a minimised virtualised environment (see Figure 1) and has access to the process-specific PML data. This information can then be used as a basis to plan and enforce page placement and page migration strategies in tiered memory systems. To the best of our knowledge, we are the first to observe isolated user space processes with virtualisation-based CPU extensions. With our contribution, we have determined that

- virtualised execution of user space programs is viable,
- the overhead is moderate,
- OSs benefit from the kernel integration of *vmmload*.

2 Fundamentals

Apart from PML, there are already established approaches to estimate memory access behaviour. We briefly overview these approaches and discuss their applicability regarding functional requirements and the tradeoff between runtime cost and precision.

2.1 Page Table Scanning

The classification regarding which page should be placed in what kind of memory is often done by tracking memory accesses. This can lead to substantial performance penalties, as it usually requires enforcing page faults by deliberately marking memory as unavailable in page tables by scanning the page tables for the memory management unit (MMU)-provided information to identify accessed and modified pages [3]. Any page-table-based approach requires the OS to scan the page tables of each process regularly to detect modifications. The method is indifferent to varying page size granularities and can be used with huge pages as well as their smaller variant. As a consequence, however, no page accesses smaller than the used granularity can be detected – i.e., a 2 MiB huge page has no information about the 4 KiB range it is composed of. Hence, page-table-based strategies can not be used to measure the effectiveness of huge pages in tiered-memory systems – only a small part of a huge page may be accessed frequently [11]. Furthermore, it is mandatory to reset the access information frequently, which results in costly TLB shootdowns [11].

2.2 Sampling

As an alternative, it is possible to use hardware support to trace memory access. These approaches are often sampling-based, as with Intel PEBS [15, 25]. PEBS allows for detailed memory access tracking at byte granularity. It is possible to adjust the level of detail with a configurable sampling interval. However, there is a risk of missing memory accesses in rapid succession, and the amount of data being stored with each sampling result is huge [28].

2.3 Binary Translation

It is also possible to introduce memory tracing into programs, by instrumenting them with binary translation [14, 18]. The byte-accurate memory accesses traces come at the cost of high-performance degradation. Even though, a page granular statistic is sufficient for page placement.

2.4 Page-Modification Logging

With the introduction of hardware-assisted virtual machines, other variants of introspection have been made possible. In particular, there is the Intel PML extension to aid the VMM, identifying memory pages used by a guest system [9]. PML creates a log of pages modified by a guest system. Hence, it serves the same goal as page-access tracking without the need of traversing page tables or being restricted by a sampling interval. In a similar manner, this logging information can be used to deduce the access frequency of pages and to derive decisions on page placement within the OS, which is crucial for tiered-memory system performance [20].

PML, as part of the virtual-machine extension (VMX), works within a virtualised environment and is built on top of extended page tables (EPTs). VMX offers hardware-assisted virtualisation by switching into a special CPU mode to isolate the host system from the virtualised *guest* environment. Any privileged instruction issued from within the guest has either no effect on the host or will pause the execution in the virtual machine and transfer control back to the host (*vm exit*). The host can then acknowledge or emulate the action performed by the guest and transfer the control to the guest (*vm enter*) [24]. The guest is provided with the same architecture as the host and can enforce privileges for software within its domain.

A virtual environment is defined by its guest physical address space, a memory range that can be used as physical memory by the guest (OS). The *guest* physical memory is mapped to *host* physical memory by the host OS via EPTs. When a *guest* physical page is accessed, the corresponding address is resolved using EPT. When PML is active, the CPU will log any *guest* physical address that sets the `modified`-bit for the corresponding EPT entry. After a certain amount of log entries has been gathered, a *vm exit* event is triggered and the VMM can gather the logged page access statistic [9].

3 Design

In order to use PML for classifying page access behaviour of user space programs, we implemented *vmmload* – an userspace ELF loader intertwined with a VMM. Due to the comparably high costs associated with the communication between the VM, the VMM and the host OS, *vmmload* is designed to reduce the number of messages. *vmmload* prepares a minimal virtual machine by using the FreeBSD-supplied virtual monitor module. The module helps to create and manage hardware-based VMs. *vmmload* arranges the minimal environment for x86_64 long mode execution, loads the targeted application into the *guest* physical memory and sets up the appropriate virtual memory translation. Additionally, the guest's physical memory is mapped into *vmmload*'s virtual address space. Afterwards, *vmmload* starts executing the guest application, while being responsible for the VM as VMM – it has to react to *vm exit* events which occur on guest system calls and partially interpret them. The most significant difference to conventional VMs is the omission of a guest kernel within the VM. Instead, our goal is to make the guest application able to communicate directly with the host OS, enabling shared memory and IPC with other (non-virtualised) processes via system calls. System calls lead to a *vm exit*, which transfers the control to *vmmload*. Depending on the kind of system call, *vmmload* can either relay the system call to the original host or process it directly. Since the relayed system calls are performed in the context of the VMM, *vmmload* has to ensure that they do not interfere with its own state. Hence, some system calls must be emulated by *vmmload*. In addition to the emulated system calls, any guest pointer passed as a parameter must be updated to reference *vmmload*'s address space instead of the guest's.

As soon as the guest process runs in a virtualised environment, the PML extension can track memory accesses. *vmmload* controls the size of the PML buffer, which is mapped into *vmmload*'s address space for efficient access. The buffer is constantly updated by the PML extension when the guest process is running. Even though PML only logs *write* access to pages, *reads* can be detected by observing modifications on the guest's last-level page table [20]. If a page table's address is in the PML buffer, the table can be scanned for the `accessed` bit.

Apart from system calls and shared memory, *vmmload* also supports dynamically linked executables by loading the dynamic linker as well.

4 Implementation

vmmload has to satisfy multiple requirements to support the execution of user space programs in a virtualised environment. On the one hand, it has to implement parts of an OS by preparing the guest process's runtime environment. On the other hand, it acts as a minimal VMM and can observe the execution. Finally, *vmmload* serves as a proxy between the host OS and the guest process.

4.1 Environment

vmmload can load statically and dynamically linked ELF binaries and execute them. This is done by setting up a virtual environment based on the FreeBSD VMM-module. The environment includes a guest physical address space of a pre-defined size. The guest memory is split into two partitions: 1) one for OS-related data structures and 2) another with guest physical page frames used for the guest application. The whole guest physical address space is continuously mapped into *vmmload*'s virtual address space. Therefore, *vmmload* can directly read and manipulate the guest's data without co-operation of the virtualised guest or interaction with FreeBSD's VMM-Module. The guest does not run while *vmmload* is active, making explicit synchronisation between operations on both sides unnecessary. As with normal user space processes, the guest cannot directly manage its virtual address space. Instead, all page allocations and memory mappings are performed via system calls (such as `mmap`) and processed by *vmmload*.

Before the guest application can be executed, it is loaded into the guest's memory. First, a virtual address space is provided based on page tables allocated from the system partition. Each loadable segment of the targeted program is mapped by setting the appropriate guest page tables in the guest physical memory. For dynamically linked binaries, the appropriate linker is loaded as well, followed by shared objects at runtime.

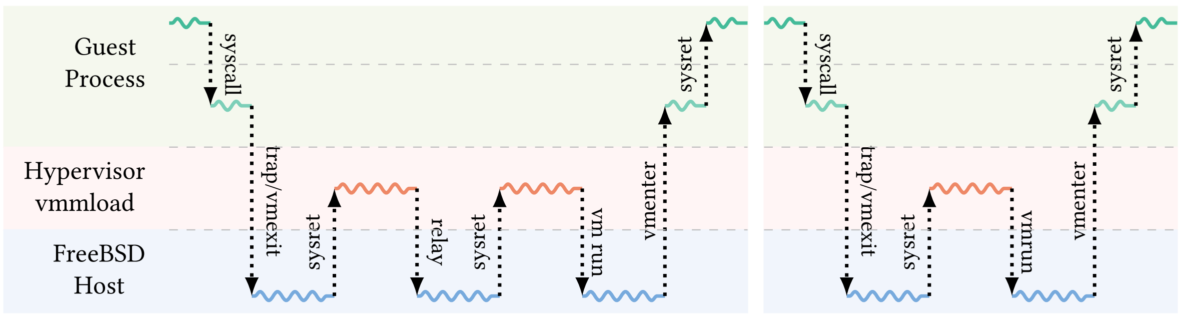
Next, the guest stack is prepared with the environment variables (as defined by POSIX) and command line arguments. Additional information about the runtime (*auxargs*) and *vdso* are copied over as well, as it is expected by the application and dynamic linker.

4.2 Runtime

The guest's runtime is composed of two parts. The first is *vmmload* itself, which sets up the guest environment for execution in the *x86_64 long mode*. In addition, it is responsible for page allocations in the guest's virtual memory. The other runtime is the host OS that receives the system calls relayed by *vmmload* and is responsible for managing the guest's process state. For the *long-mode* execution, *vmmload* has to set up some data structures expected by the CPU. These comprise the *global descriptor table (GDT)* with a description of privilege levels, an *interrupt descriptor table (IDT)* for setting up fault handlers and the virtual address space. Furthermore, floating-point and SSE/MMX-support are enabled.

4.3 System Calls

In order to make the guest process interact with the host OS and other processes, it is necessary to serve all of the guest's system calls. Since *vmmload* only supplies a rudimentary OS layer to set up the minimal required hardware runtime for the guest, most of the system calls have to be processed by the host OS. This is achieved by



(a) System calls relayed by *vmmload* generate multiple context switches. **(b) System calls that can be answered by *vmmload* directly.**

Figure 2: A detailed enumeration of the context switches necessary for different kinds of system calls.

relaying most system calls in *vmmload* to the host OS and writing the result back to the guest. Due to the design of the FreeBSD VMM abstraction, each system call requires several context switches. However, *vmmload* tries to keep the number as low as possible by handling some of the communication between the guest and the VMM with shared memory. *vmmload* installs a system call handler within the guest's memory during the initialisation and registers it with the `syscall` instruction for the guest. So, any execution of the `syscall` instruction will transfer the execution to the guest system call handler with guest supervisor privileges. The handler then proceeds to save the content of all registers used for system call parameters in the memory range known to *vmmload* and triggers an *vm exit* to resume *vmmload*. Due to the memory-mapped stub page, *vmmload* can access the stored parameters directly and relay the system call to the host OS. The result of the system call is stored on the same page, and the guest process is resumed. Following the transition to the guest context, this result will be loaded into the respective registers, and the handler will jump back to non-privileged execution.

The host OS handles most of the guest's system calls, while a few must be caught by *vmmload*. All system call parameters forwarded to the host must be checked for pointers to the guest's virtual memory. All references to guest virtual memory have to be translated to *vmmload*'s addresses, pointing to the area of mapped guest physical memory.

System calls issued by the guest process trigger up to three state switches until *vmmload* receives them, as displayed in Figure 2a. First, there is the privilege elevation within the guest process from issuing the `syscall` instruction. Since the process' address space does not change and most registers stay intact, this switch is quite fast. Next, a deliberate debug trap instruction within the guest system call handler triggers the transition into the FreeBSD kernel. The debug trap is configured to enforce a *vm exit*, which is handled by the kernel. A *vm exit* saves the current VM registers and address space and loads the host's state, invalidating various caches [9]. Since *vmmload* is responsible for the guest, the kernel will perform a normal context switch into the VMM. If the system call parameters were not passed via shared memory, more context changes would be necessary to query the system call parameters from the guest.

A few system calls require special care, as they directly impact the guest runtime environment. If any of these system calls is identified by their system call number, a special system call handler provided by *vmmload* is executed within *vmmload*'s context. Examples are `mmap` and `sysctl`. If any `mmap` issued by a guest process were to be relayed by *vmmload*, it would manipulate *vmmload*'s address space. Instead, *vmmload* checks whether the `mmap` is an anonymous allocation, in which case it can directly modify the guest's virtual address space. This variant of the system call does not need another round trip into the OS, as visualised in Figure 2b, and will, therefore, be faster. File-backed mappings are supported by a new system call, which instructs the OS to map files into the guest physical address space. *vmmload* can then set the guest virtual mapping accordingly to allow guest access. The `sysctl` family of system calls can be used to query information about the current runtime, e.g. the top of stack. Since *vmmload* and the guest are distinct processes, this information can not be accurate if *vmmload* relays the system call.

4.4 Memory

The responsibility for memory management within the minimal VM is divided in a similar way to a regular OS and its user space. The guest process must manage its memory at a byte-granular level but can request more coarsely managed memory from the OS. In the case of *vmmload*, the memory requests are partially served by *vmmload*, from the initial guest physical memory requested from the OS. Whenever the guest requests anonymous memory via the `mmap` system call, *vmmload* reserves a chunk of consecutive guest physical memory. The memory is then eagerly mapped in the guest's virtual address space, defined by the conventional page tables. The underlying EPT-tables, translating the guest physical to host physical addresses, are not updated. Any valid page access within the guest environment cannot trigger a page fault. However, the access can cause an EPT-fault, resulting in a *vm exit* handled by the kernel. The kernel will then adjust the mapping in the EPT tables and resume the guest's execution.

4.5 Page Modification Logging

We extended FreeBSD 13.1 with support for the PML extension by providing an interface for any VMM to enable PML and control the

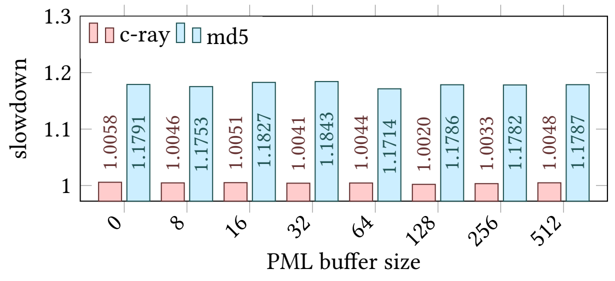


Figure 3: The slowdown for different classes of guest applications. A lower slowdown is better.

desired PML buffer size. The buffer is allocated by the kernel and can be memory-mapped by the VMM to access the results. Lowering the number of table entries increases the quality of results, but may deteriorate the performance. When the CPU tries to write on a full table, a *vm exit* event will occur. The kernel will clear the *modified* bit on all EPT entries referenced in the PML table so that any subsequent modification can be logged again. Then, the kernel will propagate the exit reason to the VMM. At this point, *vmmload* can read the memory-mapped table and create access statistics.

Since not all guest page accesses are equally significant, excluding memory from the PML statistic is possible. The two memory partitions, set up by *vmmload* (see Section 4.1), differ in their initial configuration of the *modified* bit in the corresponding EPT tables. If the *modified* bit is pre-set, any subsequent access will *not* result in a PML entry, and the bit will remain set. Any modifications to entries with the bit cleared will trigger the PML action. The VMM module will reset the modified bit of any PML buffer entry when the buffer is full. The separation is helpful for pages that should not be part of any access statistic, such as the system call trampoline or any *vmmload* managed page tables. In the case of page tables, a fine-grained distinction can be made by placing any page table level but the last in the partition ignored by PML.

5 Evaluation

We tested our FreeBSD 13.1-based *vmmload* on a tiered memory system with two Intel Xeon Gold 6330 processors. The system is equipped with 256 GiB of fast tier DRAM memory and 1024 GiB of capacity tier Intel Optane Persistent Memory 200. So, the system covers the typical pitfalls of tiered memory systems with NUMA behaviour and different memory technologies.

For our evaluation, we begin by measuring the performance impact of the minimal virtualisation environment and the Intel PML extension. Afterwards, we examine the quality of the gathered page access statistic. The tests show a mix of the *c-ray* benchmark [23], and the I/O-bound *md5* program bundled with the default FreeBSD installation. *c-ray* implements ray tracing and is mostly CPU-bound.

5.1 Performance Impact

We determine the slowdown induced by *vmmload* and compare it to the native execution of the test programs as base line in Figure 3. *c-ray* issued a total of 3676 system calls per run for an image with the resolution 2560x1440. Most of the system calls save the generated image to disk. *md5* calculates the md5-sum of a 4.3 GiB file and uses the majority of the 280 078 issued system calls to fetch the input.

The graphs present multiple runs of *vmmload* with different PML buffer size configurations. A smaller size results in a more accurate page access statistic but requires more context switches between the VMM and the guest. The entry with a PML buffer size of zero shows the overhead of *vmmload* with PML disabled.

For CPU-intensive workloads, the overhead of *vmmload* is negligible. There is less than 0.6 % slowdown which is mostly induced by occasional *vm exit* events due to scheduling and other system activity. The size of the PML buffer does not affect the runtime too much and the differences in slowdown seem to be caused mainly by the system's background noise.

I/O-bound programs experience a far more severe impact on their runtime performance. The *md5* benchmark has a slowdown of around 18 %. The slowdown correlates to the number of context switches for each system call, as displayed in Figure 2a. A conventional virtual machine does not encounter this problem because system calls are handled within the virtualised environment, and the interaction between the guest OS and the VMM is managed via interrupts and DMA [16].

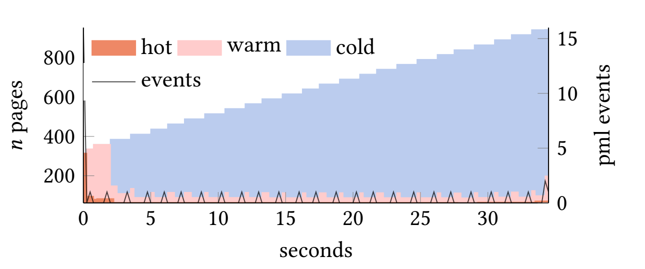
5.2 Practical Application of PML

We show the effectiveness of *vmmload* by analysing the gathered statistics. We highlight the total memory consumption and amount of frequently used pages (hotness) over the lifetime of our test programs. In addition, we unveil the memory access behaviour over different stages of the programs. These insights allow the OS to migrate pages between different tiers of memory based on the frequency of accesses and their load/store-ratio. The statistics help to identify an uneven distribution of huge page accesses, which may be better of split into smaller chunks within different memory tiers. We have cross-validated the measured memory consumption with *valgrind* [14].

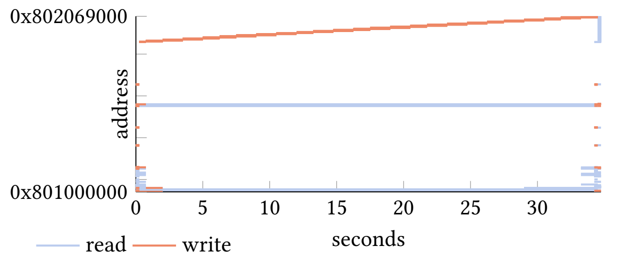
The hotness statistic in Figure 4a counts the total number of pages allocated by the respective process. All pages are distinguished as *hot*, *warm*, or *cold* based on the exponential moving average of the access frequency detected by PML and grouped by 250 ms intervals. A page gradually cools down within 2 s, if it has not been touched. In addition, the number of triggered PML events is displayed. The graph highlights *c-ray*'s growing memory demand over time. Most of the pages cool down very quickly, and the set of hot pages is very small (even barely visible in the Figure 4a).

Complementary information can be gathered from Figure 4b where the addresses of read/write accesses to the heap memory over time are visualised. All accesses within a PML event are associated with their virtual address and colour the page range and time frame. The graph indicates that set of hot pages with frequent writes roams during the runtime. However, there is also a memory region in the middle and lower address range that is continuously accessed throughout the runtime in a read-only manner.

The *vmmload* information indicates that the continuous streak of *read* accesses may be a good candidate for fast-tier memory. The *write* accesses however, display a poor locality – migration of these pages might not yield enough gain.



(a) A continuously growing memory demand by *c-ray*, with frequent PML events.



(b) Heap memory access for *c-ray* with different hotspots over the runtime.

Figure 4: Observation of memory accesses issued by *c-ray* with *vmmload* (with image resolution 1050x600).

6 Related Work

Research into both reducing the footprint of a guest within a virtualised environments and efficient memory tracing for hypervisors has gained momentum in recent years. In prior work, potential solutions to these challenges were addressed using virtualisation in a non-tradition form. Belay et al. proposed *Dune*, a system that provides non-privileged user-space application access to otherwise privileged operations using virtualisation [1]. Hereby, the virtual machine provides the application with a process-like environment instead of an abstraction of a computer system. The actual guest operating system is replaced by a library within the application that offers minimal resource management. In conjunction with a separate kernel module, *Dune* implements a facility for sandboxing using system-call filtering or efficient garbage collection based on direct page-table scanning. While our approach shares the idea of using virtualisation in an unique way, the overall goal is different: We focus on providing hardware-accelerated memory monitoring (PML) for commodity applications without any modifications.

Similarly, the *Out of Hypervisor* (OoH) principle exposes hypervisor-oriented hardware virtualisation features, such as Intel PML, to the guest operating systems [2]. Hereby, OoH realises direct access to these features using three components: The Xen hypervisor, a Linux guest kernel with the OoH module and the application itself with the OoH library. The former solely implements the interactions with the PML hardware extension. The adapted guest kernel, in turn, creates an application execution environment and allows direct access to the collected PML data. Finally, the library processes the collected data and makes it available for the application. In contrast, *vmmload* does not require any modification of the program under observation. However, the main difference is the primary objective: OoH passes the PML information to the guest user space, and *vmmload* shares the information with the host.

Other projects, such as *Nimble* [26], *Memtis* [11] and *MTM* [19] present a holistic mechanism for classification and migration of memory between memory tiers. *vmmload* can extend these aproaches by determining the set of hot pages.

7 Conclusion and Future Work

We have shown that *vmmload* can gather the memory access information necessary for tiered memory systems. Even though the frequent context switches triggered by system calls are costly, the virtualisation overhead is barely noticeable. The issue about context switches can be alleviated by integrating *vmmload* directly into the OS kernel, so that no relaying of system calls is necessary. Then, it would also be possible to enable *vmmload* statistics on demand, by migrating running processes to a virtualised environment. Following the *vmmload* kernel integration, the page access statistics can be directly used by the OS kernel to improve the performance of tiered memory systems.

We intend to use *vmmload* to determine *locality sets* [6], namely the set of referenced pages within a sampling interval. Such interval corresponds to a certain *phase of the execution* of the machine program under consideration. The characteristic runtime features of these phases form the basis for efficient process operation in a tiered main memory. Analogous to [5], we understand the tiered-memory requirements of a process as a sequence of locality sets and their holding times. Our intention is to structurally capture the relevant program sections by means of static program analysis and to quantify the properties of these sections using dynamic analysis. The latter would be the use case for *vmmload*, namely 1) when a machine program is executed for the first time in order to determine the best possible initial page distribution of the virtual process address space across the tiered main memory and 2) during all subsequent runs in order to control page migration between the memory tiers and thus ensure the most efficient address space management in the global. So *vmmload* will be a vehicle to maintain a *property-based scalable tiered virtual memory* (PAVE) for FreeBSD.

Acknowledgments

The author order corresponds to the SDC (sequence determines credit) model: that is, the nth author receives the $1/n$th share of the credit of the whole impact [22]. This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 501993201. The project is licensed under the FreeBSD (kernel) and GPLv2+ (*vmmload* user space) open source license. It is available at https://doi.org/10.5281/zenodo.14950325.

References

[1] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: safe user-level access to privileged CPU features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, USA, 335–348.

[2] Stella Bitchebe and Alain Tchana. 2022. Out of hypervisor (OoH): efficient dirty page tracking in userspace using hardware virtualization features. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 87, 14 pages.

[3] Jinyoung Choi, Sergey Blagodurov, and Hung-Wei Tseng. 2021. Dancing in the Dark: Profiling for Tiered Memory. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 13–22. doi:10.1109/IPDPS49936.2021.00011

[4] Debendra Das Sharma, Robert Blankenship, and Daniel Berger. 2024. An Introduction to the Compute Express Link (CXL) Interconnect. *ACM Comput. Surv.* 56, 11, Article 290 (jul 2024), 37 pages. doi:10.1145/3669900

[5] Peter J. Denning. 2005. The locality principle. *Commun. ACM* 48, 7 (jul 2005), 19–24. doi:10.1145/1070838.1070856

[6] Peter J. Denning. 2021. Working Set Analytics. *ACM Comput. Surv.* 53, 6, Article 113 (Feb. 2021), 36 pages. doi:10.1145/3399709

[7] Oliver Giersch, Dustin Nguyen, Jörg Nolte, and Wolfgang Schröder-Preikschat. 2024. Virtual Memory Revisited for Tiered Memory *(APSys'24)*.

[8] Hongjing Huang, Zeke Wang, Jie Zhang, Zhenhao He, Chao Wu, Jun Xiao, and Gustavo Alonso. 2022. Shuhai: A Tool for Benchmarking High Bandwidth Memory on FPGAs. *IEEE Trans. Comput.* 71, 5 (2022), 1133–1144. doi:10.1109/TC.2021.3075765

[9] Intel 2022. *Intel® 64 and IA-32 Architectures Software Developer's Manual: System Programming Guide, Volume 3 (3A, 3B, 3C & 3D)*. Intel.

[10] Manuel Le Gallo and Abu Sebastian. 2020. An overview of phase-change memory device physics. *Journal of Physics D: Applied Physics* 53, 21 (mar 2020), 213002. doi:10.1088/1361-6463/ab7794

[11] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. MEMTIS: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination. In *Proceedings of the 29th Symposium on Operating Systems Principles* (, Koblenz, Germany,) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 17–34. doi:10.1145/3600006.3613167

[12] Collin McCurdy and Jeffrey Vetter. 2010. Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. 87–96. doi:10.1109/ISPASS.2010.5452060

[13] Seonjin Na, Geonhwa Jeong, Byung Hoon Ahn, Jeffrey Young, Tushar Krishna, and Hyesoon Kim. 2024. Understanding Performance Implications of LLM Inference on CPUs. In *2024 IEEE International Symposium on Workload Characterization (IISWC)*. 169–180. doi:10.1109/IISWC63097.2024.00024

[14] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 89–100. doi:10.1145/1250734.1250746

[15] Aleix Roca Nonell, Balazs Gerofi, Leonardo Bautista-Gomez, Dominique Martinet, Vicenç Beltran Querol, and Yutaka Ishikawa. 2018. On the Applicability of PEBS based Online Memory Access Tracking for Heterogeneous Memory Management at Scale. In *Proceedings of the Workshop on Memory Centric High Performance Computing* (Dallas, TX, USA) *(MCHPC'18)*. Association for Computing Machinery, New York, NY, USA, 50–57. doi:10.1145/3286475.3286477

[16] OSIS 2019. *Virtual I/O Device (VIRTIO) Version 1.1*. OSIS.

[17] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. 2010. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *SIGOPS Oper. Syst. Rev.* 43, 4 (Jan. 2010), 92–105. doi:10.1145/1713254.1713276

[18] Mathias Payer, Enrico Kravina, and Thomas R. Gross. 2013. Lightweight Memory Tracing. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX Association, San Jose, CA, 115–126. https://www.usenix.org/conference/atc13/technical-sessions/presentation/payer

[19] Jie Ren, Dong Xu, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. 2024. MTM: Rethinking Memory Profiling and Migration for Multi-Tiered Large Memory. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 803–817. doi:10.1145/3627703.3650075

[20] Sai Sha, Chuandong Li, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. 2023. vTMM: Tiered Memory Management for Virtual Machines. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 283–297. doi:10.1145/3552326.3587449

[21] Yupeng Tang, Ping Zhou, Wenhui Zhang, Henry Hu, Qirui Yang, Hao Xiang, Tongping Liu, Jiaxin Shan, Ruoyun Huang, Cheng Zhao, Cheng Chen, Hui Zhang, Fei Liu, Shuai Zhang, Xiaoning Ding, and Jianjun Chen. 2024. Exploring Performance and Cost Optimization with ASIC-Based CXL Memory. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 818–833. doi:10.1145/3627703.3650061

[22] Teja Tscharntke, Michael E Hochberg, Tatyana A Rand, Vincent H Resh, and Jochen Krauss. 2007. Author sequence and credit for contributions in multiauthored publications. (2007). doi:10.1371/journal.pbio.0050018

[23] John Tsiombikas. [n. d.]. https://github.com/jtsiomb/c-ray/tree/6443d91189a7317385b15af5292c4cf5b3863667. ([n. d.]).

[24] R. Uhlig, G. Neiger, D. Rodgers, A.L. Santoni, F.C.M. Martins, A.V. Anderson, S.M. Bennett, A. Kagi, F.H. Leung, and L. Smith. 2005. Intel virtualization technology. *Computer* 38, 5 (2005), 48–56. doi:10.1109/MC.2005.163

[25] Lukas Wenzel, Sven Köhler, Henriette Hofmeier, and Felix Eberhardt. 2023. Quick-and-dirty memory access tracing with instruction-based sampling. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)*. doi:10.13154/294-10270

[26] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Nimble Page Management for Tiered Memory Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 331–345. doi:10.1145/3297858.3304024

[27] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 169–182. https://www.usenix.org/conference/fast20/presentation/yang

[28] Jifei Yi, Benchao Dong, Mingkai Dong, and Haibo Chen. 2020. On the precision of precise event based sampling. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '20)*. ACM Digital Library, 98–105. doi:10.1145/3409963.3410490