Computer Science Review 29 (2018) 56–73

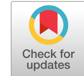
Contents lists available at ScienceDirect

Computer Science Review

journal homepage: www.elsevier.com/locate/cosrev

Survey

A survey of memory management techniques in virtualized systems



Debadatta Mishra [a,*], Purushottam Kulkarni [b]

[a] Department of Computer Science and Engineering, Indian Institute of Technology Kanpur, India
[b] Department of Computer Science and Engineering, Indian Institute of Technology Bombay, India

HIGHLIGHTS

- Discussion of resource virtualization challenges for virtualization techniques.
- Research challenges of memory partitioning and management in virtualized systems discussed.
- Survey of memory virtualization techniques and their implications.
- Classification and detailed description of dynamic memory management techniques.

ARTICLE INFO

Article history:
Received 22 July 2016
Received in revised form 19 June 2018
Accepted 20 June 2018
Available online 28 June 2018

Keywords:
Operating system
Virtualization
Memory virtualization
Memory management

ABSTRACT

Virtualization technology allows multiple operating systems to share hardware resources of a computer system in an isolated manner. Traditionally, memory is shared by an operating system using segmentation and paging techniques. With virtualization, memory partitioning and management has several new challenges. For isolated and safe execution, hypervisors do not provide direct access to hardware resources. Lack of direct access to the memory management hardware like page tables disqualifies direct usage of virtual memory solutions used on native (non-virtualized) setups. Further, aspects of dual control of the memory resource (by the guest OS and the hypervisor) and lack of semantics regarding memory usage in virtual machines present additional challenges for memory management. This paper surveys different techniques of memory partitioning and management across multiple guest OSs in a virtualized environment.

An important goal of virtualization is to increase the physical machine utilization in order to save costs. With varying application demand for memory and diverse memory management policies of the guest OSs, ensuring optimal usage of memory is non-trivial. In this survey, challenges of memory management in virtualized systems, different memory management techniques with their implications, and optimizations to increase memory utilization are discussed in detail.

© 2018 Elsevier Inc. All rights reserved.

Contents

1. Introduction ... 57
 1.1. Scope ... 58
2. Memory virtualization challenges and techniques ... 58
 2.1. Hypervisor memory allocation model ... 58
 2.2. Virtualizing the memory management unit ... 59
 2.3. Shadow paging—software-based MMU virtualization ... 59
 2.4. Direct paging—para-virtualized MMU virtualization ... 60
 2.5. Nested paging—hardware assisted MMU virtualization ... 61
 2.6. Beyond hardware assisted MMU virtualization ... 61
3. Memory resource management in virtualized systems ... 62
 3.1. Memory management complexity of operating systems ... 62
 3.2. Hypervisor memory management challenges ... 63
  3.2.1. Loss of hypervisor control of the allocated memory ... 63

* Corresponding author.
E-mail addresses: deba@cse.iitk.ac.in (D. Mishra), puru@cse.iitb.ac.in (P. Kulkarni).

https://doi.org/10.1016/j.cosrev.2018.06.002
1574-0137/© 2018 Elsevier Inc. All rights reserved.

3.2.2. Limited access to guest OS information ... 63
3.2.3. No direct influence on guest OS policies ... 63
3.2.4. Diverse nature of guest OSs ... 63
4. Memory management approaches ... 63
4.1. Exploiting dynamic memory demands ... 63
4.2. Content deduplication ... 64
4.3. Hypervisor and guest OS symbiosis ... 64
5. Dynamic memory provisioning techniques ... 64
5.1. Dynamic provisioning enablers ... 64
5.1.1. Memory ballooning ... 65
5.1.2. Memory hot-plugging ... 65
5.2. Controllers for dynamic memory provisioning ... 65
5.2.1. Black-box approaches of balloon control ... 66
5.2.2. Gray-box balloon controllers ... 67
6. Exploiting content similarity for memory management ... 67
6.1. Page sharing by out-of-band scanning ... 68
6.2. Out-of-band sharing at multiple granularity ... 68
6.3. Deduplicating memory on the I/O access path ... 69
7. Memory management through hypervisor and guest OS symbiosis ... 69
7.1. System level second chance caching ... 70
7.2. Collaborative memory state maintenance ... 71
8. Discussion ... 71
9. Conclusion ... 72
Conflict of interest ... 72
References ... 72

1. Introduction

Historically, operating systems have been designed to be sole owners and managers of hardware resources in a computer system. A modern multitasking operating system provides software abstractions of hardware devices and enables interfaces to access and use the devices in an isolated manner. For example, operating systems provide interfaces to create and run processes in a concurrent manner by sharing the CPUs. The file abstraction and related file operations (`create, read, write` etc.) is another example to enable shared access of hard disk by different applications.

Virtualization technology raises the abstraction beyond individual resources to provide an abstraction of the computer system. A Virtual Machine (VM) is the equivalent of a physical computer that can host an operating system to manage the virtual hardware just like an operating system running on a physical hardware. An important advantage of virtualization is that computing systems become *detachable* from the underlying hardware. Virtual machines – hardware independent software computers – can enable mobility, controlled execution, state replication etc. However, the transition from resource level multiplexing to system (VM) level multiplexing is not trivial. With system virtualization, a software layer known as the Virtual Machine Monitor (VMM) or the hypervisor[1] owns the physical hardware and orchestrates sharing of resources to provide virtual hardware abstractions.

Towards the design of hypervisors, a set of requirements (as stated by Popek and Goldberg [1]) are,

- **Equivalence:** Provide the same *hardware and instruction interfaces* as that of the bare metal (physical) hardware for which the operating system is designed. This condition allows operating systems designed for native hardware to execute inside a virtual machine without any modifications.
- **Resource Control:** All physical resources of a machine are completely controlled and managed by the hypervisor. A guest OS should not use any resource that is not allocated to it and the hypervisor can *gain control* of any resource at any point of time. This property is a mandatory requirement for hypervisors to provide isolation, security and improved resource utilization.
- **Efficiency:** As much as possible, the guest OS operations should be executed natively *without involvement of the hypervisor*. This is a performance requirement that separates pure software emulators like QEMU [2] from hypervisors.

Hypervisors have been designed and built with varying adherence to the above requirements, mostly exploring the tradeoff between efficiency and equivalence [3–7].

Virtualization based provisioning [8,9] is a growing trend for hosting services and applications for data centers. Such a deployment model has several benefits—elastic resource provisioning [10–12], run time migration [13], checkpointing [14], sandboxing of applications [15] etc. Towards enabling a *pay-on-use* model of provisioning and maximization of resource utilization, dynamic management of resources allocated to virtual machines plays a vital role.

Efficient utilization of physical resources—CPU, memory and I/O devices, has a direct implication on the cost in terms of hardware, maintenance, power consumption and performance of applications (commonly specified in terms of service level agreement or SLA). Even though a virtual machine is provisioned with certain resources, it may not use *all* the resources at all times. For example, a virtual machine configured with 1 GB of main memory may require 1 GB of memory only during peak demand periods. During "average" load conditions, the unused memory can potentially be used by any other virtual machine.

Following are the challenges involved to achieve efficient resource management in virtualized systems,

- **Darkness:** The hypervisor by design is *in the dark* about the resource usage quality of the guest OS. By quality we mean whether resources allocated to a virtual machine are used for *as important a purpose as* by the other VMs. For example, a VM may use the CPU resources to run an anti-virus scanner when two other VMs urgently need additional CPU for transaction processing.
- **Duality:** Resource management decisions are controlled by *two entities* in virtualized systems—(i) by the VM as per the OS design and administrator policies, and, (ii) by the hypervisor dictated by the virtualization technique and the administrator policies determined by customer service level

[1] VMM and hypervisor are used interchangeably in this document.

agreements. For example, I/O schedulers of the guest OS and the hypervisor can make contradictory decisions regarding the order in which I/O requests are issued. This duality can result in inefficient resource utilization and impact performance.

- **Diversity:** Even though the guest OS policy and the hypervisor policy may not be in sync, if a single type of guest policy is assumed, hypervisor resource management can be relatively easily tuned to be in sync with the guest VM policies. However in a real setup, OSs and applications executing within different VMs are heterogeneous in nature. As a consequence, it is difficult to develop *generic techniques* at the hypervisor level to complement policies employed by different guest OSs.
- **Dynamism:** Dynamic levels of resource demand in different VMs enables *over-commitment* of resources. Overcommitment requires the hypervisor resource manager to *detect, adapt and act* to the varying resource needs of the VMs in order to maximize utilization and adhere to the performance requirements.

The focus of this work is the main memory subsystem of a computing system. Memory management of a computer system in a multitasking OS is a well researched area [16–18]. Operating systems partition memory into independently managed chunks and assign them to different execution entities like processes and threads at run-time. Segmentation and paging are common instances of partitioning based memory management techniques. Hardware support like segmentation hardware, Memory Management Unit (MMU) and Translation Look-aside Buffer (TLB) for efficient implementation of segmentation and paging are commonly available in most of the modern commodity processors. Memory virtualization adds an additional layer of memory management for which traditional MMU based approaches and the meta-data management need to be controlled (and if required owned) by the hypervisor to ensure *correctness and isolation*.

Memory management in system virtualization is challenging because of the general resource management challenges of virtualized environments—*darkness, duality, diversity and dynamism*. Additionally, unlike the time multiplexed resources like CPU and I/O devices, memory is space multiplexed (partitioned). With time-multiplexed resources, the hypervisor periodically gains control of the resource for decision making. For example, hypervisor level CPU scheduler may be invoked periodically to change the CPU allocations to VMs. With space-partitioned resources such periodic decision making on the resource-access or allocation path is not possible and adds to the challenge of efficient resource provisioning.

1.1. Scope

In this survey, we focus on virtualization and management of memory on a *single* host, where a hypervisor controls and manages the memory sub-system of the machine. Memory virtualization solutions, implemented both in software and with hardware assistance, are discussed and evaluated against virtualization requirements. Further, we discuss existing work related to performance implications of the memory virtualization techniques.

The complexities and challenges of efficient memory management are discussed before delving into classification of individual techniques. A high-level classification based on different design philosophies is also presented. Individual techniques, optimizations, usage of the techniques and performance implications are discussed in detail.

The rest of the paper is organized as follows. Memory virtualization techniques are discussed in Section 2. Memory management challenges and complexities in a system virtualized solution are presented in Section 3. Section 4 presents a broad classification of memory management techniques which are discussed in detail in Sections 5–7. Finally, we discuss future research directions of memory management in virtualized systems in Section 8 and present conclusions in Section 9.

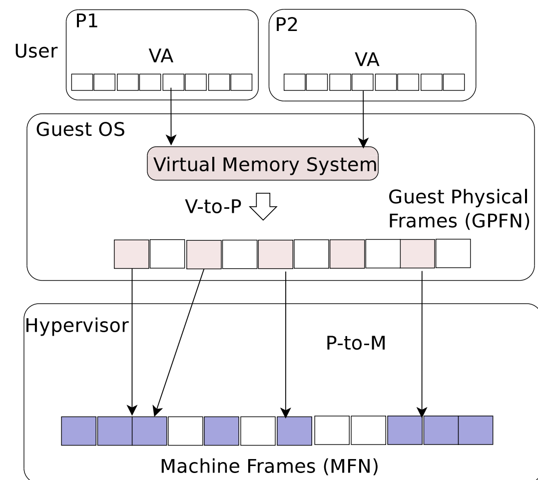


Fig. 1. Flexible page-level memory allocation in virtualized systems with dynamic pseudo (guest) physical pages to machine pages mapping.

2. Memory virtualization challenges and techniques

There are several techniques to partition and allocate memory among virtual machines in a virtualized system. Allocation can be simple continuous fixed partition based or a dynamic page level allocation scheme like *demand paging*. Further, guest OSs employ virtual memory mechanisms to support multi-tasking making use of MMU hardware to allow controlled memory access for the processes. By implication, process level memory isolation requires MMU hardware features to be accessible from the virtual machine. Allowing direct access to the hardware without hypervisor control is not a straightforward design feature as it presents challenges w.r.t. isolation requirements and efficiency aspects.

2.1. Hypervisor memory allocation model

Most modern operating systems manage physical memory by dividing it into fixed size regions called pages. The basic page entity is used by an OS to manage allocation, permissions and translate virtual address to physical address. Virtualization solutions employ the same notion of paged memory for management of memory across different virtual machines.

Linearly addressable physical memory starting at physical address zero is the underlying assumption of operating systems. If the guest OS has no explicit knowledge of the underlying hypervisor (unlike in para-virtualization techniques like Xen [4]), memory allocation by the hypervisor should ensure that the guest operating system does not need special handling for physical memory layout. To address this fundamental requirement, hypervisors implement pseudo-physical memory as a logically continuous physical memory abstraction for each virtual machine. Fig. 1 shows a generic memory allocation model employed by hypervisors for memory virtualization.

Processes executing in a virtual machine share the guest physical address by virtue of the virtual to physical mapping provided by the virtual memory subsystem of the guest OS (referred as V-to-P in the figure). As shown in Fig. 1, from a guest OS perspective,

each virtual address (VA) of processes P1 and P2 is translated to the guest physical address (referred to as P and implies guest physical frames in the figure). The guest physical page frame number[2] (GPFN) to actual machine frame number (MFN) translation is performed by the hypervisor (P-to-M mapping as shown in Fig. 1).

Given this generalized two layer model for mapping memory addresses, the central question is, how physical memory is allocated and managed across virtual machines? Following are the possibilities,

- Statically allocating fixed region physical memory to each virtual machine, while this scheme is simple to implement, large portions of memory allocated to virtual machines could never be utilized and lead to inefficient memory usage.
- Dynamic mapping similar to the way a process in an operating system is allocated memory and allocations are changed if the situation demands. To implement dynamic mapping for virtual machines, hypervisors need to maintain and manage guest to machine memory mappings in a dynamic manner.

Dynamic mapping facilitates better memory resource management. The mapping from guest physical address to machine physical address can be provided in an on-demand basis. For example, consider a virtual machine configured with 1 GB memory, and not all of it is mapped to machine memory. A particular guest physical page number (GPFN) can be mapped to machine frame number (MFN) when the guest allocates the page to any process running inside the guest OS. Additionally, the hypervisor can invalidate some GPFN to MFN mappings and reassign the machine pages to other virtual machines. As shown in Fig. 1, for many guest PFNs there are no machine frame mappings. The advantages of dynamic GPFN to MFN mapping are as follows,

- *Memory over-commitment* is possible with page level dynamic allocation. With hypervisor level page swapping, the hypervisor can swap-out guest pages to disk before invalidating the GPFN to MFN mapping and swap-in the guest page when the guest access causes a page fault. Similar to virtual memory systems, the virtual machine can be promised more memory than the actual physical memory.
- Page level dynamic allocation acts as a building block to support memory content deduplication at the allocation granularity. Consider two GPFNs (from same or different guest VMs) mapped to two distinct MFNs with same data contents. The GPFN to MFN mapping can be changed to a shared mapping to release one MFN (in Fig. 1 one such shared mapping is shown). As a consequence of deduplication, memory efficiency of the virtualized system can be improved.

2.2. Virtualizing the memory management unit

The virtual memory subsystem in an operating system provides the illusion of unlimited memory (as large as the address space) to processes. The virtual to physical address mapping is performed at run time using specialized hardware, the Memory Management Unit (MMU). Virtual to physical mapping is maintained at a page granularity and is indexed at several levels (the page tables) to reduce space requirements for storage of translation information. When a process is scheduled on a CPU, pointer to the level one page table physical page (also known as Page Directory or PGD) that contains the second level mappings is loaded into a designated CPU register (the CR3 register in x86 architectures [19]). The MMU uses the memory location stored in the CR3 register to walk through the page table hierarchy to reach to the physical page corresponding to a given virtual address.

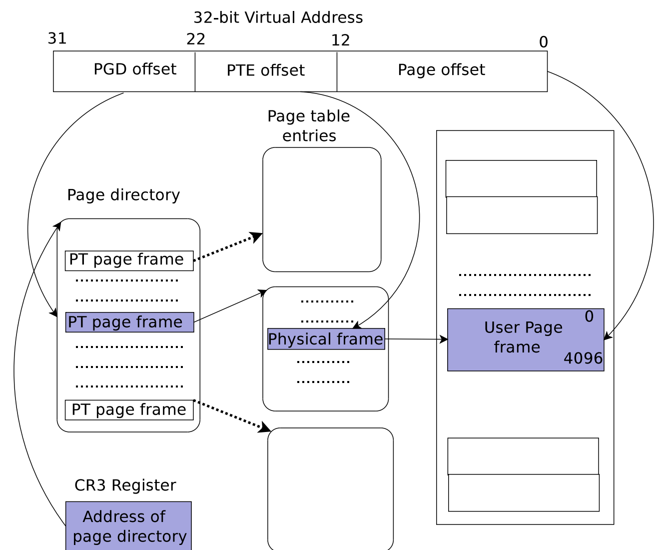


Fig. 2. Example of MMU based virtual to physical address translation in 32-bit x86 architectures with 4 KB pages.

An example of page table structures and translation for 32-bit x86 systems is shown in Fig. 2. On a context switch, the CR3 register is loaded with the physical page address that contains the PGD of the process. When the process accesses a 32-bit virtual address, 10 most significant bits of the virtual address are used as an offset into the PGD page frame to locate address of the page frame that contains the Page Table Entries (PTEs). The next 10 significant bits are used to offset into the PTE page frame to locate the user accessed page frame. The least significant 12-bits are used as an offset into the physical page to access the address inside the page.

The page table walk is performed entirely in hardware without any software involvement. The operating system gets involved either on a process context switch or to handle hardware exceptions during page table walk. *Page fault* is the most common exception generated in a demand paging system. One of the most common scenarios of page faults is when the hardware page table walker encounters a non-existent mapping for a requested virtual address. The operating system allocates a free page frame to the process (swaps out some other page if required) on a page fault exception and updates the page table accordingly before returning from the exception handler.

In the context of virtualization, virtualizing the MMU has additional factors to consider.

- In virtualized systems, the page table structures are managed by the guest OS which operates with pseudo physical pages (GPFNs). The hardware MMU page table walk requires machine page frames (MFNs) for translation of virtual addresses to physical addresses.
- To ensure isolation and correctness, access to the CR3 register is not granted to guest OSs by the hypervisor. This implies that page table modifications may need hypervisor intervention.

2.3. Shadow paging—software-based MMU virtualization

Waldspurger [5] proposes a basic framework of shadow page tables for VMware ESX server. In shadow paging, the page table

[2] A frame number indicates logical number of the page frame when memory is divided into page frames.

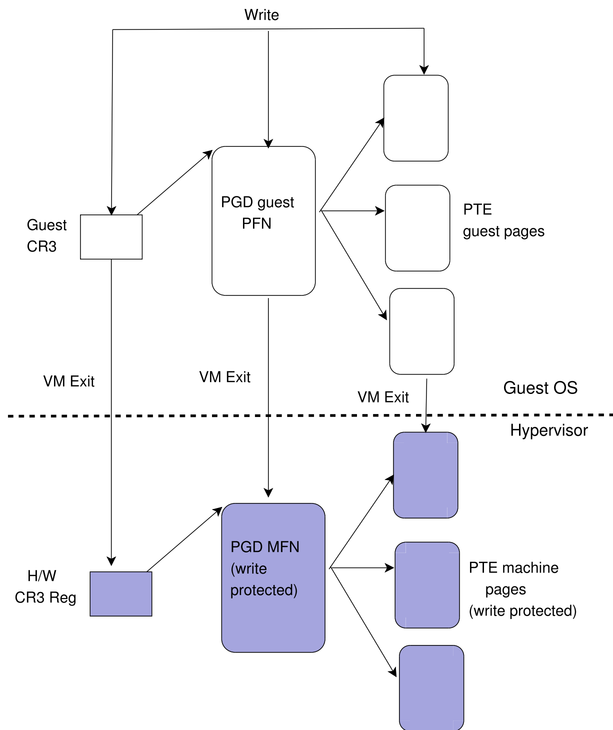


Fig. 3. Hypervisor maintains shadow page tables for each process of the guest OS and facilitates guest-transparent MMU virtualization ensuring isolation and correctness.

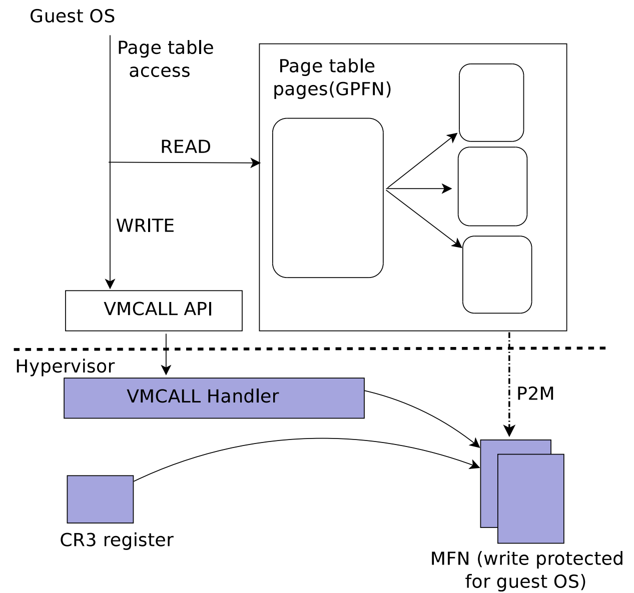


Fig. 4. Para-virtualized guest OS issues VMCALLs (hypercalls) to update the page table entries. Read operations on the page table are allowed without the hypervisor involvement.

physical resources—the page table base register (e.g., CR3 in x86) and the machine pages that contain the guest process page tables, are protected by the hypervisor. CR3 protection is achieved by architecture provided protection mechanisms—CPU privilege levels (x86 rings) and/or explicit hardware support for virtualization like Intel VT-X [20] and AMD SVM [21]. Fig. 3 shows the guest OS view of page table (above the dotted line) and the actual page table used by the MMU for address translation. The machine pages that contain the page table structures are protected by making them read-only for the guest OS. Note that, the hypervisor maintains a per-process *shadow page table* for each guest OS which is used by the MMU when a process executing within a virtual machine accesses any virtual address.

Page table reads execute without an exception, which implies that as long as a process executing inside a virtual machine performs read or write operation on the pages those are already mapped in the page table, the hypervisor is not involved. Whenever page table entries are modified because the guest OS allocates or de-allocates a guest PFN, an exception occurs. The exception is handled by the hypervisor by updating the shadow page tables. Page fault trap handler at the hypervisor looks into the guest page tables to check if the page table entry is valid. If not valid, the hypervisor injects this fault into the guest OS which is handled by modifying the page table. If the virtual address to guest PFN mapping is valid in the guest page table, the shadow page table is updated with a machine frame and the guest PFN to MFN mapping is updated in the hypervisor.

The advantage of shadow paging is that it requires no support from the hardware and the guest OS does not require any modifications. Further, isolation and equivalence requirements, as stated by Popek and Goldberg [1], are met by shadow paging. The overheads of software MMU virtualization can be substantial because it requires per process shadow page table to be maintained the hypervisor. Moreover, a large number of exits to the hypervisor (a.k.a. VMExit) may have to be handled if the memory map of processes changes frequently. Wang et al. [22] show that, for certain workloads this overheads can be substantial compared to other techniques that we will discuss shortly. All virtualization solutions that support full virtualization (Xen [4], VMware [5] and KVM [3]) implement shadow paging to support execution of unmodified operating systems on virtualization-agnostic hardware architectures. In spite of the software complexity associated with shadow page tables, this technique performs very similar to hardware page walks in the best case and does not require any additional hardware support. Therefore, even with hardware extensions to aid memory virtualization, shadow paging remains a relevant technique for the research community. For example, Gandhi et al. [23] propose architectural extensions to design an agile paging system with shadow page table used as the base technique.

2.4. Direct paging—para-virtualized MMU virtualization

Barham et al. [4] propose the concept of para-virtualization where a subset of instructions in x86 Instruction Set Architecture (ISA) is available for the guest OS. Change in ISA was primarily motivated to address architectural limitations with regard to non-deterministic behavior of sensitive instructions resulting in non-equivalent virtualization. Robin et al. [24] provide an excellent analysis of issues and challenges in design of virtualization solutions on x86 platforms of that era. One of the major issues in x86 systems was that several instructions were silently ignored without causing any trap when executed from a lower privilege level. With the para-virtualized approach, modifications to the guest OS are proposed to carry out these sensitive operations through the hypervisor. VMCALL or hypercall API is provided to guest OSs by para-virtualization platforms to execute privileged operations by the hypervisor ensuring isolation and correctness. The relaxation achieved due to guest OS changes can also be used for efficient MMU virtualization.

In Fig. 4, direct paging used in a para-virtualized setting is shown. From the guest OS view, the per-process page table structures (shown above the dotted line in the figure) are stored in the GPFNs (mapped to write protected MFNs by the hypervisor). Page

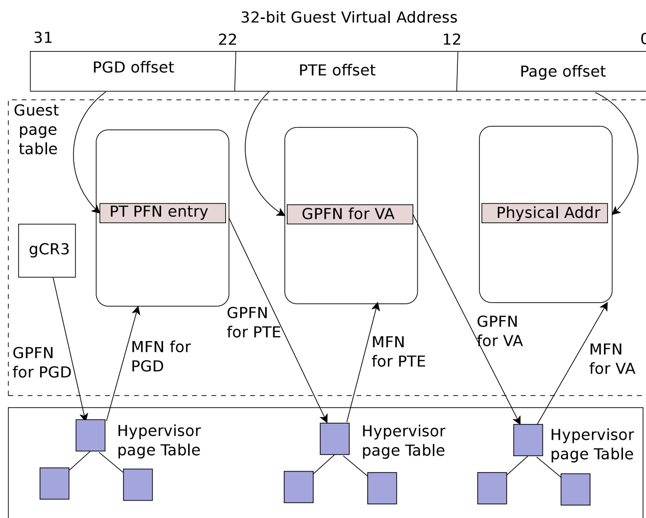


Fig. 5. MMU virtualization using in-hardware support for two levels of address translation—guest virtual to GPFN through guest page table, and GPFN to MFN using hypervisor page table.

table read by the guest OS or the MMU hardware does not require hypervisor involvement. The page table updation interfaces in the guest OS are modified to update (write) the page tables via hypercalls to the hypervisor.

The guest page table MFNs are marked read-only and any guest modification to the page table structures for a particular process is achieved by a hypercall from the guest OS to the hypervisor (as shown in Fig. 4). Hypercall handler implemented by the hypervisor validates the changes requested to ensure isolation before applying the changes to the machine pages that correspond to the guest page tables (shown as write protected MFNs in the figure). Requests can be bundled together as an optimization to reduce the hypercall overheads. Similar to the case of shadow page tables, normal page table reads execute without hypervisor intervention. The MMU register that points to the page directory (the CR3 register in case of x86) points to the applicable machine page on a process context switch in the guest OS. On a page fault, the guest OS finds a free GPFN and makes a hypercall to update the page table for the faulting virtual address. The hypervisor translates the GPFN to MFN and updates the page table entry. The primary advantage of direct paging is efficiency, which comes at the cost of losing equivalence as guest OS modifications are required to realize this solution.

2.5. Nested paging—hardware assisted MMU virtualization

As virtualized platforms are becoming an ubiquity, hardware vendors and researchers continuously strive to design virtualization-aware hardware. Advancements of this nature that are applicable to memory virtualization are extended page tables (EPT) [20] and nested page tables (NPT) [25]. These techniques provide two MMU states per CPU—one for the virtual machine and other for the hypervisor. Guest OSs and the hypervisor have access to their private CR3 registers. The guest OS CR3 register points to the guest PFN of page directory (PGD) and the hypervisor CR3 points to the page table structures maintained by the hypervisor to map guest physical frames to machine frames.

The hypervisor page table structures are same for all the processes executing within a VM. The guest accesses and manages the guest page tables without any exits to the hypervisor. The guest PFN to MFN mapping is maintained by the hypervisor.

A sample page table walk for a 32-bit guest virtual address is shown in Fig. 5. Every entry in the guest page table is in terms of a GPFN. The guest CR3 (gCR3) register containing the guest PFN of the page directory (PGD) is converted to MFN by performing a nested hypervisor page table walk. The 10-bit offset into the PGD is used to find the guest PFN that contains the page table entry (PTE). The corresponding MFN is determined by a nested walk before offsetting into the MFN that contains the GPFN for the virtual address. One more nested walk in hardware determines the machine page and the physical address is accessed using the 12 LSB bits of the virtual address. All these walks happen in hardware without the guest OS or the hypervisor involvement.

On a page fault, if the fault is due to missing or invalid entries in the guest page table, then the hypervisor injects the page fault into the guest OS. If the fault is because of the failure in nested walk, then the hypervisor allocates a machine page and updates the page table before returning from the exception. Using EPT/NPT, equivalence is achieved easily without additional software complexity unlike the earlier solutions. Drawback of nested page tables is the TLB-miss penalty which is the cost associated with multiple nested walks required to translate a single virtual address. Specifically, on a TLB miss, nested page table with n levels will result in $(n+1)^2 - 1$ number of memory accesses to translate a given virtual address in the worst case [23].

2.6. Beyond hardware assisted MMU virtualization

Wang et al. [22] provide a very good comparative analysis of MMU virtualization efficiency with shadow paging and hardware assisted EPT. The authors established that neither shadow paging nor extended page tables were clear winners for all workload scenarios. For workloads resulting in significant TLB misses, EPT experiences higher MMU translation overhead (up to 15% more) compared to shadow page tables [26,22]. On the other hand, workloads resulting in a lot of page faults, the performance of shadow page tables is shown to be 35% slower than the hardware assisted page tables. Wang et al. [22] used these insights to propose a dynamic switching between the two paging schemes and demonstrate that the hybrid scheme is capable in leveraging the best of both worlds to achieve improved virtualization efficiency.

Hardware vendors (both AMD and Intel) have proposed several optimizations like extended TLB support, hardware MMU caches [25,27] for the nested mapping etc. to circumvent the problems arising due to nested walks. Gandhi et al. [28] show that revived usage of hardware functionalities like segmentation at the hypervisor level to map guest physical to machine physical address can improve the translation overheads. However, this approach is applicable for big memory workloads (e.g., in-memory graph processing, key–value stores etc.) executing on systems with huge memory capacity. Agile paging (Gandhi et al. [23]) builds on the intuitions provided by Wang et al. [22] to propose modifications in the page table hardware. In agile paging, depending on the modification possibilities of any page table entry, extended paging is enabled selectively for the virtual address range covered by the page table entry. In the best case, agile paging incurs 4% overhead compared to native systems.

Summary: Two levels of memory translation required for memory virtualization is the primary source of complexity and overhead in virtualized systems. In our opinion, the optimal page table layout in virtualized system depends on the workload, more specifically on the memory usage and access footprint of applications. Dependence on application characteristics is the most challenging aspect in designing generic MMU hardware frameworks. Further, hardware vendors provide limited flexibility in terms of configuring the page table hardware. For example, the hypervisor cannot dynamically customize the number of levels in page table translations because of the hardware constraints. We believe, if future hardware architectures enable more software customization possibilities in the MMU hardware, memory virtualization overheads can be addressed more efficiently.

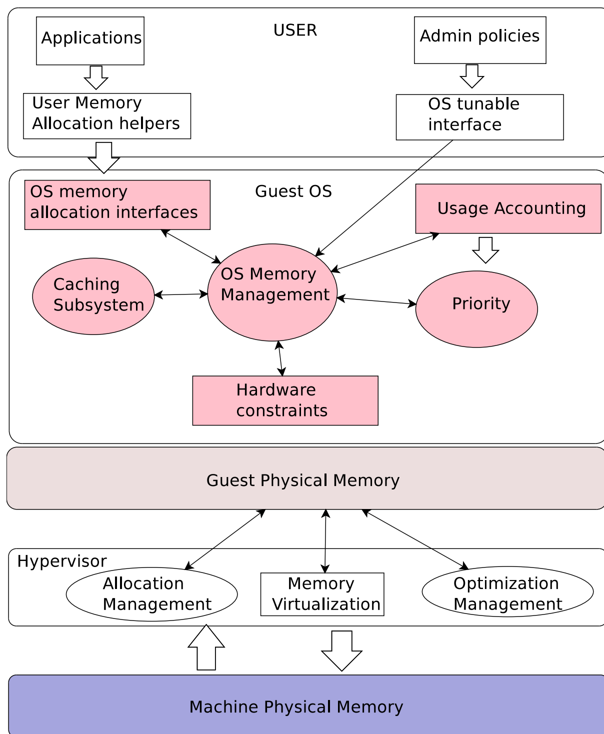


Fig. 6. Parameters influencing memory management in virtualized systems.

3. Memory resource management in virtualized systems

Memory management in a virtualized system involves three major entities—user-level applications, guest OS, and hypervisor (shown in Fig. 6). The applications executing within a virtual machine allocate memory using the memory allocation routines provided by the guest OS. Almost all operating systems provide administrator controlled policies (accessible from user space) that can influence the OS memory management. For example, an administrator may toggle the huge page feature in Linux (support for page size more than 4 kB) [29], in turn impacting memory management policies of Linux kernel related to allocation, evictions and de-fragmentation etc.

Operating system memory management is designed with the assumption that the OS is the *sole owner* of physical memory. A typical guest OS implements memory management policies to maximize local application performance, conforming to priorities enforced either by design or by administrator policies.

The hypervisor manages memory allocation across different VMs depending on the allocation policy and virtualization methods (described in previous section). Several memory optimizations are proposed and used by hypervisors for efficient management of memory. Memory management techniques of hypervisors take advantage of dynamic memory demands of the virtual machines to provision memory in a dynamic manner.

An efficient memory management method in virtualized systems should ensure high resource utilization and meet application performance and other SLA guarantees. The complexity of guest OS memory management presents a non-trivial challenge to correctly size the guest VM memory to meet memory requirements of user applications.

3.1. Memory management complexity of operating systems

Memory subsystem management is an important aspect of typical Von Neumann architectures where instructions and data reside in memory. The access speed of memory is an order of magnitude slower than the CPU. Latencies increase with the virtual memory mechanism due to additional memory accesses required for address translation. The situation is far more worse when a swap partition on a storage device is required to be used as the last resort to support multiplexing of memory across processes. Both hardware extensions (TLB and faster CPU caches) and software techniques (e.g., compiler optimizations) are employed to decrease wasted CPU cycles for memory subsystem access.

The guest OS (shown in the middle part of Fig. 6) employs memory management techniques based on several parameters. Application priority, disk block caching and hardware constraints are example parameters influencing memory management policies of an operating system. Some memory management design parameters of a typical operating system are as follows,

- **Physical memory range divisions:** Physical memory is divided into multiple logical partitions based on the way they are allocated for different purposes. The partitioning is required for two primary reasons—(i) hardware constraints (e.g., DMA address range for old devices), (ii) efficiency reasons (e.g., memory used by Linux kernel is preferably placed in a particular physical memory range for cache/translation efficiency).
- **Multiple types of allocations:** Operating systems allow interfaces for multiple types of memory allocation. Some devices require physically continuous memory to operate while operating system code and data structures may be placed in physically continuous memory to improve locality. Memory allocated for OS code sections and memory used by certain device drivers may require that allocated memory should not to be evicted to a swap device. Normal user allocation requests reserve only the virtual address during the allocation while the actual physical allocation takes place in an on demand manner.
- **Prioritization:** Memory allocation and eviction are done based on priority defined by either the entity that is requesting/using memory (an OS page is not normally selected as a victim to evict) or the usage history of the page (LRU, ARC [30] and adaptive CLOCK [31] are classical example of such priorities) or both.
- **Memory usage thresholds:** Different watermarks of memory usage are common in most of the operating systems that can drastically change the way memory is managed. A free memory low watermark may trigger swapping as a precautionary measure to maintain a minimum free memory in the system.
- **Caching content of slower storage:** To expedite access to slower block devices, most operating systems cache contents with the hope that those device blocks will be accessed in near future. Along the same lines there can be prefetching [32,33] optimizations to take advantage of temporal locality of access. Linux page caches [34] is an example cache implementation for files stored in disk storage.

Apart from the above stated general design parameters, there can be many micro level parameters and tunables which influence the memory management of an operating system. A couple of examples are explained here for the interested readers. In Linux, two kernel threads `pdflush` and `kswapd` [35] take care of flushing memory pages onto disk. Scheduling of these threads impacts the Linux kernel decisions regarding future allocation requests. Another example of a tunable is known as `swappiness` determines under what memory load conditions swapping can start.

Application designers may develop applications that impact the memory management inside the guest OS. Consider an application that tries to maintain its own cache (a database for example)

and employs direct block device access bypassing the OS disk caching layer. Memory used by the application layer disk cache will be incorrectly accounted as anonymous memory usage by the operating system. Salomie et al. [36] reported similar behavior for database applications and applications executing within JAVA runtime environment. Further, administrators (operating system distributions for that matter) may want to change default behavior of the operating system by configuring different OS configurations. This would in turn trigger changes in the OS memory management decisions.

3.2. Hypervisor memory management challenges

The complex and diverse nature of operating systems present several challenges to design efficient memory management policies in virtualized systems.

3.2.1. Loss of hypervisor control of the allocated memory

Memory is space partitioned across VMs—hypervisor does not have any idea regarding the usage of the allocated memory. For a time multiplexed resource like CPU, the hypervisor can take back the control and access the state of the CPU (e.g., program counter) to gain insights regarding its use. The hypervisor is in the *dark* regarding the utility of the allocated memory unless some explicit communication channel between the virtual machine and the hypervisor exists.

3.2.2. Limited access to guest OS information

The resource usage statistics provided by operating systems are not standardized. The nature and semantics of the information changes between operating systems, even between different versions of the same operating system. In light of guest OS *diversity*, designing hypervisor memory management policies based on information from guest OSs is challenging. Another source of information is indirectly deduced information from the page table updates depending on the memory virtualization method employed (Fig. 6). Assuming page table updates can be tracked by the hypervisor (not easy to track EPT/NPT updates), this method presents two difficulties—first, data gathering incurs larger overheads and second, deduction of usable information from the raw page table data to take memory management decisions is non-trivial.

3.2.3. No direct influence on guest OS policies

The policy decisions of the guest OS are influenced by many factors as shown in Fig. 6. The algorithms used by the guest OS are not easy to manipulate from the hypervisor. For example, changing the victim page selection algorithm of a guest OS from the hypervisor is difficult because either equivalence has to be compromised or a complex dynamic execution time binary translation [26,37] scheme is required. Several indirect designs to influence the guest OS memory management decisions are proposed and we will discuss them in Section 7.

3.2.4. Diverse nature of guest OSs

The design objectives of operating systems are complex and diverse. If the hypervisor can assume generic memory management features of guest OSs, it can indirectly influence the memory management in the guest OSs. The involvement of the hypervisor in the guest VM without guest OS modifications is a desirable design to support heterogeneous guest OSs. Moreover, guest OS intrusive techniques to improve memory efficiency cannot be applied if the guest OS cannot be modified because of commercial or other restrictions.

Summary: Memory management in virtualized system is complicated as several constraints and requirements need to be simultaneously met for an efficient solution. For example, meeting the requirements of high memory utilization under the constraint of adhering to application performance objectives is a non-trivial proposition. Further, design of generic techniques to address the *diversity* aspects of guest OSs and applications is particularly difficult. The rest of the paper discusses solutions that deal with these tradeoffs and constraints.

4. Memory management approaches

The objective of resource management involving multiple entities is to allocate resources as greedily as possible and still meet the individual resource needs of each entity. In case of memory, the hypervisor must allocate enough memory to each VM so that no SLAs (or higher level policies or goals) are violated. Such a policy ensures *tight packing* and saves cost by increasing the number of VMs that can be hosted on a physical machine.

With a page level allocation that allows flexibility to support allocation and de-allocation of machine memory to guest VMs, the hypervisor is required to maintain the mapping dynamically. Few optimizations to increase the global memory utilization may require modifications to the guest OS while other optimizations are transparent to the guest OS. As shown in Table 1, the methods for managing memory efficiently can be broadly classified into three categories.

1. Techniques that exploit varying levels of guest OS memory requirements
2. Techniques to reduce physical memory usage by content deduplication
3. Hypervisor and guest OS symbiosis based techniques.

Each of these techniques can be further classified based on the exact implementation methods. The classifications are not mutually exclusive; and if more than one of these techniques are combined, then the hypervisor should be aware of the implications and impact of one on the other.

4.1. Exploiting dynamic memory demands

Memory requirement of applications varies over time. The hypervisor can exploit this fact to make memory allocation decisions such that the amount of memory allocated to a virtual machine at any point of time is just enough for the applications executing on it.

Techniques to enable dynamic memory allocation – hypervisor swapping, `ballooning` and memory `hotplug` –take advantage of the flexibility of guest physical memory to machine physical memory mappings, to achieve dynamic memory resizing of the virtual machines. However, the adjustments need to be carefully designed such that either the guest OS has knowledge about its implications or is transparent to the guest OS. If transparency is not achievable and modifications are required, then the design should target minimal and non-intrusive changes in the guest OS. An illustration of intrusive and complex change is to modify the page eviction algorithm of a guest OS to consume inputs from the hypervisor and the other guest OSs to meet system level objectives.

Once a technique is devised to enable dynamic memory allocations to the VMs in a transparent manner, the challenge lies in design of a controller that is *accurate* and *prompt*. Accurate estimation of the memory requirement of the virtual machines in a dynamic manner is non-trivial. If the estimation of memory requirements is conservative then memory is under-utilized. An inaccurate estimation on the other hand may result in degraded application performance due to memory thrashing. The promptness to detect change in memory requirement and take actions is another aspect of the controller design. If the control interval is small, frequent change in allocations may result due to transient memory

Table 1
Classification of memory management techniques in virtualized systems.

Optimization	Optimization aspects	Challenges	Solutions
Exploiting varying memory demands	Dynamism, darkness (indirect)	WSS estimation, control interval	**(i)** Dynamic allocation enablers—Demand paging by hypervisor [5], ballooning and memory hotplug [5,38]. **(ii)** Black box controllers—WSS estimation based [5,39,40], Miss Ratio Curve (MRC) based [41,42], Guest IO monitoring [43], hypervisor exclusive caching [44]. **(iii)** Gray-box controllers—Guest OS self-ballooning [45], True working set based [46], Application driven ballooning [36]
Content deduplication	Darkness, Diversity	Transparency, sharing overheads	**(i)** Out-of-band scanning—Page level deduplication (`VMWare` [5], KSM [47–49], `singleton` [50]), Sub-page deduplication (`Difference engine` [51]). **(ii)** Deduplication in IO path—`satori` [52]
VMM and guest symbiotic memory management	Darkness, dynamism, duality	Guest operating system changes	**(i)** Efficient management of memory used for disk block caching—Transcendent memory (`tmem`) [53–55], `tmem` enabled for distributed applications (`Mortar`) [56], hypervisor exclusive caching [44]. **(ii)** Collaborative memory management [57].

demands of the guest resulting in additional overheads. A large control interval on the other hand may be too slow to respond to changing memory requirements. Techniques for memory demand estimation and the control methods are discussed in Section 5.

4.2. Content deduplication

The memory content of different virtual machines can be similar if the operating systems and the applications running in the VMs are similar. A content deduplication method identifies similar content and maintains a single copy, resulting in increased free memory. However, to make the deduplication process transparent to the guest VM, the hypervisor needs to handle the writes to the shared content by any virtual machine involved in sharing.

The method to find content similarity and the granularity at which the content similarity is searched are two important aspects of this technique. Different search methods and the associated parameters determine the additional resource overheads e.g., CPU cycles, and the savings due to sharing. Lower the granularity of sharing, higher is the sharing benefits while resource requirement to achieve the sharing can be higher. Deduplication techniques based on out-of-band memory scanning (e.g., VMWare [5]) can address the problem of *diversity* as their design is generic and transparent to the guest OSs.

While content deduplication is an elegant optimization for managing memory better, it does not guarantee optimal resource utilization when used as the only technique. For example, consider a trivial counter example where two virtual machines are completely idle but they do not have any content similarity. As a result, stand alone controllers for content deduplication are not common and are used to complement other optimizations. Content deduplication is elaborated in Section 6.

4.3. Hypervisor and guest OS symbiosis

Typically, a guest OS does not know whether it is executing on a physical machine or a hypervisor. The hypervisor is not aware of the guest OS memory requirements and the memory management policies. Changes in the memory management policies of the guest OS as per some standard enforced by the hypervisor when running on a virtualized platform is one approach to solve the problem. This requires several modifications in the guest OS and all the virtual machines hosted on the hypervisor are required to implement these changes. If some VMs do not change their policies, fairness of resource usage can be an issue.

Alternatively, the hypervisor can take inputs from the guest OSs regarding their memory management policies and state information to make intelligent decisions regarding memory management at the hypervisor level. This approach is less intrusive for the guest OS memory management because some counters need to be shared with the hypervisor. However, as discussed earlier, the complexity and the *diversity* of OS design makes the choice of exact information exchange and decision making based on those information at the hypervisor non-trivial.

The extent of intervention in guest OS memory management plays an important role in symbiotic memory management. Drastic changes in the OS design is not desirable but changes that can be made with small modifications may be implemented if the overall memory utilization is optimized. In Section 7, the symbiotic approaches are discussed in detail.

Summary: Memory management techniques in virtualized systems can be classified into three broad categories—techniques enabling dynamic memory management, content deduplication techniques, and techniques based on guest OS and hypervisor cooperation. Dynamic memory management requires dynamic memory allocation and control techniques. Content deduplication techniques are based on identification and removal of duplicate memory content. Extent of hypervisor and guest OS cooperation and the exact nature of cooperation are two important aspects of symbiotic memory management techniques. In next three sections, we provide details for each class of memory management approaches.

5. Dynamic memory provisioning techniques

5.1. Dynamic provisioning enablers

Swapping combined with paging to implement virtual memory subsystems in the OSs is a well known technique to support multiprogramming and increase memory utilization. Hypervisor level swapping is a natural extension of OS employed page-level swapping to support dynamic memory allocation to VMs. In a dynamic allocation scheme, a time varying set of guest physical pages are not backed by machine pages. In the event of a page fault caused due to the missing PFN to MFN mapping, the hypervisor

handles the fault by allocating a machine page for the faulting guest page. If no free machine pages are available, the hypervisor swaps out used MFNs to the swap device. Hypervisor swapping has the advantage of addressing *diversity* issues by operating in a guest OS transparent manner. However, there are three basic challenges with hypervisor swapping.

- In order to implement hypervisor level swapping, a demand paging like system needs to be implemented by the hypervisor. Xen [4] for example, does not implement hypervisor level swapping by design to keep the hypervisor thin.
- The hypervisor may not have explicit information (e.g., usage and access recency) to prioritize the memory pages used by a VM. This may result in an inefficient victim page selection mechanism at the hypervisor level.
- Waldspurger [5] describes the possibility of *double swapping* when the hypervisor swaps out a page and afterwards the guest OS decides to swap out the same page based on its eviction policy. To serve the guest OS swap-out, the hypervisor has to swap-in the memory page, only to be swapped out by the guest. This problem arises because of multiple management points (duality) for the same resource where one party (hypervisor) takes an un-informed decision.

5.1.1. Memory ballooning

To overcome the problems of hypervisor swapping, a well known technique known as *memory ballooning* (Waldspurger [5] and Schopp et al. [38]) is used to adjust the guest memory allocations at run-time. A balloon process inside the guest VM (see Fig. 7) allocates memory on hypervisor request and gives back allocated memory to the hypervisor. Once the hypervisor gets the guest physical frames (GPFNs), it frees up their corresponding association with the machine frames (MFNs) and reuses them for allocations to other virtual machines. This process is known as *balloon inflation*. If the hypervisor decides to give back some memory pages to the VM, it first assigns MFNs to the balloon allocated GPFNs and instructs the balloon process to free up allocated memory to the guest OS. This process is known as the *balloon deflation*. Artificially created memory pressure through balloon inflation triggers guest OS actions according to its memory management policies to readjust memory allocations.

The main idea of ballooning is to divide the responsibility of managing a single resource between two decision making entities. During memory pressure, the hypervisor decides the victim VM from which machine pages are to be obtained. Ballooning helps to indirectly push the selected VM to adjust itself to the reduced amount of memory (mostly using swapping). Therefore, unlike hypervisor swapping, ballooning has the advantage of better victim selection at both the levels which results in efficient memory management decisions. However, memory ballooning requires careful design and guest OS intrusive changes which is not needed in hypervisor swapping. Some subtle points related to design and implementation of ballooning technique are highlighted below,

- Memory allocated to the balloon process should not be reclaimed by the guest OS. Typically, operating systems provide APIs to allocate non-evictable memory which may be used by the balloon process.
- Typically, it is guaranteed that the guest will not swap out or assign pages that are allocated to the balloon process. In the worst case of guest OS not supporting page pinning, page faults caused due to guest OS access to the ballooned pages should be handled by the hypervisor.
- Repeated balloon inflation and deflation may cause the actual machine pages allocated to the guest VM to be scattered all over the physical memory. This may impact the applications that assume physically continuous memory.

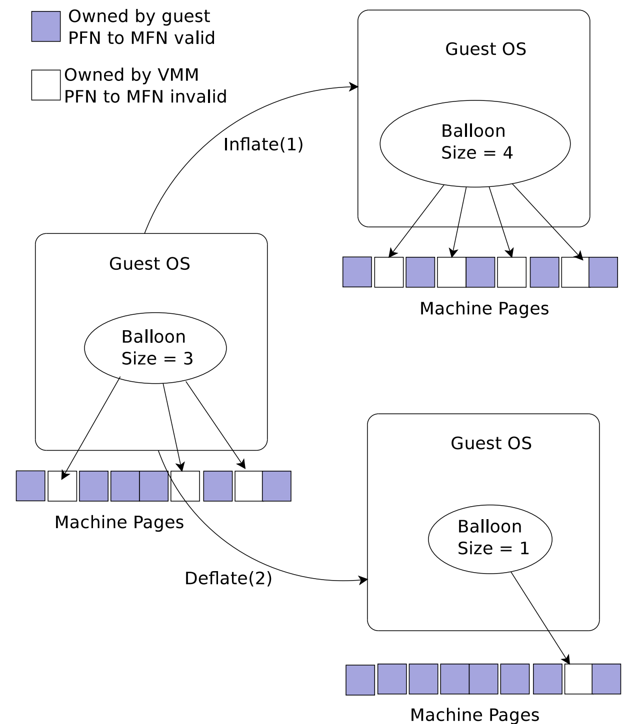


Fig. 7. Variation in mapping of physical mapping of pages to a virtual machine based on the inflation and deflation of the balloon.

5.1.2. Memory hot-plugging

Schopp et al. [38] proposed Linux kernel support for memory hot-plug and hot-unplug. Hot-plug and hot-unplug expand memory by inserting a new memory chip and removing a memory chip, respectively, while the OS is *live and running*. This feature is better applicable in virtualized systems where the memory chip is a virtual hardware resource enabled by the hypervisor for the VMs. Therefore, unlike the infrequent addition or extraction of physical memory chips, a virtualized system can perform frequent change in memory allocations by creating *memory chips in software*. The primary challenges with memory hot-plug are,

- The memory granularity at which the hot-plug and unplug work is restricted by the hardware architecture. Unmodified OSs running on top of virtual hardware implement hot-plug at the same granularity. For adjusting memory size of virtual machines, granularity constraints restrict the optimization possibilities.
- OS support for memory hot plug may not be available because of the complexity of the feature. For instance, to implement hot-unplug, the OS should support memory page migration.

Memory ballooning supports dynamic memory sizing at a page granularity, which is more flexible compared to memory hot-plugging. To overcome the memory granularity problem, Schopp et al. [38] proposed a combination of ballooning and hot-plug techniques to implement dynamic memory allocations.

5.2. Controllers for dynamic memory provisioning

One of the main challenges in designing a dynamic memory provisioning controller is to ensure that virtual machines execute with enough amount of memory without causing degradation to the application performance. To achieve this objective, controllers

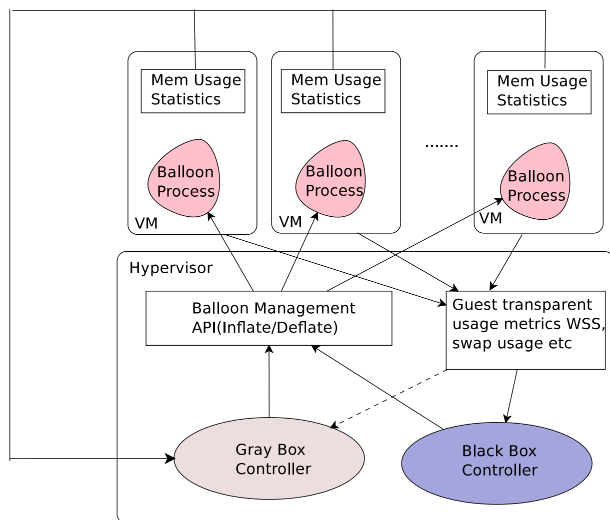


Fig. 8. Black Box and Gray Box approaches for ballooning based memory management.

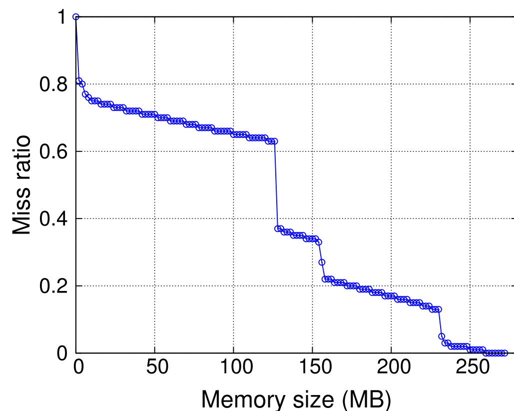


Fig. 9. Miss ratio curve (MRC) for a webserver workload with least recently used eviction policy.

should be aware of the memory requirement of each individual VM at any point of time and take a prompt action when the memory needs change. Denning [39,40] proposed *Working Set Size (WSS)* model to estimate memory requirement of applications, a widely accepted and standard technique for efficient memory management. Working set size is the amount of physical memory that is being actively accessed by an application during a recent time interval. WSS is a good approximation of the memory requirement of a VM provided the time interval used to decide the active nature of memory pages is appropriately chosen. Another important parameter of balloon controllers is the *decision interval*. If periodic estimation of WSS is taken as input to the balloon controller and the period happens to be very large, the reaction to change in memory requirements of the virtual machine may not be prompt. This in turn will result in either wastage of memory or degradation in application performance. Choosing small intervals may lead to balloon deflation and inflation for transient loads leading to ballooning overheads (GPFN to MFN map and unmap operations).

A typical controller in a dynamic ballooning based virtualized setup is shown in Fig. 8. The balloon controller resides in the hypervisor or the control domain (e.g., domain-0 of Xen) or in the host (in a hosted VM architecture like KVM). Information regarding memory requirement and usage of virtual machines can be collected in a guest transparent manner or by querying the guest OS memory usage statistics. The controllers consume this information to control the balloon size of individual VMs through the hypervisor provided API for balloon inflation or deflation. Other alternate implementations integrate the balloon controller inside the balloon process (e.g., Xen self ballooning [45]).

As shown in Fig. 8, black box balloon controllers deduce the memory usage and requirements at the hypervisor level in a guest transparent manner. The gray box balloon controllers access guest OS exposed metrics related to memory usage to deduce memory requirements and usage of a virtual machine.

5.2.1. Black-box approaches of balloon control

One of the primary challenges in black box balloon controller design is to estimate the working set size (WSS) in a guest OS *transparent manner*. The first challenge in WSS estimation is to determine the *time interval* to mark a page active or inactive depending on its access or idleness, respectively. Secondly, accurate calculation of the WSS of a virtual machine at the hypervisor in a black box manner efficiently is difficult because of the *darkness* in virtualized platforms as explained in Section 1. Several techniques to estimate the WSS of VMs to aid black box balloon controller design are described below,

Access interception based techniques: Interception of each memory page access by invalidating the guest physical page (GPFN) to the machine physical page (MFN) mapping is the simplest mechanism to estimate the WSS. This approach is applicable to both shadow paging and EPT/NPT based memory virtualization techniques. However, this method potentially incurs high CPU overhead as every page access results in a page fault. To reduce the page fault trap overheads, a sample set of pages can be invalidated and accesses to them can be tracked. Extrapolation based on WSS estimate of the sampled set yields WSS of the virtual machine [5,42]. The accuracy of sampling based method depends largely on the representativeness of the sampled memory w.r.t. the executing workload(s). Previous works proposed by Waldspurger [5] and Zhao et al. [42] provide configurables to find a tradeoff between estimation accuracy and page fault trap overheads. For example, VMWare ESX server (Waldspurger [5]) employs indirect estimation through sampling idle pages through a configurable to find a balance between accuracy and sampling overheads.

Miss ratio curves: Zhou et al. used Miss Ratio Curve (MRC) [41] to provision memory to applications in an accurate manner. MRC is determined by calculating miss ratios of application memory access for different memory allocation sizes. The miss ratio for any memory allocation size is given by,

$$MissRatio = 1 - \frac{\#PageHits}{\#PageAccesses} \quad (1)$$

The miss ratio curve can be obtained by plotting the page miss ratio against the physical memory allocation. In Fig. 9, the miss ratio curve for a `webserver` is shown. For physical memory allocations of size more than 250 MB, the miss ratio becomes negligible which implies that the WSS of the application is approximately 250 MB. Eviction algorithms conforming to the *inclusion property* may use Mattson's stack algorithm [58] to generate MRC curve without actually varying the physical memory allocation. An eviction algorithms satisfies inclusion property, if for a given memory access sequence, all cache hits with cache size M will also be cache hits for cache size greater than M. An accurate estimation of WSS of virtual machines can be derived from the MRC if guest VM memory access pattern is stable. Waldspurger et al. [59] proposed effective sampling based methods, as explained earlier to generate

MRC with low overheads. Zhou et al. [42] have shown that MRC based WSS estimation with a sampling based approach incurs CPU overheads in the range of 7%–10%.

Monitoring at hypervisor: Geiger [43] and hypervisor exclusive caching [44] infer the page cache activities i.e., promotion and eviction of the pages inside the guest OS by monitoring disk reads and writes by the hypervisor. Promotion implies that, a page is added to page cache to store a disk block while eviction implies a page cache page is evicted to the disk. This information can be used to calculate the working set size of a VM by observing the evictions and subsequent reloads by the VM using methods like ghost buffering [60], a buffer emulation technique to track cache hits and misses without actually allocating buffers. The hypervisor monitoring methods assume that the block I/O operations can be intercepted at the hypervisor level. This assumption holds in most of the hypervisors due to split driver (Xen [4] and KVM virtio [61]) implementation of block devices. In split driver model, the guest OS and the hypervisor communicate explicitly (through a shared memory a.k.a. I/O rings) to provide efficient device virtualization.

Hardware performance counters: Zhao et al. [62] proposed establishing correlation between different Performance Monitoring Unit (PMU) counters of CPU (e.g., TLB miss) [20] and the working set size. Given the existence of a relationship between the WSS and the hardware event(s), this method proves to be the one with the least overhead. However, as Zhao et al. [62] have pointed out, there can be workloads where the direct correlation does not always exist. Intermittent Memory Tracking (IMT)—a combination of WSS estimation using hardware events correlation and software trap method, is used to amortize the cost of WSS estimation. Using this method, CPU overheads due to WSS estimation can be reduced to less than 6%.

Hardware extensions: Zhou et al. [41] proposed explicit hardware extensions to accurately estimate the WSS. However, this idea is not yet incorporated into commercially available hardware. The main idea is to design added circuits integrated to the memory bus access path to profile the accessed physical pages. The operating system can configure the hardware such that it can query the page access history, flush the history and handle overflows by registering a trap handler. In this way, overhead of any of the above techniques that rely on software trap method to build knowledge about WSS can be drastically reduced.

5.2.2. Gray-box balloon controllers

Most OSs maintain statistics regarding the memory usage to aid the OS decision making with regard to memory allocation and deallocations. A balloon controller can take advantage of this memory usage information to estimate the memory need of different guest VMs at different points of time. Standard OS accounting metrics considered as parameters for gray box balloon controllers are as follows,

- OS level memory information—amount of free memory in the system, anonymous memory usage, memory used for caching disk blocks (like page cache), active and inactive memory. Gray box memory controllers like Xen self-ballooning [45] and true working set (TWS) based balloon controller by Chiang et al. [46] use these metrics to make an intelligent guess on the WSS of the VM.
- Swap activity in a given VM implies that the guest OS is under memory pressure. A balloon controller can use the extent of swap activity to estimate the memory requirements of the guest VM.
- Linux provides a single metric—`Committed_AS`, which is the OS estimate of the amount of memory required for the system to avoid swapping [63]. Xen self-ballooning [45] and TWS [46] use this metric for balloon based memory management.

Most of the balloon controllers use `Committed_AS` or an equivalent metric to dynamically determine the size of the balloon. Chiang et al. [46] proposed a sophisticated control based on multiple metrics like `refault and swap-in` along with `Committed_AS`. `Swap-in and refault` events occur when an anonymous page is accessed after it is evicted to swap and a file block is accessed after the page is evicted from the page cache, respectively. These events indicate that more memory can benefit the guest OS by reducing the number of disk access to serve page faults or file reads. True Working Set (TWS) [46] based balloon controller uses `swap-in` and `refault` in addition to `Committed_AS` to estimate the working set size of the virtual machines. The TWS based controller results in up to 18% increase in memory savings and reduces the ballooning overhead by 14% compared to Xen self ballooning [45] which uses only `committed_AS` as the decision parameter.

Server applications like databases and application servers manage memory in an independent manner by-passing the OS memory management layer. For example, databases maintain and manage their own cache for faster query processing. The OS page cache layer is bypassed using synchronous I/O (e.g., O_DIRECT flag in POSIX systems). Salomie et al. [36] have shown that a balloon controller based on guest OS memory statistics fail to estimate the memory requirements of the virtual machine accurately for these applications. Further, Salomie et al. [36] propose extensions to such applications to guide the ballooning process by controlling the balloon size depending their memory requirement and usage.

Summary: Techniques like ballooning and memory hot-plugging enable the hypervisor to dynamically resize memory allocation to VMs which is essential to address dynamism challenges of memory management. The next important challenge is the accurate estimation of working set size of VMs, which is central to the effectiveness of techniques exploiting dynamic memory demands in virtualized systems. Black box techniques employ sampling and heuristic based approaches to minimize the CPU overheads due to page faults while maintaining accuracy of estimation. Black box memory estimation techniques do not depend on the guest OS to derive memory usage related information and therefore can be applied to virtualization setups hosting VMs with diverse OSs. However, with changing application memory footprints and access behaviors, the black box techniques require renewed evaluation to determine the accuracy and propose new techniques, if required. On the other hand, gray box techniques rely on guest OS statistics which can be non-standard and require a channel of communication between the guest OS and the hypervisor. Further, ever changing OS distributions adds to the challenges of employing gray box controllers in a universal manner.

6. Exploiting content similarity for memory management

Elimination of duplicate data in memory can increase the free memory which can be used by other applications or virtual machines. Barker et al. [64] have empirically shown that sharing opportunities can exist intra-VM or inter-VM and both forms of sharing opportunities should be exploited by a deduplication process. Deduplication is a two step process, first the duplicates have to be identified and second, duplicates need to be eliminated in a manner transparent to the guest OSs.

The granularity at which duplicates are identified is an important aspect in design of this optimization. Identification and management of duplicates at fine granularity (e.g., byte-level), while potentially identifies large number of duplicates, but incurs high overheads. Alternatively, searching for duplicates across coarse granularity objects (e.g., files) incurs relatively lesser overheads, but miss out on duplicates smaller than the size of objects. In virtualized systems, deduplication techniques at page granularity are most commonly used.

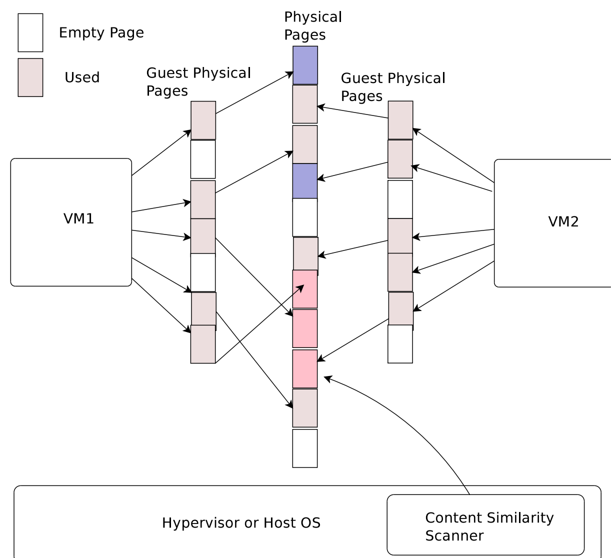


Fig. 10. Same page deduplication via out-of-band periodic scanning. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Offline or out-of-band scanning of memory is a well known method for identifying duplicates. A scanner process periodically scans through the physical memory to locate same memory content and eliminates duplicates. While this method is relatively simple and finds all possible similarities, additional overhead of scanning and merging is required. Finding similarities in the I/O access path, *in-band sharing* (mostly disk access path) avoids periodic scanning but misses out sharing opportunities in anonymous memory.

6.1. Page sharing by out-of-band scanning

In most operating systems memory management policies like ownership, access, mapping etc. are applied at a page level. Therefore, out-of-band scanning techniques proposed in VMWare [5] and KSM [47] periodically scan the physical memory to identify similar content at the page-level. As shown in Fig. 10, an out-of-band page level content similarity scanner scans through all the physical pages allocated to virtual machines. For each page scanned, the process generates a hash value for quick comparison of potential similar content pages, and then performs a byte-by-byte comparison for validating similarity. Typically, the scanner process executes within the hypervisor e.g., VMware ESX or as part of the host OS e.g., Kernel Same-page Merging or KSM in Linux KVM hypervisor. In Fig. 10, the physical pages shaded with the same color represent pages with same content.

Once a page is found with exactly same content as another page, a single copy of the pages is maintained in memory and the duplicate page is marked free. This requires additional changes to the guest PFN to MFN mapping maintained by the hypervisor. The guest PFNs with same content are mapped to a single MFN. Further, writes to a shared page need explicit handling. Each shared physical page is marked *Read Only and Copy-on-Write (CoW)*. CoW is a well known mechanism used in operating systems in different memory related optimizations. For example, modern OS `fork()` implementation to create a child process does not make copy of the parent process memory pages, instead the OS marks the parent pages CoW and makes a copy only when a child or parent process tries to modify any memory page. Similarly, when a virtual machine tries to modify a page that is marked CoW, it causes a trap that is handled by the hypervisor. The hypervisor allocates a new page, copies the content, changes the guest PFN to MFN mapping appropriately and allows write on the new copy. Periodic scan based deduplication techniques combined with CoW fault handling at the hypervisor level results in VM transparent memory deduplication. Therefore, these techniques can effectively address the *diversity* challenges in virtualized setups consisting of VMs with different OSs and applications.

Periodic scanning of memory pages to find similar pages requires CPU resources. Scanning memory aggressively ensures finding short term sharing opportunities at the expense of additional CPU cycles. Alternatively, with a low rate of scanning, short-term sharing opportunities can be missed. XLH [48] and Share-o-meter [65] propose techniques to decide an appropriate scan rate along with adaptive schemes to dynamically determine scan rate based on temporal memory usage trends of applications.

One basic approach to improve efficiency of out-of-band scanning is to provide hints about the likely pages for sharing and guide the deduplication process [48,50,66]. Cross Layer I/O-based Hints (XLH) [48] increases priority of pages used for disk I/O, a heuristic to scan recently modified pages sooner than other pages. The approach not only increases chances of early detection of shareable content but also detects short-term sharing opportunities. XLH reports up to four times more memory savings compared to KSM which follows cyclic scan order.

Singleton [50] proposes several optimizations in hardware and software to make the scanning more targeted and efficient. Nested Page Tables (NPT) proposed by AMD [21] provides a mechanism to notify the software (hypervisor) to track page modifications. This information can be used to compare and merge only the modified pages. A para-virtualized hint based approach—using periodic hints from the guest OS regarding the page modifications (dirty memory pages), can reduce the overhead of periodic scanning.

Catalyst [66] offloads hash computation and comparison at a page granularity to general purpose graphics processing unit (GPGPU). GPU computation can be used to provide hints to the scanning process to perform *targeted* deduplication. Catalyst demonstrates up to 50% savings in CPU utilization for the deduplication process compared to the baseline KSM.

6.2. Out-of-band sharing at multiple granularity

Memory usage optimizations like compression and delta encoding are widely researched. Douglis [67] and Tuduce et al. [68] propose OS-level memory compression techniques to improve memory usage efficiency.

Difference Engine [51] combines three techniques—page sharing, delta encoding or patching and memory compression to improve memory efficiency in virtualized systems. The need for patching is motivated by presence of *pages with almost similar content* in a system. For example, if memory page P2 differs marginally from page P1, then the delta can be stored along with page P1 to reconstruct P2 when required. There are two important aspects to implement patching.

- Finding *similar* pages requires scanning of guest pages and hash generation at sub-page granularity. This requires significant amount of CPU and memory, more than that of a page level matcher. Once such a page pair is found the delta is stored in a designated memory location and the guest PFN to MFN mapping is changed.
- Access to a patched page generates a page fault that the hypervisor handles by applying the patch to the page content and creating a separate copy. While selecting pages for patching, idle memory is preferred to reduce the overheads of patching. However, finding idle pages from the hypervisor requires guest OS memory access monitoring which may incur significant overheads.

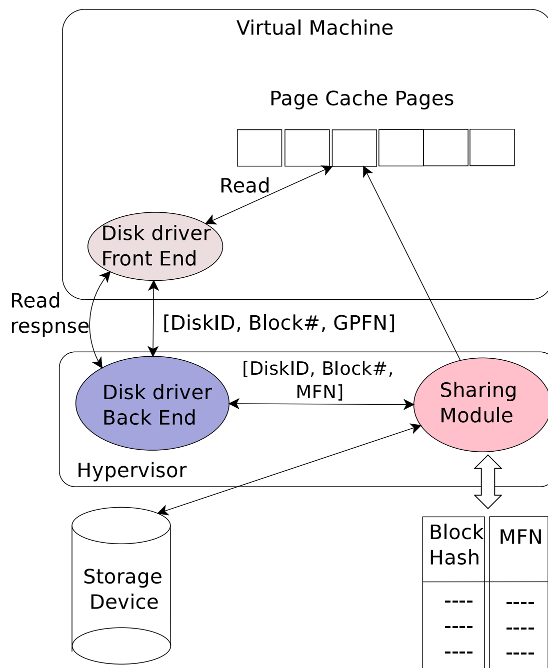


Fig. 11. Same page sharing via I/O access interception.

Compression of memory content by dictionary based compression schemes like LZS [69] or deflate [70] requires careful selection of pages to compress. If the compression ratio for selected page is not significant, then the overhead of compression/decompression will negate the benefits. Similar to patching, the compressed pages ideally should not be accessed frequently so that the benefit can last longer. Access to a compressed page by the guest OS results in a page fault that the hypervisor handles by de-compressing the page content onto a free page and allocating the page to the virtual machine.

Difference engine [51] applies page level sharing as the first preferred method due to efficiency considerations. As an approximate measure of idle pages, it periodically scans page table entries for VM allocated memory pages to check accessed and modified bits. Patching and compression are applied in the order for non-shareable and less frequently accessed memory pages.

6.3. *Deduplicating memory on the I/O access path*

Before an operating system boots, there is no data in main memory. The kernel code, application code, libraries, files etc. are loaded from secondary storage to memory before they are used. Under the assumption that memory allocated by processes and operating system (like stack and heap) are temporary and likely to change, it is fair to expect maximum sharing possibilities in the content read from the disk. Sharing pages *in-line*, on the disk access path, targets the operating system page cache or buffer cache which store frequently accessed disk blocks. Page sharing opportunities in the anonymous memory region of processes cannot be detected by this method. For example, if a process allocates several zeroed pages and uses only a few pages, the sharing potential because of zero pages will not be realized.

Satori [52] proposes in-band page sharing on the disk I/O path as shown in Fig. 11. The disk driver front end in the virtual machine and the back end implementation in the hypervisor are components of a split-driver storage device virtualization framework [4,61]. In a split driver model, the guest and the hypervisor communicate explicitly (through a shared memory a.k.a. I/O rings) to provide efficient device virtualization. A sharing module executing either as part of the hypervisor or the host OS intercepts all read requests originating from the virtual machine. The steps involved in the sharing process are as follows:

(i) The guest OS file-system issues a read request to the virtual device driver with the block number of the virtual device and GPFN(s) to which the disk content will be read. This step is performed after a unsuccessful check for the disk block in the page cache.
(ii) In the split driver implementations (e.g., Xen [4]), the front-end driver in the guest OS issues a read request to the back-end driver providing the guest PFN(s), the virtual disk identifier and the block numbers.
(iii) The back-end driver maps the virtual disk ID to the physical disk ID, the virtual device block number to the physical device block number and the guest PFN to MFN before forwarding the request to the sharing module.
(iv) The sharing module reads the block from the device, calculates a hash of the disk block contents to detect similarity with MFNs. If a similar MFN exists (based on hash comparison), a byte-by-byte comparison verifies the similarity. The current MFN is then marked free, the guest VM mappings are changed to share the previous (same content) MFN and the MFN is marked CoW, if it is not already CoW.
(v) If no match is found, a new hash entry is created with the hash value of the current MFN that contains the disk block.

Note that, a byte level comparison is required for MFN entries not marked CoW because guest OS modifications to a non-CoW MFN do not result in an exception.

Summary: Content deduplication is a best effort mechanism to increase memory efficiency in a virtualized system. Effectiveness of deduplication depends on the memory content which is determined by the VMs and applications hosted in a physical machine. Deduplication at page granularity requires periodic scanning (with configurable scan rate) to locate duplicate memory pages across the VMs. Share-o-meter [65] has shown that overheads of scanning at a higher scan rate can outweigh its benefits. To balance the cost and benefits, simple scan rate controllers like `ksmtuned` [49] consider lower threshold of free memory in the system as the triggering point to increase the scan rate. Sub-page level techniques like Difference Engine [51] incur additional overheads compared to page level deduplication techniques but result in higher memory savings. In-band sharing techniques like Satori [52] has advantage over out-of-band techniques like KSM [47] in terms CPU overheads as Satori does not require periodic memory scanning. However, Satori misses out on short term anonymous memory sharing opportunities and may impact disk access latencies as it performs additional operations in the disk I/O path. VM placement schemes like Memory Buddies [71] are built on top of memory deduplication techniques to use an out-of-band scanner to derive sharing potential between VMs for tight packing (placement) of VMs across a given set of physical machines.

7. Memory management through hypervisor and guest OS symbiosis

In a virtualized setup, the hypervisor is in the dark about the memory usage quality across different virtual machines. For example, a virtual machine may be starving for memory while other VMs are using memory for best effort content caching (e.g., disk cache or page cache). To address these issues, the hypervisor can intervene in the guest OS memory management policies to improve *memory utility* in a system-wide manner. For instance, in the above example scenario of a memory starving VM, the hypervisor may trigger

disk cache eviction in the other VMs. Depending on the levels of hypervisor intervention, the symbiotic memory management can be further classified as follows.

- The hypervisor can employ mechanisms to indirectly force some of the OS features (like disk block caching) to align with the global memory demands.
- Moving a step further would be to modify the guest OS to allow the hypervisor to intervene in the memory management decisions of the guest OS.

7.1. System level second chance caching

All modern operating systems use up the free memory to cache content of secondary device blocks (like page cache in Linux). The page cache is an optimistic memory store that utilizes unused memory to take advantage of frequently accessed disk blocks. The extent of benefit is dependent on the nature of the workload and the I/O access patterns. When only a few disk blocks are found in the page cache, most of the memory used for caching can be used by other VMs. In a virtualized environment, where increasing the memory utilization is one of the primary objectives, the liberty to the guest OS to waste memory in prediction based caching can be detrimental.

Lu et al. [44] propose hypervisor exclusive caching, an initial direction towards maintaining an exclusive hypervisor cache along with the guest OS level disk caches. Transcendent memory a.k.a. `tmem` [53] realizes exclusive hypervisor memory store for the guest OSs to store content and access them at a latter point of time. `tmem` provides two storage semantics—*ephemeral and persistent*. In ephemeral mode, the storage service is best effort i.e., memory content stored is not guaranteed to be present when accessed at a latter point of time. On the other hand, persistent `tmem` semantics guarantees to return the content that was previously stored.

Ephemeral `tmem` store, can be used as a global second chance cache for disk blocks of the guest OSs. This can provide better management of memory used for disk block caching—the hypervisor can evict any block (because of ephemeral semantics) based on a holistic view of the memory requirements of all the VMs and available system memory. Further, ballooning may be used to squeeze out memory from the guest OSs which can be used for `tmem` second chance cache store. Further, the `tmem` pools can implement optimizations like deduplication and compression [54,55,72] to increase memory efficiency.

Guest OSs require modifications to use the `tmem` memory store as shown in Fig. 12. `tmem` provides APIs to create memory pools, `put` disk blocks into the `tmem` pool with a key, access (`get`) pages from the `tmem` pool with a key and purge a disk block (`flush`) for a given key. The key for any file system can be combination of an `inode` number and the block offset within the file which is unique for every file system. The block device subsystem of the guest OS (the disk cache and the block I/O interface) is modified to invoke `get`, `put` and `flush` depending on the actions performed on the page cache. The `tmem` guest interface (Fig. 12) is required to invoke `tmem` pool actions to the `tmem` hypervisor interface (by a VMCALL or hypercall).

A read call from the Virtual File System (VFS) layer is mapped to a disk block by the file system. If the file block is not present in the guest page cache (i.e., page cache lookup fails), a `get` call to `tmem` cache is issued via the `tmem` guest interface (Fig. 12). If the block is found in the `tmem` backend cache, the hypervisor copies the content of the disk block on to the provided page. In case of a failure, the guest OS reads from block device and stores it in the guest page cache.

The guest OS `puts` an unmodified (clean) disk block into the `tmem` cache when a disk block is evicted from the page cache. If the disk block is modified in memory, the contents of the disk block is purged from `tmem` by invocation of a `flush` call.

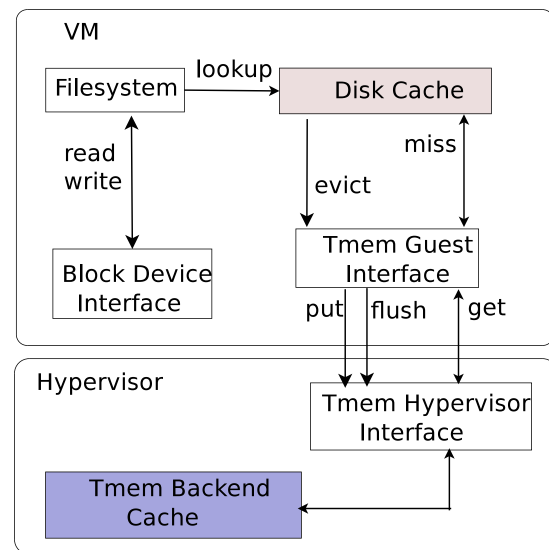


Fig. 12. Hypervisor caching backend (`tmem` backend cache) accessed by `tmem` enabled guests for system wide caching.

The hypervisor maintained page cache (like `tmem`) provides *exclusive* storage semantics with flexibility to apply policies at a system level. However, Mishra et al. [55] empirically demonstrate that `tmem` requires additional CPU cycles (up to 30k cycles) for the VMCALL and data copy between the guest and the hypervisor. Note that, this really does not impact the application level disk I/O performance as disk access speeds are orders of magnitude higher than the access from a hypervisor memory store.

Assuming that all the guest OSs can be modified, hypervisor based (page) caching provides a mechanism to push the speculative optimizations of guest OS to the system level. Further, handing over the page caching to the hypervisor enables application of system level constraints for page cache management – partitioning, eviction and compression – depending on application performance requirements. Additionally, with disk caching in both the guest OS and the hypervisor, deterministic control on the size of cache on a per VM basis may be required. Default page cache design in Linux systems (which is used by KVM in the disk I/O path) maintain a unified inclusive cache. A burst I/O load from a VM can disproportionately fill up the cache or worse compel the host OS to swap-out pages mapped to a virtual machine. Page cache partitioning across VMs (Sharma et al. [73], Synergy [72]) address this issue by providing controls to adjust sizes of the page cache on a per VM basis and also to implement custom per-VM cache eviction policies. The approach enables end-to-end system-wide deterministic memory partitioning across virtual machines.

Mortar [56] extends the basic ephemeral `tmem` to implement a distributed caching store that benefits applications by providing unused memory over the network as a second chance cache. Using the un-utilized memory across a data center, Mortar improves the performance of web applications up to 35%. Mortar also proposes the usage of `tmem` for aggressive disk pre-fetching to improve the disk access performance of guest OSs. Up to 45% improvement in video application performance is reported using disk pre-fetching.

The *persistent* storage model of `tmem` can be used as a second chance memory store for *anonymous* guest memory. A swap cache is a hypervisor maintained persistent memory pool to overcome the problems caused by memory demand bursts of the applications within a VM. The reaction time for such a demand in presence

of techniques like ballooning with a controller can be large. This leads to degraded application performance due to guest OS level swapping. Also, the guest OS can invoke Out of Memory (OOM) killers as the last resort in case of excessive swapping that results in application or OS shutdown. In such a situation, if a memory pool is created in `tmem` to store the swapped content temporarily instead of swapping the page into the swap device, the performance of applications would improve dramatically. The hypervisor can design policies to keep aside some amount of free memory to be used for this kind of burst memory demands that requires guaranteed storage of pages.

While `tmem` is a novel approach to push OS features into a global level of memory management, it requires the guest OS to be cooperative. A non-confirming VM may fool the hypervisor `tmem` implementation by `put`-ing a lot of content into the global memory pool to impact the `put` of other virtual machines. Therefore, `tmem` like solutions are incapable of addressing diversity issues in virtualized systems. Further, to enforce high level application objectives, `tmem` should support dynamic partitioning of cache across virtual machines. Xen and KVM implementations of `tmem` [54,55] provide the basic framework to implement a system-wide hypervisor caching solution.

7.2. Collaborative memory state maintenance

The guest OS and the hypervisor can maintain and explicitly share information regarding each page that is allocated to the virtual machine. A collaborative memory management approach (Schwidefsky et al. [57]) proposes maintenance and sharing of per-page state on two axes—*page usage state* and *page residency state*. Page usage state maintained by the guest OS refers to what a particular page content means to the guest OS. The residency state associated with each page is assigned by the hypervisor to decide what is the nature of the page.

The guest page usage state can be one of the following,

- **Used:** The page is actively used by the guest OS and the content is not discardable.
- **Unused:** The page content is irrelevant to the guest OS and the hypervisor can overwrite without notification to the guest OS.
- **Volatile:** The page content can be overwritten, but it may impact the guest performance. This state is normally assigned to page cache pages which if overwritten can be read back from the disk.
- **Potential Volatile:** The page content is volatile only if it is not dirty. This state suggests that disk flush is pending for the page cache page after it is modified in memory.

The residency state of a page can be one of the following,

- **Resident:** The guest PFN is mapped to a machine page frame and can be accessed without causing any page fault.
- **Backed up:** The guest page content is backed up by a swap device at the hypervisor level, so no machine frame is associated. The hypervisor has the responsibility of handling page fault for such pages.
- **Unused or Idle:** The guest page is neither associated with any MFN nor backed up into the disk. These pages are currently idle or free at the guest level. If they are used, the hypervisor should handle the page fault by assigning a free MFN.

For every page in the guest, a combination of usage status and residency status is maintained and modified when the status of the page changes. Using this information the hypervisor can—(i) accurately determine the guest memory usage and requirements, (ii) take prompt decisions regarding guest PFN to MFN mapping changes to adjust to dynamic memory requirements by guest VMs.

Memory usage and residency state maintenance by the hypervisor and guest OS in collaborative memory management technique can be combined with other techniques like content deduplication and dynamic ballooning. Further, the information sharing framework can be extended to aid efficient memory management controller design. However, this framework expects high levels of symbiosis between the guest OS and the hypervisor which may not be practical in a public cloud setup.

Summary: Efficient memory management in virtualized systems can be realized with different levels of guest OS and hypervisor cooperation. Hypervisor caching is an elegant method to efficiently manage memory used to cache disk content for disk I/O efficiency. The cooperation required to realize hypervisor caching solutions require modifications in the guest OS disk cache layer (e.g., `cleancache` in Linux). Similarly, page swapping functionality in guest OSs can be augmented to use hypervisor maintained swap cache to aid ballooning based dynamic memory management. Collaborative memory management using tightly coupled page level information sharing is another interesting approach to manage memory in a more effective manner. While symbiotic memory management offers scope for narrowing the semantic gap between the hypervisor and guest OS, potentially minimizing memory wastage, it comes at the price of guest OS dependence. Therefore, diversity issues not only remain unaddressed but also become more pronounced for the above techniques.

8. Discussion

Memory management in virtualized systems has been a topic of interest in the research community for the last decade. Several techniques like ballooning, sharing, hypervisor based caching etc. have been proposed as methods for optimized memory management. Over-commitment of memory is an important provisioning requirement for increasing memory efficiency. The level of over-commitment possible in a given setup depends on the memory management techniques employed and the nature of the workloads running inside the VMs. For example, content deduplication based techniques will be limited by the content similarity that is present in a given setup. Study of possible levels of over-commitment and the overheads associated with the technique(s) employed is required to understand the efficiency and applicability of the memory management techniques in virtualized systems.

Guest modifications: Some of the techniques discussed in this paper require guest cooperation. However, not all guest OSs can be modified because of commercial and legal constraints. For example, implementation of `tmem` like solution for Microsoft Windows VMs is not possible because of the unavailability of source code for Windows OS. The implication of memory management technique(s) in presence of a mixture of VMs (co-operative and not co-operative) needs to be studied in greater detail in order to ensure fairness in terms of memory allocation. For example, reward model based mechanisms can be employed to favor conforming OSs for improved management of resources in presence of non-conforming VMs.

Combination of techniques: Achieving maximum level of over-commitment with low overheads may require combination of techniques. While each individual technique has its complexity and overheads, the combination of techniques may present a completely different set of challenges. For example, combination of ballooning and deduplication may not complement each other in a seamless manner because the balloon process (in the virtual machine) may balloon out shared pages impacting the deduplication efficiency. In this scenario, to select the techniques to combine and design controllers for the same need a clear examination of the individual methods and implications when used together.

Symbiotic management: Design of new techniques using better symbiosis between the guest OS and the hypervisor to target

the guest features (e.g., disk block caching) that create problems for efficient memory management at the system level is a new direction of research. This is possible because of increased use of open source operating systems like Linux. Another reason that allows guest OS modification is due to the platform as a service (PaaS) model of cloud service providers. In case of PaaS, the guest OS is in control of the cloud provider and thus can be standardized. However, `tmem` like techniques require further study to explore—co-existence with deduplication and ballooning, and cache sizing to enforce high level application objectives/priorities.

Hardware evolution: Existing memory management techniques in virtualized system may assume some features/properties of underlying memory multiplexing (virtualization) techniques. Memory virtualization techniques change due to the evolution of underlying hardware. The memory virtualization changes may break the assumptions of memory management technique(s). For example, support for large pages (page sizes greater than 4 kB) in MMU and TLB hardware (for MMU translation efficiency) is leveraged by most of the hypervisors. Techniques like page level content deduplication (designed for 4 kB pages) fail to leverage sharing benefits because there may be very few 4 kB pages in a large page enabled system. When free memory in the system is low, the benefits of large pages can be sacrificed [74] for increased sharing benefits (to avoid swapping) by breaking large pages to 4 kB pages. A renewed juxtaposition of memory virtualization and management techniques is required in tune with evolution of hardware evolution and increased support for virtualization.

9. Conclusion

In this survey, we covered two important aspects related to memory resource in virtualized systems. First, the problem of memory multiplexing (virtualization) across multiple VMs was outlined along with the underlying challenges and techniques to address the challenges. Second, the problem of efficient memory management in a multi-VM setting was discussed in detail.

Techniques to multiplex memory across different VMs – shadow paging, direct paging and hardware assisted paging – meet the isolation requirements which is mandatory in a multi-hosting setup. However, direct paging compromises equivalence requirements (requires guest OS modifications) to achieve efficient multiplexing. Hardware assisted paging requires two-dimensional page walks which can be detrimental in terms of efficiency for applications with low locality. Shadow paging performs like a native system albeit with additional overheads for maintaining shadow page tables for every process across all the VMs. So far as efficiency is concerned, neither shadow paging nor hardware assisted paging is a clear winner (Wang et al. [22]). We discussed recent techniques like Agile paging [23] which attempts to combine shadow paging and hardware assisted paging techniques to leverage the best of both techniques.

Challenges in efficient memory management in virtualized system has four different dimensions—darkness, duality, dynamism and diversity. At the core of the problem lies the following necessary evil: two independent system softwares i.e., the guest OS and the hypervisor, try to manage resource as per the policies at their respective levels. In Table 1, a high level classification of memory management techniques, their effectiveness in addressing the challenge(s) was outlined. Three broad categories of techniques – dynamic provisioning enablers, content deduplication and symbiotic memory management – were proposed and a high level view of each was provided. Techniques within each of the high level categories was further elaborated with respect to their design aspects, implications, applicability and extents of benefit. Memory management techniques like ballooning along with policy controls addressing the issues of dynamism were discussed. Efficiency of deduplication techniques and several optimizations were discussed in this work. Further, techniques like Transcendent memory based on hypervisor and guest OS cooperation were presented along with their effectiveness towards efficient memory management in virtualized system.

In Section 8, we summarized the future directions and possible areas of exploration in the scope of memory virtualization and management. While we know that the directions provided were not exhaustive, we expect that this document will be useful to understand existing techniques, explore new directions and further the research in this scope.

Conflict of interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work.

References

[1] G.J. Popek, R.P. Goldberg, Formal requirements for virtualizable third generation architectures, Commun. ACM 17 (7) (1974) 412–421. http://dx.doi.org/10.1145/361011.361073.

[2] F. Bellard, Qemu, a fast and portable dynamic translator, in: Proceedings of the USENIX Annual Technical Conference, 2005, pp. 41–46.

[3] A. Kivity, kvm: the linux virtual machine monitor, in: OLS '07: The Ottawa Linux Symposium, 2007, pp. 225–230.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield, Xen and the art of virtualization, SIGOPS Oper. Syst. Rev. 37 (5) (2003) 164–177. http://dx.doi.org/10.1145/1165389.945462.

[5] C.A. Waldspurger, Memory resource management in vmware esx server, SIGOPSOper. Syst. Rev. 36 (SI) (2002) 181–194. http://dx.doi.org/10.1145/844128.844146.

[6] Oracle. Oracle vm virtualbox [online]. www.virtualbox.org.

[7] Microsoft. Virtualization for your datacenter and hybrid cloud [online]. www.microsoft.com/en-us/server-cloud/solutions/virtualization.aspx.

[8] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, M. Zaharia, A view of cloud computing, Commun. ACM 53 (4) (2010) 50–58. http://dx.doi.org/10.1145/1721654.1721672.

[9] A. Corradi, M. Fanelli, L. Foschini, Vm consolidation: A real case based on openstack cloud, Future Gener. Comput. Syst. 32 (2014) 118–127. http://dx.doi.org/10.1016/j.future.2012.05.012.

[10] T. Wood, P. Shenoy, A. Venkataramani, M. Yousif, Sandpiper: Black-box and gray-box resource management for virtual machines, Comput. Netw. 53 (17) (2009) 2923–2938. http://dx.doi.org/10.1016/j.comnet.2009.04.014.

[11] T.C. Ferreto, M.A.S. Netto, R.N. Calheiros, C.A.F. De Rose, Server consolidation with migration control for virtualized data centers, Future Gener. Comput. Syst. 27 (8) (2011) 1027–1034. http://dx.doi.org/10.1016/j.future.2011.04.016.

[12] R. Iyer, R. Illikkal, O. Tickoo, L. Zhao, P. Apparao, D. Newell, Vm3: Measuring, modeling and managing vm shared resources, Comput. Net. 53 (17) (2009) 2873–2887. http://dx.doi.org/10.1016/j.comnet.2009.04.015.

[13] C. Clark, K. Fraser, S. Hand, J.G. Hansen, E. Jul, C. Limpach, I. Pratt, A. Warfield, Live migration of virtual machines, in: Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2, in: NSDI'05, 2005, pp. 273–286.

[14] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, A. Warfield, Remus: High availability via asynchronous virtual machine replication, in: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation, in: NSDI'08, 2008, pp. 161–174.

[15] S. Santhanam, P. Elango, A. Arpaci-Dusseau, M. Livny, Deploying virtual machines as sandboxes for the grid, in: Proceedings of the 2nd Conference on Real, Large Distributed Systems, USENIX Association, Berkeley, CA, USA, 2005, pp. 7–12.

[16] J.B. Dennis, E.C. Van Horn, Programming semantics for multiprogrammed computations, Commun. ACM 9 (3) (1966) 143–155. http://dx.doi.org/10.1145/365230.365252.

[17] J.B. Dennis, Segmentation and the design of multiprogrammed computer systems, J. ACM 12 (4) (1965) 589–602. http://dx.doi.org/10.1145/321296.321310.

[18] R.C. Daley, J.B. Dennis, Virtual memory, processes, and sharing in multics, Commun. ACM 11 (5) (1968) 306–312. http://dx.doi.org/10.1145/363095.363139.

[19] INTEL. Intel 64 and ia-32 architectures developer's manual: Vol. 3a, [online]. www.intel.com.

[20] INTEL. Intel 64 and Ia-32 architectures developer's manual: Vol. 3B [online]. www.intel.com.

[21] AMD. Amd64 architecture programmers manual volume 2: System programming, [online]. www.developer.amd.com/wordpress/media/2012/10/24593_APM_v21.pdf.

[22] X. Wang, J. Zang, Z. Wang, Y. Luo, X. Li, Selective hardware/software memory virtualization, in: Proceedings of the international conference on Virtual execution environments, 2011, pp. 217–226. http://dx.doi.org/10.1145/1952682.1952710.

[23] J. Gandhi, M.D. Hill, M.M. Swift, Agile paging: Exceeding the best of nested and shadow paging, in: Proceedings of the 43rd International Symposium on Computer Architecture, in: ISCA '16, 2016, pp. 707–718.

[24] J.S. Robin, C.E. Irvine, Analysis of the intel pentium's ability to support a secure virtual machine monitor, in: Proceedings of the 9th conference on USENIX Security Symposium - Volume 9, 2000, pp. 129–144.

[25] AMD. Amd-v nested paging [online]. www.developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf.

[26] K. Adams, O. Agesen, A comparison of software and hardware techniques for x86 virtualization, SIGOPS Oper. Syst. Rev. 40 (5) (2006) 2–13. http://dx.doi.org/10.1145/1168917.1168860.

[27] T.W. Barr, A.L. Cox, S. Rixner, Translation caching: Skip, don't walk (the page table), in: Proceedings of the 37th Annual International Symposium on Computer Architecture, ACM, 2010, pp. 48–59. http://dx.doi.org/10.1145/1815961.1815970.

[28] J. Gandhi, A. Basu, M.D. Hill, M.M. Swift, Efficient memory virtualization: Reducing dimensionality of nested page walks, in: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, in: MICRO-47, 2014, pp. 178–189.

[29] J. Navarro, S. Iyer, P. Druschel, A. Cox, Practical, Transparent operating system support for superpages, SIGOPS Oper. Syst. Rev. 36 (SI) (2002) 89–104. http://dx.doi.org/10.1145/844128.844138.

[30] N. Megiddo, D.S. Modha, Arc: A self-tuning, low overhead replacement cache, in: Proceedings of the 2nd USENIX Conference on File and Storage Technologies, 2003, pp. 115–130.

[31] S. Bansal, D.S. Modha, Car: Clock with adaptive replacement, in: Proceedings of the 3rd USENIX Conference on File and Storage Technologies, 2004, pp. 187–200.

[32] P. Cao, E.W. Felten, A.R. Karlin, K. Li, A study of integrated prefetching and caching strategies, SIGMETRICS Perform. Eval. Rev. 23 (1) (1995) 188–197. http://dx.doi.org/10.1145/223586.223608.

[33] X. Ding, S. Jiang, F. Chen, K. Davis, X. Zhang, Diskseen: Exploiting disk layout and access history to enhance i/o prefetch, in: Proceedings of the USENIX Annual Technical Conference, 2007, pp. 20:1–20:14.

[34] R. Love, Linux Kernel Development (2nd Edition) (Novell Press), Novell Press, 2005.

[35] Kernel documentation [online]. https://www.kernel.org/doc/.

[36] T.-I. Salomie, G. Alonso, T. Roscoe, K. Elphinstone, Application level ballooning for efficient server consolidation, in: Proceedings of the 8th ACM European Conference on Computer Systems, 2013, pp. 337–350. http://dx.doi.org/10.1145/2465351.2465384.

[37] P. Feiner, A.D. Brown, A. Goel, Comprehensive kernel instrumentation via dynamic binary translation, in: Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems, 2012, pp. 135–146. http://dx.doi.org/10.1145/2150976.2150992.

[38] J. Schopp, K. Fraser, M. Silbermann, Resizing memory with balloons and hotplug, in: Proceedings of Linux Symposium, 2006, pp. 313–319.

[39] P.J. Denning, The working set model for program behavior, Commun. ACM 11 (5) (1968) 323–333. http://dx.doi.org/10.1145/363095.363141.

[40] P.J. Denning, Working sets past and present, IEEE Trans. Softw. Eng. 6 (1) (1980) 64–84. http://dx.doi.org/10.1109/TSE.1980.230464.

[41] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, S. Kumar, Dynamic tracking of page miss ratio curve for memory management, in: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, 2004, pp. 177–188. http://dx.doi.org/10.1145/1024393.1024415.

[42] W. Zhao, Z. Wang, Dynamic memory balancing for virtual machines, in: Proceedings of the International Conference on Virtual Execution Environments, 2009, pp. 21–30. http://dx.doi.org/10.1145/1508293.1508297.

[43] S.T. Jones, A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, Geiger: Monitoring the buffer cache in a virtual machine environment, SIGOPS Oper. Syst. Rev. 40 (5) (2006) 14–24. http://dx.doi.org/10.1145/1168917.1168861.

[44] P. Lu, K. Shen, Virtual machine memory access tracing with hypervisor exclusive cache, in: Proceedings of the USENIX Annual Technical Conference, 2007, pp. 3:1–3:15.

[45] Memory overcommit... without the commitment [online]. www.xen.org/files/xensummitboston08/MemoryOvercommitXenSummit2008.pdf.

[46] J.-H. Chiang, H.-L. Li, T.-c. Chiueh, Working set-based physical memory ballooning, in: Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13), 2013, pp. 95–99.

[47] A. Arcangeli, I. Eidus, C. Wright, Increasing memory density by using ksm, in: OLS '09: Proceedings of the Linux Symposium, 2009, pp. 19–28.

[48] M. Konrad, F. Fabian, R. Marc, H. Marius, B. Frank, XLH: More effective memory deduplication scanners through cross-layer hints, in: Proceedings of the USENIX Annual Technical Conference, 2013, pp. 279–290.

[49] Fedora documentation: KSM on fedora [online]. http://docs.fedoraproject.org/en-US/Fedora/18/html/Virtualization_Administration_Guide/chap-KSM.html.

[50] P. Sharma, P. Kulkarni, Singleton: system-wide page deduplication in virtual environments, in: Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing, 2012, pp. 15–26. http://dx.doi.org/10.1145/2287076.2287081.

[51] D. Gupta, S. Lee, M. Vrable, S. Savage, A.C. Snoeren, G. Varghese, G.M. Voelker, A. Vahdat, Difference engine: harnessing memory redundancy in virtual machines, Commun. ACM 53 (10) (2010) 85–93. http://dx.doi.org/10.1145/1831407.1831429.

[52] G. Miłós, D.G. Murray, S. Hand, M.A. Fetterman, Satori: enlightened page sharing, in: Proceedings of the USENIX Annual Technical conference, 2009, pp. 1–14.

[53] D. Magenheimer, C. Mason, D. McCracken, K. Hackel, Transcendent memory and Linux, in: Proceedings of Linux Symposium, 2009, pp. 191–200.

[54] D. Magenheimer, Update on transcendent memory on Xen [online]. https://oss.oracle.com/projects/tmem/dist/documentation/presentations/TranscendentMemoryXenSummit2010.pdf.

[55] D. Mishra, P. Kulkarni, Comparative analysis of page cache provisioning in virtualized environments, in: Proceedings of International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS '14), 2014, pp. 213–222. http://dx.doi.org/10.1109/MASCOTS.2014.35.

[56] J. Hwang, A. Uppal, T. Wood, H. Huang, Mortar: Filling the gaps in data center memory, in: Proceedings of the 10th International Conference on Virtual Execution Environments (VEE), 2014, pp. 53–64. http://dx.doi.org/10.1145/2576195.2576203.

[57] M. Schwidefsky, H. Franke, R. Mansell, H. Raj, D. Osisek, J. Choi, Collaborative memory management in hosted linux environments, in: Proceedings of Linux Symposium, 2006, pp. 123–138.

[58] R.L. Mattson, J. Gecsei, D.R. Slutz, I.L. Traiger, Evaluation techniques for storage hierarchies, IBM Syst. J. 9 (2) (1970) 78–117. http://dx.doi.org/10.1147/sj.92.0078.

[59] C.A. Waldspurger, N. Park, A. Garthwaite, I. Ahmad, Efficient mrc construction with shards, in: Proceedings of the 13th USENIX Conference on File and Storage Technologies, 2015, pp. 95–110.

[60] R.H. Patterson, G.A. Gibson, E. Ginting, D. Stodolsky, J. Zelenka, Informed prefetching and caching, SIGOPS Operating Systems Review 29 (5) (1995) 79–95. http://dx.doi.org/10.1145/224057.224064.

[61] R. Russell, Virtio: Towards a de-facto standard for virtual i/o devices, SIGOPS Oper. Syst. Rev. 42 (5) (2008) 95–103. http://dx.doi.org/10.1145/1400097.1400108.

[62] W. Zhao, X. Jin, Z. Wang, X. Wang, Y. Luo, X. Li, Low cost working set size tracking, in: Proceedings of the USENIX annual technical conference, 2011, pp. 1–6.

[63] Featured article: /proc/meminfo explained [online]. www.redhat.com/advice/tips/meminfo.html.

[64] S. Barker, T. Wood, P. Shenoy, R. Sitaraman, An empirical study of memory sharing in virtual machines, in: Proceedings of the USENIX Annual Technical Conference, 2012, pp. 273–284.

[65] S. Rachamalla, D. Mishra, P. Kulkarni, Share-o-meter: An empirical analysis of ksm based memory sharing in virtualized systems, in: Proceeding of 20th International Conference on High Performance Computing (HiPC), 2013, pp. 59–68. http://dx.doi.org/10.1109/HiPC.2013.6799096.

[66] A. Garg, D. Mishra, P. Kulkarni, Catalyst: Gpu-assisted rapid memory deduplication in virtualization environments, in: Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, 2017, pp. 44–59.

[67] F. Douglis, The compression cache: Using on-line compression to extend physical memory, in: Proceedings of 1993 Winter USENIX Conference, 1993, pp. 519–529.

[68] I.C. Tuduce, T. Gross, Adaptive main memory compression, in: Proceedings of the USENIX Annual Technical Conference, 2005, pp. 1–14.

[69] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Trans. Inform. Theory 23 (3) (2006) 337–343. http://dx.doi.org/10.1109/TIT.1977.1055714.

[70] P. Deutsch, Deflate compressed data format specification version 1.3, 1996.

[71] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, M.D. Corner, Memory buddies: exploiting page sharing for smart colocation in virtualized data centers, in: Proceedings of tHe International Conference on Virtual Execution Environments, 2009, pp. 31–40.

[72] D. Mishra, P. Kulkarni, R. Rangaswami, Synergy: A hypervisor managed holistic caching system, IEEE Trans. Cloud Comput. pre-print (2017) 1–14.

[73] P. Sharma, P. Kulkarni, P. Shenoy, Per-vm page cache partitioning for cloud computing platforms, in: Proceedings of the 8th International Conference on Communcation Systems and Networks, 2016.

[74] F. Guo, S. Kim, Y. Baskakov, I. Banerjee, Proactively breaking large pages to improve memory overcommitment performance in vmware esxi, in: Proceedings of the 11th International Conference on Virtual Execution Environments, 2015, pp. 39–51. http://dx.doi.org/10.1145/2731186.2731187.