COORDINATED MEMORY MANAGEMENT IN VIRTUALIZED ENVIRONMENTS

A Thesis
Presented to
The Academic Faculty

by

Dushmanta Mohapatra

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
December 2015

COORDINATED MEMORY MANAGEMENT IN VIRTUALIZED ENVIRONMENTS

Dedicated to my parents Dibakar, Sabita, and my lovely wife Ellina, for they were as much a part of this journey, as me.

ACKNOWLEDGEMENTS

Completing a dissertation is a long and difficult road that I never could have navigated alone. There are too many people I must thank, but a few deserve special note. First and foremost among them is my advisor, Prof. Kishore Ramachandran, without whose research advice, unwavering support, and constant encouragement I never could have done this.

I would also like to thank my committee members, Prof. Mustaque Ahamad, Prof. Santosh Pande, Prof. Milos Prvulovic and Dr. Kalyan Perumalla whose guidance has helped me in completing this dissertation.

I would like to take this opportunity to thank all the members of Embedded Pervasive Lab with whom I have interacted over the years. I have learned a lot from them and I will cherish the times spent in this lab for the rest of my life.

Finally, I sincerely appreciate and thank my friends, both at Georgia Tech and in Atlanta at large, who have stood by me and supported me through this roller coaster journey.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

In this age of bloated software systems, there is a direct correlation between system performance and available physical memory. Also, virtualization technology is gradually becoming ubiquitous. It has already become a staple in the utility computing space (i.e., the Cloud). In virtualized systems, owing to the ever-increasing processing capacity of the CPUs, more and more virtual machines are being consolidated on a single hardware platform, thus resulting in increasing demands for memory. On top of that, virtual machine based systems inherently demand more physical memory to support the total workloads than an equivalent single OS process environment would because of the overhead of the duplication of OS code, data, data structures and greedy OS memory algorithms. Due to the time-varying nature of the memory needs of various applications running inside a virtual machine (VM), it is difficult to predict the future memory requirements of the VMs. There are two options for dealing with the application dynamism in the context of memory usage: (1) over-provisioning of the memory, and (2) dynamic balancing of the available memory among the set of virtual machines. In case of VMs containing critical applications and/or system components, over-provisioning of resources is desirable. But in case of virtual machines containing non-critical applications, over-provisioning results in under-utilization of resources for a majority of operating time. With the assumption that, not all virtual machines (hosted on one physical platform) experience memory spikes simultaneously, situations like over-provisioning and under-utilization of memory resources could be avoided by crafting effective dynamic memory balancing and management mechanisms in the virtualization layer. Memory over-commitment mechanisms like ballooning and transcendent memory are examples of such mechanisms.

Adoption of virtualization into embedded platforms such as smart-phones and handheld devices is a fairly new phenomenon. Virtualization offers the potential for multiple-use and versatility of such platforms by enabling the co-location of diverse software platforms and sand-boxing of critical and non-critical components on the same hardware platform. It has already been introduced in smartphones and we expect that other high-end embedded platforms such as automobiles to follow suit in the near future. However, the application space of such platforms differ significantly from the traditional server world where virtualization has hitherto found extensive use. Applications that run on such platforms are highly latency sensitive and display bursty requirements for memory. Therefore, effective memory management and careful orchestration of memory balancing mechanisms is an important research endeavor for virtualized embedded platforms. Low operational latency becomes a vital requirement of any such mechanism. This necessitates deep analysis of the individual components of the overall operation and optimizing them by appropriate design and implementation.

The problem being investigated in the research presented in this dissertation is about the most optimal way to architect a mechanism to achieve dynamic memory partitioning in a virtualized environment with minimal impact on system performance. Through appropriate design, implementation and runtime adjustments, we alleviate the various performance issues associated with memory balancing process in order to decrease the latency overheads.

This dissertation consists of various subparts. We start by laying the base for the dissertation by exploring relevant memory management mechanisms and their suitability towards catering to the aforementioned needs of virtualized embedded devices. Subsequently, we present the details of a coordinated memory management mechanism designed in accordance with the split driver principle used in other para-virtualized drivers in xen setup, which uses xen-store and xen event-channel for data transfer and coordination purposes. Next, we present detailed analysis of this memory management mechanism comprising of latency results, results showcasing the interference from scheduling policy and an analysis of latency involved in various steps of the operation. Armed with these measurement driven insights, we delve into improving the latency involved in the operation of the coordinated memory management mechanism in the subsequent chapters. First, we explore the latency reduction possible by minimizing the scheduler interference. This is achieved by effecting appropriate coordination constructs between the operation of the mechanism and the schedulers inside Xen hypervsior and Linux kernel and we present results and analysis towards validating that. Next, we address another finding of our previous analysis which showed that using a xen-store based communication and coordination approach has some inherent latency overhead, which could be avoided by appropriate modification in design and implementation. We achieve this by using the up-call mechanism available in xen hypervisor and transferring much of the central logic into the hypervisor from that of Dom-0. We also present results from our measurement experiments which showcase the benefits of this design approach over earlier approach. In the next section, we compare the latency of our mechanism, which is built for inter-VM operations, to that of the inherent latency of intra-VM memory management mechanisms ascertained by measuring the latency experienced by a user mode process for getting access to various memory amounts.Finally, we conclude by summarizing the key findings of this dissertation and discussing about possible future work.

CHAPTER I

INTRODUCTION

1.1 Virtualization

In computing, the term virtualization corresponds to creating a virtual version of a device, or resource, such as a CPU, storage, network or even an operating system, where the framework divides the resource into one or more execution environments. This dissertation operates in the context of virtual machines. The concept of virtual machine(VM) was invented by IBM as a method of time-sharing extremely expensive mainframe hardware [9, 13]. Figure 1 represents a traditional organization of a virtual machine system. A software layer called a virtual machine monitor(VMM)(/hypervisor) takes complete control of the machine hardware and is responsible for managing and exporting the real resources to the virtual machines. Each VM is given the illusion of being a dedicated physical machine that is fully protected and isolated from other virtual machines and usually runs its own operating system(OS) and applications. Multiplexing of physical resources happens at the granularity of entire operating systems.

In the current day, there are different types of virtualized setups: 1) fully virtualized setups such as VMware [47] and QEMU [5], which allow unmodified OSes to run on top of hypervisors 2) paravirtualized setups such as Xen [3], in which the guest OS is modified to

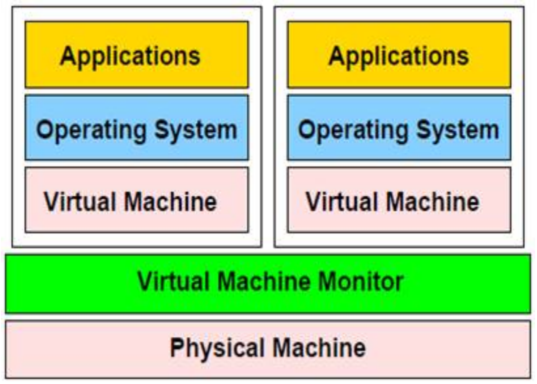


Figure 1: Organization of a Virtualized System

be able run on top of a hypervisor and 3) Hosted operating system (OS) level virtualization setups such as KVM [26], where a OS kernel (like Linux) acts as the base virtualization layer and allows other VMs to run on top of it.

1.2 Proliferation of Virtualization

Virtualization as a software construct is being used in a variety of domains, ranging from cloud computing infrastructures and data centers to modern day smartphones. The purpose of introducing virtualization into different domains varies. The enterprise scale virtualization in cloud computing setups and data centers was primarily designed to avoid underutilization of individual servers (by allowing them to be consolidated as virtual machines on a single physical server) and provide necessary isolation and security guarantees by sand-boxing of the individual VMs. Additional benefits like reduction in energy usage, appropriate usage accounting etc. made virtualization technology very critical for the emergence and success of cloud computing era and paved way for its subsequent introduction into other domains.

In the embedded and hand-held devices arena, features like versatility (enabling the usage of multiple applications designed for different OSes) and security are more important. Virtualization, with its ability to run multiple computing domains on a single device allows a user to segregate applications into multiple VMs according to their security demands and functionality. Moreover, it enables to run different OSes according to the application characteristics. For example, a smart-phone application could run on a real-time OS while a downloaded game application could run on a general purpose OS that provides rich graphical user interfaces.

The current trend of automobiles becoming software heavy also creates the near future possibility of virtualization being introduced into their software stack. The increasing number and sophistication levels of software modules in the automobiles results in increasing demand for computing power. This in turn requires car manufacturers and suppliers to introduce powerful multi-core electronic control units(ECU) into their components, which offer features such as higher levels of parallelism [36]. The resulting system is quite complex and calls for new framework and methodologies to ensure their correct operation. Moreover, achieving strong isolation guarantee between the software modules responsible for critical components and those responsible for feature rich infotainment applications is one of the critical system design factors. Virtualization with its sandboxing features is able to provide security and fault-containment guarantees, which potentially may be the driving factor behind introduction of virtualization into automobile software systems [38].

1.2.1 Virtualization Use Cases

Virtualization already has a strong foothold in the server and data-center world where it is primarily used for consolidation of virtual machines into physical machines and dynamic balancing of workload on physical machines by utilizing virtual machine migration functionality. In the recent past, virtualization has also made inroads into desktop and laptop domains. Desktop virtualization was introduced for flexible and secure management of desktop environments. Virtual desktop infrastructure (VDI)[1] pushes virtual desktops into centralized server farm to which users connect via network. In order to mitigate the issues with network disconnection and bad user experience, client side virtual desktops, which involve checking out of VM images from servers to a local device, were introduced.

With the growing popularity of laptops, people have started using laptops both for personal purposes and for business tasks and this significantly increases the possibility of corporate information leak by malicious attacks and viruses. Virtualization has been used to provide strong isolation guarantee between the business and personal environments. A similar story has been repeated in the case of consumer electronic devices, mostly in the form of smart-phones. Due to increase in their computational power, memory and storage capacity, smart-phones nowadays are capable of doing many PC-like tasks (multimedia playback, games, social networking activities, wi-fi downloads etc.) apart from their primary functions of calling and texting. With the growing usage of smart-phones for security sensitive applications like Internet-banking, there is a strong need for protecting the primary functionalities and critical applications from the rest of the system, for which virtualizing

[1]a variant of thin client computing

the system is the only viable solution. Considering the usage mode, be it the case of desktops and laptops or smart-phones, the user will mostly interact with only one virtual machine at one time. Thus, it makes sense to allocate as much resource as possible to the virtual machine with which the user is interacting. Achieving this goal is not much difficult for CPU and I/O devices, but in the case of memory, it is significantly harder to achieve.

From a futuristic point of view, when virtualization finds a place in the software stack of automobiles, its primary purpose will be to isolate the critical components and software modules from the non-critical ones. A real-time OS will be catering to the critical components and corresponding software modules, whereas the other modules (for infotainment, media streaming, application downloading etc.) will be running on top of one or more general purpose OSes. It is imperative to have an optimum and static allocation of resources for the real time OS pertaining to the critical modules. Among the general purpose OSes(/VMs), resource allocation can happen based on their priority and requirements, more in line with dynamic resource allocation in virtualized smart-phones and other hand held devices.

1.3 Memory Resource Management in Virtualized Environments

Irrespective of the cause behind the introduction of virtualization into any system, the primary mechanisms and features on top of which a virtualized setup operates are more or less similar. There is a need for appropriate multiplexing of CPU, I/O cycles and proportionate partitioning of available memory and disk, among the operating VMs. Though, most physical resources, such as processor cores and I/O devices are shared among virtual machines using time slicing and can be multiplexed flexibly based on priority, allocating an appropriate amount of physical memory to virtual machines is more challenging.

Owing to the limited capacity (unlike CPU, I/O cycles which are unlimited) of available memory and a rather direct impact of allocated memory size on system performance, effective memory partitioning becomes critical in a virtualized setup. The uncertainty about the future memory needs of applications inside various VMs adds to the complexity and renders static memory partitioning completely ineffective for the task at hand. Introduction of virtualization into non-traditional domains like client devices and automobiles also has significant implications for appropriate memory management mechanisms. Due to the resource constrained nature of these devices and their growing usage for resource intensive tasks like video streaming, gaming etc., the role of fast and dynamic memory allocation and balancing mechanisms becomes crucial.

So, from the early days of modern virtualization, researchers and system designers have been looking into dynamic approaches for memory partitioning, memory page sharing etc. *Ballooning* and *content based page sharing* mechanisms were introduced with VMWARE's ESX-Server and have since been replicated into almost all mainstream hypervisors. Researchers have devised mechanisms for estimating the working set size of the VMs. More recently, the concept of *transcendent memory* was introduced into Xen and other open source hypervisors.

1.3.1 Intra-VM Memory Management

Inside an OS, there are two different types of memory allocation: memory allocation for the kernel and the memory allocation for user-mode processes. In the context of Linux and other POSIX[2] compliant OSes, memory allocation for the user-mode process occur by the function call 'malloc' and its variants like 'calloc', 'realloc' etc. These allocation routines are usually implemented with the 'sbrk' system call, which either expands or contracts the heap of the process based on the outcome of the operation.

Memory allocation inside the OS kernel is not as straightforward as memory allocation for user mode processes. Unlike user-space, the kernel cannot always sleep and cannot easily deal with memory errors. Linux kernel has interfaces for memory allocation at the granularity of memory pages ('alloc_pages', __get_free_pages) or byte sized chunks ('kmalloc', 'vmalloc'). Because of hardware limitations (some pages because of their physical address in memory, cannot be used for certain task like DMA), the kernel divides pages into various zones in order to group pages of similar properties. The memory allocation routines have

[2]an acronym for "Portable Operating System Interface", is a family of standards specified by the IEEE for maintaining compatibility between operating systems.

the option to allocate pages from a particular zone.

1.3.1.1 Low Memory Situations

In the otherwise usual operation of an OS (or VM), a shortage of available physical memory, with regards to the demand for memory pages, could lead to either *thrashing* or *OOM (out of memory)* conditions. Thrashing occurs when a computer's virtual memory subsystem is in a constant state of paging, rapidly exchanging data in memory for data on disk, to the exclusion of most application-level processing. Out of memory (OOM) is a state of operation (often undesired) where no additional memory can be allocated for use by programs or the operating system. In such a state, the system will be unable to load any additional programs or allocate any additional memory to any existing process and thus resulting in their erroneous functioning. The primary cause of such a situation is that all available memory, including disk swap space, has been allocated.

1.3.2 Inter-VM Memory Management

In a virtualized environment, multiple VMs operate simultaneously on top of a single hypervisor and *memory overcommitment* is used for dynamic management of available memory among the set of virtual machines. Overcommitment means that the total size configured for all running virtual machines exceeds the total amount of actual machine memory and the system manages the allocation of memory to individual VMs at runtime based on their need, system load and other parameters.

When memory is overcommitted, the hypervisor must employ some mechanism to reclaim memory from one or more virtual machines in order to give them to others. There could be two primary approaches for achieving memory overcommitment:

- **Dynamic Partitioning:** Guest OSes coordinate with the hypervisor to dynamically change their memory size to accommodate a global memory strategy.

- **Host Paging:** The hypervisor transparently swaps out some of the memory pages allocated to a VM, when needed. Conceptually this is similar to the way any operating system kernel does paging for the various processes and applications running on top of it.

Host paging is possible when the hypervisor is actively involved in the memory management of the VMs running on top of it. This extra level of paging requires a meta-level page replacement policy. The hypervisor must choose not only the VM from which to revoke memory, but also which of its particular pages to reclaim. In general, meta-level page replacement at the hypervisor layer has to make relatively uninformed resource management decisions. The best information about which pages are least valuable, and thus could be reclaimed, is known only by the guest operating system within each VM. A sophisticated meta level policy is likely to introduce performance anomalies due to unintended interactions with the native memory management policies in guest OSes. Different OSes may have varied priorities and policies for page replacement (based on the needs of the applications running inside them), which creates further complexities for meta-level page replacement efforts. The fact that paging is transparent to the OSes running in the VMs can result in *double paging* even when the meta-level policy is able to select the same page that the native guest OS would choose.

For the reasons mentioned above and the security advantages of having thin hypervisors (in terms of reduced size of trusted computing base), all modern hypervisors are designed to not play any role in the paging activities of the VMs running on top of them. All the discussions in this dissertation are in the context of Xen hypervisor and para-virtualized mode of operation of the virtual machines. In this mode, the page-tables are para-virtualized, i.e. the hypervisor merely ensures that the VMs create only valid mappings for the memory pages assigned to it. Dynamic partitioning is the only viable means of memory management in such cases and a technique called *ballooning* is used to achieve that. Ballooning operates by introducing a pseudo device driver (or kernel service) called balloon driver inside each VM. The basic function of the balloon driver is to pass memory pages back and forth between the hypervisor and the kernel memory management unit in the VM, which it achieves by either inflating or deflating of the balloon . When the hypervisor needs to reclaim some memory from a VM, the corresponding balloon driver is activated and it inflates itself by asking for memory pages from the memory management unit and releasing the pages to the hypervisor. Deflating involves deallocating of previously-allocated pages and thus making them available for general use by guest OS. Recently, a parallel approach at memory management called *Transcendent Memory* was also devised. This operates with the philosophy of collecting un-utilized and under-utilized memory pages from the various VMs to form a centralized memory pool from them. Later the VMs are provided with an indirect page copy based access to these central pool, which they can use in time of memory scarcity.

1.3.3 Memory management in the context of current and future applications

The aforementioned mechanisms and their variants and hybrids are currently part of the ever-evolving virtualization ecosystem. But, a comparative analysis of the existing mechanisms and their appropriateness for various scenarios, in which they would be used, is not available at present. So there is a need for a detailed evaluation of the mechanisms with respect to the needs of the various applications. Among these applications, streaming media and video-games are noteworthy. Owing to their increasing popularity, resource intensiveness and interactive nature, they require fast response for their resource needs. Also missing from existing literature is, an analysis of how the memory management mechanisms affect and get affected by the rest of the system policies and mechanisms. With the possibility of virtualization getting introduced into software stacks, where some of the critical components(having stringent resource requirements) co-exist with components which are not system critical, but are memory intensive nonetheless, the mechanisms should be designed to satisfy the needs of all these components.

1.4 Problem Statement

Two recent advancements are the primary motivating factors for the research in this dissertation. First, virtualization is no longer confined to the powerful server class machines. It has already been introduced into smart-phones and will be a part of other high-end embedded systems like automobiles in the near future. Second, more and more resource intensive applications are being used in devices which are rather resource constrained and introducing virtualization into the software stack just exacerbates the resource allocation issue.

Existing memory-management mechanisms were designed for server class machines and their implementations are geared towards the applications running primarily on data centers and cloud setups. In these setups, appropriate load balancing and achieving fair division of resources is the goal and over-provisioning may be the norm [16]. Latency involved in resource management mechanisms may not be a big concern. But in case of smart phones and other hand held devices, applications like media streaming, social-networking are prevalent, which are both resource intensive and latency sensitive. Moreover, the bursty nature of their memory requirement results in spikes in memory needs of the virtual machines. As over-provisioning is not an option in these domains, fast and effective (memory) resource management mechanisms are necessary.

Usually, operating system (kernel) owns the entire physical memory and allocates a portion of the memory to a process requesting for more memory (by invoking the 'malloc' system call or its variants like 'kmalloc'/'vmalloc'etc.). When there is shortage of free memory, the kernel tries to free-up some memory by writing pages to disk files, flushing the contents of clean pages etc., or in extreme situations terminating existing processes. In virtualized environments, these low memory situations could also be dealt with by getting memory from other domains (that are not experiencing memory pressure at that instant) or hypervisor. Dynamic memory management mechanisms like *ballooning* and *transcendent memory* are designed for this purpose. But due to the latency and uncertainty associated with these inter-VM memory balancing mechanisms, they are not directly integrated with the intra-VM (in-kernel) memory management units and operate more or less decoupled from them. Further, these mechanisms act in a non-coordinated manner and are oblivious to what is happening inside the other VMs, and as such are not geared towards handling the different levels of tolerability for latency for the varied mix of applications in the different virtual machines.

So, achieving memory over-commitment and coordinated dynamic memory balancing, with low latency and as little performance penalty as possible, is the primary research problem that is being addressed in this dissertation.

1.5 Thesis Statement

On-demand and coordinated memory management mechanism is essential for dealing with unexpected and bursty memory needs of various applications and the resulting spikes in VM memory requirements. Towards this goal, we created a memory management mechanism built on top of ballooning principle, where multiple VMs cooperate to alleviate a memory pressure situation in one of the VMs. While the idea is simple and straightforward, this process of coordinating between a number of VMs to achieve memory balancing is inherently distributed in nature and as such needs careful orchestration of the individual components and the coordination constructs for achieving goals of low operating latency. Specifically, the critical functionalities that need careful consideration are the process of choosing VMs for releasing the pages, the process of making the selected VMs release the pages, the mechanism used for communication among the VMs. Also, the interference from external components, especially the schedulers, needs to be considered. If all these issues are appropriately catered to, inter-VM memory balancing at its core is not very different from intra-VM memory allocation, albeit with more number of players. This background allows us to state our thesis statement as "*With appropriate design and implementation, it is possible to devise a coordinated Inter-VM memory management scheme with a latency that is within a few factors of the latency involved in Intra-VM memory management schemes*".

1.6 Contributions

We make the following contributions through this dissertation:

- We analyze the memory requirement pattern of some prevalent applications in smart phone world and showcase the need for low latency memory management mechanisms in virtualized environments for these devices.

- Realizing that over-provisioning may not be a viable option in these resource constrained devices, we propose a coordinated memory management mechanism (CMM) that is appropriate for adoption of virtualization in the resource-constrained embedded platforms.

- We present a detailed measurement infrastructure built into the Xen hypervisor in order to accurately identify the various sources of latency in CMM operation.

- We provide detailed analysis of the operation of CMM and identify avenues for reduction in operational latency, one of which is interference from schedulers. We propose and implement optimizations in the Linux CFS scheduler and Xen credit scheduler towards reducing the interference and bounding the operational latency.

- Based on our analysis, we find that our baseline implementation approach using xen-bus and xen-store based communication and coordination constructs has inherent latency implications due to the file backed implementation of xen-store and its transaction based interface for writes. We address this issue by creating an optimized implementation of CMM by changing our design and implementation to that of xen-upcall and hypercall based implementation, which also requires us to transfer the centralized logic from Dom-0 to the hypervisor. This allows us to achieve significant improvement in the operational latency over our previous design.

- We provide measurement results comparing latency of CMM with relevant mechanisms for inter-domain and intra-domain memory management mechanisms.

- To summarize, through system implementation and empirical system evaluation, we argue towards validating our thesis statement from previous section.

1.7 Roadmap

This dissertation is composed of ten chapters. Chapter 2 2 presents a brief overview of the other research works that are related to the interest space of this dissertation. We introduce and discuss about core mechanisms for memory management in virtualized environments: *Ballooning*, its prevalent implementation form of *Self-Ballooning* and *Transcendent Memory.* We also present a brief analysis of the memory requirement behavior of frequently used applications, and thus provide a context for much of the discussion in this dissertation. Next in chapter 3 3, we present the design and implementation of a new coordinated memory management mechanism, built according to the split-driver principle. Chapter 4 4 presents detailed experimental analysis of the coordinated memory management mechanism and a comparison with other relevant mechanisms. In chapter 5 5, we analyze the effect of scheduling on the operation of the coordinated memory management scheme and also focus on doing a detailed time analysis of the mechanism to enumerate the time spent during various operational phases of the mechanism. Chapter 6 6 explores avenues for improving the mechanism's operational latency by reducing the scheduler interferences. In chapter 7 7, we provide the design and implementation details of the hypervisor based implementation where much of the centralized logic is inside the hypervisor and communication is done using hypercalls and xen-upcalls. Chapter 8 8 contains the results of our experimental analysis of Intra-VM memory management mechanisms and provides comparative analysis with our devised mechanism for Inter-VM memory management. Chapter 9 9 discusses related work and chapter 10 10 concludes with a brief discussion about possible future work.

CHAPTER II

BACKGROUND AND NEED FOR A NEW MECHANISM

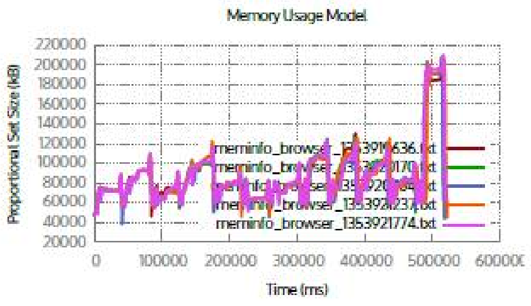
As mentioned earlier, two recent advancements are the primary motivating factors for the research presented in this dissertation: (1) Virtualization is no longer confined to the powerful server class machines. It has already been introduced into smart-phones and will be a part of other high-end embedded systems like automobiles in the near future. (2) More and more memory intensive and latency sensitive applications are being used in devices which are rather resource constrained and introducing virtualization into the software stack just exacerbates the memory (resource) allocation issue.

So there is a need for careful analysis of the needs of these new class of applications and provide appropriate mechanisms in the software stack for their suitable usage. Given that memory resource management in the context of these new class of applications is the focus of our research, we begin by analyzing memory behavior of applications.

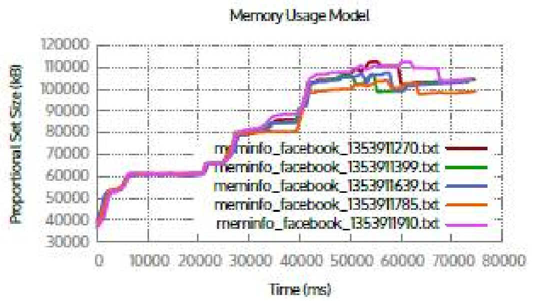
2.1 Analyzing Application Behavior

The primary objective of the discussion in this section is to analyze and understand the memory requirement patterns of various applications prevalent in the present and will be very relevant in the context of end devices in the future.

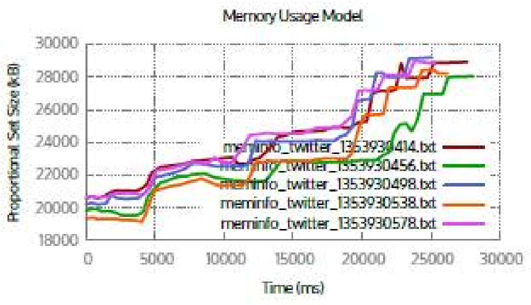
Figure 2 presents the results of experiments conducted on an Android platform, where we are tracking the memory requirements (proportional set size) of some commonly used applications. Browser is used to read news, and access product information web pages, Facebook and Twitter are used for their commonplace uses and finally a H.264 video is played in case of video player. Although, there is no denying of the fact that the actual memory usage is subject to the implementation of the application and their usage, it is clear from the observations that, the memory needs of the applications displays a bursty nature. This is most evident in the case of browser, primarily because of its aggressive garbage collection policy.



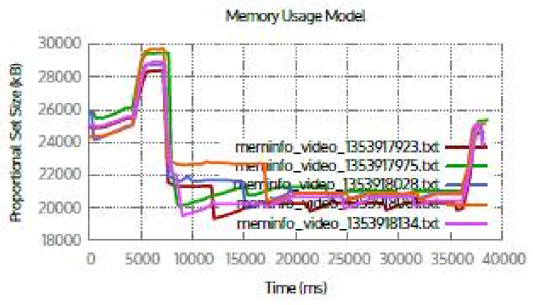
(a) Browser



(b) Facebook



(c) Twitter



(d) Video Playback

Figure 2: Tracking memory needs of various applications

In the next set of experiments, given the relative prevalence and importance of streaming media applications among the current set of applications and its rapidly growing popularity, we focus on applications used for playing streaming media. Figure 3 shows the results of tracking (physical) memory size of a youtube client, while the network bandwidth available is varied using a traffic shaper (ipfw, dummynet). The bandwidth variance invokes the adaptive feature of youtube client, where it changes the bitrate of the video being played according to changing network conditions. This has effect on the buffer size used by the application for storing of data and this gets reflected in the bursty nature of the memory needs of this application.

In order to further explore the memory requirements of streaming media applications in varying conditions, we conducted experiments using video playback clients built in conformance with DASH (Dynamic Adaptive Streaming over HTTP) protocol [45], wherein the client requests video files of various bit rates according to its local network conditions.

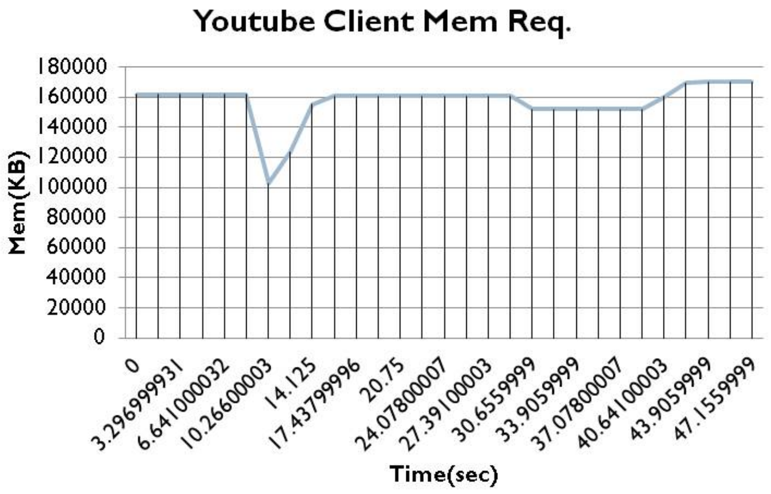


Figure 3: Tracking memory needs of Youtube client

Adaptive HTTP streaming is the evolving paradigm for video streaming on the web and it relies on off-the-shelf web servers, instead of dedicated video servers. A video file is divided into multiple small segment files that have the same play-out duration, typically a few seconds long. Each of the segments from the same video file is also encoded into different bit-rates and quality levels. A XML file called media presentation description (MPD) file describes the video segments and quality levels, and is present along with the segment files in web-servers. A video client can then request segments at different bit-rates depending on its estimation of the state of the local network. As this technology is gradually becoming prevalent and it has implications towards the increasing sophistication of the client devices, it is important to explore its characteristics for our research.

We use the ipfw [19] and dummynet [12, 6] to shape the available bandwidth at the client machine (a 2.4 GHz laptop with latest version of Ubuntu). Apache 2.4 Webserver installed on a 2.4 GHz machine (with Windows XP as the OS) is used for serving up the various bitrate encoded video files as per client demand. The physical memory size of the client (GPAC client) is tracked (at 0.2 sec interval), as we vary the available network bandwidth. Figure 4 and Figure 5 show the results of the experiments. In case of the experiment represented by Figure 4, we observed a max spike of 97 MB per sec and second

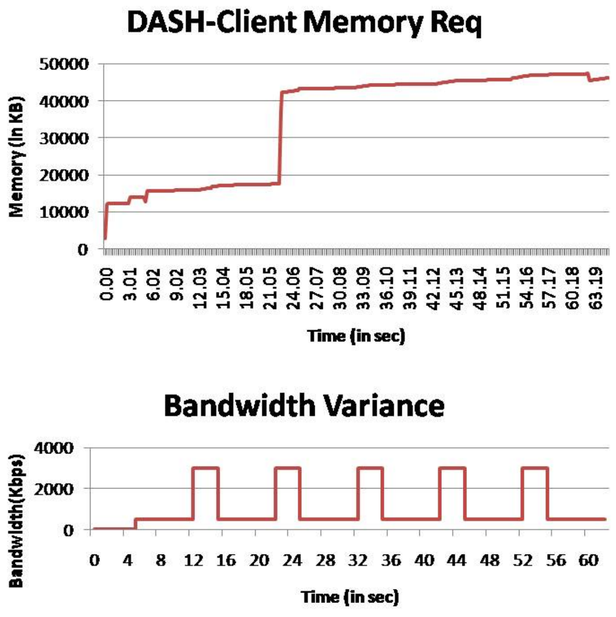


Figure 4: Tracking memory needs of a DASH-client 1

max spike of 26 MB per sec. In Figure 5, the corresponding figures were 112 MB per sec and 31.2 MB per sec.

Apart from exhibiting the apparent bursty nature of the memory requirement of these class of applications, these figures also reveal the importance of having really latency sensitive memory management mechanisms in the virtualized setups which will be catering to these kind of applications. So, with the gradual change in the application landscape of the virtualized environments, there is a need to rethink the design and implementation of some of the core mechanisms.

It is noteworthy to mention that, although the work presented in this dissertation draws its motivation from the low latency needs of the applications prevalent in end-devices, the principles and techniques discussed in this context are equally applicable to server class setups having need for low latency memory management mechanisms. Network emulation frameworks [41] are one such class of applications.

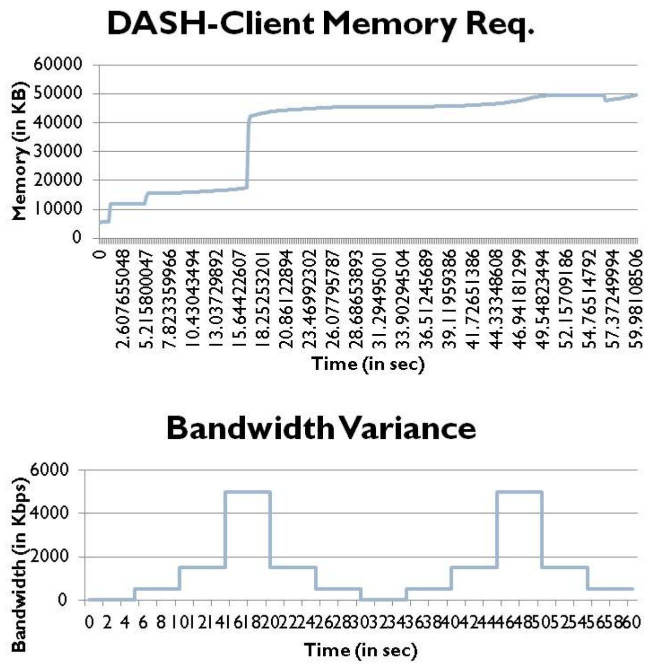


Figure 5: Tracking memory needs of a DASH-client 2

2.2 Existing Mechanisms

The focus of the research presented in this dissertation is on balancing and dynamic partitioning of physical memory across a number of VMs running simultaneously within one physical machine. Other researchers have looked into utilizing virtual machine migration techniques [49] to deal with overload in memory, CPU need etc.. But they are more geared towards balancing the overall load across available physical machines and are orthogonal to the problem of catering to occasional spikes in memory pressure experienced by a VM. Mechanisms like page sharing help in reducing the overall memory pressure and hence do not serve the same cause as transferring memory from one VM to another.

Other works in the area of working set size estimation are complementary to page transfer mechanisms. They enable the user/administrator to determine the need for a memory balancing across the running VMs, but need to use one of the page transfer mechanisms in order to effect the balancing. So page transfer mechanisms are at the core of memory balancing in virtual environments and hence it is essential to have a detailed understanding of the pros and cons of the basic page transfer/ memory balancing mechanisms and their suitability for various purposes. Keeping this in consideration, we will look more into the two core page transfer based memory management mechanisms: *ballooning* and *transcendent memory* in the rest of this chapter.

2.2.1 Ballooning

The memory requirements of a VM are not static. There may be occasional spikes in its memory need but once these spikes subside, most of the allocated memory may simply be dormant in the page/buffer cache of the VM. Although there is no denying of the importance of page/buffer cache, there is no guarantee about whether the pages in the page/buffer cache will really be accessed in the future or not. So over-provisioning of memory for VMs for catering to the spikes will lead to sub-optimal memory usage for most of the running time and would result in wastage of the memory resource if the memory allocated to a given VM is fixed. Also, in some of the domains, over-provisioning may not be an option altogether.

When multiple VMs are running in one physical machine simultaneously, they may not experience memory spikes at the same time. This is where resource over-commitment comes to the rescue. Memory over-commitment is a principle by which multiple VMs, whose maximum memory requirements sum up to more than the actual memory, may coexist. One of the underlying principles in many of the over-commitment based memory management approaches is *ballooning*. Every virtual machine, when it starts, is provisioned for a physical memory amount in accordance to its maximum memory requirement. But, at any given time, the amount of physical memory available to a VM should be in accordance to its need. This is achieved with the help of a pseudo driver inside the kernel called 'balloon driver' which has two basic functionalities: *inflate* balloon, *deflate* balloon (balloon symbolizes the amount of memory that has been taken away from the VM and given back to the hypervisor). Inflating of balloon results in memory pages being taken away from the VM and released to the hypervisor. Deflating of balloon results in the hypervisor backing the pseudo pages inside a VM with real physical pages (thus increasing the memory allocation of a VM).

The existing mechanism in the setup for achieving ballooning is called self-ballooning. Self ballooning [35] is a mechanism in which a VM makes its own decision about whether ballooning needs to be activated and whether to inflate or deflate the balloon. It monitors the memory usage of the VM and periodically tries to determine the working set size (WSS) of the VM (either on its own or finds out the value of WSS from some other source). Based on the working set size estimation and the amount of memory that is allocated to the VM, it determines whether to release some pages back to the hypervisor (inflation) or attempt to get some pages from the hypervisor (deflation). All the decision are taken by the balloon-driver without any consultation (/direction) from the hypervisor or other VMs. With self ballooning, Xen may over-commit memory resources amongst VMs and every VM makes sure that it uses only the amount of memory that it actually requires.

In a slightly different form, the ballooning principle is implemented as a hypervisor-driven mechanism in ESX server (and its descendants) [48, 46], where the hypervisor periodically assesses the free memory available in the system and also periodically receives memory status information from the VMs. If the available free memory goes below a certain predefined level, the ballooning mechanisms in the individual VMs is activated and the VMs release the memory pages during their next iteration cycle.

2.2.2 Transcendent-Memory

Transcendent memory (T-Mem) [33, 34] was developed primarily to provide an alternate memory management mechanism to that of ballooning and was designed to improve upon the shortcomings of ballooning. The primary issues with ballooning that the authors of transcendent memory tried to address are:

- difficulty associated with satisfying urgent memory needs of the virtual machines due to memory inertia
- difficulty associated with the correct prediction of the future memory needs of the virtual machines and thus determination of an optimum rate of ballooning

The authors argue that, an effective solution would be to use the various unutilized and under-utilized memory pages (fallow memory) in such a way that it could be provided back to the needy VMs for usage as and when necessary. They strive to achieve this by reclaiming the idle memory not only from the hypervisor but also from the guests to provide an aggregated pool of unused memory. This pool of memory could then be put under the supervision of the hypervisor and Dom-0 and could be provided for usage by the domains as needed. In the current form, transcendent memory implementation provides a page copy based interface for getting and putting memory contents into this pool, through ‘get page’ and ‘put page’ operations. From the perspective of a guest virtual machine, transcendent memory is a fast pseudo-RAM of indeterminate and varying size that is useful primarily when real RAM is in short supply and is accessible only via a page copy based interface.

The transcendent-memory in its current state provides interfaces for creating four kinds of pools:

- Private persistent pools: A persistent memory pool is one, which has the characteristic that pages put into this pool remain in the pool as long as the virtual machine that put the page there remains active. A private pool is meant for storing pages from only one virtual machine.

- Shared persistent pools: A shared persistent pool is a pool that is shared between more than one virtual machines.

- Private ephemeral pool: An ephemeral pool has the characteristic that pages put into this pool might not exist in the future. So it is the guest VM’s responsibility to put only the pages, whose content loss would not affect the VM.

- Shared ephemeral pool: An ephemeral pool shared between more than one virtual machines.

Using the aforementioned constructs, the authors of transcendent memory have introduced two mechanisms into the guest domain kernels called ‘Front-swap’ and ‘Clean-cache’.

Front-swap is built as a private persistent pool for storing dirty anonymous pages, which otherwise would have been written to swap area. It behaves as a fast swap disk. Whenever a page needs to be swapped out, Linux kernel swap subsystem first contacts front-swap rather than going to the disk directly. If front-swap has space for storing the contents of the page, disk access is avoided. Clean-cache works as a victim cache for pages evicted from page cache. Due to its ephemeral nature, it is meant only for putting and getting clean pages.

Apart from the above constructs, transcendent memory implementation provides for additional mechanisms to ensure the proper usage of these pools. To ensure that front-swap does not hold pages which might not be used for a long time, an additional mechanism called ‘front-swap self-shrinking’ is used to evict pages from front-swap and return them back to kernel memory. This ensures that front-swap has free memory for future memory needs of the guests. There is also provision for compressing the contents of the pages while storing in the transcendent memory pools. We are more interested in evaluating the core transcendent memory mechanism and its usability for effective memory management and hence will not be discussing further about these additional mechanisms.

2.2.3 Drawbacks of existing mechanisms

2.2.3.1 Self-Ballooning

A primary drawback of the self ballooning approach is that there is no coordinated effort from the VMs to resolve a spike in memory requirement of one of the VMs. It is possible that some other VM might be having spare memory pages, but the needy VM has no way to communicate its need to the other VM. As a result of this, there is a possibility that, many times a VM’s request for additional memory may remain unfulfilled by the hypervisor due to lack of memory or, at the least, not fulfilled at the moment it is needed most. For self ballooning to work in a resource conscious manner, the driver has to aggressively monitor the memory need of the VM it is in and this could result in unnecessary overhead. Furthermore, due to the uncertainty of future memory needs, aggressive “inflation” by the driver could result in additional unnecessary page transfers. While these problems may not be apparent in resource-rich server class machines that cater to longer running predictable workloads endowed with resource over-provisioning, the same is not true for embedded domains.

The hypervisor driven approach for ballooning in ESX server is very specifically geared towards the data center setups, where the primary goal is to keep free memory available with the hypervisor. It is important to note that this approach is not best suited for memory constrained environments, where it may not be appropriate to keep memory lying fallow with the hypervisor, rather than allocating them to the VMs that could benefit from the additional memory. Other than that, although not designed to be used in this manner, the setups where the policy is for the hypervisor to keep very little free memory with it, this approach has the similar drawbacks of the self ballooning approach.

These problems could be solved by taking an on-demand approach to ballooning and providing a channel for making collective effort at memory balancing.

2.2.3.2 Transcendent Memory

Transcendent memory mechanism has its strengths and weaknesses. The primary drawback with T-Mem's approach for memory management and handling memory need spikes is that of the indirect page copy based interface. This approach has inherent latency overheads, as we will show later by experimental results. Other than that, in this approach, the kernel in the virtual machine does not get full access to the pages. It has to discard some existing pages in order to free up the memory and memory management unit of the VM cannot do with the pages in these memory pool as it sees fit.

Also, another key ingredient in the operation of transcendent memory based mechanisms is the pooling together of memory pages to create a central pool. How to get the pages for this pool becomes a key decision point. At present, the pages for the free pool are acquired either from the unallocated memory at the disposal of the hypervisor or by doing aggressive ballooning inside the guest domains. Using the unallocated memory inside the hypervisor is fine, except for the fact that, in a real working system hypervisor may not have too much free memory lying unallocated. Also, if there is free memory with the hypervisor, then a virtual machine in need of more memory could just get it from the hypervisor. On the other hand, using ballooning (inside the virtual machines) for getting memory pages for the central pool raises questions about the rate and aggressiveness of the ballooning inside the individual domains. This has the same issue of virtual machines not having an estimate of their future memory needs.

2.3 Concluding Discussion

From the discussion in this chapter, it is clear that existing memory management mechanisms may not be best suited to cater to the requirements of current applications, especially the ones exhibiting bursty memory requirement behavior. So to fill this gap, in the subsequent chapters, we will be presenting about our approach towards creating a coordinated memory management mechanism (CMM) for resource constrained devices. Low operational latency being one of our primary goals, we will also present our efforts at bounding the latency of our mechanism.

CHAPTER III

COORDINATED MEMORY MANAGEMENT MECHANISM (CMM)

Having motivated the need for a better memory management mechanism in virtualized setups for resource constrained devices, we now consider how to approach this in detail. We started with three basic design goals for this mechanism:

- On-demand
- Coordinated
- Low operational latency

Our goal is to invent a mechanism that operates in an *on demand* manner as opposed to continuous balancing, since it is difficult and complex to estimate the future memory needs of a VM. Coordinated approach allows multiple VMs to act in a cooperative manner to resolve the memory pressure situation of another VM. Low operational latency is a requirement considering the characteristics of the applications whose needs the mechanism is designed to cater to. Also, taking the on-demand approach allows us to provide scheduling priority for the operation of the mechanism as will be described later.

We have used Xen based para-virtualized setup for developing and analyzing CMM, as it provides us with the advantage of being open source and also the para-virtualized approach gives us the flexibility of modifying the various components of the software stack as and when necessary.

3.1 Baseline implementation of CMM

3.1.1 Implementation choices

Keeping in mind our design goals, we devised a simple flow of operations for achieving coordinated memory management. In our mechanism, when a domain experiences memory pressure and needs more memory, it communicates with a central entity (daemon /server process) which coordinates the process of resolving the memory pressure. The key functions of this central entity include:

(a) collecting necessary information from other VMs (available memory , memory need/ wss estimation).

(b) making decision about which domains could free some memory.

(c) informing the VMs to free some memory pages and release to the hypervisor

(d) informing the original domain to get the memory pages from the hypervisor.

While the overall mechanism is simple and straight forward, the inherent distributed nature and the need for low latency raises some interesting design and implementation choices. One of the decision points is the location of the central coordinating entity. Choices for implementing this include:

- using a central entity inside the hypervisor.
- using a central entity inside the control domain (Dom 0) and designing the core driver of the mechanism according to the split driver principle (following the example of other drivers).
- using central entity inside the control domain (Dom 0) as a daemon in user mode and communicating with corresponding user-space daemons in other VMs using network-protocol.

We ruled out the third choice for a couple of important reasons: (a) the daemons live in the user space of the VMs and hence would not be well integrated with the other related components living in the kernel space; and (b) the ballooning will be heavily dependent on the network setup in the VMs, which may turn out to be a stringent restriction for the scheme to work in a latency-sensitive manner. The first choice of putting a centralized coordinator inside the hypervisor, would increase the code density of the hypervisor core, which goes against the viewpoint of keeping the hypervisor lean and thin. For this reason, and considering the general convention where most device drivers in the para-virtualized world are designed and implemented according to a split driver model, we decided to opt for the second approach to begin with. However, our experience with the split driver approach and the performance results we obtained convinced us that an “inside the hypervisor” approach may yield better dividends in shaving the latency and we validate this hypothesis later in this dissertation 7.

3.1.2 Split Driver Model

The split driver model was originally formulated to be able to reuse the existing drivers for the wide range of hardware in commodity PCs. Figure 6 provides a depiction of primary architectural components of a split driver based design. In this model the device drivers are split across two domains, a portion in a privileged domain handling the physical device (the back-end) and the other part (front-end) in the unprivileged domain acting as a proxy upon the back-end. The back-end and front-end communicate using shared event channels (light weight event delivery mechanism used for sending asynchronous notifications to a domain) and ring buffers, in an architecture known as the xenbus. Xenbus provides a bus abstraction for the domains to communicate with each other in para-virtualized environment. The basic architecture can be considered to be a chain of four components (see Figure 7), a pair of devices communicating across the xenbus, and then a pair of devices using that bus, handling the device-specific interaction with the kernel device layer.

In order to establish communication across this chain, a number of parameters need to be passed from privileged to unprivileged domain, and vice versa. All of these parameters are passed using the xenstore (a centralized configuration database). Xenstore is a hierarchical information storage space shared between domains. It is meant for configuration and status information exchange rather than for large data transfers. Each domain gets its own path in the store. When values are changed in the store, the corresponding drivers are notified.

3.1.3 CMM Components

As described earlier, we use the split driver model for our baseline implementation of CMM. Following this design principle, there is a front-end CMM driver inside the kernel of each virtual machine. It is connected with the balloon driver and uses the xenbus /xenstore

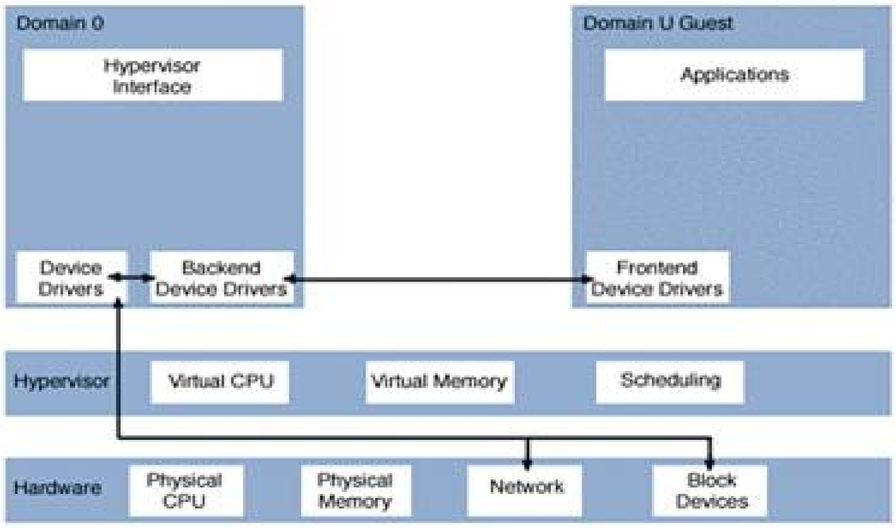


Figure 6: An example of a split driver architecture in Xen based para-virtualized setup

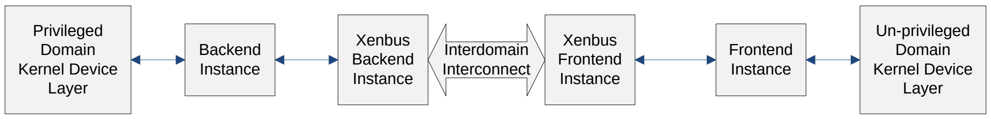


Figure 7: Components of a driver built according to split driver principle in Xen based para-virtualized setup

mechanisms to communicate with the CMM back-end driver. The back-end driver resides in the privileged domain (Domain 0) and is responsible for coordinating in the process of resolving the need for more memory of a VM and for providing a channel for communicating the memory need to other VMs active in the physical machine. The overall system structure for the implementation of the split driver model is illustrated in Figure 8).

3.1.4 Implementation

All the front-ends and back-ends have their directory (containing configuration parameters) in xenstore. To trigger the creation of a device connection, xend writes front-end and back-end entries to the corresponding directories in xenstore. This happens when a new domain is getting created and initialized. These new details are seen by the xenbus driver instances, and initialization begins. The xenbus back-end instance has a

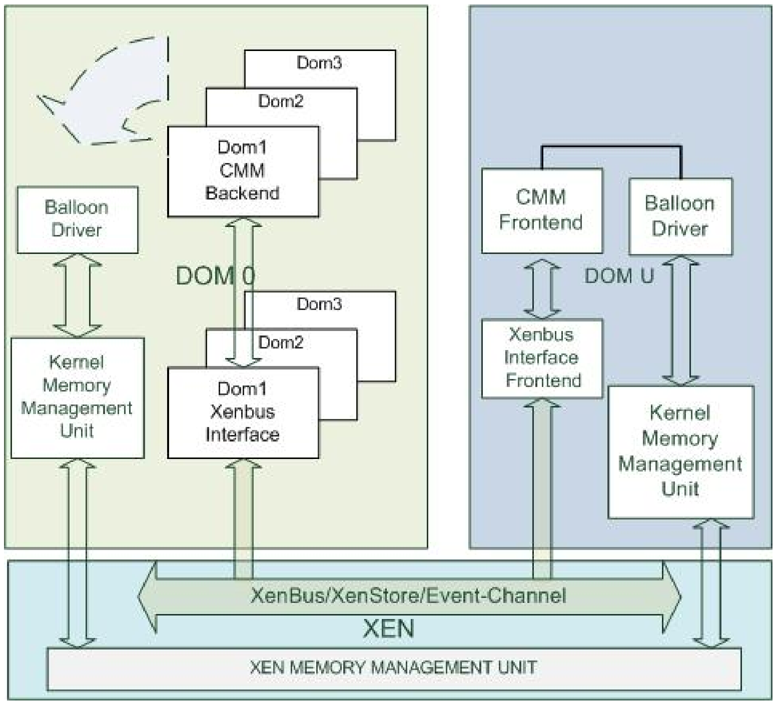


Figure 8: Software Components of the CMM Implementation in Xen using the split driver approach

watch on '/local/domain/0/backend/dirbal' and the front-end instance has a watch on '/local/domain/U/device/dirbal'. When the initializing details are written to these directories, the watches fire, and the xenbus instances begin negotiation. They go through the usual xenbus handshake mechanism of going through various state before reaching the final 'connected' state.

After the initialization phase, the drivers communicate by writing the necessary information into xenstore directories, so that the other end could get them. The entries are of two types: data entries and command entries. Data entries are meant for data exchange like the current memory, current working set size etc. The command entries are primarily boolean entries whose main function is to convey to the other end to start some activity. Typically command entries are like 'status-response', 'start-action' etc. The notification (about new entries being made, change etc) to the other end is sent by using event-channel. As there is no need for real data transfer between front-end and back-end, the ring buffer and page grant mechanisms are not used in this driver.

One aspect that is different in the design of split driver model of CMM from that of other split drivers (disk, network) is that the back-ends need to communicate and coordinate with each other. This is achieved by maintaining a linked-list of data structures meant for back-end data and operation pointers. This is a circular linked list implemented using the Linux kernel linked list construct. In this way each back-end can communicate with other back-ends and also collect necessary information by traversing the linked list.

3.1.5 Operation

Figure 9 is a depiction of the various steps involved in the operation of CMM. The front-end is responsible for triggering the sequence of operations of the CMM. The sequence of actions will be triggered when the front-end senses the need for more memory. The specific condition that triggers the sequence of actions is specific to each VM and could be based on working set size estimations and/or available memory going below a threshold. The front-end driver writes the necessary information into the xenstore entries of the front-end and sends a virtual interrupt to the back-end using event-channel.

On receiving the virtual interrupt from a front-end, the back-end (henceforth referred as *master back-end*) starts the sequence of activities associated with CMM. The first step is to collect information about the memory usage statistics of the other domains that are running as guests on this machine. This is accomplished by turning on the command entry for 'status-report' in the xenstore for the other back-ends and sending a virtual interrupt to the corresponding front-ends. Each front-end, on getting the interrupt, collects the necessary information (current memory, working set size estimation, etc.) and writes it into the xenstore entries of the front-ends and notifies its associated back-end. In our current implementation, the front-ends gather information similar to what is available in /proc/meminfo.

Each back-end, upon getting the notification interrupt from its associated front-end, reads the necessary data entries from the xenstore and populates the in-memory data structures for perusal by the master backend. Once all the in-memory data structures have been thusly populated by the peer VMs (via their respective split driver backends), the master backend gets to work. It makes decisions as to which peer VMs could be asked to free up memory without adversely affecting their performance. By following a similar handshake as just detailed, the master back-end then communicates to the various front-ends in the peer VMs, instructing them to release the required amount of memory. Thereafter, the front-end in each VM, works with its balloon driver to release the required number of pages to the hypervisor. Once the pages have been released by the balloon driver, the front-end notifies its associated back-end. Each back-end updates its in-memory data structure recording the results communicated by its front-end. Once all the relevant back-ends cooperating for this sequence of actions have received notifications from their respective front-ends, the master back-end aggregates all the results and notifies its front-end in the source VM that is currently experiencing the memory pressure. The front-end invokes the balloon driver to deflate to acquire the necessary memory pages from the hypervisor, thus increasing the available memory for the VM experiencing the memory pressure.

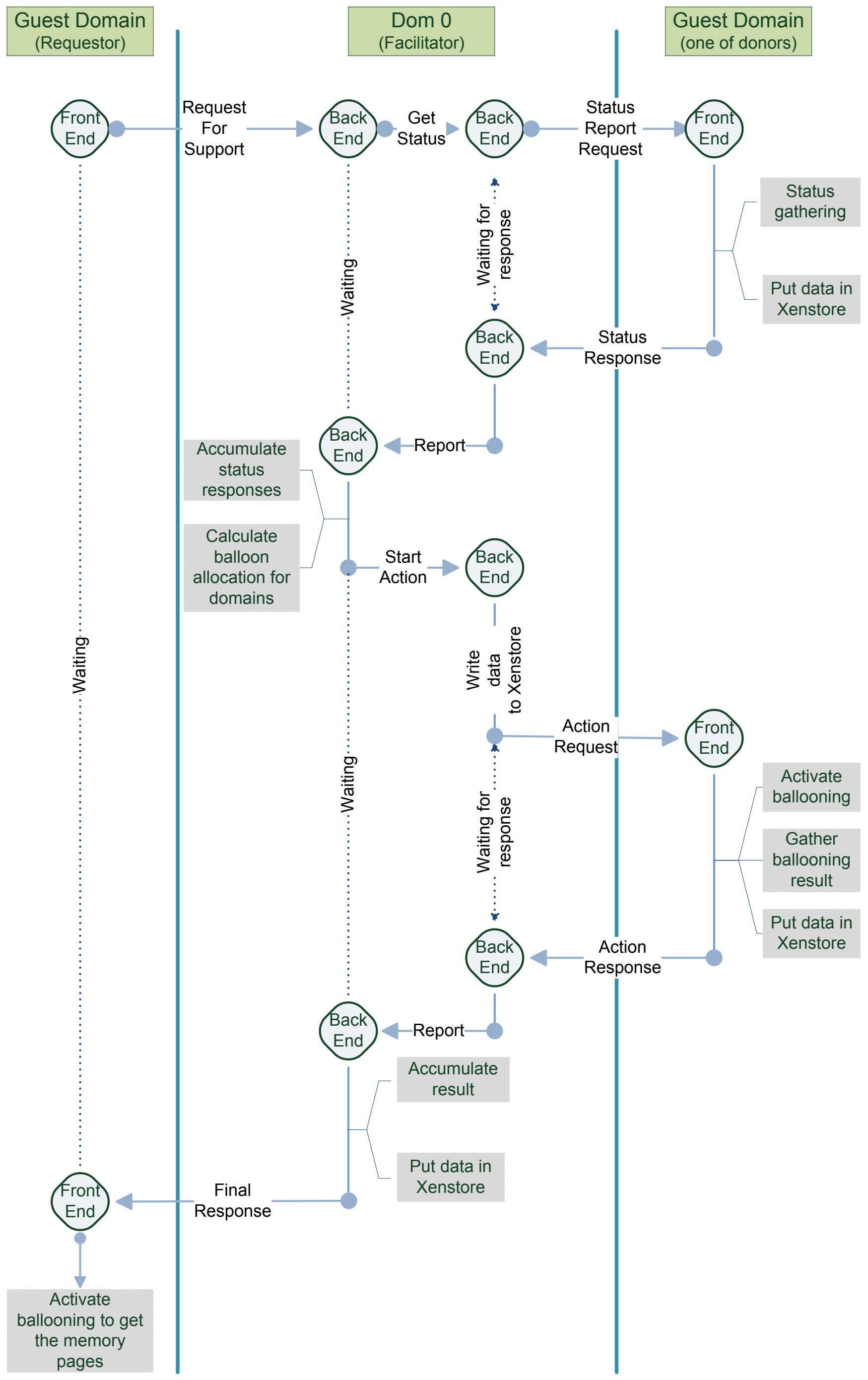


Figure 9: Detailed step diagram for an end-to-end operation of CMM

CHAPTER IV

MEASUREMENT OF BASELINE CMM

4.1 Experimental Setup

The experimental platform is a 2 GHz dual core AMD64 machine with 4 GB RAM. We use Xen-4.1.0 as the hypervisor with pvops based Linux 2.6.32.26 kernel in Dom0 and 3.1-rc7 Linux kernel inside the guest domains. All our experiments have been performed with minimal interference from other processes inside the virtual machines. We do not create any new processes or daemons in any of the virtual machines apart from the default ones. The same machine configuration is used for other experiments/evaluations, unless mentioned otherwise.

4.2 Operational Latency

First we measure the end-to-end operational latency of CMM for a needy VM to acquire the required memory from peer VMs. Our primary goal in measuring the latency is to get an the amount of time involved in an end-to-end operation of CMM for varied amounts of memory and varied number of running VMs. In this experiment, we vary the amount of needed memory and the number of active VMs as control parameters. We use two variants of the experiment: (1) *Equi distribution*: In this variant, we assign equal amount of memory to be reclaimed from each of the peer domains; (2) *Skew distribution*: In this variant, we use as few peer VMs (which is 1 for our controlled experiments) as possible to reclaim the necessary amount of memory. The results of our experiments are represented in Figure 10 and Figure 11. We vary the required memory amount from 100MB - 400MB for a fixed number of 5 running guest VMs in Figure 10. We vary the number of VMs from 2 to 5 for a fixed amount of 300 MB of memory in Figure 11.

As seen from Figure 10, for low memory amount, using skew distribution might have an advantage, as this removes the overhead of more number of VMs having to release the pages. But for higher amounts of initial memory request, the difference is negligible, because per

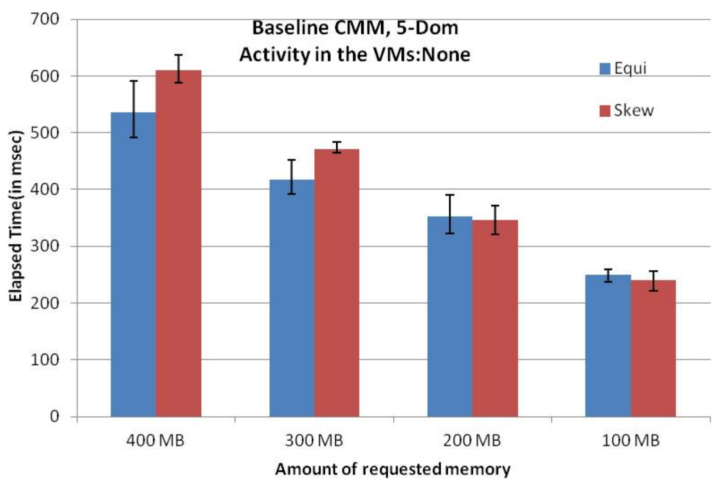


Figure 10: End-to-end latency for CMM: There are no active processes in any of the domains (VMs). Experiment varies the amount of memory for a fixed number of 5 VMs.

domain pressure to release memory pages is more in the skew distribution. Not only does it take more time to release more memory pages for any particular domain, also this process might have interference from the virtual machine scheduling (as we will be discussing in the next section). With the memory amount remaining fixed, latency increases with increase in the number of domains (in both equi and skew distribution). So it is hard to draw any inference and argue in favor of one form of distribution against the other, as the optimum distribution for any memory amount for a particular number of running virtual machines could only be obtained by more case specific measurements. But as a general rule, it could be concluded that if the number of running virtual machines is more than 3 and the amount of memory requested is of the higher order (more than 300 MB), then it is more effective to use a more equitable distribution rather than distributing the load among less number of VMs.

4.3 Comparison with Self Ballooning

Next, we compare the latency of the CMM with memory balancing using the self ballooning approach. In self ballooning approach, a daemon thread runs periodically inside each VM and depending on the memory pressure, either releases memory pages or asks for more

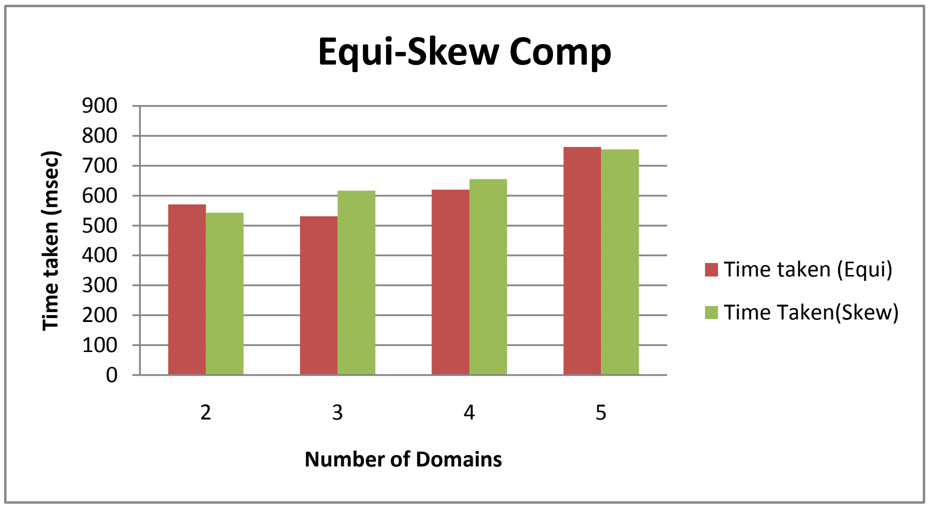


Figure 11: End-to-end latency for CMM: There are no active processes in any of the domains (VMs). Experiment varies the number of VMs for a fixed amount of 300MB of memory request .

memory pages from the hypervisor. One important tunable parameter associated with self-ballooning is *hysteresis*. The basic idea is a VM may not necessarily release all its fallow memory every time the balloon driver is activated. This is to safeguard against potential increase in the working set size of the VM. A hysteresis parameter of k results in 1/k amount of surplus memory being released by a VM during any ballooning period. Also, in order to make a fair comparison with CMM, we implemented a mechanism inside the hypervisor which controls the rate of release of memory pages to a requesting VM to limit it to only the pages released by other donor domains (from the start of the experiment). Figure 12 presents the results of our experiment for ballooning amounts of 400MB and 200MB, where we used a hysteresis parameter of 3 and the experimental setup uses the skew distribution. We present the results for self ballooning periods of 250 msec and 500 msec, both of which are rather aggressive assumptions (ESX server uses periods of 1 sec and the default set up in Xen hypervisor is 5 sec period). As can be seen from the results, latency for CMM is significantly better than that of self-ballooning by up to 4 times even in conservative settings. Equi distribution also gives comparable performance advantage. One important point to note is that it is possible to devise a setup where self ballooning latency is comparable to that of CMM by choosing aggressive periodicity of self-ballooning and hysteresis parameter setting; but such a setting would result in high ballooning overhead for normal no memory pressure situation.

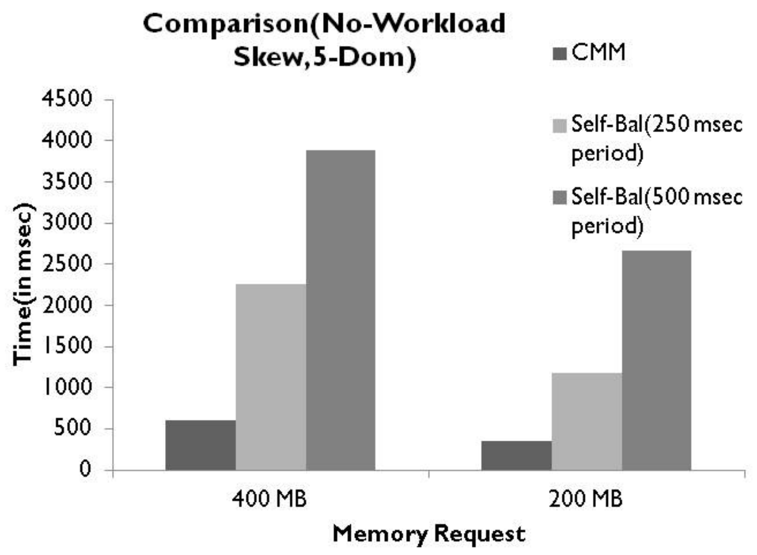


Figure 12: CMM compared with Self Ballooning

4.4 Comparison with Transcendent-Memory

4.4.1 Evaluation Criterion

Transcendent Memory (T-Mem) has some fundamental difference in its design purpose and operation mechanism from that of CMM. CMM is based on the ballooning constructs and provides a way to transfer memory pages from one domain to another in an on-demand and coordinated manner. The transferred pages become a part of the VM and could be used by the OS (running in the VM) in any manner it decides. Transcendent memory operates with the principle of collecting memory from all the domains a priori and creating a centralized pool. The domains get access to these pool(s) through page copy based mechanism. Due to the inherent difference between the two mechanisms, we use the following evaluation criterion.

In case of *transcendent memory*, the primary overhead is the page copy based mechanism to get content from and put content into the pools. Our goal in analyzing transcendent memory is to find out its suitability for being used as a mechanism to mitigate temporary memory pressure in a virtual machine. There could be two ways to deal with a low memory situation in a virtualized setup: get more memory from hypervisor (and other virtual machines) or free up some memory pages so that they could be allocated to processes demanding more memory. The structure of the transcendent memory mechanisms lends itself more suitable to be used in the latter manner.

One possible usage model is to plug-in the transcendent memory mechanism with the page frame reclamation mechanism of the kernel. At times, when the available free memory is low, and there is need for additional memory allocation (to some process), the page frame reclamation algorithm gets activated and tries to free up some page frames. This happens by writing the contents of the dirty page frames (of page cache) to disk files, of anonymous pages to swap area or simply flushing the contents of the page frame in case of clean pages. By plugging in the transcendent memory interfaces, we could save on the number of pages being written out to disk (file or swap area) and this will reduce the latency involved. Also by writing the contents of the flushed out clean pages to the memory pools we could save on potential disk access in the future, in case there is a need to bring the pages back to memory. For the purpose of making a fair comparison with our mechanism, we have focused our evaluation of transcendent memory to this aspect of using it in conjunction with page frame reclamation mechanism.

4.4.2 Performance Analysis

Based on the discussion in earlier section, we evaluate transcendent memory according to the following aspects:

- We measure the overhead involved in using the page copy based interfaces. As all the operation involving transcendent memory involves getting and putting pages into the pools, having a sense of how much latency is involved in these operations is important.

- We evaluate the performance advantage that could be obtained by plugging in transcendent memory exported interfaces with the page frame reclamation mechanism of the Linux kernel. Fortunately transcendent memory implementation in Xen and Linux kernel already provides support for clean-cache and front-swap which are easily integrated with the page frame reclamation setup.

Other plausible sources of overheads are the overheads associated with creation and maintenance of the pools. Pool creation is a one time process and pool maintenance overheads are not relevant to our evaluation criterion. So we do not discuss about them in this dissertation.

For getting an idea of the latency involved in the page copy based operations, we create an ephemeral pool solely for the purpose of this experiment and use it to put contents from the memory pages of a guest virtual machine. The amount of memory written into the ephemeral pool is varied betwwen 100MB - 400MB. Figure 13 contains the results of our experiment. To put this result in perspective, we also measure the latency involved in getting similar amounts of memory (from the hypervisor) through the ballooning phase of CMM. The latency involved in copying of page contents into transcendent memory pools does not include the time taken in creating and populating the pool with page frames. So for a fair comparison, the ballooning phase in this case is just about getting the pages from the hypervisor, and thus gives us a clear picture about the extra amount of overhead involved in page copy based memory management.

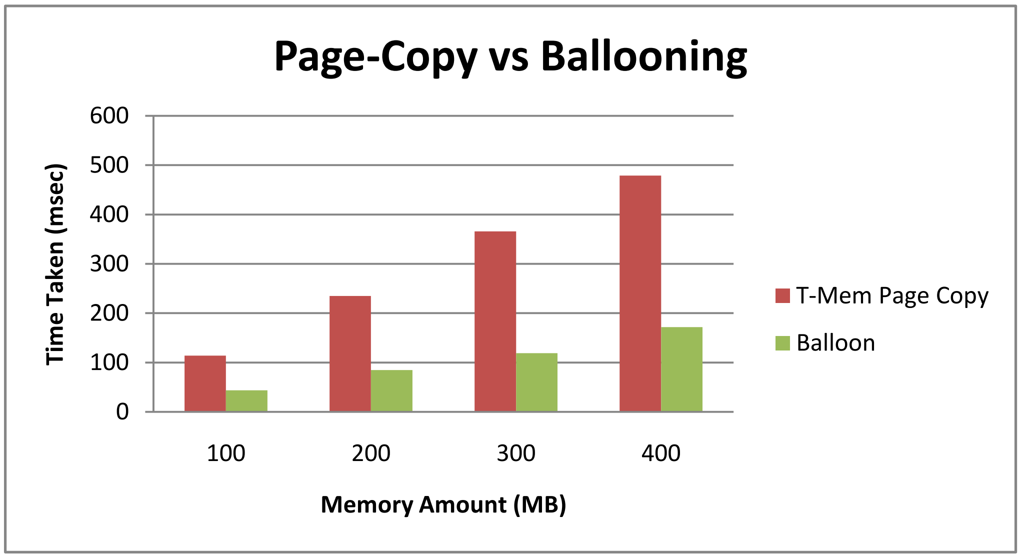
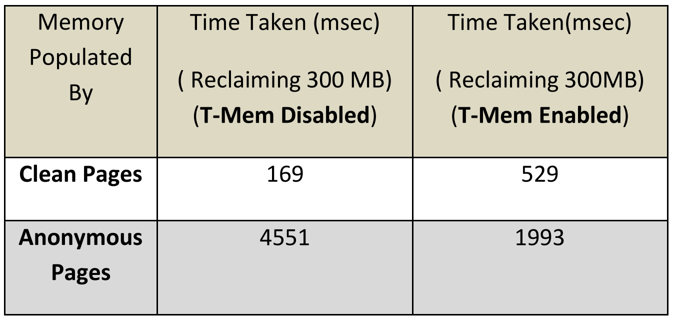


Figure 13: Experimental results showing the overhead of page copy based interface in T-Mem

In order to study the interaction between transcendent memory and the page frame reclamation mechanism of the Linux kernel, we conducted experiments in four different scenarios. Two of them correspond to the situation, where the memory of the guest virtual machine is filled with clean pages (by reading from two large files). In both these scenarios, we try to activate the page frame reclamation (for 300MB worth of pages), one with transcendent memory enabled and the other with transcendent memory mechanisms disabled. A similar set of experiments is conducted in situation where the virtual machine memory is mostly filled with anonymous pages (by creating a daemon process which does a big *malloc*, accesses the memory and holds on to it). The reason for choosing these two types of pages is that, the evicted clean pages go to clean-cache and the evicted anonymous pages could go to front-swap, respectively. We conduct the experiments with only one guest VM running, and the measurements are taken when most of the evicted pages are absorbed by transcendent memory constructs (clean-cache or front-swap). The results of these experiments are presented in Figure 14. As could be seen, in case of clean pages, use of transcendent memory actually increases the time involved. This is because, the evicted pages are put into the clean-cache, which does not happen in the normal case. In case of anonymous pages, there is a significant latency reduction by using the front-swap mechanism (due to reduction in swap activity). But if we compare this result to that of CMM, CMM could provide us with 300 MB extra memory in around 500 msec. So we really are not getting benefited by using transcendent memory.



Memory Populated By	Time Taken (msec) (Reclaiming 300 MB) **(T-Mem Disabled)**	Time Taken(msec) (Reclaiming 300MB) **(T-Mem Enabled)**
Clean Pages	169	529
Anonymous Pages	4551	1993

Figure 14: Experimental results showing the latency of page frame reclamation when operated with T-Mem enabled.

4.5 Conclusion

In this chapter, we presented a preliminary evaluation of the coordinated memory management mechanism and compared it with self-ballooning and transcendent memory mechanisms for memory balancing in virtualized setups. Our working model and experimental analysis showcases the efficacy of CMM and has established the necessity for its introduction, along side the existing core ballooning mechanism, into the Xen setup. Our analysis shows that, latency involved in an end-to-end execution of CMM compares favorably with that of transcendent memory. Although transcendent memory has been marketed as a improvement on the shortcomings of ballooning, our analysis shows that it does not necessarily provide any significant performance/ usage advantage over ballooning. Rather, the unusual page copy based interfaces do not make it very appealing for being used as a mechanism for fast and dynamic memory management.

CHAPTER V

IDENTIFYING AVENUES FOR LATENCY IMPROVEMENT IN BASELINE CMM

Having established the relative merits of the CMM over other related mechanisms, we explore avenues for further improvements towards the end goal of reducing the operational latency. In this chapter, we will present our results and subsequent analysis towards identifying scopes for improvement of latency of CMM. We will be primarily focusing on two aspects:

- Interference from scheduling policies and activities (both in the Linux kernel and Xen hypervisor) with the sequence of activities of CMM
- Analysis of the time spent in various phases of CMM

5.1 Analyzing Scheduler Interference

In this section, we will be discussing about how the default scheduling policy in the hypervisor and OS kernel interfere in the operation of CMM.

5.1.1 Kernel Scheduler Interference

CMM uses the *work queue* subsystem of the Linux kernel to create work-items. In its most basic form, the work-queue subsystem is an interface for creating kernel threads to handle work that are queued from various other subsystems in the Linux kernel. The kernel threads, called *worker threads* carry out the necessary actions of the work-items when scheduled on the processor. The work-queue subsystem is one of the three mechanisms available in the Linux kernel for implementing *bottom half* handlers [1]. The other two mechanisms, *Softirq*,

[1]The primary responsibility of bottom halves is to perform any interrupt-related work not performed by the interrupt handlers. As interrupt handlers run with the current interrupt line disabled on all processors, and it is very important to minimize the time spent with interrupts disabled, the expectation from interrupt handlers is to be as fast as possible and do the minimum necessary amount of work and delegate the rest to the bottom halves.

and *Tasklet*, are primarily used in the context of handling interrupts, wherein the bottom half handlers execute to completion without the need to block. As the work items related to CMM may be rather long running and have possibility of going to sleep (on occasions where some pages may need to be written to swap before releasing them to the hypervisor), work-queues is the only relevant bottom half mechanism that could be used for this purpose.

Linux kernel, schedules these worker threads commingled with other processes and threads. Thus the Linux scheduling policy plays a role in the latency of CMM (for e.g., if the scheduler gives higher priority to other threads or processes over the work-items enqueued by the CMM). Also, there is a need for appropriate coordination between the VM scheduler in the hypervisor and the process scheduler inside each VM.

5.1.1.1 Evaluation

To quantify the extent of scheduler interference, we repeated the experiment to measure CMM latency shown in Figure 10, by introducing a computationally intensive daemon process in each of the VMs. Figure 15 shows the result of this experiment. Comparing Figures 10 and 15, we see that the default kernel scheduling policy could significantly affect the latency of CMM in the presence of other workloads in the donor VMs that are trying to come to the rescue of the VM that is experiencing the memory pressure. For example, Figure 10 shows an end-to-end latency of 500 ms for 400 MB while the latency jumps to 2500 ms in Figure 15 for the Equi distribution in the presence of workload.

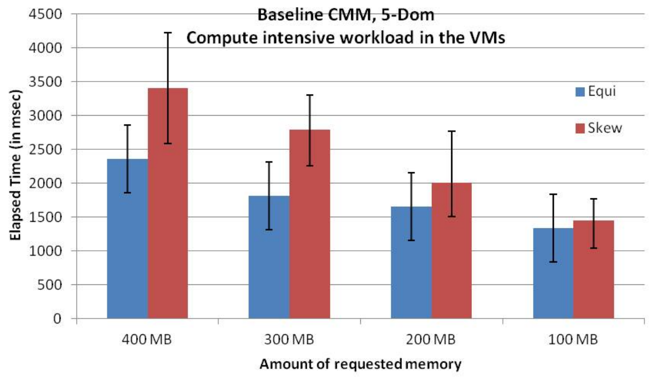


Figure 15: End-to-end Latency of baseline CMM in the presence of workload. Each VM runs a computationally intensive daemon process.

This implies that the interference due to sub-optimal (in the context of CMM) kernel scheduling policy is significant and there is a need for system level constructs to mitigate this interference.

5.1.2 Hypervisor Scheduler Interference

In the preceding section, we considered the interference caused by the scheduler in the Linux kernel. In addition to the kernel scheduler, the scheduler and scheduling policy of the Xen hypervisor also interferes in the operation of CMM.

With the split-driver model of implementing CMM, there is communication between the front-end and back-end portions of the driver in each VM, decision making to be done by the master back-end, and communication between the front-end and the balloon driver in each donor VM. The actual act of releasing of memory pages in the donor VMs and acquiring them in the needy VM through the balloon driver mechanism are on top of these communication and decision making. This inherently distributed nature of the mechanism makes it vulnerable to interference from the hypervisor scheduler. Based on the various inputs from the front-end drivers, the back-end decides on which virtual machines should free-up memory pages. Not all the domains might be in a state to donate memory pages back to the hypervisor and not all might be required to do so. Among the domains that are selected, not all the domains will be scheduled immediately to run. Thus the end-to-end latency in CMM is critically dependent on the scheduling of the VMs by the hypervisor. We refer to this as *hypervisor scheduler interference* and explore a methodology to quantify the same.

5.1.2.1 Measurement Framework

To quantify the extent of hypervisor scheduler interference in the operation of CMM, we implemented a measurement framework. The measurement framework consists of two core constructs: *event logging posterior analysis.* The event logging construct is responsible for defining, capturing and recording events and creates a trace of the various events and their associated time stamp. The posterior analysis construct uses the event logs and does a detailed analysis to calculate the possible interference.

Event logging: The event logging construct provides interface for creating a log of various events and their associated time stamp. The log records are created and maintained in an in-memory buffer created in the hypervisor memory. The buffer is pre-allocated and all the initialization tasks are done prior to the operation. It is ascertained that the buffer is big enough to hold the records for all the relevant events during the duration of the operation, so as to prevent devoting any cpu-cycles towards buffer management related tasks during the CMM operation and thus avoid adding any additional overhead. There are two types of event logs:

- Events related to the operation: The various VMs taking part in the coordinated operation access the event logging interface by making hypercalls at different stages of their involvement in the operation. Each trace-point stores information about the event, originator VM of the event, destination VM of the event (if any), etc. For example, a typical trace-point could store information about Dom-0 sending an action request (for releasing pages) to Dom-3 at time 't'. The time instance 't' is added by the hypervisor.

- Scheduling events: In addition to the above, for the duration of the operation of the mechanism, we also collect a trace of all the scheduling events. A typical scheduler trace-point stores information that looks like: "*in CPU core 1, at time t, the executing virtual machine was switched from Dom-3 to Dom-2*".

For both the type of event logs, we use the cpu cycle counter value as a representation of the time instance, in order to deal with really short time durations, which are common for scheduler events.

posterior analysis: The two types of event logs are collected and stored in (hypervisor) memory in the order of their occurrence (during the execution of the mechanism) for future analysis. Based on the above trace information we could calculate the number of CPU cycles (in both the cores) allocated to a particular VM for various phases of their involvement in the operation of the mechanism. We divide the cycles allocated to the various VMs into two types: *valid* cycles and *invalid* cycles. If at some instant, a VM is not supposed to be doing anything meaningful with regards to the progress of the CMM operation (since all VMs may not be involved in a given CMM operation), but is still allocated CPU cycles by the hypervisor (by getting scheduled in a CPU core), those cycles are counted towards invalid cycles for that VM. Otherwise the allocated cycles are counted towards valid cycles. All the cycles allocated to all the VMs are added up, and we calculate the hypervisor scheduler interference as the ratio of the total invalid cycles to that of the total cycles for one instance of operation of the mechanism. In this calculation, all the CPU cycles allocated to the idle domain are considered as invalid cycles.

5.1.2.2 Evaluation

We measured the interference due to hypervisor scheduler both with and without the presence of active workload (a computation intensive daemon process) in each of the VMs. The results obtained are shown in Figure 16 (without any workload) and 17 (with work load). There are two things that are evident from these figures

- Hypervisor scheduler interference is usually higher in cases of skewed distribution of the total memory request amount, due to the fact that, virtual machines which are not effectively contributing to the operation of CMM are also getting allocated CPU cycles.

- The total scheduler interference in the presence of workload is higher than that of the case where the VMs do not have any active workload in them. This is because, the VMs which are not actively contributing towards the progress of the mechanism's operation, on getting scheduled, are running for longer periods of time.

But the raw figures do not reveal the complete picture. In a para-virtualized setup, Dom-0 is the privileged domain which contains all the device drivers for interacting with the physical devices. Also, there is the concept of Idle-Domain in Xen, which gets scheduled either when no other VM has any work to do, or when Xen has to carry out some work in the *softirq* context. The important observation is that, it is not possible to stop or reduce the number of scheduling of the vcpus pertaining to these two internal activities of Xen, without adversely affecting the overall system operation. To put this in context, in the figures, we have further classified the total interference into the categories of interference from Idle-Dom, interference from Dom-0 and the rest. One key conclusion after this classification is, in the absence of active workload in the system, much of the interference could be attributed to the scheduling of Idle-Dom and Dom-0 and as such the potential for reducing the hypervisor scheduler interference is not much. On the other hand, in the case where the VMs have active workload, much of the scheduler interference could be attributed to the improper scheduling of the Dom-Us. This result clearly shows a scope for significant reduction of interference in this case. As our coordinated memory management mechanism is not meant to operate in vacuum, and will be operating in the presence of other applications, it is imperative to construct mechanisms for countering and reducing the interference from hypervisor scheduler and also enabling appropriate coordination between the VM scheduler in the hypervisor and the process scheduler inside each VM.

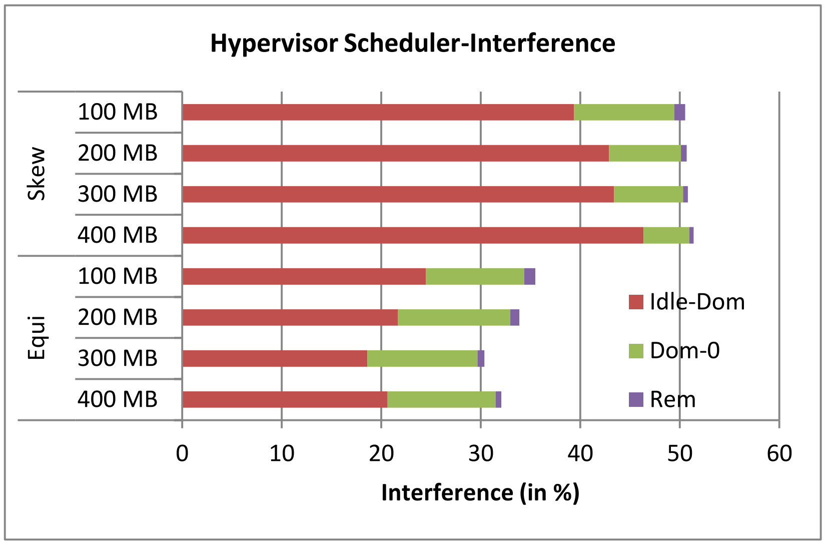


Figure 16: Hypervisor Scheduler Interference (no workload). Most of the interference is attributable to internal activities of the Xen hypervisor that are critical to the overall integrity of the system. There is not much opportunity to reduce CMM latency.

5.2 Component Time Analysis

The primary goal of our discussion in this section is to accurately enumerate the time spent in various phases of operation of CMM and thereby identify avenues for improvement in

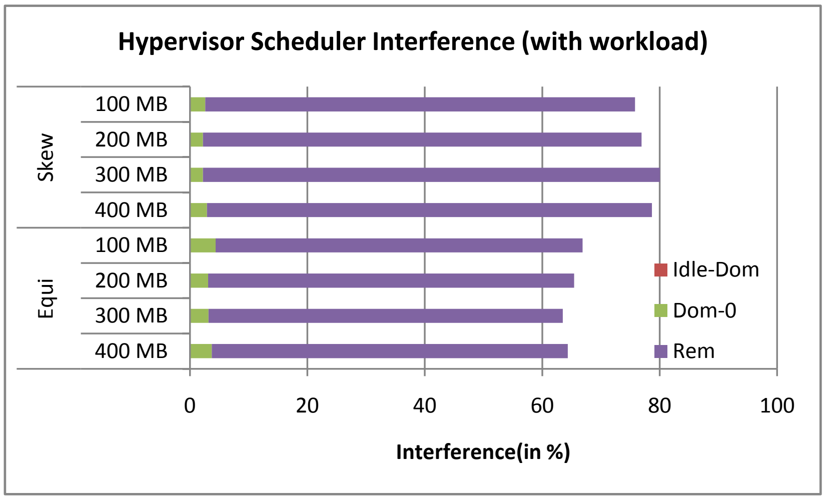


Figure 17: Hypervisor Scheduler Interference (with workload). Most of the interference comes from Xen scheduling VMs that are not in the critical path of CMM latency. This shows a tremendous opportunity to reduce the CMM latency.

the design and implementation towards the end goal of reducing the operational latency.

5.2.1 Methodology

Our measurement methodology for this section is based on the previously described procedure for measuring scheduler interference. We use hypercalls to log the time and event type of various events inside the hypervisor memory. These log records constitute a trace of the entire operation. We later analyze the trace to find the time spent in various stages of the complete operation.

The three types of primary role players in the operation of the mechanism are: (a) starter domain: The VM which is experiencing memory pressure and starts the operation of the mechanism by contacting the central coordinator. (b) controller domain: This is Dom-0, inside which the central coordinating entity operates. (c) helper domains: Other VMs which are running in the physical machine and actively participate in the mechanism.

The following table 1 provides a brief description of the various phases of execution in these entities for which we measure the time and their notation in the measurement tables in the next section:

Table 1: Notations for component time analysis of baseline CMM

Entity	Notation	Description
Starter Domain	Init_analysis	Time spent in doing the initial analysis. Populating the xenstore entries with relevant data
	Sup_req_send	Time spent in sending the support request to the controller domain.
	Wait	Time spent in waiting for the response from the controller domain.
	Sup_resp_rec	Time spent in receiving the response from the controller domain
	Action	Time spent in gathering the results from xenstore and getting the memory from the hypervisor
Controller Domain	Start_analysis	Time spent in doing initial analysis
	Stat_req_send	Time spent in sending status requests. Involves writing into xenstore entries for the various VMs.
	Stat_resp_wait	Time spent in waiting for the status responses from the domains
	Status_analysis	Time spent in analyzing the staus responses
	Action_request_send	Time spent in sending the action requests. Involves writing into xenstore entries for the various VMs.

Table 1 Continued:

	Action_response_wait	Time spent in waiting for the action responses
	Action_analysis	Time spent in analyzing the action responses
Helper Domain	Status_request_rec	Time spent in receiving the status request
	Stat_req_act	Time spent in acting on the status request
	Stat_resp_send	Time spent in sending the status response to the controller domain
	Action_request_rec	Time spent in receiving the action requests
	Action_request_act	Time spent in acting on the action requests
	Action_response_send	Time spent in sending the action responses

5.2.2 Results and Analysis

The results for our analysis are presented in the following figures. Figure 18 and 19 present the results for sample end-to-end operations of CMM in the absence of active workload in any of the VMs, for equi and skew distribution respectively. Figure 20 and 21 present the results for sample end-to-end operations of CMM in the presence of compute intensive workload in the VMs for equi and skew distribution respectively. The total amount of initial memory request is 400MB in all these cases.

There are some key observations that could be inferred from the results presented in the figures. We enlist them below:

- The time spent in activities where there is significant interaction with xenstore interfaces show a high degree of variance. It is especially true in situations where multiple VMs are simultaneously trying to access the xenstore or simultaneously trying to write to xenstore. This is evident in the values for Stat_req_act in the various figures.

- The time spent in activities where there is involvement of writing into the xenstore are usually higher than the activities where there is only need for read access to the xenstore. This is evident in Figure 18 where the Stat_analysis phase for the controller domain takes significantly less time than for Act_req_send phase. Stat_analysis involves reading the entries from xenstore, where as Act_req_send involves writing into xenstore for all the relevant domains.

- When operating in the presence of workload, the time involved in activities which involve interacting with xenstore increase significantly. At some instances these are comparable to that of activities which involve releasing of the pages. This is clearly evident from the values for Act_req_rec and Act_req_act in Figure 20.

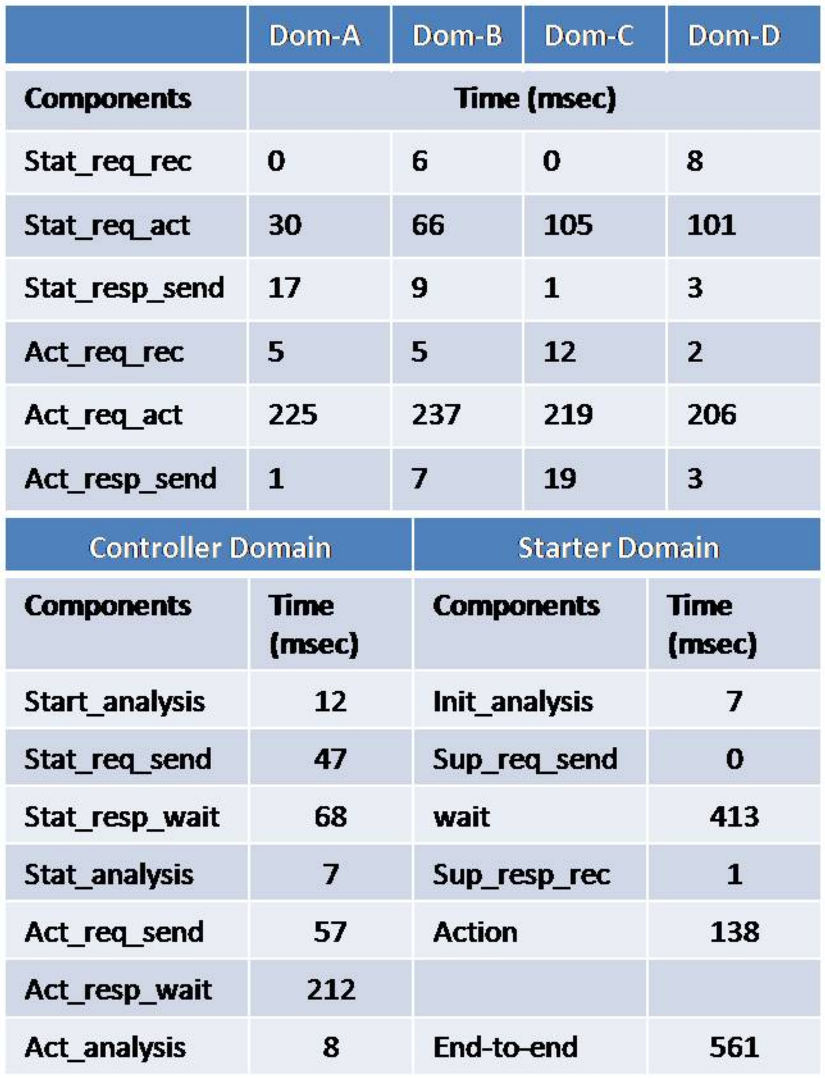
- Lastly, the time spent in coordination and communication is significant, and given the low latency goal of our mechanism there is a need to revisit this aspect.

Given the involvement of so many role players at different layers of the software stack, it is not straight forward to pinpoint the actual causes behind the above observations. However, as all of our observations have xenstore and activities involving xenstore as a common point, we have done further exploration into the overall xenstore architecture. On further analysis, we found that xenstore is implemented as a database built on top of a file which is mapped to the memory of the various VMs. Figure 22 presents an architectural representation about how xenstore fits in the overall Xen based virtualized setup. So it is essentially file backed and a daemon (xend) running in Dom-0 is in charge of maintaining the consistency by writing to the backing file. That is one of the primary reasons for higher latency of activities where multiple VMs are simultaneously trying to access xenstore. Apart from having the disadvantage of dealing with a central daemon, appropriate scheduling of Dom-0 and the xend daemon also become critical and this is one of the reasons for high variance in latency of some of the activities as we noted previously. Also, xenstore exports a transaction-oriented write interface, and this causes a fraction of write operations to be eventually repeated more than once, so as to deal with situations where the transaction is unable to be completed in one go and has to be repeated more than once.

So based on our observations and analysis, it is clear that dependency on xenstore based communication and coordination is not ideal keeping in mind the low latency goals of CMM. These findings and the dependence on Dom-0 for xenstore operations motivates us to revisit the baseline implementation of CMM.

5.3 Conclusion

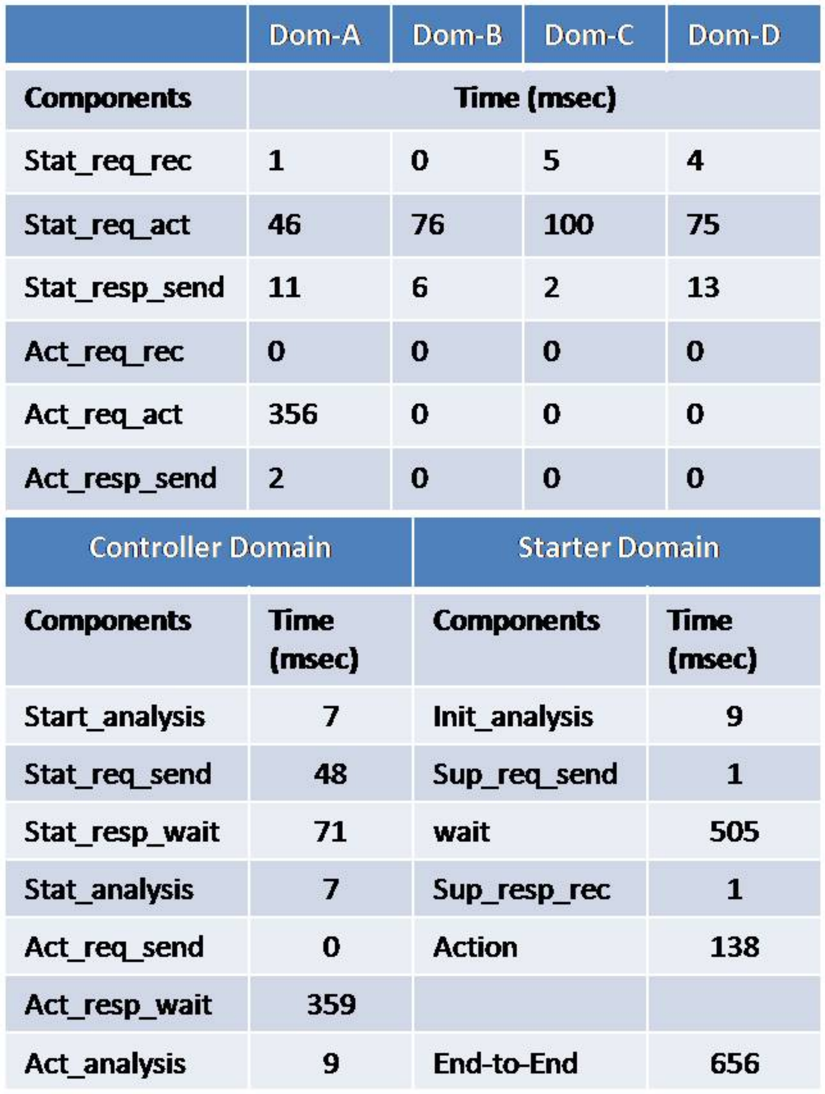
In this chapter, we evaluated the extent of interference that the schedulers and scheduling policies in the hypervisor and OS kernel can introduce in the operation of CMM mechanism and the resulting effect on latency. We will be diving into the scheduler interference issue in the next chapter and discuss methods used for mitigating scheduler interference. We have also discussed about a detailed analysis of the amount of time taken in various phases of operation of CMM. One key realization from our analysis is about the latency issues cropping up due to the split driver design principle and reliance on xenstore interfaces for communication and coordination. We will discuss design changes for dealing with this issue in a later chapter 7.



	Dom-A	Dom-B	Dom-C	Dom-D
Components	**Time (msec)**			
Stat_req_rec	0	6	0	8
Stat_req_act	30	66	105	101
Stat_resp_send	17	9	1	3
Act_req_rec	5	5	12	2
Act_req_act	225	237	219	206
Act_resp_send	1	7	19	3

Controller Domain		Starter Domain	
Components	**Time (msec)**	**Components**	**Time (msec)**
Start_analysis	12	Init_analysis	7
Stat_req_send	47	Sup_req_send	0
Stat_resp_wait	68	wait	413
Stat_analysis	7	Sup_resp_rec	1
Act_req_send	57	Action	138
Act_resp_wait	212		
Act_analysis	8	End-to-end	561

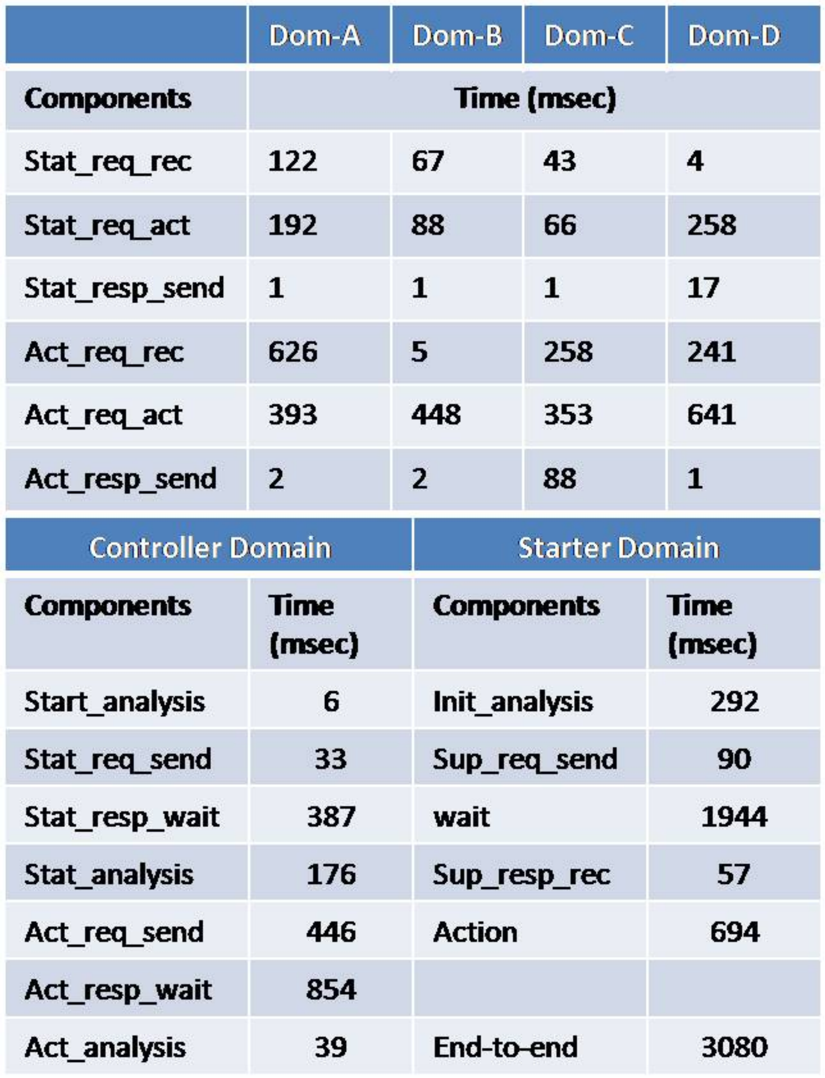
Figure 18: Results of the analysis of time spent in various phases of end-to-end operation of CMM for equi distribution of requested memory and no active workload in the donor VMs



	Dom-A	Dom-B	Dom-C	Dom-D
Components	**Time (msec)**			
Stat_req_rec	1	0	5	4
Stat_req_act	46	76	100	75
Stat_resp_send	11	6	2	13
Act_req_rec	0	0	0	0
Act_req_act	356	0	0	0
Act_resp_send	2	0	0	0

Controller Domain		Starter Domain	
Components	**Time (msec)**	**Components**	**Time (msec)**
Start_analysis	7	Init_analysis	9
Stat_req_send	48	Sup_req_send	1
Stat_resp_wait	71	wait	505
Stat_analysis	7	Sup_resp_rec	1
Act_req_send	0	Action	138
Act_resp_wait	359		
Act_analysis	9	End-to-End	656

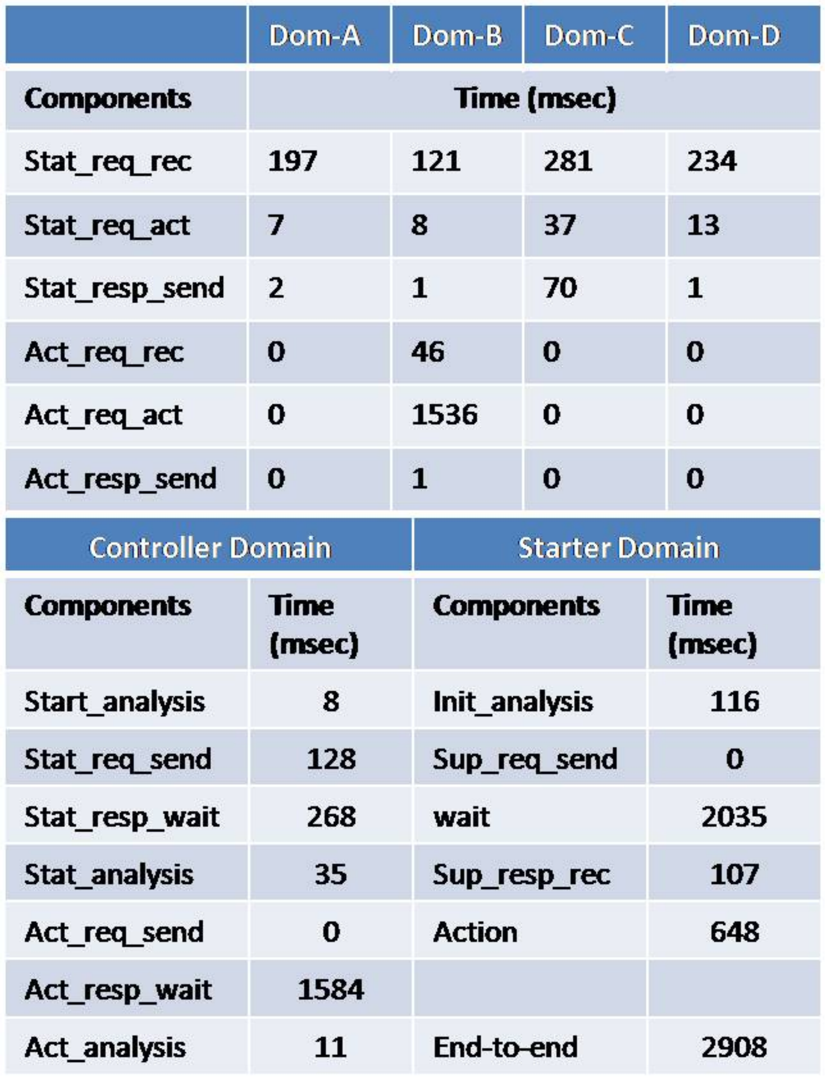
Figure 19: Results of the analysis of time spent in various phases of end-to-end operation of CMM for skew distribution of requested memory and no active workload in the donor VMs



	Dom-A	Dom-B	Dom-C	Dom-D
Components	**Time (msec)**			
Stat_req_rec	122	67	43	4
Stat_req_act	192	88	66	258
Stat_resp_send	1	1	1	17
Act_req_rec	626	5	258	241
Act_req_act	393	448	353	641
Act_resp_send	2	2	88	1

Controller Domain		Starter Domain	
Components	**Time (msec)**	**Components**	**Time (msec)**
Start_analysis	6	Init_analysis	292
Stat_req_send	33	Sup_req_send	90
Stat_resp_wait	387	wait	1944
Stat_analysis	176	Sup_resp_rec	57
Act_req_send	446	Action	694
Act_resp_wait	854		
Act_analysis	39	End-to-end	3080

Figure 20: Results of the analysis of time spent in various phases of end-to-end operation of CMM for equi distribution of requested memory and in the presence of active workload in the donor VMs



	Dom-A	Dom-B	Dom-C	Dom-D
Components	**Time (msec)**			
Stat_req_rec	197	121	281	234
Stat_req_act	7	8	37	13
Stat_resp_send	2	1	70	1
Act_req_rec	0	46	0	0
Act_req_act	0	1536	0	0
Act_resp_send	0	1	0	0

Controller Domain		Starter Domain	
Components	**Time (msec)**	**Components**	**Time (msec)**
Start_analysis	8	Init_analysis	116
Stat_req_send	128	Sup_req_send	0
Stat_resp_wait	268	wait	2035
Stat_analysis	35	Sup_resp_rec	107
Act_req_send	0	Action	648
Act_resp_wait	1584		
Act_analysis	11	End-to-end	2908

Figure 21: Results of the analysis of time spent in various phases of end-to-end operation of CMM for skew distribution of requested memory and in the presence of active workload in the donor VMs

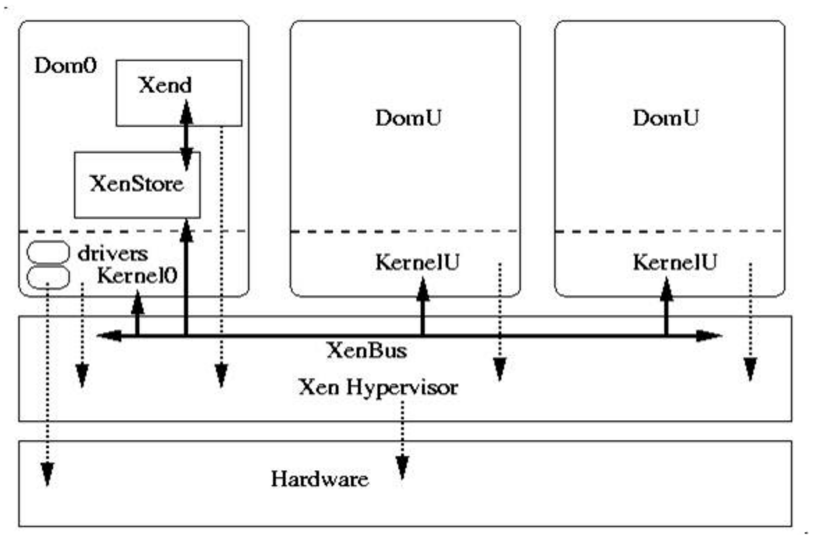


Figure 22: Xenstore architecture

CHAPTER VI

MITIGATING SCHEDULER INTERFERENCE

As is evident from the discussion in the previous chapter, the default scheduling policy of the schedulers inside the Xen hypervisor and the Linux kernel are not optimal for the distributed nature of operation of CMM. In this section we will be discussing about the optimization constructs that we have put into the hypervisor and kernel layers for minimizing the interferences.

6.1 Hypervisor

Our discussion in this section is in the context of the existing credit scheduler framework [1, 7]. It is the most commonly used (also the default choice) scheduler in Xen hypervisor based virtualized setups. However, the concepts are equally applicable to a different scheduler framework also. We start by briefly describing the credit scheduler operating principles.

6.1.1 Credit Scheduler

Credit Scheduler provides a form of proportional share scheduling. In credit scheduler, every physical core has one *run-queue*, which holds the runnable VCPUs (VCPUs having task to run). An *IDLE-VCPU* for each physical core is also created at boot time. It is always runnable and is put at the end of the run-queue. When the *IDLE-VCPU* is scheduled, the physical core becomes idle. Credit scheduler operates based upon two parameters pertaining to each VM/domain: *weight* and *cap*. *Weight* defines its proportional share, and *cap* defines the upper limit of execution time. At the beginning of every accounting period, each domain is allocated *credit* according to its *weight*, and the credit is distributed among its VCPUs (each domain/VM have one or more VCPUs, which are the schedulable entities).

VCPUs are divided into three priority categories when on the RunQ: BOOST, UNDER, and OVER. A VCPU is put into UNDER if it has not used up its allocated credit, and OVER if it runs out of credit. To accommodate low latency, the scheduler has the BOOST priority, where blocked VCPUs waiting for an I/O event are boosted upon receiving an event/interrupt. A task gets boosted for a very short time (less than 10 ms) after which it reverts to UNDERr priority. This simple optimization of task boosting is good in the delegation model for I/O-bound domains such as Dom0 which handles I/O for all domains, wakes up a lot and finishes its work within a very short time. The scheduler picks VCPU at the front of the run-queue, which is arranged in the order of VCPUs of priority BOOST, UNDER and OVER (different regions of the same run-queue). Within each category, VCPUs are scheduled in a round robin fashion. Once a VCPU is scheduled, it receives the time slice of 30 ms and runs consuming its credit (every 10 ms). This default time slice has been decided by keeping in mind the trade off between real time responsiveness and throughput. If the time slice of a running VCPU expires (or it is de-scheduled for some other reason), it is put into the tail of a list of that contains VCPUs with the same priority as the de-scheduled VCPU. If a running VCPU does not have any runnable task in spite of time remaining in its time slice, it is blocked and leaves the run-queue. The credit scheduler also performs load balancing between cores in an SMP environment. When a physical core becomes idle or has no VCPU in boosted or under priority, it checks its peer CPUs' queues to see if there is a strictly higher priority task. If so, it steals the higher priority task.

6.1.2 Scheduler Optimization

At the hypervisor layer, the primary mode of scheduler interference is in terms of a VM not getting scheduled when it is required to be doing some computation towards the progress and completion of an operation of CMM. The causes for this are two pronged: (1) lack of knowledge at the hypervisor layer about any ongoing operation of the mechanism and the various stages of involvement of the virtual machines in this operation (2) Lack of any optimization features inside the hypervisor scheduler that could be activated on-demand to provide scheduling priority to virtual machines actively involved in CMM operation.

To address the first issue, we have introduced new hypercalls which could be used by the VM kernels to inform the hypervisor about the 'start' and subsequent stages of the operation of the coordinated memory management mechanism. The invoking of these hypercalls transfers the control into the hypervisor, which subsequently takes actions to minimize the scheduler interference and latency involved in the operation. This is achieved by temporary tweaking of the scheduling policy in order to give scheduling priority to the virtual machines (VCPUs) that are actively working towards the progress of the operation.

To address the second issue, we have modified the credit scheduler. We have introduced a new *BALLOON* priority to which VCPUs could be elevated when they are doing CMM related activity. When the hypervisor receives the very first hypercall about the start of an operation of the mechanism (from the VM that initiates the operation), the hypervisor switches on the ballooning-mode of the scheduler. In this mode, when a VM starts any CMM related activity (computing the status of its allocated memory, sending information to the controller-domain/Dom-0, Dom-0 deciding about the distribution of the initial memory request amount among the active VMs etc.), it informs the hypervisor about it by invoking the hypercall and providing information about the activity by passing an integer-valued parameter. On receiving the hypercall, the hypervisor elevates the priority of all the VCPUs associated with the particular VM to *BALLOON*. Also, the VCPUs are removed from their current position in the *run-queue* and are re-inserted at the front of the run-queue (after other VCPUs of *BALLOON* priority, if present) so as to give them priority over other VCPUs. When a VM doing CMM related computation finishes with the phase of activity, it informs the hypervisor about that by invoking the hypercall and the hypervisor re-installs the VCPUs belonging to the domain back with their original priority (most often *UNDER*).

A simple deviation from this optimization mechanism is applied for Dom-0 owing to its special position in the para-virtualized architecture and split-driver model of device driver implementation. Also, the use of xenstore and xenbus for the inter-domain communication during the operation of CMM and the role of Dom-0 in the functioning of xenstore/ xenbus necessitates the special consideration of Dom-0 in our optimization mechanism. Keeping this in mind, Dom-0 and its VCPUs are kept in the *BALLOON* priority from the beginning until the end of the operation of CMM.

6.2 Kernel Scheduler

As mentioned earlier, inside the VMs, CMM related activities are performed with the help of kernel threads in accordance with work-queue mechanism, by putting work-items into work-queues. Kernel-threads are generic and serve multiple work-queue. In addition, work-items scheduled in a work-queue may end up getting served by more than one kernel-threads. So, in usual circumstances, it is not possible to decide a priori which kernel thread will end up executing the computation associated with a particular work-item. This is also true in case of CMM related work-items, as we have not used dedicated worker-queue and threads for these work-items. In these circumstances, the primary form of interference by the kernel scheduler in the mechanism's operation is in terms of not granting appropriate run-time priority to the kernel thread executing CMM generated work-items.

We illustrate our design idea by modifying the *Completely Fair Scheduler* (CFS) in Linux kernel. The features of the CFS most relevant to this discussion are *scheduling policies* and *scheduler classes*. CFS is designed and implemented as an extensible hierarchy of scheduler classes (/modules) and supports multiple scheduling policies (SCHED_NORMAL, SCHED_BATCH, SCHED_IDLE, SCHED_FIFO, SCHED_RR) with the help of *fair* and *rt* modules. *SCHED_NORMAL* is the policy that is used for regular tasks. It tracks the virtual run time of the tasks and always runs the task having the smallest virtual run time. Achieving fairness being its primary goal, this scheduling policy is not appropriate for catering to the needs of latency sensitive tasks. However, the policies SCHED_FIFO and SCHED_RR provide alternatives for (soft) real-time latency sensitive tasks. SCHED_FIFO implements a simple first-in, first-out scheduling algorithm without time slices. A runnable SCHED_FIFO task is always scheduled over any SCHED_NORMAL tasks. When a SCHED_FIFO task becomes runnable, it continues to run until it blocks or explicitly yields the processor. Only a higher priority SCHED_FIFO or SCHED_RR task can preempt a SCHED_FIFO task. Two or more SCHED_FIFO tasks at the same priority run round robin, but only yielding the processor when they explicitly choose to do so. SCHED_RR is identical to SCHED_FIFO except that each process can run only until it exhausts a predetermined time-slice. Also on account of real-time tasks operating only with static priority (no dynamic priority calculation) ensures that a real time process at a given priority always preempts a process at a lower priority.

Given the availability of the afore-mentioned scheduling policies, we use the the following approach for minimizing the latency (and kernel scheduler interference) of the CMM operation:

- *Marking*: We use a special *CMM marker* in the work-item data structure to distinguish CMM work-items from others. We use separate macros to create CMM related work-items,.

- *Priority Boosting*: When a kernel thread picks up a work-item with a *CMM marker*, the scheduling policy for thread is boosted to SCHED_FIFO and it is given the maximum possible priority.Its original priority, scheduling policy and nice value are also recorded in a data structure for later use.

- *Priority Re-instating*: When a kernel thread finishes a CMM-marker work-item, its original scheduling policy, priority, and nice values are re-instated to its pre-boost stored values.

Since this priority boosting for CMM-related work-items is on-demand and short-lived we believe that this does not affect other critical tasks in a given VM.

6.3 Evaluation

In this section, we evaluate the effect of applying the optimization constructs mentioned in the earlier section on the latency of CMM and the interference caused by hypervisor scheduling, and present the results of our experimental evaluation. In each case, we present the result for the following options (1) When no optimization construct is used (2) Only the optimization construct for Linux kernel scheduler is used (3) Only the optimization construct for Xen hypervisor scheduler is used (4) Optimization constructs in both Xen and Linux are used. We repeated the experiments with and without active workload in each of the VMs. The other experimental conditions are same as mentioned in prior chapters.

6.3.1 No Workload

Figure 23, Figure 24, present the results for the cases where equi-distribution is used to distribute the total required memory amount among the set of active VMs. Figure 25, Figure 26 show the results for skew distribution. As we already know from earlier discussion, in the case of no active workload in the VMs, the opportunity for reducing the latency and scheduler interference is minimal and this is reflected in our results.

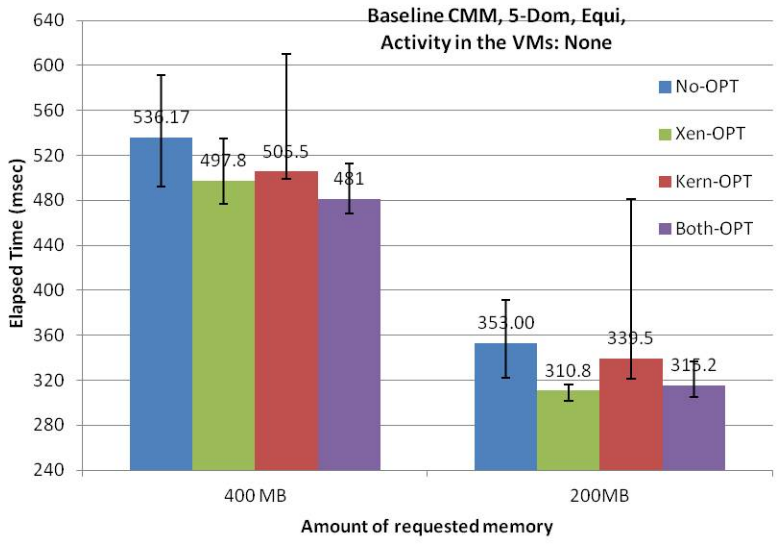


Figure 23: Effect of scheduler optimization constructs on end-to-end operational latency of CMM for equi distribution of requested memory, no active workload in the donor VMs

Although we see close to 10% latency improvement (with both optimization constructs applied) in the equi-distribution case, our optimization constructs have no significant effect for skew distribution. The primary reason behind this is, in the skew distribution (when no active workload is present in the VMs apart from the mechanism related tasks), the VM that is supposed to be doing CMM related computation is the only active VM (apart from Dom-0). Hence, in the absence of any competition from any other VM (non Dom-0), it is allocated optimum execution time-slots. A similar logic is applicable for the kernel thread executing CMM generated work-items. In such a circumstance, neither the latency nor the scheduler-interference has much scope for improvement.

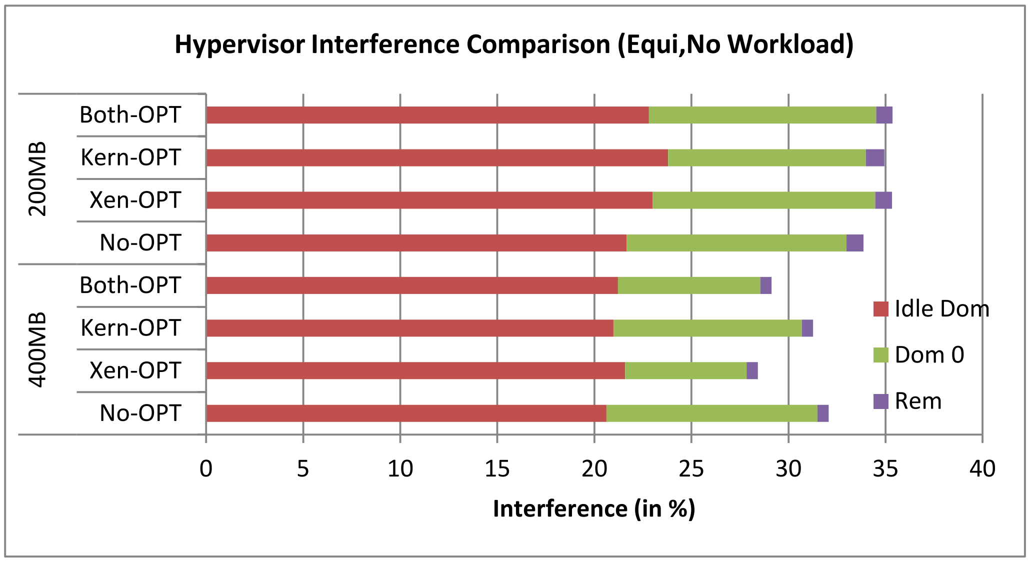


Figure 24: Effect of optimization constructs on scheduler interference for equi distribution of requested memory, no active workload in the donor VMs

6.3.2 With Workload

When the experiments are repeated in the presence of computation heavy workload in each of the VMs, we get to see the real effect of the optimization constructs. Figure 27 and Figure 28 presents the latency data for equi-distribution. Figure 29 and Figure 30 present the results for latency and scheduler interference in case of skew-distribution.

As is expected, with active workload, the effect of optimization is much more evident in the case of skew-distribution. In the case of skew-distribution, when there is active workload present in each of the VMs, the hypervisor does not possess any information regarding which VMs are doing CMM related computation and which are not working towards progress of the CMM operation. As such, all VMs are scheduled to maintain fairness and equity, which although is desirable in a normal circumstance, is not optimal considering the low latency needs of CMM. So, switching on the optimization constructs has significant benefits in this case. As can be seen in Figure 29, with both optimization constructs activated, there is close to 45% improvement in operational latency of the mechanism. A parallel trend is also observed in the case of scheduler interference. In case of equi distribution, as expected, the effects of scheduler optimization constructs is relatively less compared to that of skew

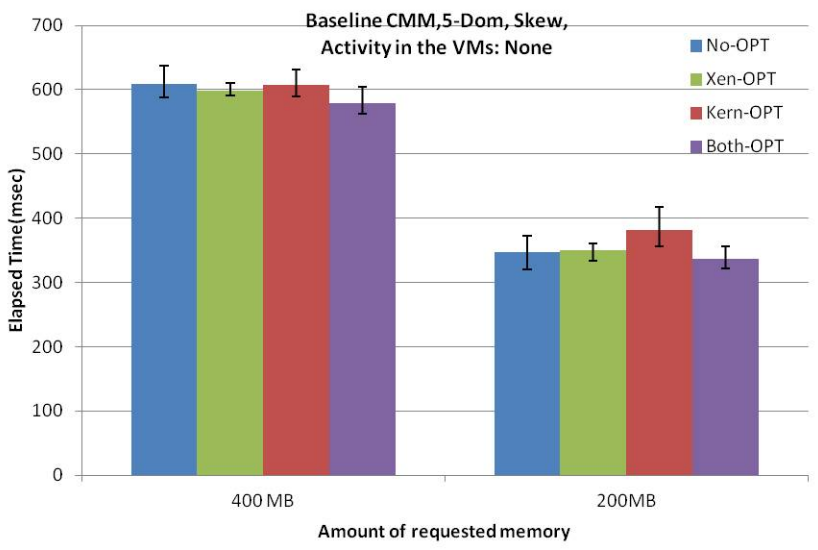


Figure 25: Effect of scheduler optimization constructs on end-to-end operational latency of CMM for skew distribution of requested memory, no active workload in the donor VMs

distribution.

One thing to note would be, in case of equi-distribution there are time periods when all the active VMs are actively doing CMM related computation and as such, all the corresponding VCPUs are elevated to the *BALLOON* priority. In such a case, the purpose of prioritizing is lost and a simple solution is to selectively switch off the optimization constructs built into Xen hypervisor when using equi-distribution.

6.4 Conclusion

In this chapter, we discussed about the optimization constructs we have built into the Xen hypervisor and Linux kernel for mitigating the scheduler interferences. In real world usage, CMM will be operated in the presence of active workload in each of the VMs and only a subset of VMs will be considered for donating memory. In such cases, the necessity of and the resulting benefits from both the optimization constructs is well established by the results presented in this chapter.

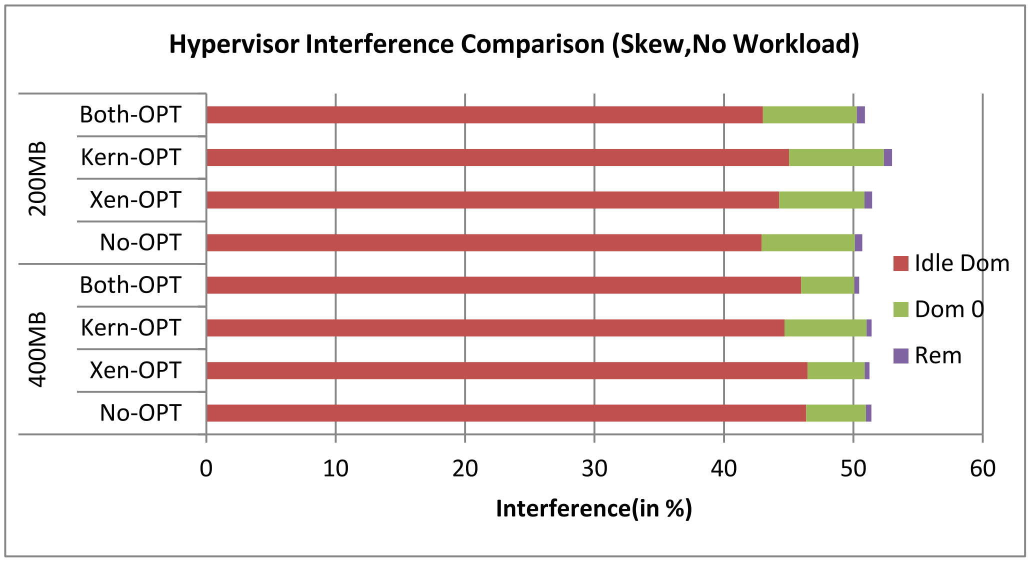


Figure 26: Effect of optimization constructs on scheduler interference for equi distribution of requested memory, no active workload in the donor VMs

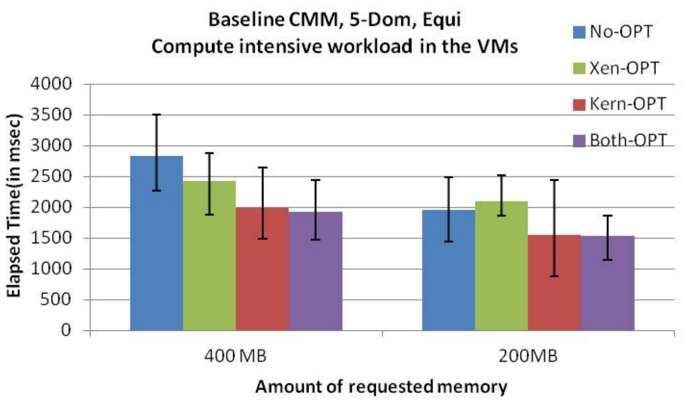


Figure 27: Effect of scheduler optimization constructs on end-to-end operational latency of CMM for equi distribution of requested memory, with active workload in the donor VMs

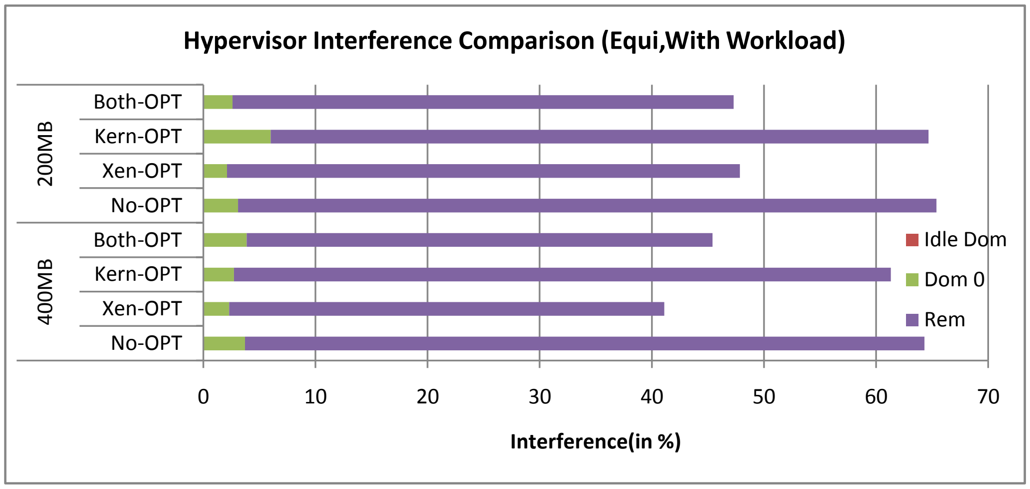


Figure 28: Effect of optimization constructs on scheduler interference for equi distribution of requested memory, with active workload in the donor VMs

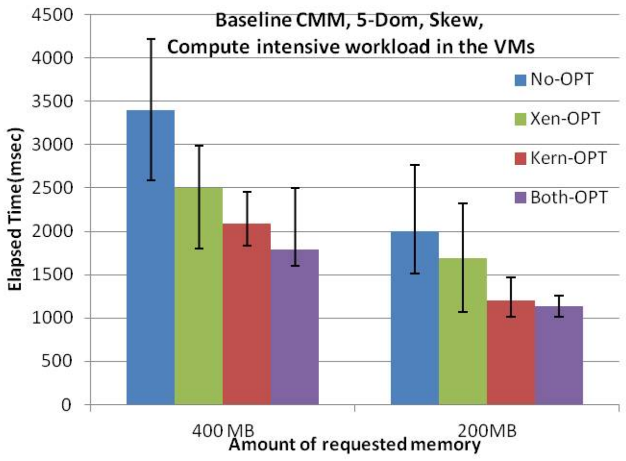


Figure 29: Effect of scheduler optimization constructs on end-to-end operational latency of CMM for skew distribution of requested memory, with active workload in the donor VMs

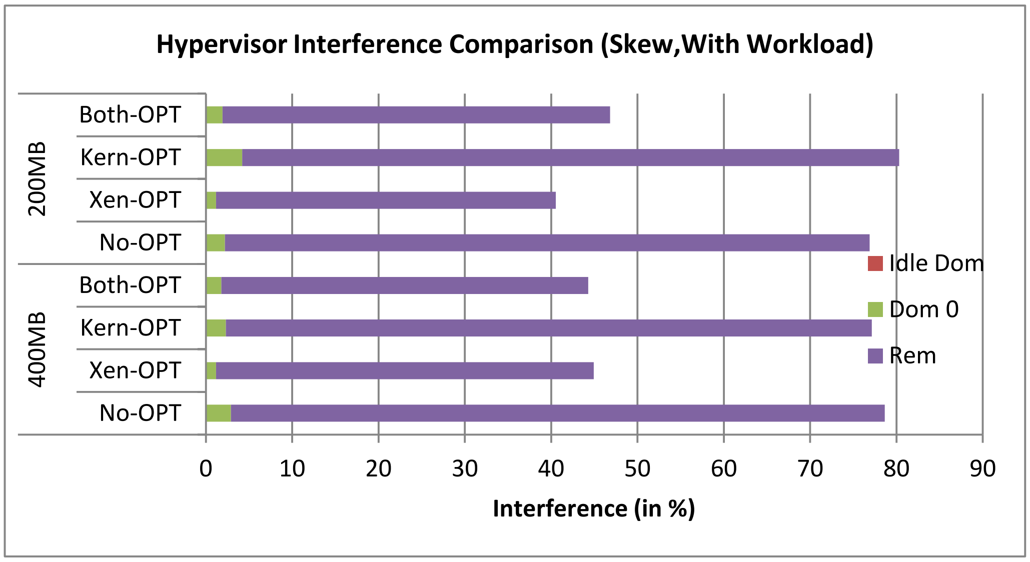


Figure 30: Effect of optimization constructs on scheduler interference for skew distribution of requested memory, with active workload in the donor VMs

CHAPTER VII

LATENCY OPTIMIZED IMPLEMENTATION OF CMM

As has been clear from the discussions in earlier chapters, it is our goal in this dissertation to come up with a low latency memory management mechanism for virtualized environments, which could be used for dynamic balancing of the memory among the set of running virtual machines. With this goal, we started by analyzing existing memory management mechanisms and designed the coordinated memory management mechanism. A key feature of CMM is the coordinated and collected approach at resolving the memory pressure being experienced by any VM. Our analysis has shown that CMM compares favorably with existing mechanisms. In addition, our analysis also provided us with some pointers regarding possible avenues to improve the operational latency. Due to the distributed nature of the mechanism, it is prone to interference from scheduler. We have implemented and analyzed constructs for mitigating interference from schedulers in Xen hypervisor and Linux kernel. But there are other aspects of CMM's operation that need to be optimized in order to further minimize the operational latency and ensure that it remains bounded in varied operating scenarios.

One such aspect is the choice of the split driver principle and choice of Dom-0 for placing the central coordination functionality. Another aspect is the choice of using xenstore and xenbus mechanisms for communication and coordination across the different VMs. In this setup, xenstore, xenbus and event-channel mechanisms are used by the VMs for communication and data transfer and the back-end driver in Dom-0 is responsible for the coordinating and decision making aspects of the mechanism's operation. Our primary reasoning behind this design decision was to keep in sync with implementation of other drivers in para-virtualized setups, and a conscious choice to not explicitly increase the hypervisor code base. This system architecture has an inherent dependency on the xenstore and xenbus operations and the latency of the operation is affected by the latency of xenstore and xenbus operations. Although xenstore is implemented as a database built on top of memory mapped files and a daemon running in Dom-0, our previous analysis shows that the latency involved in xenstore operations is a significant component of the total operational latency of the coordinated memory management mechanism built with this design principle. Also, owing to the critical role of the Dom-0 in xenstore and xenbus operations, it has to be accorded a special consideration in the hypervisor scheduler optimization feature, which is not desirable considering the goal of minimizing the scheduler interference and operational latency.

The aforementioned issues with the split-driver design motivate us to change the implementation of CMM to one in which Xen hypervisor plays a more active role in the operation, the primary hypothesis behind this decision being, by avoiding the xenbus and xenstore operations, we can achieve better operational latency of CMM. In the new architecture, hypervisor is responsible for the coordination and decision making aspects of CMM and assumes the role of Dom-0 in the earlier implementation. Instead of xenstore, xenbus and event channel operations, VMs communicate with the hypervisor by making hypercalls and the hypervisor uses up-calls to transmit control messages and data to the VMs.

7.1 Design

Keeping in sync with our designated principles of *on-demand*, *coordinated* and *low-latency*, our new design for CMM closely follows the earlier design. Figure 31 presents the key components of the new architecture. In this new architecture, the central coordination module is inside the hypervisor and being inside the hypervisor allows it to be well integrated with the memory management unit and the scheduling unit of the hypervisor. Interaction with the memory management unit allows it to keep track of the memory allocation to the various VMs, and in addition, opens up possibilities for further sophistication of CMM. Because of the focus on the latency aspect of the mechanism in this dissertation, we have not considered exploring the transparent paging of the memory pages of the VMs to the disk in times of really severe memory pressure. However, the current architecture provides a clean interface for integrating with this functionality. Central coordination module interacts with the scheduling unit for providing higher priority to the tasks and VMs working towards the progress of the CMM operation. One aspect that is different in the new design from that of the earlier design is, instead of a back-end for each VM, there is one single central coordination module.

The front-ends for the mechanism are located inside the kernel space of the VMs, as was the case in the earlier design. They in turn are integrated with the memory management unit and the ballooning unit of the kernel. Interaction with the memory management unit is for gathering statistics about the amount of memory allocated to the VMs, usage status, working set estimation etc. and the ballooning unit does the memory page release and reclamation functions. Also, they interact with the scheduling framework of the kernel to provide higher scheduling priority to the threads and work-queues actively involved in the operation of CMM. The communication and coordination between the front-end and the central coordination module happens by the help of hyper-calls and up-calls and the necessary data transfer is by the help of a shared memory page between the hypervisor and each of the VMs. We will discuss these in detail in the next section.

It is important to note that, in a parallel design approach which is more similar to the earlier approach, we could have used a shared page based data transfer and coordination mechanism between the Dom-Us and Dom-0 to alleviate the latency effects of xenbus and xenstore operations. But, in order to reduce the complications of giving Dom-0 special priority in the scheduler optimization constructs and to make Dom-0 an equal partner in the mechanism as other domains, we have followed this design approach of transferring the location of the central coordination module to inside the hypervisor.

7.2 Implementation details

In this section we will be discussing briefly about some of the key implementation and operation aspects of this revised design and how it differs from the earlier design.

7.2.1 Coordination constructs

There are three primary coordination, communication and data transfer related features that are different from the earlier design. A brief description about them is as follows:

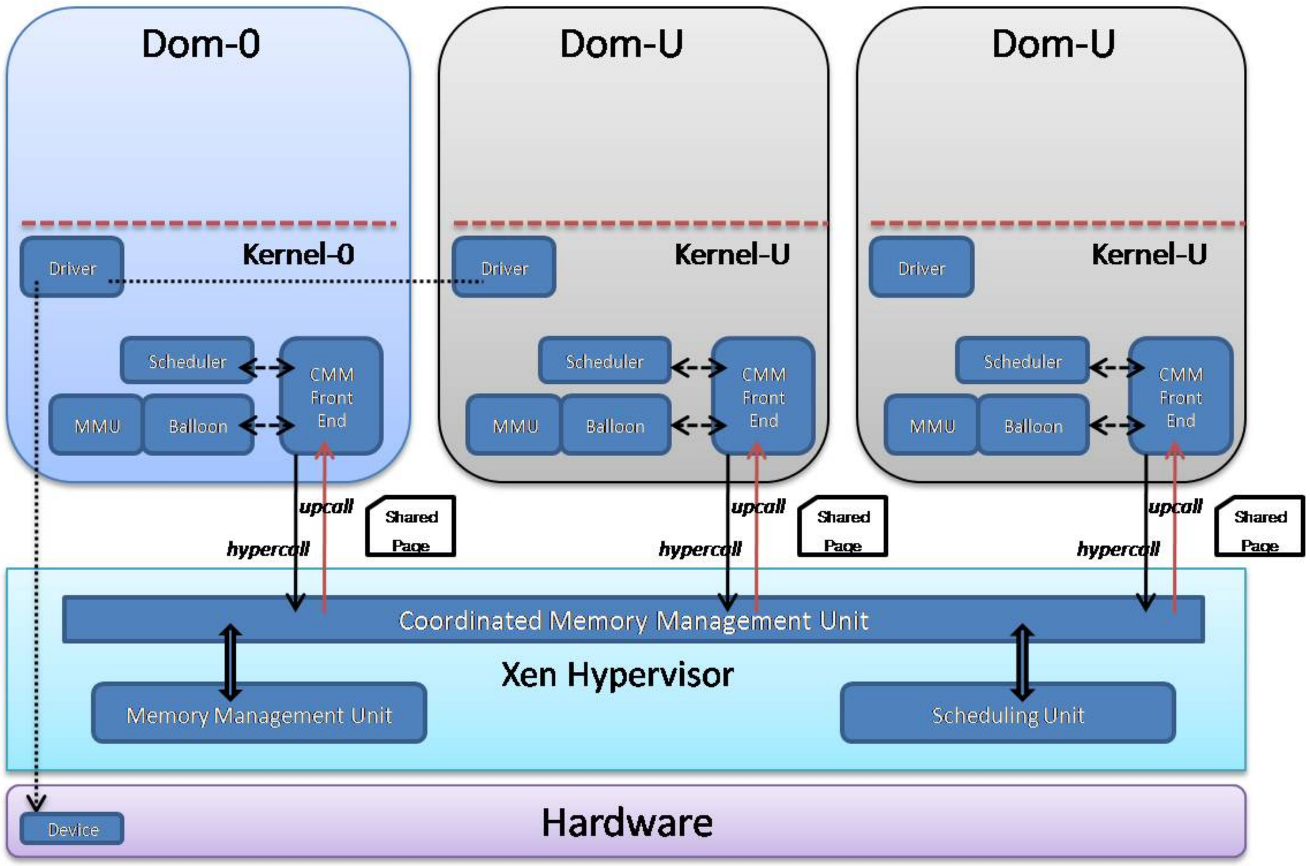


Figure 31: Architecture diagram for optimized hypervisor based implementation of CMM. Central coordination module is in the hypervisor; shared memory page, upcalls and hypercalls are used for coordination

- Shared memory page for data transfer: When a VM's memory image is created during the boot time, a separate memory page is allocated for the data transfer purposes of CMM. The primary purpose of the shared page is to host the necessary data structure for information exchange. The data structure is appropriately initialized in this shared page during boot time and has entries for both the front-end of mechanism and the central coordination module. The entries are similar to the entries in the xenstore directory for the VMs in the earlier split-driver design. Central coordination module writes to the entries it is responsible for and those entries are read by the front-ends. The reverse happens for the entries for which the front-ends are responsible. In its basic form, this functionality is similar to the start-info and shared-info pages for each of the VMs running in the Xen based virtualized setup. The start-info page contains basic information required by the VM to initialize the kernel and shared-info page is for data that is updated while the VM runs.

- Xen upcall: This is the mechanism that allows the central coordination module to send notifications related to the operation of CMM to the front-ends inside the kernel of the VMs. It is built on top of the virtual interrupt mechanism available in the Xen based virtualization setup. The virtual interrupt mechanism itself is within the purview of the events mechanism in the Xen setup for delivering notifications from the hypervisor to VMs or from one VM to another. Conceptually, the events mechanism is similar to that of UNIX signals, where each event delivers information about an occurrence of some event. Events in the Xen setup could be one of three types: *physical IRQ*, *virtual IRQ*, *inter-domain events*. Physical IRQs are the mapping of real IRQs to the events mechanism. Virtual IRQs are similar to physical IRQs but are meant for virtual devices. Inter-domain events are used for sending notifications from one VM to another. The event channel mechanism used in the earlier design for sending notification from the Dom-U front-ends to Dom-0 back-end is one manifestation of the inter-domain event construct. The upcall mechanism used in the new design is based on the virtual IRQ construct. Each virtual CPU (VCPU) has flags relating to virtual interrupts. The evtchn_upcall_pending and evtchn_pending_sel flags are used to denote that events or interrupts are pending. There is another evtchn_pending field, which is a eight machine words long bit-field and which indicates the channels which have events waiting. Bits in this field are set by the hypervisor and cleared by the guest. For our purpose, we have used an unused bit position in this field for delivering the upcalls related to CMM. We also install appropriate handlers for this event/interrupt in the VMs so that they can appropriately respond to the upcalls from the hypervisor.

- Xen hypercall: Hypercalls are quite common in the Xen setup and are used in the earlier design as well for generating the trace of events and for integration between the scheduler optimization constructs inside the VMs and the scheduler optimization construct in the hypervisor. Hypercalls are analogous to system calls of non-virtualized setups, but are instead used by the kernels of the VMs to trap into the hypervisor and get a particular privileged task done. The implementation approach for the hypercalls in Xen is a bit different from the usual implementation of system calls. In recent implementations of Xen, for para-virtualized setups, hypercalls are issued by the VM kernels by calling a function in a shared memory page mapped by the hypervisor. For our purposes, we added additional hypercalls for sending CMM related control signals from the VMs to the hypervisor. They act as replacement for the xenbus and event-channel based inter-domain notifications in the earlier design. Some of these hypercalls are also responsible for additional functions of helping the creation of the event-trace and integrating with the scheduler optimization constructs inside the hypervisor.

7.2.2 Operation

The operating steps of the new design very closely resemble that of the earlier design. After the initialization phase, the drivers communicate by updating the necessary values in the data structure hosted in the shared page, so that the other end can get them. As all the necessary setups are done during the boot-up time, writing to these shared pages is as straight forward as getting a pointer to data structure and updating the values. As before, the entries in the data structure are of two types: data entries and command entries. Data entries are meant for data exchange like the current memory, current working set size etc.. The command entries are primarily boolean entries whose main function is to convey to the other end to start some activity like 'status-response', 'start-action' etc. The notification (about new entries being made, change etc.) to the other end is sent by using the hypercalls and upcalls described in the previous section.

Figure 32 is a depiction of the various steps involved in the operation of optimized CMM. As before, the front-end is responsible for beginning the sequence of activities of CMM. On receiving information about need of memory from a front-end, the central coordination module starts the decision making process. The first step is to collect information about the memory state of other VMs running in the machine. This is accomplished by turning on the entry for 'status-report' in the shared page data structure for the various VMs and sending an upcall to the corresponding front-ends. The front-ends, on getting the upcall, read the entries from the shared page and get to know that they have been asked to report the status.They collect the necessary information (current memory, working set size estimation etc) and write them back to the entries in the shared page. After this they turn on the entry for 'status-response' in the shared page and inform the central coordination module by the help of a hypercall. As before, the front-ends gather similar information as is exported to /proc/meminfo.

The central coordination module waits until it receives response from all the relevant front-ends, and then reads the necessary entries from the shared page data structures, and populates its internal data structure for further analysis. After analyzing the responses, the central coordination module makes decision about which VMs should be asked to free up memory pages without adversely affecting their ability to cater to their own work loads. It may use policy configurations like the relative importance and criticality of the VMs etc. in making this decision. It then starts the action phase by communicating to the various front-ends about how much memory they should try to free up. This is achieved by writing the necessary information to the corresponding shared page data structure, turning the 'start-action' entry on and sending upcalls to the front-ends.

As the front-ends receive the upcall and find the 'start-action' entry turned on, they read the data about what is expected of them and start acting on it. Mostly, it is a command from the back-end to release memory pages to the hypervisor (balloon-inflation). So the front-end driver works with the balloon driver to release the target number of pages back to the hypervisor. After the page release operation is over, the front-end writes the result of the operation into the shared page and notifies the central module by making a hypercall.

After getting responses from all the front-ends, the central coordination module updates its internal data structures to reflect the result of the various front-end operations, and accordingly notifies the front-end which started the operation on behalf of the VM experiencing memory pressure. This front-end then invokes the kernel balloon driver to start the final operation of getting memory pages from the hypervisor (balloon-deflation).

7.3 Evaluation

To make a fair comparison with the results presented in earlier chapters for the split-driver design of CMM, all the evaluation in this section has been done in the same 2 GHz dual core AMD64 machine with 4 GB RAM, as for earlier experiments. We use Xen-4.1.0 as the hypervisor with pvops based linux 2.6.32.26 kernel in Dom0 and 3.1-rc7 Linux kernel inside the guest domains. All our experiments have been performed with minimal interference from other processes inside the virtual machines. We do not create any new processes or daemons in any of the virtual machines apart from the default ones. Also for the sake of fair comparison, we do not explicitly involve Dom-0 in any of the memory page release or reclamation activity.

7.3.1 Latency

In this section, we present the results of our experiments for measuring the latency of an end-to-end operation of the optimized CMM implemented according to the new hypervisor based design.

As in the earlier cases, to get an idea about the effect of distribution of total memory among the running domains, we use two types of distribution. In one, we distribute the requested memory amount evenly among the virtual machines (named as *equi*-distribution). In the other case, we try to use as less number of domains as possible to get the required number of free pages (named *skew*-distribution). We vary the required memory amount from 100MB - 400MB for a fixed number of 5 running guest VMs. The results of our experiments are represented in Figure 33 and Figure 34. Figure 33 presents the results when the experiments are run without any additional workload in any of the VMs. Figure 34 presents the results when the same experiments are run with a compute intensive workload in each of the VMs.

The following are the primary takeaways from these results:

- Lower latency compared to the earlier split driver based implementation: This inference is obvious by comparing the results of Figure 33 and Figure 34 to that of the results in Figure 10 and Figure 15. When the mechanism is operated in the absence of any workload, there is 30-35% latency improvement over the earlier design for higher memory balancing amount and close to 50% latency improvement for lower memory balancing amounts. This trend is observed for both equi and skew distribution cases. In the split-driver design, for lower memory amounts, the time spent for communication and coordination using the xenstore and xenbus interfaces is a higher fraction of the total end-to-end operation time. This is no longer the case in the new design which uses fast upcalls and hypercalls for coordination purposes. So in the new design, in the absence of any workload, the latency reduction is relatively higher for lower memory amounts. A similar trend is observed for the results in the presence of workload. Also, the latency reduction in the case of equi distribution is higher (close to 50%) in the presence of workload. This is because equi distribution involves more coordination and communication and reducing the coordination cost has a more pronounced effect.

- *Equi* distribution almost always performs better than *Skew* distribution: The primary reason behind this observation is again because of the lower cost of communication and coordination. With the communication costs minimal, equi has advantage over skew distribution because of the possibility of parallel release of memory pages by various VMs and also due to the general fact that releasing lower amounts of memory pages (which is the case in equi distribution) is much less time consuming than releasing higher amounts of memory pages.

7.3.2 Time analysis of the components

Building on the results from the earlier section, we are focusing on identifying the core reasons behind the observed latency improvements for the optimized CMM. For this reason, we present a similar time analysis of the various phases of the end-to-end operation of the CMM based on the new design.

As before, our measurement methodology for this section is based on the previously described procedure for measuring scheduler interference. We use hypercalls to log the time and event type of various events inside the hypervisor memory. These log records constitute a trace of the entire operation. We later analyze the trace to find the time spent in various stages of the complete operation.

The three types of primary role players in the operation of the mechanism are: (a) starter domain: The VM which is experiencing memory pressure and starts the operation of the mechanism by contacting the central coordinator. (b) hypervisor: The central coordination module operates inside the hypervisor. (c) helper domains: Other VMs which are running in the physical machine and actively participate in the mechanism.

The following table 1 provides a brief description of the various phases of execution in these entities for which we measure the time and their notation in the measurement tables in the next section:

Table 2: Notations for component time analysis of the optimized CMM

Entity	Notation	Description
Starter Domain	Sup_req_send	Time spent in sending the support request to the controller domain. Also includes the time spent in populating the entries in the shared memory page
	Wait	Time spent in waiting for the response from the central coordination module
	Sup_resp_rec	Time spent in receiving the response from the central coordination module
	Action	Time spent in gathering the results from the shared memory page and getting the memory from the hypervisor
Hypervisor	Start_analysis	Time spent in doing initial analysis

Table 2 Continued:

	Stat_req_send	Time spent in sending status requests. Involves writing into the entries in the shared pages for the various VMs.
	Stat_resp_wait	Time spent in waiting for the status responses from the VMs
	Status_analysis	Time spent in analyzing the status responses and gathering the result from the various shared pages
	Action_request_send	Time spent in sending the action requests. Involves writing into the corresponding entries for the various VMs selected for memory page donation.
	Action_response_wait	Time spent in waiting for the action responses
	Action_analysis	Time spent in analyzing the action responses
Helper Domain	Status_request_rec	Time spent in receiving the status request
	Stat_req_act	Time spent in acting on the status request
	Wait	Time spent in waiting for the response from the central coordination module
	Action_request_rec	Time spent in receiving the action requests
	Action_request_act	Time spent in acting on the action requests

It is noteworthy to mention that, because of the manner in which the traces are collected for the time analysis of the components of the optimized CMM, we are not being able to measure the time for sending of status response and action response by the various front-ends, as we measured it for the previous design. The same hypercall that is used to notify the sending of the response is also used for notifying the event on the event trace. However, as is clear from other related values, this time is not significant because of the fast hypercalls.

The results for our analysis are presented in the following figures. Figure 35 and 36 present the results for sample end-to-end operations of CMM where there is no active workload in any of the VMs, for equi and skew distribution respectively. Figure 37 and 38 present the results for sample end-to-end operations of CMM in the presence of compute intensive workload in the VMs for equi and skew distributions respectively. The total amount of initial memory request is 400MB in all these cases.

The primary conclusions from the results in these figures and comparing them to the corresponding values in the Figures 18, 19, 20, 21 are as follows:

- Latency overhead due to communication and coordination is significantly less. This is clear from the values for Stat_req_send for the hypervisor in the new design and the controller domain in the old design, especially if we look at the values for no workload situations. It is negligible(0) in the new design and is not so in the case of earlier split-driver implementation.

- The latency involved in the release and reclamation of memory pages are almost similar in the various situations. This can be observed from the values for Act_req_act for the helper domains and Action for the starter domain, in the various cases. So evidently, most of the latency improvements is due to the change in the coordination constructs. This result is not surprising, given that coordination and communication constructs are the only change from the previous design.

- Operation in the presence of workload showcases interference from schedulers: This is evident more clearly by comparing the values for the Act_req_act and Action for the helper domains and the starter domain, respectively. These are higher in the case where the mechanism operates in the presence of workload, and thus showcasing the interference from the schedulers inside the VM kernel. Also other latencies like

Stat_req_rec, Stat_req_act, Act_req_rec for the helper domains, which are negligible (0) in the absence of workload, are no longer so in the presence of workload. This is primarily due to the interference from the hypervisor scheduler. As all the VMs have active workload, at times they are running for their full allocated time slot, before another VM which has some pending CMM related computation is scheduled. We have not done accurate analysis of the extent of scheduler interference for the new design, as we already have the interference mitigation constructs available and we provide the results for their effects in the next section.

7.3.3 Effect of scheduler optimization constructs

Although we have not done detailed analysis of the extent of scheduler interference for the latency optimized implementation of CMM, it is evident that there is interference from the schedulers. So in this section, we evaluate the effect of applying the optimization constructs (mentioned in the earlier chapter) on the latency of the coordinated memory management mechanism operation. The optimization constructs are the same as before and operate inside the kernel of the VMs and the hypervisor. As before, the primary responsibilities of the optimization constructs are identifying the entity involved in doing active contribution towards the progress of CMM operation and provide a prioritized cpu access to that entity. At the hypervisor layer, the entity is the vcpu of the VM and at the kernel layer, the entity is the kernel thread executing a CMM related work item. Inside the VM, the identification of the work items pertaining to CMM happens by specifically marking them at the time of creation. The corresponding identification for the VMs and vcpus happens by explicit hypercalls to the hypervisor.

We present the results of our experimental evaluation in the following figures. As before, in each case, we present the result for the following options (1) When no optimization construct is used (2) Only the optimization construct for Linux kernel scheduler is used (3) Only the optimization construct for Xen hypervisor scheduler is used (4) Optimization constructs in both Xen and Linux are used. We repeated the experiments with and without active workload in each of the VMs. The other experimental conditions are same as that mentioned in prior sections.

Figure 39 and Figure 40 present the results for the effects of scheduler optimization constructs in the absence of any active workload in any of the VMs for equi and skew distribution of the requested memory amount respectively. Figure 41 and Figure 42 present the results for the operations in the presence of computation intensive workload in each of the VMs.

The primary observations from the results are as follows:

- As before the effect of optimization constructs is minimal in the case where the operation of the mechanism is activated in the absence of any significant workload in any of the VMs. The cause remains the same: in the absence of any active workload in any of the VMs, only the VMs that have to do something related to the operation of the mechanism are scheduled and as such the potential for any latency improvement is minimal.

- But when CMM is operated in the presence of active workload, there is significant benefit in terms of latency reduction by switching on the scheduler optimization constructs. The relative magnitude of latency reduction by activating both the scheduler optimization constructs is definitely higher than those in the case of earlier split-driver design. This is evident by comparing the results in Figure 41 with Figure 27 and results in Figure 42 with Figure 29. With the new design, the overall latency reduction is higher than 60% in equi distribution case and around 80% in the case of skew distribution. The higher magnitude of latency reduction for skew distribution is expected, because skew distribution inherently involves selected number of VMs in the later stage of the operation and giving higher priority to those VMs results in greater benefit. The core reason for the overall higher latency reduction by the scheduler optimization constructs in the new design is because of removal of the dependency on Dom-0 and xenstore related operations.

- An extension of the last observation is that, with the new design, and with both the optimization constructs enabled, the latency of CMM operation in the presence of workload is just marginally higher than the operational latency of CMM in the absence of any workload. This fact re-affirms the effectiveness of the optimization constructs in minimizing the interference from the schedulers.

To summarize our efforts towards creating a low latency dynamic memory management mechanism (CMM) and bounding its operational latency, we present a comparative picture of the latency of baseline CMM without any scheduler optimization features enabled and the latency of optimized CMM with the scheduler optimization features enabled in Figure 43 and 44. Figure 43 and 44 show the results of repeating the same set of experiments as reported in Figures 10 and 15, respectively but for the optimized CMM with the scheduler optimization features enabled. We present results only for 400 MB and 200 MB memory request amounts and for the *equi* distribution. We get similar results for the *skew* distribution.

For the no workload case (Figure 43), the optimized CMM performs better than the baseline implementation by 30% for the 400 MB memory request; and 50% for the 200 MB memory request. Similarly, for the experiments with workloads in the donor VMs (Figure 44), the optimized CMM performs better than the native implementation by 80% for the 400 MB memory request; and 84% for the 200 MB memory request. In Figure 12, we have already shown that even the baseline implementation offers better latency (by as much as by a factor of 4)compared to self-ballooning for the chosen aggressive parameter settings. The optimized CMM (Figure 43) does anywhere between 6 to 8 times better than self-ballooning for the memory request sizes we experimented with.

7.4 Conclusion

In this chapter, we presented the details for the latency optimized design and implementation of CMM where the central coordination module is inside the Xen hypervisor (instead of Dom-0 as was the case earlier) and the communication happens through upcalls and hypercalls instead of writing to xenstore. We also presented experimental results validating the advantages (in terms of lower latency) of this revised design. In addition to the advantage of lower operational latency, the new design also reaps higher benefits from the scheduler optimization constructs as a result of its more straightforward design and elimination of unnecessary dependence on xenstore and Dom-0.

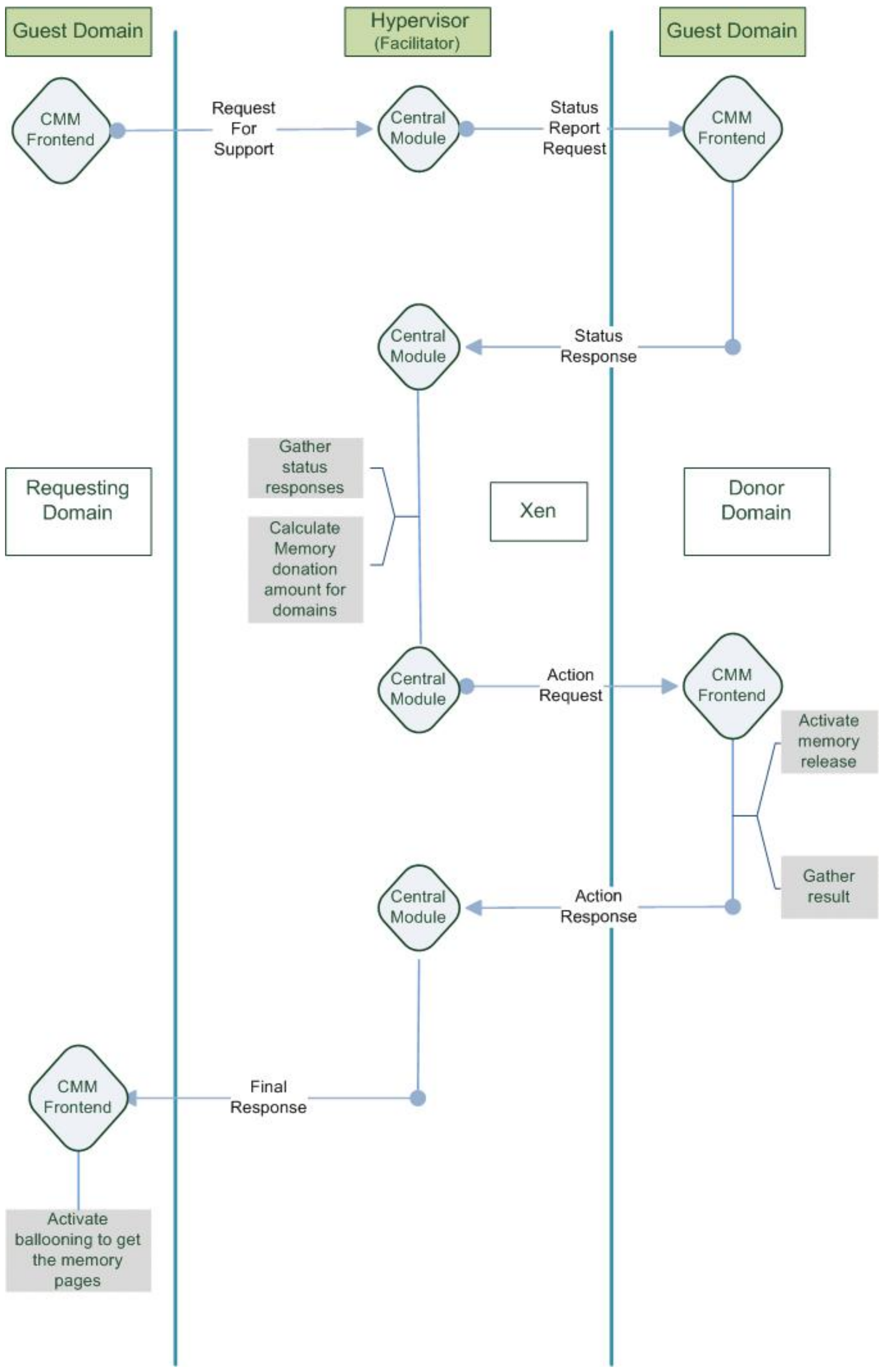


Figure 32: Operational Steps for the optimized hypervisor based implementation of CMM

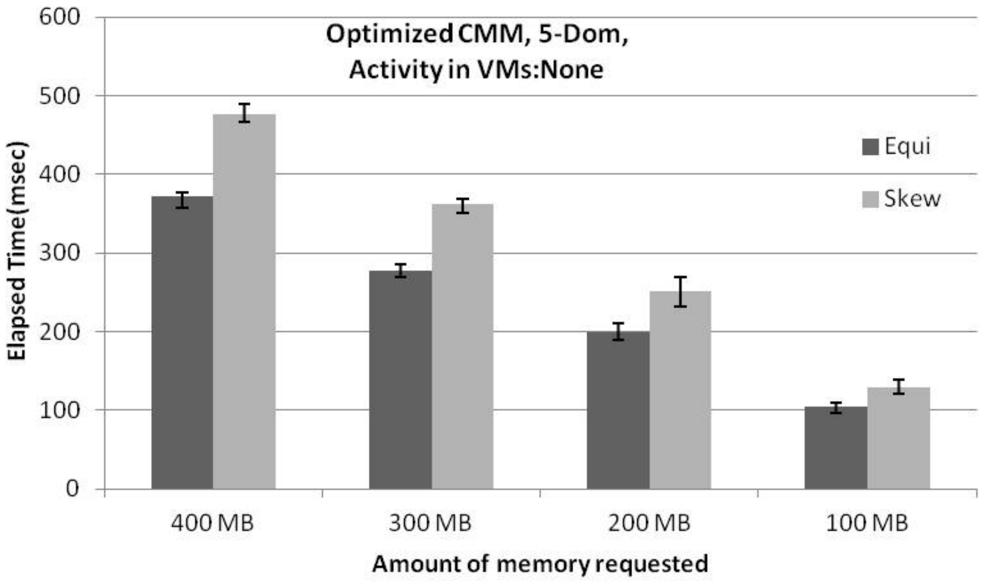


Figure 33: End-to-end latency for optimized hypervisor implementation of CMM: There are no active processes in any of the domains (VMs). Experiment varies the amount of memory for a fixed number of 5 VMs.

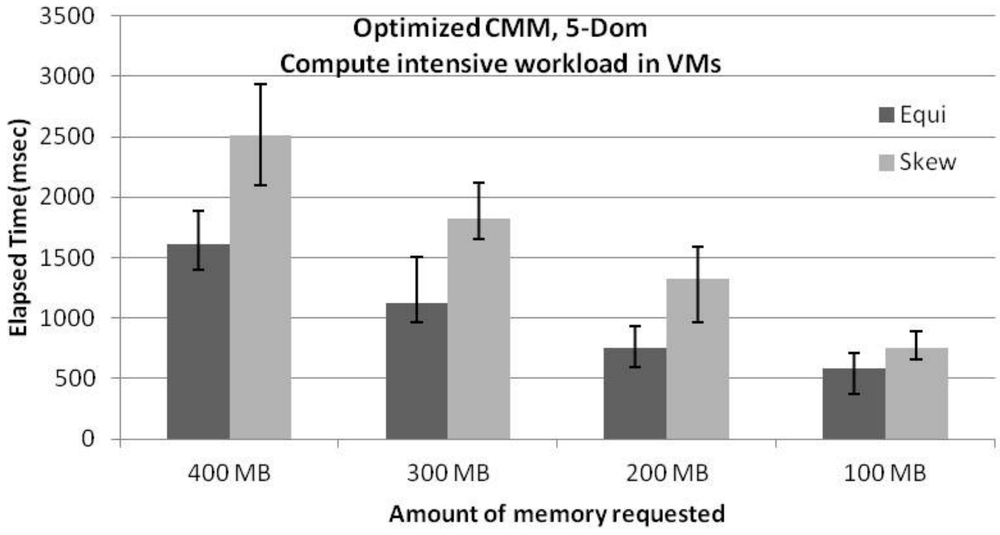
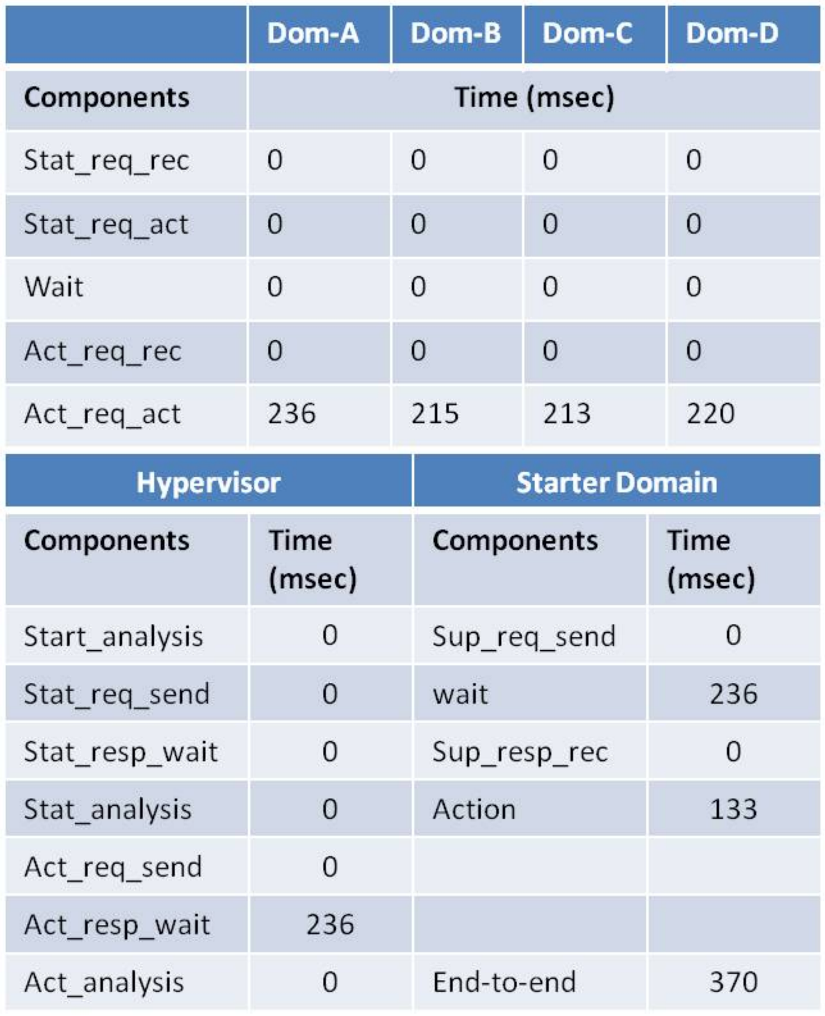


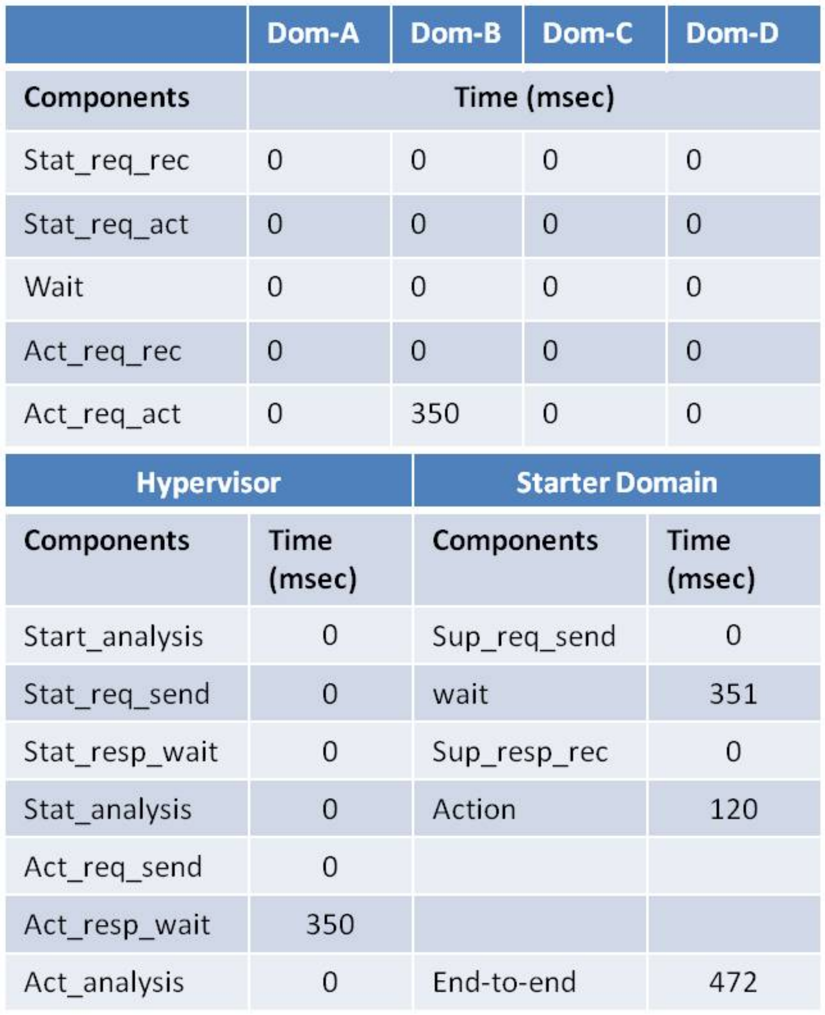
Figure 34: End-to-end latency for optimized hypervisor implementation of CMM in the presence of workload: There is active compute intensive process in the domains (VMs). Experiment varies the amount of memory for a fixed number of 5 VMs.



	Dom-A	Dom-B	Dom-C	Dom-D
Components	**Time (msec)**			
Stat_req_rec	0	0	0	0
Stat_req_act	0	0	0	0
Wait	0	0	0	0
Act_req_rec	0	0	0	0
Act_req_act	236	215	213	220

Hypervisor		Starter Domain	
Components	**Time (msec)**	**Components**	**Time (msec)**
Start_analysis	0	Sup_req_send	0
Stat_req_send	0	wait	236
Stat_resp_wait	0	Sup_resp_rec	0
Stat_analysis	0	Action	133
Act_req_send	0		
Act_resp_wait	236		
Act_analysis	0	End-to-end	370

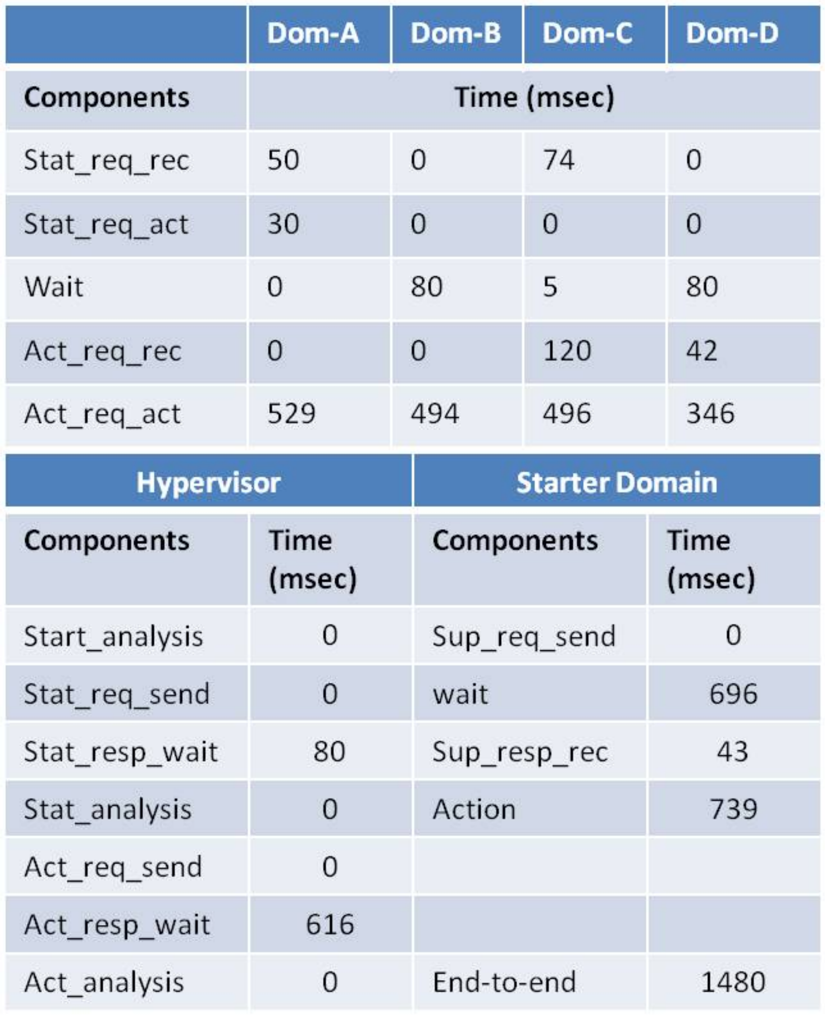
Figure 35: Results of the analysis of time spent in various phases of end-to-end operation of optimized CMM for equi distribution of requested memory and no active workload in the donor VMs



	Dom-A	Dom-B	Dom-C	Dom-D
Components	Time (msec)			
Stat_req_rec	0	0	0	0
Stat_req_act	0	0	0	0
Wait	0	0	0	0
Act_req_rec	0	0	0	0
Act_req_act	0	350	0	0

Hypervisor		Starter Domain	
Components	Time (msec)	Components	Time (msec)
Start_analysis	0	Sup_req_send	0
Stat_req_send	0	wait	351
Stat_resp_wait	0	Sup_resp_rec	0
Stat_analysis	0	Action	120
Act_req_send	0		
Act_resp_wait	350		
Act_analysis	0	End-to-end	472

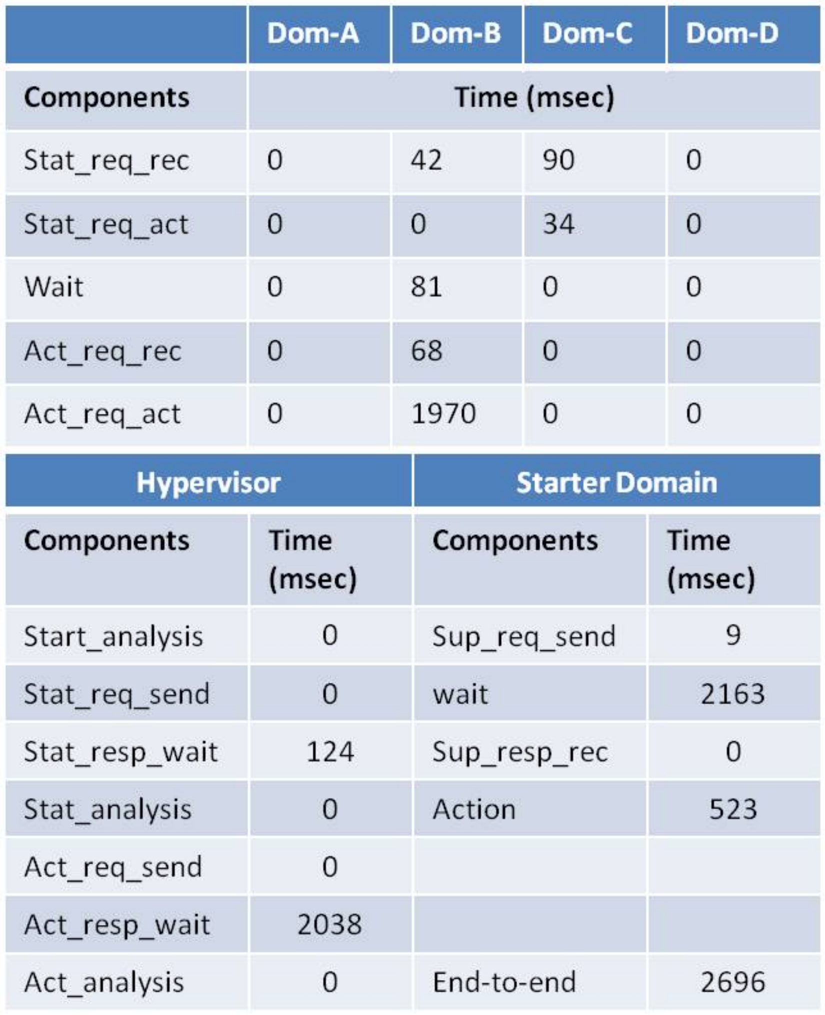
Figure 36: Results of the analysis of time spent in various phases of end-to-end operation of optimized CMM for skew distribution of requested memory and no active workload in the donor VMs



	Dom-A	Dom-B	Dom-C	Dom-D
Components	Time (msec)			
Stat_req_rec	50	0	74	0
Stat_req_act	30	0	0	0
Wait	0	80	5	80
Act_req_rec	0	0	120	42
Act_req_act	529	494	496	346

Hypervisor		Starter Domain	
Components	Time (msec)	Components	Time (msec)
Start_analysis	0	Sup_req_send	0
Stat_req_send	0	wait	696
Stat_resp_wait	80	Sup_resp_rec	43
Stat_analysis	0	Action	739
Act_req_send	0		
Act_resp_wait	616		
Act_analysis	0	End-to-end	1480

Figure 37: Results of the analysis of time spent in various phases of end-to-end operation of optimized CMM for equi distribution of requested memory and in the presence of active workload in the donor VMs



	Dom-A	Dom-B	Dom-C	Dom-D
Components	Time (msec)			
Stat_req_rec	0	42	90	0
Stat_req_act	0	0	34	0
Wait	0	81	0	0
Act_req_rec	0	68	0	0
Act_req_act	0	1970	0	0

Hypervisor		Starter Domain	
Components	Time (msec)	Components	Time (msec)
Start_analysis	0	Sup_req_send	9
Stat_req_send	0	wait	2163
Stat_resp_wait	124	Sup_resp_rec	0
Stat_analysis	0	Action	523
Act_req_send	0		
Act_resp_wait	2038		
Act_analysis	0	End-to-end	2696

Figure 38: Results of the analysis of time spent in various phases of end-to-end operation of optimized CMM for skew distribution of requested memory and in the presence of active workload in the donor VMs

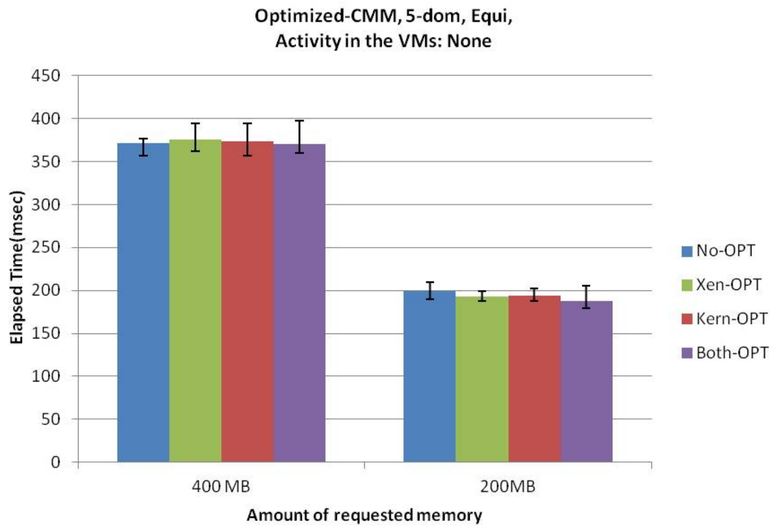


Figure 39: Effect of scheduler optimization constructs on end-to-end operational latency of optimized CMM for equi distribution of requested memory, no active workload in the donor VMs

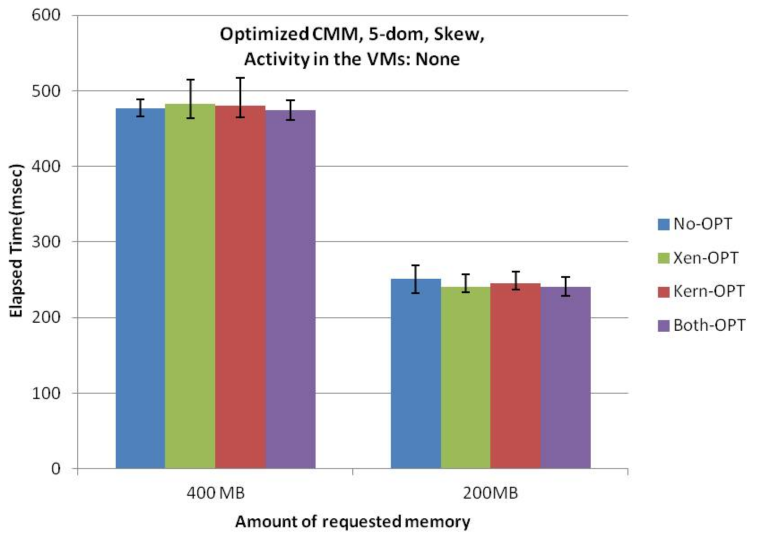


Figure 40: Effect of scheduler optimization constructs on end-to-end operational latency of optimized CMM for skew distribution of requested memory, no active workload in the donor VMs

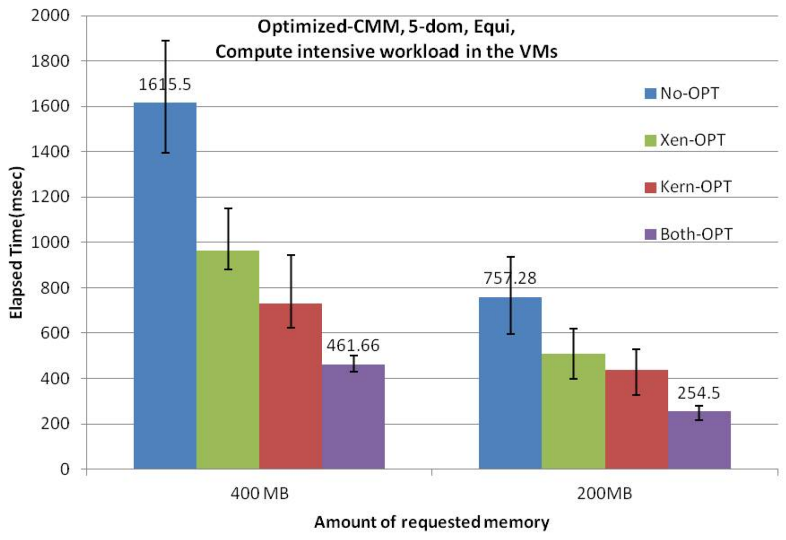


Figure 41: Effect of scheduler optimization constructs on end-to-end operational latency of optimized-CMM for equi distribution of requested memory, with active workload in the donor VMs

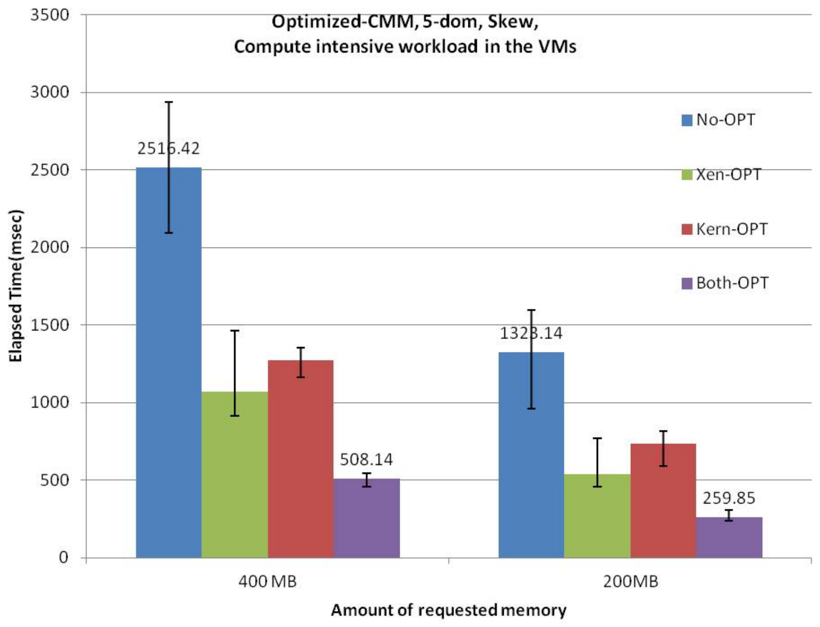


Figure 42: Effect of scheduler optimization constructs on end-to-end operational latency of CMM for skew distribution of requested memory, with active workload in the donor VMs

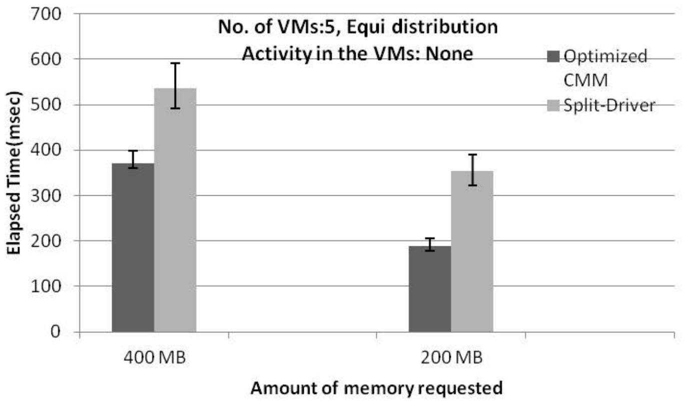


Figure 43: Latency comparison of Optimized CMM *vis a vis* split-driver, equi, no workload

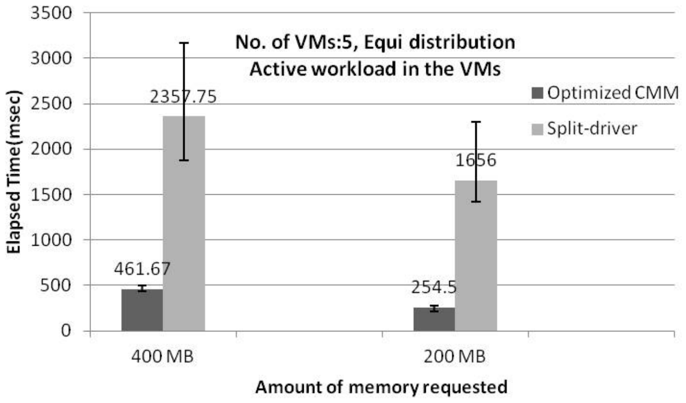


Figure 44: Latency comparison of Optimized CMM *vis a vis* split-driver, equi, with a compute-intensive daemon process in each donor VM

CHAPTER VIII

COMPARISON WITH INTRA-VM MEMORY MANAGEMENT MECHANISMS

Our thesis statement is, it is possible to devise an inter-VM memory management mechanism with a latency within a few factors of the latency involved in intra-VM memory management constructs. With the goal of validating the thesis statement, we are focusing on presenting a comparative analysis of the latency aspects of the intra-vm and inter-vm memory management mechanisms, as the final contribution of this dissertation. We evaluate the user mode memory management mechanisms in various operating scenarios and provide a comparative analysis with CMM that we have discussed over the course of this dissertation.

8.1 Introduction

It is our goal through this dissertation to create a low latency memory management mechanism for virtualized setups. Our primary focus is memory balancing across a set of active VMs and all the discussions up until this point has been for inter-VM memory management and balancing. But, in any system, the end users do not directly interact with the lower layers of the system software stack like the kernel of the operating system or the hypervisor. End users interact with applications and applications use various interfaces like APIs and system calls to interact with other components and layers of the software stack. In the Linux world, 'malloc' and other related routines are the primary memory management related interfaces available to the applications. With our focus being on memory management, from the end user perspective, the key factor is the latency of these memory management related interfaces exposed to the upper layer applications, and the lower layer mechanisms just act as an extension to the the upper layer memory management mechanisms. In this chapter, we provide an analysis of the latency associated with this class of memory management mechanisms, and discuss about how the coordinated memory management mechanism presented in the earlier chapters of this dissertation compares with them with respect to latency.

8.1.1 Why malloc analysis?

When we consider intra-VM memory management mechanisms, it can be either of the following:

- *malloc* and related interfaces like *calloc*.
- memory management interfaces available inside the kernel like *kmalloc*, *vmalloc*
- page allocation routines available inside the kernel like the variants of *alloc_pages* and *__get_free_pages*.

'Malloc', 'calloc' are interfaces available to applications for allocating virtual memory. The memory allocated by malloc is not initialized and calloc does a zero initialized allocation. The allocation happens from the heap of the virtual address space and at the time of allocation all the virtual memory may not be backed with physical pages. Internally these may be using a system call like sbrk(), which just increases or decreases the virtual address space of a process by a specified number of bytes. But sbrk is not a full fledged memory manager and routines like malloc operate in a more sophisticated manner by remembering previous allocation chunks and optimizing certain aspects of the operation of backing up the virtual allocation with physical pages.

kmalloc is the kernel version of the malloc and is an interface for getting kernel memory in byte-sized chunks. It returns a region of memory, that is at least the requested size bytes in length and the region of memory allocated is physically contiguous. vmalloc works in a similar way to kmalloc, but it allocates memory that is only virtually contiguous and not necessarily physically contiguous. The primary difference between kmalloc and vmalloc is: kmalloc guarantees that the pages are physically (and virtually) contiguous, whereas vmalloc only ensures that the pages are contiguous within the virtual address space. vmalloc allocates potentially noncontiguous chunks of physical memory and appropriately modifies the pages table to map the memory into a contiguous chunk of the logical address space. In that sense, the functionality of vmalloc is similar to malloc. But, despite the fact that physically contiguous memory is required only in certain cases, most kernel code uses kmalloc instead of vmalloc (because of performance reasons) and the usage of vmalloc is only confined to situations where very large amount of memory is needed.

The routines alloc_pages, __get_free_pages and some of their variants are at the root of most memory allocation and management mechanisms and provide interfaces for allocating memory pages. Other kernel memory routines directly or indirectly end up calling these page allocation routines.

But, as explained earlier, malloc and its variants are direct interfaces for user mode applications and therefore analyzing the latency involved in their usage is more meaningful from the end-user perspective. In that sense, it is more important to analyze the latency behavior of malloc than those of their in-kernel variants like kmalloc. The other memory allocation and management routines (like page allocation) may be involved in a malloc operation and the subsequent usage of the allocated memory. Lastly, the coordinated inter-VM memory management mechanism (CMM) described over the course of previous chapters in this dissertation internally uses page allocation routines (through the ballooning construct) and we provide latency involved in such operations as part of our time analysis of the various phases of operation of CMM. So due to these reasons, we focus on analyzing latency involved in usage of malloc and related routine calloc in this chapter.

8.2 Evaluation

8.2.1 Experimental scenarios

In this section, we briefly describe the various scenarios for which we present the results in the next sub-section. Given that malloc only does virtual memory allocation without any guarantee of backing the allocation by physical pages, for the malloc operations, we follow by trying to write a byte into each of the pages of the allocated memory. This ensures that the allocation is backed up by physical pages. This is denoted by 'malloc-write' in the figures. Also malloc optimizes its operating latency over a number of allocations by remembering the previous allocations. So an initial invocation of malloc (and the subsequent writes) is invariably of higher latency than the subsequent malloc invocations. To atone for this fact, we only present the measured latency for the malloc and write operations after repeating the similar cycle twice before. This should take care of any warm-up overhead for the malloc. After that, we do a calloc operation for the specified amount of memory and try to read a byte from each of the page of the allocated memory. This is denoted by ‘calloc-read’ in the subsequent figures. This result is to show us the latency for reads as opposed to writes. Next, we showcase the result for the situation when malloc is invoked after file read operations have happened. This is designed to account for the situations when the physical memory is filled with clean pages, and for backing up the virtual allocation, some form of page frame reclamation operation has to happen in the background. In this case, the reclamation does not have to do a lot of write into disk or swap, as the memory is filled with clean pages. The results for this experiment are denoted by ‘malloc-write-fread’. We repeat a similar experiment after doing file write operations to fill up the physical memory with dirty pages, so that page frame reclamation has to deal with disk or swap writes. The results for this experiment are denoted by ‘malloc-write-fwrite’.

8.2.2 Experimental Setup

For measuring the results of the aforementioned scenarios, we conduct two sets of experiments. In the first set of experiments, there is just a Dom-0 running in the physical machine, without any Dom-U. As before, the Dom-0 has a 2.6.36 pvops Linux kernel and is allocated 1 GB memory. The machine configuration remains the same as in earlier experiments. For the second set of experiments, along with the Dom-0, there are 5 Dom-Us also running in the physical machine. This configuration is the same as the configuration for most of the other experiments discussed in this dissertation. In the second set of experiments, we also introduce computation intensive workload into each of the Dom-Us. The workload is the same as used for earlier experiments for measuring the latency of CMM in the presence of active workload.

8.2.3 Results and Analysis

Figure 45 presents the results when the malloc experiments are run in the Dom-0 only, and there is no other workload introduced into the Dom-0. We present the results for allocating and accessing (writing/reading) 400MB and 200MB of memory respectively for the various scenarios mentioned earlier. The results definitely show that the latency involved in operation of intra-VM mechanisms is significant. For 400MB of memory, basic malloc operation (malloc-write) takes around 1000 msec. That is quite higher than the latency of less than 400 msec observed for the optimized CMM in no workload situation (refer Figure 33). In the best case scenario for calloc-read, the latency is comparable to the operational latency of the coordinated mechanism.

The latency for malloc operations is still higher when operated after file read and file write activities. This shows two things: (i) Latency is affected by the overall memory situation, and may usually be higher than the values obtained for basic malloc-write experiments, because, in a normal working setup, file read and write operations constitute a significant portion of the activities in any OS or VM (ii) In a virtualized setup, if the memory situation in any particular VM is not very good (free pages not readily available) and there is memory pressure, it may make sense to try getting memory from other VMs running in the physical machine, with the assumption that there are VMs which are not experiencing memory pressure at that instant and they have enough free memory available.

The results for a similar set of experiments in the presence of other VMs with active workload in them is presented in Figure 46. The results in this case further corroborate our observations from the earlier set of experiments. As expected, the latencies are considerably higher in this case, because of the presence of additional VMs and the workload in them. To put this in perspective, the latencies for the corresponding cases of CMM (refer Figure 41) are usually lower, even when the optimization constructs are not used. The latency for calloc-read in the presence of VMs with active workload is comparable to the latency of CMM for similar memory amounts. The use of optimization constructs tilts the balance much in the favor of CMM. It is important to note that, for all the results in this section, we have used the malloc and calloc functionalities as it is and have not looked into their implementation details for any optimization. Also, a single process (/thread) is used for all the experiments. The results may be different in multi-threaded situations, or if some possible latency optimization constructs are used for the malloc class of routines.

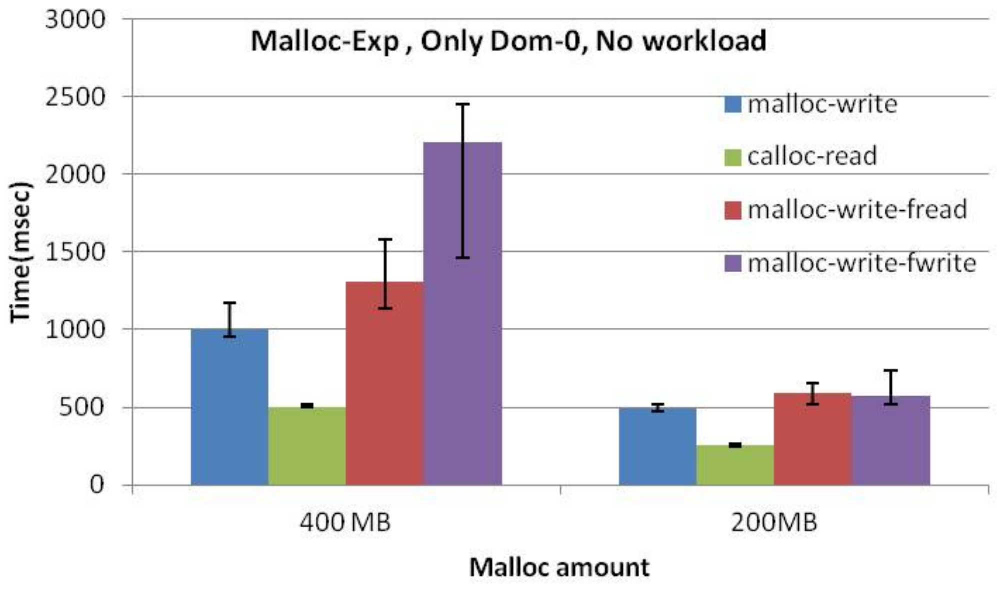


Figure 45: Latency of intra-VM memory management mechanisms like malloc and calloc when operated in the absence of significant workload; only Dom-0 running

8.3 Conclusion

In this chapter, we discussed about the latency involved in the usage and operation of the intra-VM memory management mechanisms, and our analysis validates our initial thesis statement that, operational latency of inter-VM memory management mechanisms is comparable to that of the intra-VM mechanisms.

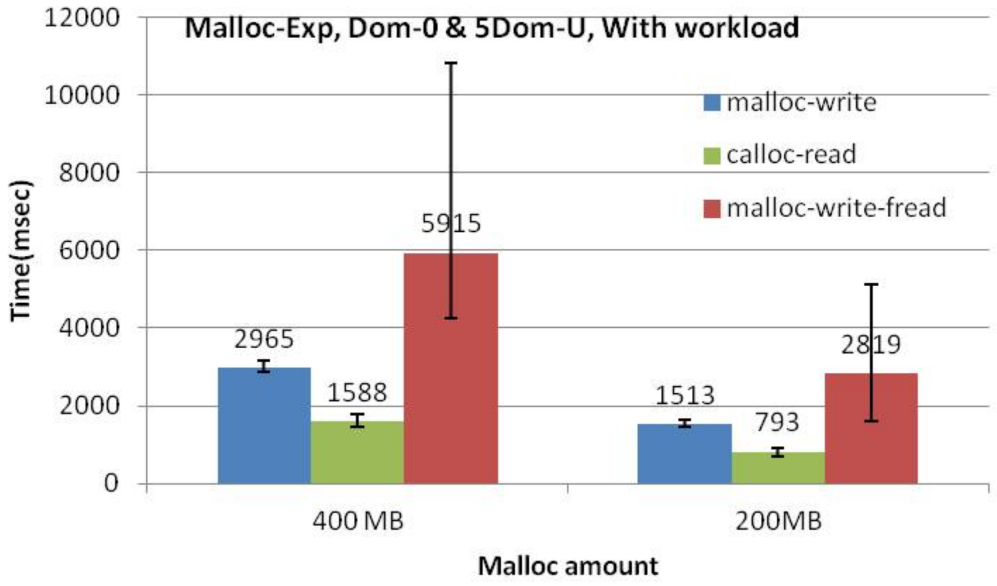


Figure 46: Latency of intra-VM memory management mechanisms like malloc and calloc when operated in the presence of VMs carrying compute intensive workload; Dom-0 and 5 Dom-Us carrying compute intensive daemon

CHAPTER IX

RELATED WORK

The research works related to the discussion in this dissertation can be broadly categorized into various sub-categories.

The first category comprises of the various studies targeted towards design and implementation of memory management schemes through page transfer mechanisms. Ballooning mechanism was introduced in [48] and has been implemented in all subsequent hypervisors (including Xen) in one form or other. We have discussed about ballooning and its variants like self-balooning in detail in this dissertation. Transcendent memory (T-Mem)[33] [34] was designed and implemented to overcome some of the issues associated with ballooning. T-Mem provides a different approach for memory management in a virtualized environment by claiming under-utilized memory in the system, putting them into a central pool and providing a page copy based interface to this pool. The VMs can offload some of the pages to these centralized pool in times of memory scarcity. We have discussed about T-Mem in this dissertation and we found that the overhead associated with page copy based interface render T-Mem unsuitable for catering to situations with a need for low latency. In a related paper [11], Umesh Deshpande et al. have looked into utilizing distributed memory and network paging mechanisms to allow the VMs to use more DRAM than is provided by the host machine. A similar idea has been explored in the context of memory blade servers in [30].Hwang et al., have looked into hypervisor level paging and intelligent page reclamation for memory balancing in [18], but in this approach hypervisor needs to be aware of page cache accesses of VMs. Amit et al., in VSwapper [2], are also focusing on optimizing hypervisor based paging activities, and are specifically focusing on optimization of scenarios where ballooning functionality is not sufficient to resolve the memory pressure situations.

The second category primarily comprises of research works dealing with sharing of memory pages. Waldspurger et al., introduced content-based page sharing in ESX server [48].

The basic idea is to identify page copies by their contents. Pages with identical contents can be shared regardless of when, where, or how those contents were generated. The cost for this is that work must be performed to scan for sharing opportunities. Generally, hashing is used to identify pages with potentially identical contents. A hash value that summarizes a page's contents is used as a lookup key into a hash table containing entries for other pages that have already been marked copy-on-write (COW). If the hash value for the new page matches an existing entry, it is very likely that the pages are identical, although false matches are possible. A successful match is followed by a full comparison of the page contents to verify that the pages are identical. In Difference Engine [15], the authors looked at finding page sharing opportunities at sub-page level. They use patching and compression to achieve greater memory savings than sharing alone. In another related work, Satori [37], the authors have tried to reduce the inherent overhead in page sharing and increase the utilization of page sharing opportunities by using enlightenment in the guest OSes. They use sharing-aware block devices as a low overhead mechanism for detecting duplicate pages. Since a large amount of sharing originates within the page cache, they monitor data as it enters the cache. They have a mechanism which distributes the saved pages to the various VMs in accordance with the amount of pages they have contributed to sharing. In another study [4], the authors find that the amount of page sharing opportunities within a single VM/OS far outweighs the sharing opportunities across VMs and conclude that page sharing opportunities should be explored in non-virtualized systems too.

There is a set of research work dealing with working set size estimation and other works which are built on top of basic page transfer and page sharing mechanisms. Memory resource management was first formalized for operating systems by Denning in 1968, with the introduction of the working set model [10]. The working set of a process at time t is the set of pages that it has referenced in the interval $(t - \tau, t)$. Pages can then be allocated to each process so that its working set can fit in memory. Since it is challenging to calculate the working set exactly, an alternative approach is to monitor the page fault frequency for each process. If a process incurs too many page faults, its allocation of pages is increased and vice versa. In virtualized environments there is a need for calculating the working set size of the entire VM. Zhao et al., have used the LRU-histogram technique to predict the page miss rate and use it to estimate the working set size of a VM [56]. In Geiger [20], the authors detect memory pressure and calculate the amount of extra memory needed by monitoring disk I/O and inferring major page faults. Lu et al., [31] use a hypervisor exclusive cache for calculating the page miss ratio curve of each VM and use that to estimate the WSS of the VM. In this design, each VM gets a small amount of machine memory, and the rest of the memory is managed by the VMM in the form of an exclusive cache. This approach can track all memory accesses above the minimum memory allocation and both WSS increase and decrease can be deduced. Kim et al., [23] have designed and implemented a cooperative caching mechanism for globally managing the buffer caches of the virtual machines running in a physical machine. In their approach, if a block cached in the buffer cache of one virtual machine is decided to be evicted, then it is given a second chance to be stored in the memory of some other virtual machine. The goal is to reduce the number of disk accesses and this mechanism has high degree of similarity with some of the constructs available in transcendent memory.

A section of this dissertation is dealing with interference of scheduling activities in the CMM operation. So, a set of related work consists of research works directed at evaluating various facets of virtualized environments and interference among the VMs and scheduler enhancements targeted at improving certain aspects of a virtualized setup. Koh et al., [25] have studied the effects of performance interference between two virtual machines hosted on the same hardware platform by looking at system-level workload characteristics. Xing Pu et al., [42] study the performance interference between VMs executing on the same hardware platform and running network I/O workloads that are either CPU bound or network bound. Research efforts such as [40, 29, 14] study the impact of VM scheduling on I/O virtualization performance and highlight the need for improved scheduling techniques that help I/O-bound VMs while maintaining fairness. In the research papers [24, 22], the authors have looked into enhancing the scheduling support for multimedia related tasks inside VMs. In a different but related set of papers [52, 54, 53], the authors look at the scheduler modifications necessary for efficient simulations in virtualized environments. In Bubbleflux [51], authors have looked into creating an integrated dynamic interference measurement and online QoS management mechanism with the end goals of providing accurate QoS control and maximize server utilization.

We have looked at the latency associated with ‘malloc’ class of operations in this dissertation. Malloc being a fundamental interface for memory allocation at user level, researchers have looked into the performance aspects of malloc in varied scenarios. Lever et al., [28] have researched about the performance of malloc in multi-threaded environments. The latency numbers presented in this paper are comparable to the numbers we have observed as part of our study and we will be looking into the details of their experimental setups for our future studies about malloc. Publications of Poul-Henning Kamp [21] and Doug Lea [27] are very helpful in understanding the internal implementations of malloc.

CHAPTER X

CONCLUSION AND FUTURE WORK

10.1 Conclusion

There is a gradual proliferation of virtualization into platforms where virtualization has not traditionally been a part of the software ecosystem. Due to the promises of versatility and security, mobile and embedded devices like smart-phones are the most recent addition to the list of environments where virtualization plays a noteworthy role and environments like automobile software system are in the pipeline for introducing virtualization into the software stack. However, this proliferation also redefines the expectations for the various mechanisms in the virtualization layer in order to cater appropriately to the needs of the new breed of applications that are now part of the upper layers of the software stack. Time varying and bursty memory requirements of the applications is one such characteristic that has to be appropriately dealt with by the virtualization layer. We have addressed this issue through the research presented in this dissertation.

We started with by analyzing the memory requirement patterns of some of the prevalent applications in the mobile devices arena, like browsing, email and video streaming etc., and showed the inherent bursty behavior of their memory needs. This finding, and the latency sensitive nature of these applications, calls for low latency memory management mechanisms at various layers of the system software stack. With our primary focus being on the virtualization layer, we designed and implemented a coordinated memory management mechanism (CMM), a novel approach where multiple VMs cooperate among them to come to the rescue of another VM experiencing sudden memory pressure. Subsequently, we also present experimental analysis comparing CMM with some similar memory management mechanisms in the virtualization layer. Our first implementation of the mechanism is in accordance with the split driver principle followed by other para-virtual device drivers in the Xen setup, where the front-ends of the driver are inside the Dom-Us and the back-ends reside in Dom-0. Communication and coordination between the VMs happens by writing into xenstore and sending signals across to the other end by xenbus.

Due to our primary focus on the operational latency of the mechanism, we have done an extensive analysis of CMM in order to identify avenues for latency improvement. Given the distributed nature of the implementation and operation of the mechanism, scheduling policy plays a critical role in the overall operational latency. Based on this hypothesis, we evaluated the extent of interference caused by the scheduling activity in the hypervisor and VM kernels and showed that the interference may be significant when CMM operates with other active workload in the VMs. The primary cause of interference is due to schedulers not being aware of any ongoing CMM operation and not having any means of giving it higher priority over other tasks. We implemented optimization constructs inside Linux credit scheduler and Xen scheduler to minimize this interference and analyzed the effect of these optimization constructs to showcase their benefit.

With a goal of further bounding the operational latency of CMM, we also analyzed the time taken in various phases of operation and found that using xenstore interfaces for information exchange adds unnecessary overhead. Also, the general design of the split drivers and the internal dependency of xenstore on a daemon running in Dom-0, creates a complex scenario for scheduler optimization constructs, where Dom-0 has to be accorded a special status. In order to atone for these pain points, we revised the design and implementation of CMM where we moved the central coordination functionalities out of Dom-0 and put it into the hypervisor. In order to remove the dependency on xenstore, we also modified the communication and coordination constructs. The revised design uses shared memory page based information exchange, and Xen upcall and hypercall based coordination. We provide detailed experimental results to showcase the latency gains due to this revised design. There are two primary takeaways: (i) The revised design provides a latency reduction of of 30% -50% over the earlier design when scheduler optimization constructs are not enabled (latency reduction is higher when scheduler optimizations are enabled) in various experimental scenarios (ii) Due to the relatively simpler and straightforward coordination mechanism and removal of unnecessary dependence on Dom-0, the effect of the optimization constructs is more pronounced for the optimized CMM. More noteworthy is the fact that, in the new design, with the scheduler optimization constructs enabled, the operational latency of CMM in the presence of workload is very near the operational latency in the absence of any workload. This definitely shows the effectiveness of the optimization constructs.

As the final contribution of our dissertation, we measure the latency of intra-VM mechanisms like ‘malloc’ and ‘calloc’ and show that the best case results for these intra-VM mechanisms is comparable to that of the corresponding latency of CMM.

In a nutshell, we make the following contributions through this dissertation:

- we propose a coordinated memory management mechanism that is appropriate for adoption of virtualization in the resource-constrained embedded platforms;
- we have implemented a detailed measurement infrastructure in the hypervisor, which will be useful in of itself for other researchers;
- we propose latency-conscious optimizations in the Linux CFS scheduler framework, which will also be useful in of itself for other researchers;
- we propose latency conscious optimizations in the Xen hypervisor credit scheduler, which will also be useful in of itself for other researchers;
- we report on the design details and performance results of an optimized implementation of our CMM mechanism;
- we provide comparative analysis of CMM with other relevant inter-VM and intra-VM memory management mechanisms to showcase its effectiveness.

Finally it is important to mention that, although the work presented in this dissertation draws its motivation from the low latency needs of the applications prevalent in end-devices, CMM and the latency reduction techniques discussed in this dissertation are equally applicable to server class setups having need for low latency memory management mechanisms. Network simulation frameworks [39, 41] are one such class of applications.

10.1.1 Discussion

The coordinated mechanism (CMM), that has been developed and optimized for operational latency over the course of this dissertation, is built for a very specific purpose i.e., to help a VM experiencing memory pressure alleviate the situation quickly by getting support from other VMs. For this purpose, the system has been designed to enable prioritized execution of the operation of this mechanism, with the assumption that, with quick execution, the overall system can get back quickly to a steady state and for this duration of execution giving priority to the mechanism does not hamper other tasks. But, in certain situations, this assumption may not be true and for those situations, non activation of the optimization constructs should be the norm. Also, the mechanism is effective only when there are VMs which are not experiencing memory pressure at the same instant and are willing to donate memory to the VM experiencing memory pressure. This may not be true in all circumstances and therefore alternate considerations are necessary for dealing with such situations.

In our approach to mitigating scheduler interference, we follow the principle of creating a framework for identifying the entities that need prioritized access to CPU (because of their association with CMM operation) and giving them higher priority over others. Another approach of dealing with this could be through static or dynamic modification of CPU time slices allocated to VCPUs or kernel threads. Theoretically, reduction of the time slices has the potential to reduce the level of scheduler interference because unwanted VCPUs (or threads) are quickly scheduled out. However, it needs to be verified by experimental analysis. Moreover, during the detailed analysis of CMM to quantify the extent of scheduler interference, we observed that the average time amount for which a VCPU runs (on being scheduled) is far less than the time slice amount, primarily because of a quick context switch to that of idle domain in order to deal with interrupts or softirqs. Other researchers have also observed this behavior [32]. Therefore, the actual extent of benefit that is achievable by modification of scheduler time slices is probably not much, and can only be quantified by detailed experimental analysis. A similar argument could be made for the case of Linux kernel schedulers.

Also, in the work presented in this dissertation, we have not specifically looked into the scalability aspects of CMM. Given our focus on the needs from virtualization layer in the context of end-devices, scalability has not been one of our design goals. But the architecture and implementation of both the split-driver model and the optimized CMM do not create any additional constraint from the point of view of scalability. One limitation in the existing architecture is that, it works on satisfying the memory pressure situation of one VM at a time using the coordinated approach. For dealing with situations where multiple VMs may be experiencing memory pressure simultaneously, appropriate design and implementation changes to CMM are necessary. One simple solution is to serialize the memory request (and corresponding resolution steps), but a more flexible solution is desirable and could be a subject for future research.

Finally, we have implemented and analyzed CMM in the Xen hypervisor based para-virtualized setup, due to the flexibility associated with the para-virtualization approach which allows modification of guest VMs. However, the design principle of CMM is transferable to other forms of virtualization frameworks like full virtualization (VMware) or hosted virtualization (KVM). One aspect of CMM that is not readily transferable to these other virtualization environments are the constructs built into the scheduler optimization features for coordination between the hypervisor scheduler and kernel scheduler. This is achieved by making hypercalls at the beginning and end of various phases. There is a need to devise appropriate work around for this issue, depending on the virtulization framework to which CMM is being introduced.

10.2 Future Work

There are quite a few avenues for possible future work building upon the research presented in this dissertation. One possible avenue for future work is to look more closely at the memory page release and reclamation phases of the operation of the coordinated mechanism. These phases could potentially be implemented in a multi-threaded approach, depending on the number of vcpus allocated to the various VMs and the number of cores available in the physical machine. We have not touched upon this aspect in our current research. This could possibly reduce the operational latency by a few factors, depending on the machine configuration and the effectiveness of the multi-threaded algorithm.

Another avenue for further exploration is to look at the internals of the working set size estimation. Simple estimation of the current working set size is not enough from the point of view of fast and dynamic memory balancing. Memory once allocated to a virtual machine is quite difficult to recover, as the operating system (inside the guest VMs) uses it for different purposes, mostly for page and buffer cache contents. Moreover, as has been shown in Chapter 8 (although in a different context), the amount of time required to get a certain amount of memory pages depends on the overall situation of the physical memory at that instant. So, gathering more detailed statistics in terms of the number of free pages, clean and dirty pages etc. and evaluating their impact towards the latency aspects of a coordinated memory management mechanism is an important area for further exploration. In a related research front, development of a system for fast dissemination of these relevant meta-data about the internal memory situation of the various VMs to the central coordinating module is also necessary. Developing a dynamic algorithm to take these various inputs into consideration for coming up with the best possible memory distribution for various VMs can help the mechanism perform optimally in varied setups.

Lastly, as we have shown through this dissertation, in varied scenarios, the latency involved in the operation of intra-VM memory management mechanisms is comparable to the latency involved in an inter-VM mechanism like CMM discussed in this dissertation. Building upon this observation and accurately identifying the reasons behind the latency of malloc operations is important and we will be doing further explorations towards this goal. Also, in the existing software setups, these two classes of memory management mechanisms operate more or less independent of each other and in a non-integrated manner. But in virtualized setups, with multiple VMs running in one physical machine, not all the VMs will experience memory pressure at the same instant. So for such situations, integrating the intra-VM and inter-VM memory management mechanisms may help the VMs alleviate their memory pressure situations in a better way, as compared to the existing approaches of activating the paging out to disks or ending execution of active processes in extreme situations. This integration is one of the necessary steps for showcasing and quantifying the benefits of the memory management mechanisms in the lower layers of the software stack (like CMM) for the applications located in the top layer of the stack. So, this is one avenue of future work which we will be diving into in the immediate future and strive towards showcasing the benefits of CMM for upper layer applications through verifiable results.

REFERENCES

[1] Ackaouy, E., "New cpu scheduler with smp load balancer." *Available at* `http://xen.1045712.n5.nabble.com/New-CPU-scheduler-w-SMP-load-balancer-td2489546.html`.

[2] Amit, N., Tsafrir, D., and Schuster, A., "Vswapper: A memory swapper for virtualized environments," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, (New York, NY, USA), pp. 349–366, ACM, 2014.

[3] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A., "Xen and the art of virtualization," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, (New York, NY, USA), pp. 164–177, ACM, 2003.

[4] Barker, S., Wood, T., Shenoy, P., and Sitaraman, R., "An empirical study of memory sharing in virtual machines," in *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, (Berkeley, CA, USA), pp. 25–25, USENIX Association, 2012.

[5] Bellard, F., "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, (Berkeley, CA, USA), pp. 41–41, USENIX Association, 2005.

[6] Carbone, M. and Rizzo, L., "Dummynet revisited," *SIGCOMM Comput. Commun. Rev.*, vol. 40, pp. 12–20, Apr. 2010.

[7] Cherkasova, L., Gupta, D., and Vahdat, A., "Comparison of the three cpu schedulers in xen," *SIGMETRICS Perform. Eval. Rev.*, vol. 35, pp. 42–51, Sept. 2007.

[8] Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A., "Live migration of virtual machines," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, (Berkeley, CA, USA), pp. 273–286, USENIX Association, 2005.

[9] Creasy, R. J., "The origin of the vm/370 time-sharing system," *IBM J. Res. Dev.*, vol. 25, pp. 483–490, Sept. 1981.

[10] Denning, P. J., "The working set model for program behavior," *Commun. ACM*, vol. 11, pp. 323–333, May 1968.

[11] Deshpande, U., Wang, B., Haque, S., Hines, M. R., and Gopalan, K., "Memx: Virtualization of cluster-wide memory," in *ICPP*, pp. 663–672, IEEE Computer Society, 2010.

[12] dummynet, "dummynet details." *Available at* `https://www.freebsd.org/cgi/man.cgi?query=dummynet&sektion=4`.

[13] Goldberg, R. P., "Survey of virtual machine research," *Computer*, vol. 7, pp. 34–45, Sept. 1974.

[14] Govindan, S., Nath, A. R., Das, A., Urgaonkar, B., and Sivasubramaniam, A., "Xen and co.: communication-aware cpu scheduling for consolidated xen-based hosting platforms," in *Proceedings of the 3rd international conference on Virtual execution environments*, VEE '07, (New York, NY, USA), pp. 126–136, ACM, 2007.

[15] Gupta, D., Lee, S., Vrable, M., Savage, S., Snoeren, A. C., Varghese, G., Voelker, G. M., and Vahdat, A., "Difference engine: Harnessing memory redundancy in virtual machines," in *OSDI* (Draves, R. and van Renesse, R., eds.), pp. 309–322, USENIX Association, 2008.

[16] Hess, K., "Best practices for resource overprovisioning." *Available at* *http://www.zdnet.com/article/best-practices-for-resource-overprovisioning-oxymoron-or-real-thing/.*

[17] Hines, M. R. and Gopalan, K., "Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, (New York, NY, USA), pp. 51–60, ACM, 2009.

[18] Hwang, W., Roh, Y., Park, Y., Park, K.-W., and Park, K. H., "Hyperdealer: Reference-pattern-aware instant memory balancing for consolidated virtual machines," in *IEEE CLOUD'10*, pp. 426–434, 2010.

[19] ipfw, "ipfw details." *Available at* *https://www.freebsd.org/doc/handbook/firewalls-ipfw.html.*

[20] Jones, S. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H., "Geiger: monitoring the buffer cache in a virtual machine environment," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, (New York, NY, USA), pp. 14–24, ACM, 2006.

[21] Kamp, P.-H., "Malloc(3) in modern virtual memory environments.." *Available at* *http://docs.freebsd.org/44doc/papers/malloc.html.*

[22] Kim, H., Jeong, J., Hwang, J., Lee, J., and Maeng, S., "Scheduler support for video-oriented multimedia on client-side virtualization," in *Proceedings of the 3rd Multimedia Systems Conference*, MMSys '12, (New York, NY, USA), pp. 65–76, ACM, 2012.

[23] Kim, H., Jo, H., and Lee, J., "Xhive: Efficient cooperative caching for virtual machines," *IEEE Trans. Comput.*, vol. 60, pp. 106–119, January 2011.

[24] Kim, H., Lim, H., Jeong, J., Jo, H., and Lee, J., "Task-aware virtual machine scheduling for i/o performance.," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, (New York, NY, USA), pp. 101–110, ACM, 2009.

[25] Koh, Y., Knauerhase, R., Brett, P., Bowman, M., Wen, Z., and Pu, C., "An analysis of performance interference effects in virtual environments," in *In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS*, 2007.

[26] KVM, "Kernel virtual machine." *Available at* `http://www.linux-kvm.org/page/Main_Page`.

[27] Lea, D., "A memory allocator." *Available at* `http://g.oswego.edu/dl/html/malloc.html`.

[28] Lever, C. and Boreham, D., "Malloc() performance in a multithreaded linux environment," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '00, (Berkeley, CA, USA), pp. 56–56, USENIX Association, 2000.

[29] Liao, G., Guo, D., Bhuyan, L., and King, S. R., "Software techniques to improve virtualized i/o performance on multi-core systems," in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '08, (New York, NY, USA), pp. 161–170, ACM, 2008.

[30] Lim, K., Turner, Y., Santos, J. R., AuYoung, A., Chang, J., Ranganathan, P., and Wenisch, T. F., "System-level implications of disaggregated memory," in *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2012.

[31] Lu, P. and Shen, K., "Virtual machine memory access tracing with hypervisor exclusive cache," in *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, (Berkeley, CA, USA), pp. 3:1–3:15, USENIX Association, 2007.

[32] Lv, H., Dong, Y., Duan, J., and Tian, K., "Virtualization challenges: A view from server consolidation perspective," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, (New York, NY, USA), pp. 15–26, ACM, 2012.

[33] Magenheimer, D., Mason, C., Mccracken, D., and Hackel, K., "Transcendent memory and linux," 2009.

[34] Magenheimer, D. and others, "Transcendent memory website." *Available at* `http://oss.oracle.com/projects/tmem/`.

[35] Magenheimer, D. J., Mason, C., McCracken, D., and Hackel, K., "Paravirtualized paging," in *Workshop on I/O Virtualization* (Ben-Yehuda, M., Cox, A. L., and Rixner, S., eds.), USENIX Association, 2008.

[36] Monot, A., Navet, N., Bavoux, B., and Simonot-Lion, F., "Multi-source software on multicore automotive ecus - combining runnable sequencing with task scheduling," *IEEE Transactions on Industrial Electronics*, vol. 59, no. 10, pp. 3934–3942, 2012.

[37] Murray, D. G., H, S., and Fetterman, M. A., "Satori: Enlightened page sharing," in *In Proceedings of the USENIX Annual Technical Conference*, 2009.

[38] Navet, N., Delord, B., and Baumeister, M., "Virtualization in automotive embedded systems : an outlook," in *RTS Embedded Systems 2010*, (Paris, France), Mar. 2010.

[39] OMNET, "Omnet suite." *Available at* `https://omnetpp.org/`.

[40] Ongaro, D., Cox, A. L., and Rixner, S., "Scheduling i/o in virtual machine monitors," in *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '08, (New York, NY, USA), pp. 1–10, ACM, 2008.

[41] OPNET, "Opnet suite." *Available at* `http://www.riverbed.com/products/steelcentral/opnet.html?redirect=opnet`.

[42] Pu, X., Liu, L., Mei, Y., Sivathanu, S., Koh, Y., and Pu, C., "Understanding performance interference of i/o workload in virtualized cloud environments," in *Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing*, CLOUD '10, (Washington, DC, USA), pp. 51–58, IEEE Computer Society, 2010.

[43] Schopp, J. H., Fraser, K., and Silbermann, M. J., "Resizing memory with balloons and hotplug," 2006.

[44] Schwidefsky, M., Franke, H., Mansell, R., Raj, H., Osisek, D., and Choi, J., "Collaborative memory management in hosted linux environments," 2006.

[45] Stockhammer, T., "Dynamic adaptive streaming over http –: Standards and design principles," in *Proceedings of the Second Annual ACM Conference on Multimedia Systems*, MMSys '11, (New York, NY, USA), pp. 133–144, ACM, 2011.

[46] vmware, "Understanding memory management in vmware vsphere 5.0." *Available at* `http://www.vmware.com/files/pdf/mem_mgmt_perf_vsphere5.pdf`.

[47] Waldspurger, C. A., "Memory resource management in vmware esx server," *SIGOPS Oper. Syst. Rev.*, vol. 36, pp. 181–194, Dec. 2002.

[48] Waldspurger, C. A., "Memory resource management in vmware esx server," in *OSDI*, 2002.

[49] Wood, T., Shenoy, P. J., Venkataramani, A., and Yousif, M. S., "Black-box and gray-box strategies for virtual machine migration," in *NSDI*, USENIX, 2007.

[50] Wood, T., Tarasuk-Levin, G., Shenoy, P., Desnoyers, P., Cecchet, E., and Corner, M. D., "Memory buddies: exploiting page sharing for smart colocation in virtualized data centers," *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 27–36, July 2009.

[51] Yang, H., Breslow, A., Mars, J., and Tang, L., "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, (New York, NY, USA), pp. 607–618, ACM, 2013.

[52] Yoginath, S., Perumalla, K., and Henz, B., "Taming wild horses: The need for virtual time-based scheduling of vms in network simulations," in *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, pp. 68–77, 2012.

[53] Yoginath, S. B. and Perumalla, K. S., "Efficiently scheduling multi-core guest virtual machines on multi-core hosts in network simulation," *2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*, vol. 0, pp. 1–9, 2011.

[54] Yoginath, S. B. and Perumalla, K. S., "Optimized hypervisor scheduler for parallel discrete event simulations on virtual machine platforms," in *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques*, SimuTools '13, (ICST, Brussels, Belgium, Belgium), pp. 1–9, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2013.

[55] Zhang, I., Garthwaite, A., Baskakov, Y., and Barr, K. C., "Fast restore of checkpointed memory using working set estimation," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '11, (New York, NY, USA), pp. 87–98, ACM, 2011.

[56] Zhao, W., Wang, Z., and Luo, Y., "Dynamic memory balancing for virtual machines," *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 37–47, July 2009.