(No)Compromis: Paging Virtualization Is Not a Fatality

Boris Teabe Djomgwe, Peterson Yuhala, Alain Tchana, Fabien Hermenier, Daniel Hagimont, Gilles Muller

(No)Compromis: Paging Virtualization Is Not a Fatality

Boris Teabe
University of Toulouse, France

Peterson Yuhala
University of Neuchatel, Switzerland

Alain Tchana
ENS Lyon, France

Fabien Hermenier
Nutanix, France

Daniel Hagimont
University of Toulouse, France

Gilles Muller
INRIA, France

ABSTRACT

Nested/Extended Page Table (EPT) is the current hardware solution for virtualizing memory in virtualized systems. It induces a significant performance overhead due to the 2D page walk it requires, thus 24 memory accesses on a TLB miss (instead of 4 memory accesses in a native system). This 2D page walk constraint comes from the utilization of paging for managing virtual machine (VM) memory. This paper shows that paging is not necessary in the hypervisor. Our solution *Compromis*, a novel Memory Management Unit, uses direct segments for VM memory management combined with paging for VM's processes. This is the first time that a direct segment based solution is shown to be applicable to the entire VM memory while keeping applications unchanged. Relying on the 310 studied datacenter traces, the paper shows that it is possible to provision up to 99.99% of the VMs using a single memory segment. The paper presents a systematic methodology for implementing *Compromis* in the hardware, the hypervisor and the datacenter scheduler. Evaluation results show that *Compromis* outperforms the two popular memory virtualization solutions: shadow paging and EPT by up to 30% and 370% respectively.

CCS CONCEPTS

• **Software and its engineering → Virtual machines**.

KEYWORDS

Virtualization, Memory, Segmentation, Pagination

1 INTRODUCTION

Virtualization has become the *de facto* cloud computing standard because it brings several benefits such as optimal server utilization, security, fault tolerance and quick service deployment [1, 3, 10]. However, there is still room for improvement, mainly at the memory level which represents up to 90% [46] of the global virtualization overhead.

Memory virtualization overhead comes from the necessity to manage three address spaces (application, guest OS and host OS) instead of two (application and OS) as in native systems. Shadow paging [44] is the most popular memory virtualization solution. Each page table inside the guest OS is shadowed by a corresponding page table inside the hypervisor, which contains the real mapping between Guest Virtual Addresses (GVA) and Host Physical Addresses (HPA). Thus, shadow page tables are those used for address translation by the hardware page table walker (which resides inside the Memory Management Unit, MMU). Page tables inside the guest OS are never used.

Shadow paging leads to one-dimensional (1D) page walk on a TLB miss, as in a native system. However, building shadow page tables comes with costly context switches between the guest OS and the hypervisor for synchronization. Nested/Extended Page Table (EPT) [15, 43] has been introduced for avoiding page table synchronization cost. It improves the page table walker to walk through two page tables (from the guest and from the hypervisor) at the same time in a 2D manner. Thus, building the shadow page table does not require the protection of guest's page tables. This drastically reduces the number of context switches. However, this solution induces several memory accesses during address translation due to the 2D page walk mechanism. In a radix-4 page table [41] (the most popular case) for instance, this 2D page walk leads to 24 memory accesses on each TLB miss instead of 4 in a native system, resulting in significant performance degradation.

While many recent works proved the effectiveness of paging when dealing with processes (*e.g.*, for reducing memory fragmentation), to the best of our knowledge, there is no clear assessment of its effectiveness when dealing with virtual machines (VMs). One explanation is that the implementation of hypervisors was inspired by the bare metal implementation of OSes. By analyzing traces from two public clouds (Microsoft Azure and Bitbrain) and 308 private clouds (managed by Nutanix[1]), we show in Section 4 that paging is not mandatory for managing memory allocated to VMs. In fact, we found that memory fragmentation is not an issue in virtualized datacenters thanks to VM memory sizing and arrival rate.

This paper presents *Compromis*, a novel MMU solution which uses segmentation for VMs and paging for processes inside VMs. *Compromis* allows a 1D page walk and generates zero context switch to virtualize memory. *Compromis* is inspired by *Direct Segment* (DS) introduced by Basu *et al.* [13] for native systems. For memory hungry applications which allocate at start time their entire memory and self-manage it at runtime (*e.g.*, Java Virtual Machine), their virtual memory space can be directly mapped to a large physical memory segment identified by a triple (*Base*, *Limit*, *Offset*). This way, the translation of a virtual address *va* is given by a simple register to register addition (*va*+*Offset*). *Compromis* generalizes DS to DS-n, allowing the provisioning of a VM with several memory segments. In *Compromis*, every processor context includes 2*n* new registers for address translation. Contrary to other DS based solutions [9, 24], *Compromis* considers the entire VM memory and requires no guest OS and application modification.

This paper also investigates systems implications and presents a systematic methodology for adapting the hypervisor and other cloud services (*e.g.*, datacenter scheduler) for making *Compromis* effective. To the best of our knowledge, this is the first DS-based approach in virtualized systems which puts the entire puzzle together.

[1] Nutanix is a world wide private cloud provider.

We have implemented a whole prototype in both Xen and KVM virtualized systems managed by OpenStack. This involves: firstly, integrating our DS-aware memory allocation algorithm; secondly: improving the Virtual Machine Control Structure (VMCS) data structure for configuring new hardware registers introduced by DS-n; and finally, improving OpenStack's VM placement algorithm to minimize the number of memory segments. To evaluate our prototype, since our solution relies on hardware modifications, we mimicked the functioning of a VM running on a DS-n machine in the following fashion: We run the VM in para-virtualized (PV) mode [11], using its 1D page walk as DS-n. However in PV mode, all page table modifications performed by the VM kernel trap into the hypervisor using hypercalls. To avoid this behavior which will not exist on a DS-n machine, we modified the guest kernel to directly set page table entries with the correct HPAs, calculated in the same way as a DS-n hardware would have done.

In total, our paper makes the following contributions:

- We first studied the potential effectiveness of DS in virtualized datacenters. In other words, we answered the following question: considering VM memory demands, arrival times and departure times, is it advantageous to provision all or the majority of VMs with one large memory segment? To answer this question, we studied memory fragmentation in virtualized systems by analyzing traces from two public clouds (Microsoft Azure [21] and Bitbrains [40]) and 308 private clouds. We found that using a DS aware memory allocation system, memory fragmentation is not a critical issue in virtualized datacenters like in native ones.
- Drawing on this observation, we proposed DS-n, a generalization of DS to provision a VM with multiple memory segments. We presented the hardware modifications required by DS-n.
- We proposed a DS-aware VM memory allocation algorithm which minimizes the number of memory segments to use for each VM.
- We evaluated the performance gain of DS-n using an accurate methodology on a real environment. The main results are as follows. Firstly, the analyzed datacenter traces show that it is possible to provision up to 99.99% of VMs with one memory segment, while three segments are sufficient for provisioning all VMs. Secondly, concerning the performance gain, DS-n reduces memory virtualization overhead to only 0.35%, outperforming both shadow paging and EPT by up to 30% and 370% respectively. The results also show that our memory allocation algorithm runs faster than traditional ones. For example, Xen's algorithm is outperformed by 80%.

The remainder of the paper is as follows. Section 2 presents the necessary background to understand our contributions. Section 3 evaluates the limitations of state-of-the-art solutions. Section 4 presents the analysis of several production datacenter traces and validates the opportunity to apply DS in virtualized systems. Section 5 presents the necessary hardware and software improvements to make DS-n effective. Section 6 presents the evaluation results. Section 7 presents works relevant to our contributions. Section 8 concludes the paper.

Table 1: Benchmarks used for assessment and evaluation of our solution.

Benchmark	Description
SpecCPU 2006	Compute multi-threaded workloads
PARSEC 3.0	Compute multi-threaded workloads
Redis	In-memory database
Elastic search	In-memory database

2 BACKGROUND

This section presents two main techniques used to achieve memory virtualization, namely Shadow paging [44] and Extended Page Table (noted EPT) [15, 43].

2.1 Shadow Paging

Shadow paging is a software memory virtualization technique. In Shadow paging, the hypervisor creates a *shadow* Page Table (PT) for each guest PT. This shadow PT holds the complete translation from GVA to HPA. It is walked by the Hardware Page Table Walker (HPTW) on a Translation Look-aside Buffer (TLB) miss. Guest Page Tables (GPTs) are fake page tables not utilized by hardware. To put in place shadow paging, the hypervisor write protects both the CR3 register (which holds the current PT address) and GPTs. Each time the guest OS attempts to modify these structures, the execution then traps into the hypervisor, which in turn updates the CR3 register or the shadow PT. Using shadow paging, the HPTW only performs a 1D page walk as in a native system, leading to 4 memory accesses per TLB miss. However, the resulting context switches during these traps severely degrade the VM performance.

2.2 Extended Page Table

EPT (also called Nested Page Table) is a hardware-assisted memory virtualization solution proposed by many chip vendors such as Intel and AMD. It relies on a two layer PT: the first PT layer resides in the guest address space and is exclusively managed by its OS, at the ratio of one PT per process. This first layer PT thus contains GPAs which point to guest pages in the guest address space. Every process context switch triggers the guest OS to set the CR3 register with the GPA of the scheduled-in process's PT address. The second PT layer resides in the hypervisor, at the ratio of one PT per VM. This PT represents the address space of the guest, and includes HPAs that point to pages (real pages in RAM) in the host address space. Every vCPU context switch triggers the hypervisor to set the nested CR3 register (nCR3) with the HPA of the scheduled-in VM's PT address. On a TLB miss, the hardware page table walker translates a virtual address va into the corresponding HPA by performing a 2-dimensional page walk, leading to 24 memory accesses per TLB miss.

3 ASSESSMENT

This section presents the overhead of memory virtualization in both native and virtualized systems. Note that even in a native system, the expression "memory virtualization" is used because of the mapping of the process linear address space to the physical address space.

Table 2: Formulas to estimate the overhead of memory virtualization. ("handler" is the handler which treats the VMExit generated when the guest OS attempts to modify the page table.)

Native	total cycles of all page walks
EPT	total cycles of all (GPT+EPT) walks
Shadow	total cycles of all (hypervisor level PT walk+ VMEntry+VMExit+handler)

Methodology. Table 1 lists the benchmarks used to evaluate the performance overhead of memory virtualization. In order to evaluate huge page-based solutions, we executed the benchmarks while varying the memory page size in both the hypervisor and the guest OS. The notation ***g****X*-***h****Y* means *X* and *Y* are the memory page sizes in the **g**uest OS and the **h**ypervisor, respectively. We use the time taken by both the hardware and the software for memory virtualization as our evaluation metric. Table 2 presents how this metric is calculated for each virtualization technology. We rely on both Performance Monitoring Counters (PMC) and low-level software probes that we have written for the purpose of this paper. Details on the experimental setup are given in Section 6.2.1.

Results. Figure 1 presents our benchmark results. Firstly, even in native systems, memory virtualization takes a significant proportion of an application's execution time, up to 42% for the *mcf* benchmark. Secondly, running applications in a virtualized environment increases the said proportion up to 50.93% for Elastic Search under shadow paging. Thirdly, shadow paging incurs more overhead than EPT for the majority of applications, with up to 43.89% of difference for *vips*. Finally, we can observe that even when huge pages are used simultaneously in the guest OS and hypervisor, memory virtualization overhead is still high, at almost 31.5% for the Redis benchmark. [9, 13, 24, 36] reached the same conclusion with the use of huge pages.

Conclusion. These results show that the overhead of memory virtualization is very significant in a virtualized system, even when utilizing huge pages. The root cause of this overhead is the utilization of paging as a basis for memory virtualization in VMs.

4 PAGING IS NOT A FATALITY FOR VMS

Several research works tried to reduce the overhead of memory virtualization in virtualized systems. However, no work has questioned the relevance of paging in this context. This section studies the (ir)relevance of paging when dealing with VMs. To this end, we compared paging with segmentation, an alternative approach that is often left out.

Paging involves organizing both the virtual address space of a process and the physical address space into fixed size memory chunks (4KB, 2MB, *etc.*) called pages. Thus, each virtual page can be housed in any physical page frame. The process PT and the HPTW make it possible to find the actual mapping of a virtual page to a physical page address. Segmentation, on the other hand, organizes both the virtual and the physical address spaces in the form of variable size memory chunks called segments. The size of the latter is chosen by the programmer. The correspondence between virtual segments and physical segments is provided by a segment table. The virtual address to physical address mapping is done by a simple addition.

The main reasons to promote paging over segmentation in native systems are as follows: (R_1) Paging is invisible to the programmer; this is not the case with segmentation, which makes application programming more difficult. (R_2) Paging makes implementing OS memory allocators easier. Indeed, it only requires the use of a list of free pages; it is sufficient to simply choose any page within this list upon receiving a memory allocation request. This property is important for scalability; segmentation, in contrast, requires the OS to find an appropriate physical segment that satisfies the size of the virtual segment requested by the application. (R_3) Paging limits memory fragmentation [2], which is not the case with segmentation. (R_4) Paging allows overcommitment, which is useful for optimizing memory utilization.

The question is whether all of these reasons are valid when manipulating VMs. To answer this question, we analyzed the relevance of each reason in virtualized systems. Before continuing, note that when we talk about memory management in a virtualized system, we are talking about the allocation of physical memory to VMs and not memory allocation to applications inside VMs.

Relevance of R_1 (Segmentation makes application programming more difficult). This reason is valid in native environments (when dealing with applications) because application programmers do not have the expertise to manage segment size in a segmentation based system. Moreover, such a problem is not relevant to the business logic of their applications. On the other hand, with VMs the developers are experts in operating systems; therefore it is within their reach to have the responsibility to manage memory segments.

Relevance of R_2 (Paging makes memory allocation easier). It is necessary to facilitate the work of the memory allocator for scalability purposes. In a native system, the memory allocator is subject to thousands of memory allocation and deallocation requests per second. This is not true when dealing with VMs. Each VM performs only one allocation (at startup) and deallocation (at shutdown). Thus, the frequency of memory allocation and deallocation requests received by the hypervisor is not on the same order of magnitude as those received by the OS in a native system. Table 3 presents the average memory allocation frequency received by a server from a native system, virtualized private clouds, as well as public clouds (see Section 6 for more details on the analyzed datasets). We observed a phenomenal difference between native and virtualized systems, which are quite stable. Given the extremely low values for virtualized datacenters, finding free memory chunks with segmentation when dealing with VMs is not a difficult problem.

Relevance of R_3 (Paging limits fragmentation). Fragmentation is due to the heterogeneity of memory allocation sizes. Indeed, a system in which all allocation sizes are identical would not suffer from fragmentation. To verify whether fragmentation could be a problem in a virtualized datacenter, we analyzed the memory allocation sizes of the traces from the datacenters presented in Table 3. Figure 2 shows the CDF of these allocation sizes.

[2]Internal fragmentation within pages is always possible, but it is negligible compared to the external fragmentation caused by segmentation

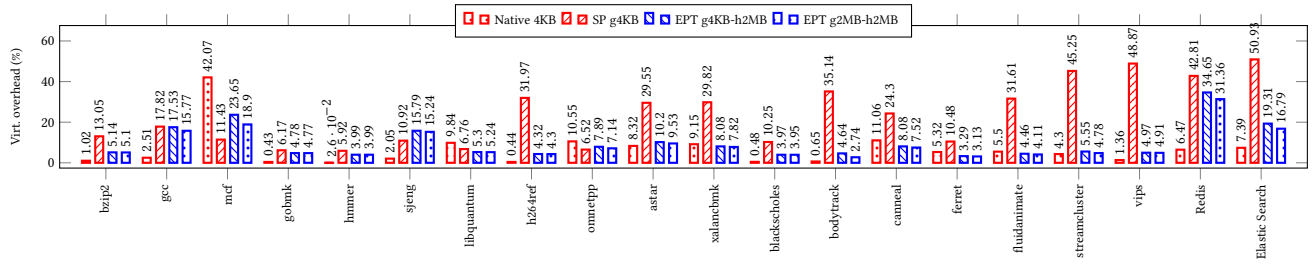

Figure 1: Proportion of CPU time used for memory virtualization in native, virtualized shadow paging (SP) and virtualized EPT.

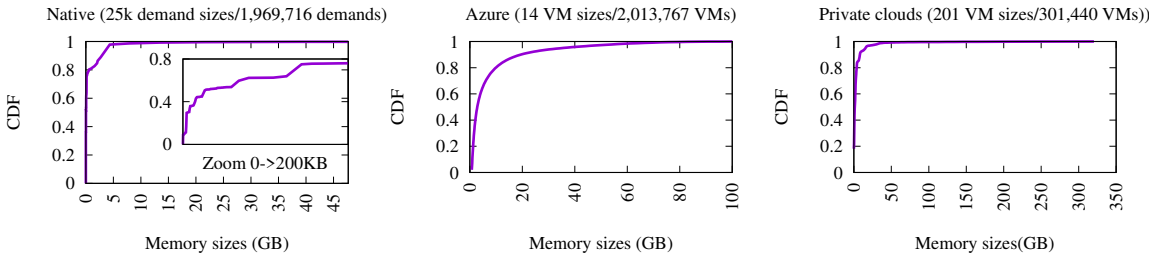

Figure 2: CDF of memory allocation sizes in different datacenter types.

Table 3: Memory allocation frequency (per hour on a server) in native and virtualized datacenters.

Dataset	Alloc./Hour/Server
Native - Our lab machine	82071.5
Virtualized - Private clouds	0.056
Virtualized - Microsoft Azure	0.31

We observe that public clouds stand out with very few different allocation sizes (14). This is because in public clouds VM sizes are imposed by the provider. Things are slightly different in private clouds (201) where there is more freedom in VM size definition. On the contrary, allocation sizes vary a lot in native systems (25k) than in virtualized environments. These results show that fragmentation is not a relevant issue when dealing with VMs.

Relevance of R_4 (Paging allows memory overcommitment). Overcommitment is a practice which allows to reserve more memory than the physical machine actually has. It exploits the fact that not all applications require their entire memory demand at the same time. As a result, overcommitment helps avoid resource waste. However, overcommitment comes with performance degradation during memory reclamation and performance unpredictability [32]. These limitations are acceptable in a native system because there is no contract between application owners and the datacenter owner; they both belong to the same company. Best-effort allocation becomes the practice in such contexts. Things are different in a virtualized datacenter, especially in commercial clouds, where the datacenter operator must respect the contract signed with the VM owner, who paid for the reserved resources. Therefore, even if a VM is not using its resources, these resources have already been amortized. The necessity to avoid resource waste is thus less critical compared to a native system. Furthermore, the implementation of overcommitment in a virtualized system is challenging because of the blackbox nature of a VM [32]. It requires expertise in the workload and the system to configure it and to react in case of performance issues. As a consequence, no public cloud supports overcommitment. Private cloud providers either do not support it (Nutanix), disable it by default (VMWare TPS, Hyper-V dynamic memory) or enable it with extra warnings (RedHat with KVM).

5 *COMPROMIS*: A DS-BASED MEMORY VIRTUALIZATION APPROACH FOR VMS

Compromis is a hardware memory virtualization solution implemented within the MMU that exploits the strengths of both direct segments (DS) and paging. The former is used by the hypervisor to deal with VMs, while the latter is used by the guest OS to deal with processes. The innovation is the utilization of DS instead of paging by the hypervisor. Considering the fact that it may be impossible to satisfy a VM allocation request using a single memory segment, *Compromis* generalizes DS to DS-n, where a VM is allocated k segments ($1 \leq k \leq n$) using the *Compromis* hardware feature. This section presents the set of improvements that should be applied to the datacenter stack in order to make *Compromis* effective.

5.1 General Overview

Figure 3 presents the general operations of a datacenter using *Compromis*. When a user requests a VM creation from a certain VM flavor (#CPU, memory size), the cloud scheduler chooses the physical machine that will host the said instance. This choice is made according to a placement policy, which generally takes into account constraints in resource availability. In a *Compromis*-aware

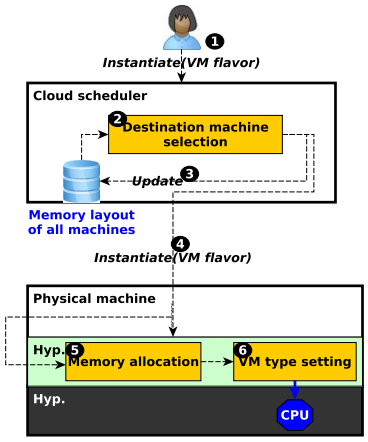

Figure 3: General functioning of a datacenter which implements *Compromis*.

datacenter, this policy is extended to choose the machine with the greatest chance of allocating large memory segments to the VM. To this end, the cloud scheduler quickly simulates the execution of the memory allocator implemented by the hypervisor or compute nodes. This simulation is built by the cloud scheduler on top of every machine's memory layout state, which is locally stored and periodically updated (see Section 5.4).

When the hypervisor of the selected physical machine receives a VM creation request, it reserves memory for the VM in the form of large memory segments rather than small page chunks as it is currently done (see Section 5.3). If the number of segments used to satisfy the VM is less than or equal to n, then the hypervisor configures the VM in DS-n mode, a new mode implemented in hardware (see Section 5.5.1). Otherwise, the VM is configured in shadow or EPT mode, depending on the datacenter operator. In DS-n mode, the hardware performs an address translation by doing a 1D page walk (instead of 2D) followed by a series of register to register operations (see Section 5.2). Notice that a *Compromis*-aware machine can simultaneously run DS-n and non-DS-n VMs. The next subsections detail the modifications that should be applied to each datacenter layer for building *Compromis*.

5.2 Hardware-Level Contribution

A hardware which implements *Compromis* includes new registers to indicate the mapping of GPA segments (in the guest address space) to HPA segments (in the host address space). The value of each register comes from a Virtual Machine Control Structure (VMCS), configured by the hypervisor at VM startup (see Section 5.3). The number of added registers is a function of n. In particular, there are $n-1$ *guest base registers* (noted $GBReg_1$, ..., $GBReg_{n-1}$, no such registers in DS-1), n *host base registers* (noted $HBReg_0$, ..., $HBReg_{n-1}$), and the *limit* register. These registers indicate the mapping as follows: the GPA segment $[0, GBReg_1 - 1]$ is mapped to the HPA segment $[HBReg_0, GBReg_1 - 1]$, and each GPA segment $[GBReg_{i-1}, GBReg_i - 1]$ is mapped to a HPA segment $[HBReg_{i-1}, HBReg_{i-1} + (GBReg_i - GBReg_{i-1})]$ (where $GBReg_i - GBReg_{i-1}$ is the size of this segment). For a VM with k segments, the mapping of the last GPA segment
$GBReg_{k-1}, GBReg_{k-1} + (limit - HBReg_{k-1})$
is the HPA segment $[HBReg_{k-1}, \text{limit}]$. Once the hypervisor finishes configuring these registers, the translation of a virtual address *va* to the corresponding HPA *hpa* for a DS-n VM type (whose number of segments is less than n) is summarized in Figure 4. Firstly, the MMU performs a 1D GPT walk, taking as input *va* and returning a GPA *gpa*. *hpa* is then calculated as follows:

$$hpa = HBRreg_i + (gpa - GBRreg_i) \quad (1)$$

with $[GBRreg_i, *]$ being the smallest GPA segment which contains *gpa*. If no such segment exists, a boundary violation is raised and trapped in the hypervisor as a "DS-n violation" exception. More generally, for each *gpa* extracted from a GPT layer, we perform an offset addition followed by a comparison, meaning that every EPT walk is replaced by these two operations. For instance, when the VM has only one segment, the computation of *hpa* is as follows

$$hpa = HBRreg_0 + gpa \quad (2)$$

We raise a boundary violation if *hpa* is greater than *limit*. The performance benefit of these operations compared to the 2D page walk done in EPT is discussed in Section 6.

5.3 Hypervisor-Level Contribution

The hypervisor needs two main changes: the integration of a memory allocator for VMs and the configuration of the VMCS to indicate DS-n type VMs.

5.3.1 A DS-Based Memory Allocator for VMs. We assume that the physical memory is organized in two parts: the first part is reserved for the hypervisor and privileged VM tasks, while the second part is dedicated to user VMs. This memory organization is found in almost all popular hypervisors. In *Compromis*, the first memory part is managed using the traditional memory allocator. Concerning the second memory part, a new allocation algorithm is used to enforce large memory segment allocation to VMs. This section describes this new allocator.

Implementing a memory allocator requires answering three questions: (Q_1) which data structure to use for storing information about free memory segments? (Q_2) how do we choose elements from this data structure for responding to an allocation request? (Q_3) how do we insert an element into this data structure when there is a memory de-allocation?

Answer to Q_1: data structure. We use a doubly-linked list to describe free memory segments (a list is the data structure used by most memory allocators). Each element in the list describes a segment using three variables:

- *base*: start address of the segment;
- *limit*: end address of the segment;
- *date*: allocation date of the segment containing *base-1*.

The elements of the list are ordered in an ascending order of *base*. We hereafter note each element $[base, limit, date]$.

Answer to Q_2: allocation policy. When the hypervisor receives a request to start a VM with a memory allocation M, it goes through the list described above to find out which segments should be allocated to the VM. If it finds a segment of size M, then that

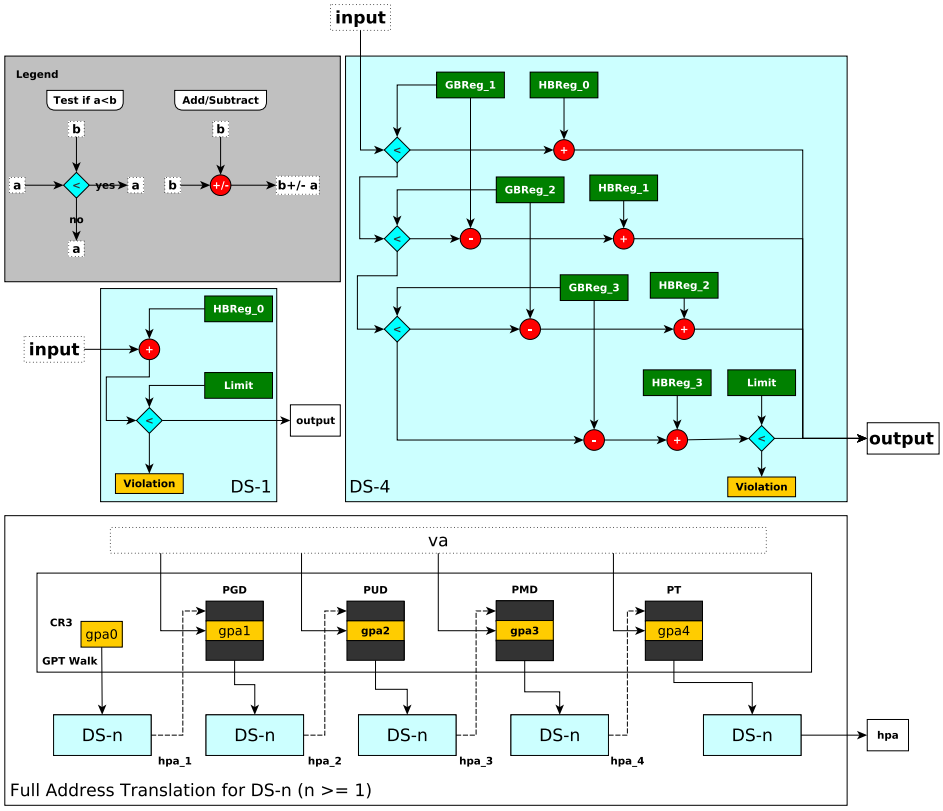

Figure 4: Address translation handling in two DS-n machine types (left: DS-1, right: DS-4).

segment is taken off the list and allocated to the VM. Otherwise, the allocator chooses the largest segment $S_b[base, limit_i, date]$ among those whose sizes are greater than M. The VM allocation is satisfied with a portion of S_b. Note that taking the largest segment prevents the aggregation of small segments, which is bad for a DS based approach. If there is no segment larger than M, then two options are possible. The first option (*Opt1*) satisfies the VM with smaller segments. This gives a chance for VMs coming later to use the big free segments. The second option (*Opt2*) chooses the largest segment that exists and executes the above algorithm with a new memory size M', where M' equals M minus the size of the chosen segment.

Compromis offers these two options to support various workload patterns and datacenter constitutions. A workload pattern is the set of VM creation and shutdown requests submitted to the datacenter during a period of time. The constitution of a datacenter is the physical machine sizes. It is the responsibility of the cloud scheduler, which has a global view of the datacenter, to choose among these options (see Section 5.4).

Answer to Q_3: freed memory taken into account. Stopping a VM results in the freeing of its memory, which has to be inserted into the list of free memory segments. Let S be a memory segment to insert into the list. If S coincides with the beginning or the end of a segment S' in the list, then S' is simply extended (forward or backward). If this extension causes the new big segment to coincide with the beginning or the end of the segments that follow or precede it, then the extension continues. If there is no shared border between S and the existing segments in the list, then S is inserted in the list so that the ascending order is respected.

5.3.2 VM Type Configuration. Let k be the number of memory segments allocated to a VM. If $k \leq n$ then the VM is of type DS-n, otherwise it is configured with EPT or shadow paging according to the datacenter administrator's choice. The type configuration of a VM is done by modifying the VMCS of its vCPUs. To indicate that a VM is of type DS-n, a new bit of the *Secondary Processor-Based VM-Execution Controls* is set. Otherwise, this bit remains at zero. For DS-n VMs, the hypervisor also positions new VMCS fields that will be used to populate *GBReg*, *HBReg*, and *limit* registers. The fields are populated in ascending order of crossing segments. The values of the fields which map to *HBReg* and *limit* registers come from the list of segments allocated to the VM. Concerning the fields which map to *GBReg* registers, their values are calculated as they are filled. When $k < n$, the remaining fields are set to zero.

5.4 Cloud Scheduler Level Contribution

The cloud scheduler is improved for two purposes: a DS-n-aware VM placement algorithm and selection of a memory allocation option.

VM placement algorithm improvement. The placement algorithm determines the physical machine that will instantiate the VM. Traditionally, this algorithm has a particular objective, e.g. load balancing. For example, the schedulers of OpenStack [23] and CloudStack [20] consist of a list of filters. Each filter implements a set of constraints such as resource-matchmaking [37], VM-VM or VM-host (anti-) affinities. A filter receives as arguments a set of possible machines for the VM to boot on, and removes among them those not satisfying its constraints. Each time the scheduler is invoked to decide where to place a VM, it chains the filters to eventually retrieve the satisfactory machines and picks one among them.

To enjoy the benefit of DS-n, the VM scheduler must integrate inside its objective the maximization of the number of VMs of type DS-n. For filter-based schedulers, this consists of implementing a new filter to append to the existing list. This filter maintains a local copy of the free memory segments on every machine, and uses a simulator to evaluate the number of segments that will be used if the VM is instantiated on each machine. It then selects the machine leading to the fewest number of segments. For schedulers that do not rely on filters, the strategy is to weigh the existing objectives against the ones that consists of picking the machine minimizing the number of memory segments. Note that this cloud scheduler modification does not reduce the hosting capacity of the datacenter, because the destination machine is selected among the original cloud scheduler candidates.

Memory allocation option selection. Section 5.3.1 noted that the cloud scheduler has the responsibility to select the memory allocation option that all hypervisors will use. To this end, it embeds a memory allocator simulator which implements the two options presented in 5.3.1. Then it periodically (e.g. every week) replays the recorded VM startup and shutdown logs in its simulator. This is done while varying the memory allocation option. The selected option is the one that produces the most DS-n VMs. All hypervisors are then notified of the selected option and the log repository is reset.

5.5 Prototype

We implemented *Compromis* in two popular hypervisors (Xen and KVM), as well as in OpenStack's Nova scheduler.

5.5.1 Implementation in the Hypervisor.

Implementation in Xen. The implementation of *Compromis* in Xen is straightforward. Firstly, Xen already organizes the main memory in two parts as we wished. The first part is managed by the Linux memory allocator subsystem hosted within the privileged VM (*dom0*). The memory allocator for user VMs resides in the hypervisor core, and is invoked by the *dom0* during the VM instantiation process. We simply replaced this allocator with the one described in Section 5.3.1. We validated the effectiveness of this algorithm by starting VMs in hardware-assisted virtualization (HVM) mode with single segments, while the hypervisor still uses EPT for address translation.

Concerning the configuration of the VM type, the modification of Xen does not require any particular description other than what has been said in Section 5.3.2. Concerning the handling of cloud scheduler notifications related to the changing of the memory allocation option, we defined a new hypercall that informs the hypervisor of the selected option.

Implementation in KVM. Unlike Xen, KVM does not hold memory in two blocks. KVM relies on the Linux memory allocator which sees VMs as normal processes. To implement *Compromis* in KVM, we first enforced the organization of the physical memory in two blocks using the *cgroup* mechanism. The default Linux memory allocator is then associated to the first block while our memory allocator manages the second block. The */proc* file system is used to record the memory allocation option imposed by the cloud scheduler.

5.5.2 Implementation in the Cloud Scheduler. The implementation of *Compromis* in OpenStack Nova is quite straightforward because Nova's placement algorithm is easy to identify. Its execution steps are also easy to identify, making its extension with a memory allocation simulator very simple. Concerning the periodical selection of the memory allocation option, we implemented a separate process which starts at the same time as Nova. That process relies on existing OpenStack logs for obtaining VM startup and shutdown requests.

5.6 Discussion

Memory over commitment and VM migration. Since *Compromis* allows DS-n, it is possible to implement memory over commitment by performing dynamic segment resizing, addition or removal, combined with a slight cooperation between the guest OS and the hypervisor. A VM which needs more memory gains new segments or sees its segments extended. Inversely, a VM whose memory needs to be reduced will see either its segment sizes or number reduced. The cooperation between the guest OS and hypervisor is only necessary in this case. In fact, the hypervisor should indicate to the guest OS the range of GPAs that should be released by the VM (using the balloon driver mechanism). Indeed, the hypervisor is the only component which knows segment ranges.

Memory Mapped IO (MMIO) region virtualization. IO device emulation and direct IO are two IO virtualization solutions implemented by hypervisors in HVM mode. The former and most popular solution involves protecting virtual MMIO ranges seen by the guest OS so that all IO operations performed by the guest trap into the hypervisor. With this IO virtualization solution, the utilization of *Compromis* is straightforward since virtual MMIO regions are at the GPA layer. The validation step presented in Section 5.5.1 was performed under this solution. With direct IO virtualization, the guest OS is directly presented with the physical MMIO ranges configured by the hardware device. This solution requires *Compromis* to use several memory segments. Note that this solution is not popular in today's clouds because it limits scalability (only enables very few virtual devices) and dynamic consolidation (VM live migration is not possible).

6 EVALUATIONS

We evaluated the following aspects of our solution: (1) Effectiveness (see section 6.1): the capability to start a large number of VMs using the DS-n technology; (2) Performance gain (see section 6.2): whether DS-n VMs improve the performance of applications running within; (3) Startup impact (see Section 6.3): whether there's any potential

positive/negative impact on VM startup latency. If not otherwise indicated, the hypervisor and cloud management system utilized in our evaluations are Xen and OpenStack respectively.

6.1 Effectiveness

Effectiveness evaluation is done by simulation using real datacenter traces.

6.1.1 Methodology. We developed a simulator which mimics a datacenter managed with OpenStack [23], improved with our contributions. The simulator replays VM startup and shutdown requests collected from several production datacenters, the details of which are presented in Section 6.1.2. The simulator considers each VM startup request to include a number of CPU cores and a memory size. For each simulated VM startup request, the simulator logs two metrics: the number of segments used for satisfying the VM memory demand, and the time taken by our changes (extension of the cloud scheduler and the utilization of our memory allocation algorithm in the hypervisor).

To highlight the benefits of each *Compromis* feature, we evaluated different versions of our improvements, including: – *BaseLine*: the simulator implements both the native OpenStack scheduler and Xen's memory allocation algorithms;
– *ImprovPlacement*: in this version, the VM placement algorithm is improved to choose for every VM the machine which will use the minimum number of memory segments (as described in Section 5.3.1);
– *DynamicOptionSelec*: in this version, the cloud scheduler calculates every week the best memory allocation option to be used (as described in Section 5.4).

6.1.2 Datasets. We used the traces of 2 public clouds (Bitbrains [40] and Microsoft Azure [21]) and 308 private clouds. Among other fields, each trace includes: the VM creation and destruction time, and the VM size (#CPU and memory size).

Bitbrains. This cloud is a service provider specialized in managing hosting and business computation for many enterprises. The dataset consists of 1,750 VMs, collected between August and September 2013. Bitbrains does not include physical machine characteristics.

Azure. This is a public Microsoft cloud. The dataset comprises 2,013,767 VMs running on Azure from November 16^{th}, 2016 to February 16^{th}, 2017.

Private clouds. This group aggregates data of 308 private IaaS clouds running diverse workloads between November 1^{st}, 2018 to November 29^{th}, 2018. For a given cloud, we collected one or more consistent snapshots of the cluster state at the moment the cluster triggered its hotspot mitigation service, which indicates that a machine is getting close to saturation. A snapshot depicts the running VMs, their sizing (in terms of memory and cores) and their host (in terms of available memory and cores). The collected dataset includes 301,440 VMs. As the dataset contains snapshots and not the VM creation and destruction time, we derived from each snapshot a *bootstorm* scenario where all the VMs are created simultaneously. This dataset includes server characteristics.

Composition and characteristics of servers used by Bitbrains and Azure. Having no hardware information about the first two datasets, we consider that they are composed of server generations presented in Table 4. We chose these server generations as they are used in Azure according to this Youtube video [2]. *Gen6* and *Godzilla* are new generations while *Gen2 HPC*, *Gen4* and *Gen5* are older ones. All server generations have the same proportion.

Table 4: Server generations used in the replay of Bitbrains and Azure traces.

Name	RAM (GB)	Cores	% in the traces
HPC	128	24	20
Gen4	192	24	20
Gen5	256	40	20
Gen6	192	48	20
Godzilla	512	32	20

Table 5: Number of memory segments allocated to VMs from Bitbrains and Azure.

Bitbrains				
Solution	**1 seg.**	**2 seg.**	**3 seg.**	**>3 seg.**
BaseLine	12.816	44.376	30.078	12.728
ImprovPlacement+Opt1	100	0	0	0
ImprovPlacement+Opt2	100	0	0	0
DynamicOptionSelec	100	0	0	0
Azure				
Solution	**1 seg.**	**2 seg.**	**3 seg.**	**>3 seg.**
BaseLine	3.581	11.171	9.996	75.252
ImprovPlacement+Opt1	99.9736	0.026	0	6.18E-05
ImprovPlacement+Opt2	99.947	0.007	0.022	0.021
DynamicOptionSelec	99.999	7.07E-04	0	0

6.1.3 Results. **Bitbrains and Azure - Table 5**. *BaseLine* provides better results in Bitbrains (up to 81% of VMs are satisfied with less than four memory segments) compared to Azure (only about 24% of VMs are satisfied with less than four memory segments). This is because the VMs running on Bitbrain have a longer life time than Azure. However, our solutions satisfy more VMs than *BaseLine* (99.95%-100%). This is because *BaseLine*, which implements Xen, organizes the physical memory in the forms of small memory chunks which are then used for allocation. As a naive algorithm, Xen cannot enforce DS to a VM even if it exists a free memory segment which is larger than the memory demand. In contrast, *Compromis* enforces DS to near-perfection (more than 99% of VMs are satisfied with only one memory segment). Our two memory application options discussed in 5.3.1 show their slight difference in Bitbrains: *ImprovPlacement+Opt1* satisfies more VMs with only one segment in comparison with *ImprovPlacement+Opt2*. Finally, dynamically switching between the two options (*DynamicOptionSelec*) is the best solution, with 99.99% of VMs using only one memory segment.

Private clouds - Figure 5. We plot the results for these clouds separately from the previous ones because of the multitude of different cloud environments. We can make the same observation as above. Our solutions satisfy almost all VMs with only one memory segment; observe the "wall" at 1 memory segment for all 3 versions of our improvement.

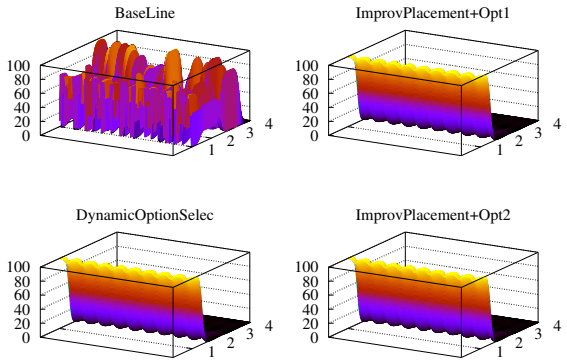

Figure 5: Number of memory segments allocated to VMs from 308 private clouds (longitude=cloud, latitude=#segment, depth=proportion).

6.2 Performance Gain

This section evaluates the performance gain brought by the utilization of DS-n.

6.2.1 Methodology. A DS-n machine handles a TLB miss using a 1D page walk followed by a set of register-to-register operations. We mimic this functionality by running the VM in para-virtualized (noted PV) mode [11] which also uses a 1D page walk. However in PV, all page table modifications performed by the VM kernel trap into the hypervisor using hypercalls. We modified the guest kernel to directly set page table entries with the correct HPAs, calculated in the same way as a DS-n hardware would have done. The reader could legitimately ask why use PV to simulate a hardware-assisted solution. We claim that our approach makes sense in our context because the benchmarks do not solicit the PV machinery: all disks are in-memory (*tmpfs*) based and all network requests use the *loopback* interface. Accordingly, only the memory subsystem is utilized.

The evaluation methodology we used is as follows. Let T_{1D} be the execution time of the VM in this modified PV context. We estimated the cost (noted $T^n_{reg2reg}$) of the register-to-register operations performed by the DS-n hardware on a TLB miss using an assembly code which executes these operations. It is adaptable depending on the value of n. Let N_{tlb} be the number of TLB misses (collected using PMC) generated by the application when it is executed in a native system. We estimate the execution time T_{DS-n} of a VM on a DS-n using the following formula:

$$T_{DS-n} = T_{1D} + N_{tlb} \times T^n_{reg2reg} \quad (3)$$

We evaluated different values of n from 1 to 3. We compare DS-n with EPT (in which the execution time is noted T_{ept}) and shadow paging (in which the execution time is noted T_{sha}). We used 4KB page size in guest VMs as the standard size. The characteristics of the experimental machine are presented in Table 6. Note that this machine includes a page walk cache [12]. The list of benchmarks we used (as previous work) are presented in Table 1. Each benchmark runs in a VM having a single vCPU and 5 GB of memory. The hypervisor and OS used in our evaluations are Xen 4.8 and Ubuntu 16.04 (Linux kernel 4.15) respectively.

Table 6: Characteristics of the experimental machine.

Processor	Single socket Intel(R) Core (TM) i7-3768 @2.40GHz 4cores
Memory	16GB DDR4 1600MHz
DTLB	4-way, 64 entries
ITLB	4-way, 128 entries

Table 7: The total cost (in seconds) of each memory virtualization technology for Redis, gcc and Elastic Search.

Technology	Redis	gcc	Elastic Search
C_{DS-n}	3	13	14
C_{EPT}	17	17	46
C_{Sha}	25	62	201

6.2.2 Results. Figure 6 presents the evaluation results. We only presented the results for DS-1 because we obtained almost the same results with DS-2 and DS-3. This is because the cost of register-to-register operations realized in DS-1, DS-2 and DS-3 is extremely low compared to the cost of a 2D page walk.

Figure 6 is interpreted as follows. First, obviously CPU-intensive-only applications (*e.g.*, hmmer from PARSEC) do not benefit enough from DS-n. Second, we confirm that DS-n almost nullifies the overhead of memory virtualization and leads the application to almost the same performance as in native systems. In fact, all black histogram bars are very close to 1. DS-n outperforms both EPT (up to 30% of performance difference for mcf) and shadow paging (up to 370% of performance difference for Elastic Search). Finally, we observe that DS-n produces a very low, close to zero, overhead (0.35%) but also a stable overhead (0.42 standard deviation). While a smaller overhead is always appreciated, a stable overhead can also be a requirement to host latency-sensitive applications, e.g. databases or real-time systems.

To justify the origin of this significant performance gap between these memory virtualization technologies, we analyzed the values of the internal metrics focusing on applications such as Redis, gcc, and Elastic Search. For DS-n, the cost of memory virtualization is $C_{DS-n} = C_{1D} \times N^{DS-n}_{tlb}$, where C_{1D} is the number of CPU cycles for performing a 1D page walk and N^{DS-n}_{tlb} is the number of TLB misses. For EPT, that cost is $C_{EPT} = C_{2D} \times N^{EPT}_{tlb}$, where C_{2D} is the number of CPU cycles used to perform a 2D page walk and N^{EPT}_{tlb} is the number of TLB misses. For shadow paging, the cost is $C_{Sha} = C_{1D} \times N^{Sha}_{tlb} + N^{Sha}_{exit} \times (C_{exit} + C_{enter} + C_{handler})$, where N^{Sha}_{tlb} is the number of TLB misses; N^{Sha}_{exit} is the number of VMExits related to page table modification operations; C_{exit} is the cost of performing VMExit followed by VMEnter; and $C_{handler}$ is the average execution time of memory management handlers in the hypervisor. Table 7 presents these particular costs on our experimental machine. We observe that C_{DS-n} is much lower than C_{EPT} (e.g., ×6 for Redis) and C_{Sha} (×14).

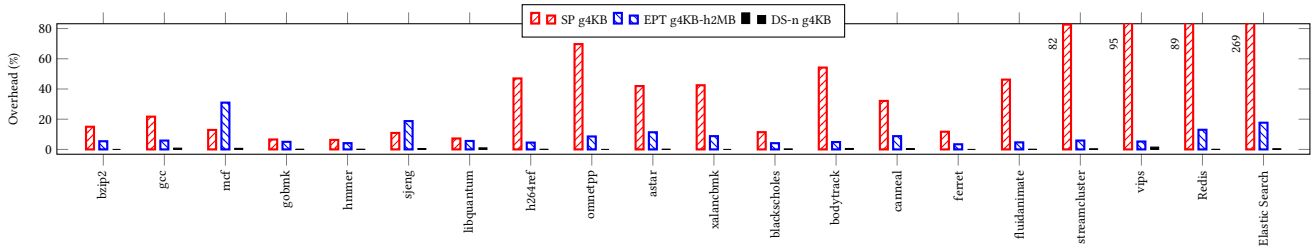

Figure 6: Performance overhead of DS-n compared with shadow paging (SP) and EPT. Lower is better.

Table 8: Memory allocation latency (mean-stdev) in ms.

Solution	Bitbrains	Azure	Private clouds
BaseLine	6.42-139.47	17.76-520.55	1.92-18.27
DynamicOptionSelec	3.57-1.23	3.42-1.18	0.098-0.011

6.3 Startup Impact

Recall that *Compromis* extends the Cloud scheduler (which intervenes at VM startup time) and changes the default memory allocator used by the hypervisor (also at VM start time). Therefore, one may legitimately ask where would these changes have an impact regarding VM startup latency. We answer this question by summing the cost of the extension with the cost of our memory allocation algorithm, then comparing it with the cost of the default Xen memory allocation algorithm. We rely on simulation logs generated during the evaluations presented in Section 6.1. The experiment shows that almost all of the different versions of our solution have the same complexity, thus we focus the results for *DynamicOptionSelec* in Table 8. These results are interpreted as follows. Firstly, we observe that our solution reduces the startup time by up to 80% for Azure VMs. This is because our allocation algorithm is simpler with regards to Xen, which organizes memory in several memory chunk lists and iterates over these lists several times to satisfy memory demand. Secondly, the smaller standard deviation illustrates that VM startup time with our solution is more stable than that of Xen. Such a predictability is critical for auto-scaling services, as demonstrated by Nitu *et. al.* [33]. The Xen unpredictability comes from its complex memory allocation algorithm presented above.

7 RELATED WORK

The overhead of memory virtualization has been measured by several previous works [12–14, 16–18, 22, 26, 30, 34, 35, 46]. It has also been shown that this overhead is exacerbated in virtualized environments [6–9, 15, 19, 24, 25, 25, 36, 43, 45, 46]. This section presents existing work in the latter context. The research in this domain can be classified into two categories: software and hardware-assisted solutions.

Software Solutions. Direct paging [5] is similar to shadow paging [44] (presented in Section 2.1), but it requires the modification of the guest OS. In Direct paging [5], the hypervisor introduces an additional level of abstraction between what the guest sees as physical memory and the underlying machine memory. This is done through the introduction of a Physical to Machine (P2M) mapping maintained within the hypervisor such as in shadow paging. The guest OS is aware of the P2M mapping and is modified in a such a way that it writes entries mapping virtual addresses directly to the machine address space by using the P2M, instead of writing PTEs. Like shadow paging, direct paging uses a 1D page walk to handle a TLB miss. However, it includes two main drawbacks: context switches between the guest and the hypervisor for building the P2M table, and the modification of the guest OS (making proprietary OSes such as Windows not usable).

Hardware-Assisted Solutions. Both Intel and AMD proposed EPT [15, 43], a hardware-assisted solution which does not include the software solution's limitations. We have already presented this solution in Section 2.2. As shown in the section, EPT is far from satisfactory because of the 2D page walk that it imposes. To reduce the overhead caused by this 2D page walk, several works have proposed the extension of the page walk cache (PWC) [12] already used in native systems. Such a cache avoids page walk on PWC hit. [15] investigated for the first time this extension of PWC for EPT. The main limitation of such solutions is their inefficiency facing large working set size VMs (e.g., in-memory databases) [46]. In addition, PWC based solutions suffer from a high rate of cache misses when several VMs share the same machine due to cache evictions. [8] used a flat EPT instead of the traditional multi-level radix. This way, the authors reduced the number of memory references on each TLB miss to 9. *Compromis* totally eliminates the EPT, resulting in 4 memory references for each TLB miss.

Some solutions improved the TLB [36, 38, 46]. [38] presented *POM-TLB*, a very large level-3 in RAM TLB. *POM-TLB* brings two main advantages. First, the number of TLB misses is reduced because of the large TLB size, thus reducing the number of 2D page walks. Second, *POM-TLB* benefits the data cache to reduce RAM references. However, on a cache miss a RAM access is necessary. Additionally, on a *POM-TLB* miss, the hardware still needs to perform a 2D page walk. This solution can be used at the same time with *Compromis*.

[45] and [25] showed that neither EPT nor shadow paging can be a definite winner. They proposed dynamic switching mechanisms that exceed the benefits of each technique. To this end, TLB misses and guest page faults are monitored to determine the best technique to apply. Such dynamic solutions come with a significant overhead related to two tasks: the monitoring and computing of necessary metrics consumes a lot of CPU cycles, and switching from one technique to another requires rebuilding new page tables.

Orthogonal Solutions. Some researchers like [28, 29] proposed the utilization of huge pages [4, 39] in the guest OS and the hypervisor at the same time. This way, the number of hierarchies in the page table is reduced, thus the number of memory references during page walk is reduced too. However, using huge pages leads to two main limitations for the guest. First, it increases memory fragmentation, thus memory waste for the guest. This could lead to memory pressure in the guest OS, resulting in swapping, which is unfavourable for application performance. Second, huge pages increase average and tail memory allocation latency in the guest because zeroing a huge page at page allocation time is more time consuming than zeroing a 4Kb page.

[42] proposed *Hashed page tables* in native systems as an efficient alternative to the radix page table structure. With hashed page tables, address translation is done using a single memory reference, assuming no collision. [46] presented how this technique can be adapted for virtualized systems. The authors showed that by using a 2D hashed page table hierarchy, the page walk is done with 3 memory references instead of 24. This is one less than in *Compromis* and native systems but suffers from hash collisions. A recent work [31] proposes "Address Translation with Prefetching" (ASAP) which launches prefetches to the deeper levels of the page table, bypassing the preceding levels. These prefetches happen concurrently with a conventional page walk, which observes a latency reduction due to prefetching while guaranteeing that only correctly-predicted entries are consumed.

Direct segment (DS) based solutions. Previous work showed the benefits of DS in both native [13, 27] and virtualized systems [9, 24, 25]. presented DVMT[9], a mechanism which allows applications inside the VM to request DS allocations directly from the hypervisor. The application is responsible for mapping GVAs which are in the allocated DS address space. This is a limitation for application developers who are not experts. [24] proposed three memory virtualization solutions based on DS. Their *VMM Direct* mode is very close to *Compromis*, but DS does not concern the entire VM memory. In other words, *VMM Direct* proposes to use contiguous memory addressing for some applications and not at the VM scale. In addition, the authors mainly investigated the two other modes. Our contribution in this article is threefold: Hardware-level contribution, Hypervisor-level contribution and Cloud scheduler level contribution which is not the case with existing works.

More generally, existing solutions in this category mainly focused on hardware contributions while we study the consequences on the entire cloud stack. These solutions also relied only on simulations, while we tried to perform accurate experiments on real machines using real systems. Finally, by relying on trace analysis, we motivate for the first time the relevance of DS on virtual machines.

8 CONCLUSION

This paper presented *Compromis*, a novel MMU solution for virtualized systems. *Compromis* generalizes DS to provide the entire VM memory space using a minimal number of memory segments. In this way, the hardware page table walker performs a 1D page walk as in native systems. By analyzing several production datacenter traces, the paper showed that *Compromis* provisioned up to 99.99% VMs with a single memory segment. The paper presented a systematic implementation of *Compromis* in the hardware, the hypervisor and the cloud scheduler. The evaluation results show that *Compromis* reduces the memory virtualization overhead to only 0.35%. Furthermore, *Compromis* reduces the VM startup latency by up to 80% while providing also a predictable startup time.

REFERENCES

[1] [n.d.]. Benefits of Virtualization. https://www.thrivenetworks.com/blog/benefits-of-virtualization/.

[2] [n.d.]. Inside Microsoft Azure datacenter hardware and software architecture with Mark Russinovich. https://www.youtube.com/watch?v=Lv8fDiTNHjk.

[3] [n.d.]. Top 5 Business Benefits of Server Virtualization. https://blog.nhlearningsolutions.com/blog/top-5-ways-businesses-benefit-from-server-virtualization.

[4] [n.d.]. Transparent Hugepages. https://lwn.net/Articles/359158/.

[5] [n.d.]. X86 Paravirtualised Memory Management. https://wiki.xen.org/wiki/X86_Paravirtualised_Memory_Management.

[6] Keith Adams and Ole Agesen. 2006. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. ACM, New York, NY, USA, 2–13. https://doi.org/10.1145/1168857.1168860

[7] Ole Agesen, Jim Mattson, Radu Rugina, and Jeffrey Sheldon. 2012. Software Techniques for Avoiding Hardware Virtualization Exits. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, Berkeley, CA, USA, 35–35. http://dl.acm.org/citation.cfm?id=2342821.2342856

[8] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. 2012. Revisiting Hardware-assisted Page Walks for Virtualized Systems. In *Proceedings of the 39th Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '12)*. IEEE Computer Society, Washington, DC, USA, 476–487. http://dl.acm.org/citation.cfm?id=2337159.2337214

[9] Hanna Alam, Tianhao Zhang, Mattan Erez, and Yoav Etsion. 2017. Do-It-Yourself Virtual Memory Translation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. ACM, New York, NY, USA, 457–468. https://doi.org/10.1145/3079856.3080209

[10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. 2010. A View of Cloud Computing. *Commun. ACM* 53, 4 (April 2010), 50–58. https://doi.org/10.1145/1721654.1721672

[11] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. In *IN SOSP*. 164–177.

[12] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2010. Translation Caching: Skip, Don'T Walk (the Page Table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (Saint-Malo, France) *(ISCA '10)*. ACM, New York, NY, USA, 48–59. https://doi.org/10.1145/1815961.1815970

[13] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA '13)*. ACM, New York, NY, USA, 237–248. https://doi.org/10.1145/2485922.2485943

[14] Arkaprava Basu, Mark D. Hill, and Michael M. Swift. 2012. Reducing Memory Reference Energy with Opportunistic Virtual Caching. In *Proceedings of the 39th Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '12)*. IEEE Computer Society, Washington, DC, USA, 297–308. http://dl.acm.org/citation.cfm?id=2337159.2337194

[15] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. 2008. Accelerating Two-dimensional Page Walks for Virtualized Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (Seattle, WA, USA) *(ASPLOS XIII)*. ACM, New York, NY, USA, 26–35. https://doi.org/10.1145/1346281.1346286

[16] Abhishek Bhattacharjee. 2013. Large-reach Memory Management Unit Caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (Davis, California) *(MICRO-46)*. ACM, New York, NY, USA, 383–394. https://doi.org/10.1145/2540708.2540741

[17] Abhishek Bhattacharjee. 2017. Translation-Triggered Prefetching. *SIGARCH Comput. Archit. News* 45, 1 (April 2017), 63–76. https://doi.org/10.1145/3093337.3037705

[18] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. 2011. Shared Last-level TLBs for Chip Multiprocessors. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA '11)*. IEEE Computer Society, Washington, DC, USA, 62–63. http://dl.acm.org/citation.cfm?id=2014698.2014896

[19] Xiaotao Chang, Hubertus Franke, Yi Ge, Tao Liu, Kun Wang, Jimi Xenidis, Fei Chen, and Yu Zhang. 2013. Improving Virtualization in the Presence of Software Managed Translation Lookaside Buffers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA '13)*. ACM, New York, NY, USA, 120–129. https://doi.org/10.1145/2485922.2485933

[20] cloudstack [n.d.]. Apache CloudStack – Open Source Cloud Computing. http://cloudstack.apache.org/.

[21] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. ACM, New York, NY, USA, 153–167. https://doi.org/10.1145/3132747.3132772

[22] Guilherme Cox and Abhishek Bhattacharjee. 2017. Efficient Address Translation for Architectures with Multiple Page Sizes. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. ACM, New York, NY, USA, 435–448. https://doi.org/10.1145/3037697.3037704

[23] filter [n.d.]. Nova filter scheduler. http://docs.openstack.org/developer/nova/filter_scheduler.html.

[24] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. 2014. Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, United Kingdom) *(MICRO-47)*. IEEE Computer Society, Washington, DC, USA, 178–189. https://doi.org/10.1109/MICRO.2014.37

[25] Jayneel Gandhi, Mark D. Hill, and Michael M. Swift. 2016. Agile Paging: Exceeding the Best of Nested and Shadow Paging. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Seoul, Republic of Korea) *(ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 707–718. https://doi.org/10.1109/ISCA.2016.67

[26] Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2018. Devirtualizing Memory in Heterogeneous Systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. ACM, New York, NY, USA, 637–650. https://doi.org/10.1145/3173162.3173194

[27] Nikhita Kunati and Michael M. Swift. 2018. Implementation of Direct Segments on a RISC-V Processor. In *IN Second Workshop on Computer Architecture Research with RISC-V (CARRV), Co-located with ISCA*.

[28] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, Berkeley, CA, USA, 705–721. http://dl.acm.org/citation.cfm?id=3026877.3026931

[29] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2017. Ingens: Huge Page Support for the OS and Hypervisor. *SIGOPS Oper. Syst. Rev.* 51, 1 (Sept. 2017), 83–93. https://doi.org/10.1145/3139645.3139659

[30] Yashwant Marathe, Nagendra Gulur, Jee Ho Ryoo, Shuang Song, and Lizy K. John. 2017. CSALT: Context Switch Aware Large TLB. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, Massachusetts) *(MICRO-50 '17)*. ACM, New York, NY, USA, 449–462. https://doi.org/10.1145/3123939.3124549

[31] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. 2019. Prefetched Address Translation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 1023–1036. https://doi.org/10.1145/3352460.3358294

[32] Vlad Nitu, Aram Kocharyan, Hannas Yaya, Alain Tchana, Daniel Hagimont, and Hrachya Astsatryan. 2018. Working Set Size Estimation Techniques in Virtualized Environments: One Size Does Not Fit All. *Proc. ACM Meas. Anal. Comput. Syst.* 2, 1, Article 19 (April 2018), 22 pages. https://doi.org/10.1145/3179422

[33] Vlad Nitu, Pierre Olivier, Alain Tchana, Daniel Chiba, Antonio Barbalace, Daniel Hagimont, and Binoy Ravindran. 2017. Swift Birth and Quick Death: Enabling Fast Parallel Guest Boot and Destruction in the Xen Hypervisor. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Xi'an, China) *(VEE '17)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/3050748.3050758

[34] Ashish Panwar, Aravinda Prasad, and K. Gopinath. 2018. Making Huge Pages Actually Useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. ACM, New York, NY, USA, 679–692. https://doi.org/10.1145/3173162.3173203

[35] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. 2017. Hybrid TLB Coalescing: Improving TLB Translation Coverage Under Diverse Fragmented Memory Allocations. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. ACM, New York, NY, USA, 444–456. https://doi.org/10.1145/3079856.3080217

[36] Binh Pham, Ján Veselý, Gabriel H. Loh, and Abhishek Bhattacharjee. 2015. Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have It Both Ways?. In *Proceedings of the 48th International Symposium on Microarchitecture* (Waikiki, Hawaii) *(MICRO-48)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2830772.2830773

[37] R. Raman, M. Livny, and M. Solomon. 1998. Matchmaking: Distributed Resource Management for High Throughput Computing. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing (HPDC '98)*. IEEE Computer Society, Washington, DC, USA, 140–. http://dl.acm.org/citation.cfm?id=822083.823222

[38] Jee Ho Ryoo, Nagendra Gulur, Shuang Song, and Lizy K. John. 2017. Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. ACM, New York, NY, USA, 469–480. https://doi.org/10.1145/3079856.3080210

[39] Tom Shanley. 1996. *Pentium Pro Processor System Architecture* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[40] Siqi Shen, Vincent van Beek, and Alexandru Iosup. 2015. Statistical Characterization of Business-Critical Workloads Hosted in Cloud Datacenters. In *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2015, Shenzhen, China, May 4-7, 2015*. 465–474.

[41] Cristan Szmajda and Gernot Heiser. 2003. Variable Radix Page Table: A Page Table for Modern Architectures. In *Advances in Computer Systems Architecture*, Amos Omondi and Stanislav Sedukhin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 290–304.

[42] M. Talluri, M. D. Hill, and Y. A. Khalidi. 1995. A New Page Table for 64-bit Address Spaces. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, USA) *(SOSP '95)*. ACM, New York, NY, USA, 184–200. https://doi.org/10.1145/224056.224071

[43] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. 2005. Intel Virtualization Technology. *Computer* 38, 5 (May 2005), 48–56. https://doi.org/10.1109/MC.2005.163

[44] Carl A. Waldspurger. 2002. Memory Resource Management in VMware ESX Server. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002), 181–194. https://doi.org/10.1145/844128.844146

[45] Xiaolin Wang, Jiarui Zang, Zhenlin Wang, Yingwei Luo, and Xiaoming Li. 2011. Selective Hardware/Software Memory Virtualization. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Newport Beach, California, USA) *(VEE '11)*. ACM, New York, NY, USA, 217–226. https://doi.org/10.1145/1952682.1952710

[46] Idan Yaniv and Dan Tsafrir. 2016. Hash, Don'T Cache (the Page Table). In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science* (Antibes Juan-les-Pins, France) *(SIGMETRICS '16)*. ACM, New York, NY, USA, 337–350. https://doi.org/10.1145/2896377.2901456