VMDedup: Memory De-duplication in Hypervisor

Furquan Shaikh, Fangzhou Yao, Indranil Gupta, Roy H. Campbell
Computer Science Department
University of Illinois at Urbana-Champaign
{fmshaik2, yao6, indy, rhc}@illinois.edu

***Abstract*—Virtualization techniques are widely used in cloud computing environments today. Such environments are installed with a large number of similar virtual instances sharing the same physical infrastructure. In this paper, we focus on the memory usage optimization across virtual machines by automatically de-duplicating the memory on per-page basis. Our approach maintains a single copy of the duplicated pages in physical memory using copy-on-write mechanism. Unlike some existing strategies, which are intended only for applications and need user configuration, VMDedup provides an automatic memory de-duplication support within the hypervisor to achieve benefits across operating system code, data as well as application binaries. We have implemented a prototype of this system within the Xen hypervisor to support both para-virtualized and fully-virtualized instances of operating systems.**

I. Introduction

Virtualization has become one of the most important technologies in today's big data era. It forms the fundamental block in cloud computing solutions, such as Amazon EC2 [1] and Windows Azure [2]. Virtualization became essential in such systems, because it makes better utilization of resources and reduces the cost by allowing multiple operating systems to run concurrently on the same physical infrastructure. Multiple operating systems running in their own virtual machines (VM) share the hardware resources, while ensuring high availability for the user applications [3].

In such environments, when there are multiple similar virtual machines sharing the physical machine infrastructure, there arise many opportunities for optimizations. Various techniques for optimizations have been explored with respect to disk scheduling [23], CPU scheduling [22], TCP optimizations [21] as well as storage optimizations [17] in virtualized environments.

However, memory footprint is the single biggest hardware issue affecting performance for common services running in a VM, such as the Apache web server [31]. In order to avoid the latency introduced by frequent swapping, a user has to take memory usage into account and determine the appropriate size for the service process, and leave enough memory for other processes. The approach to solve this problem is simply to add more memory to the physical machine, since it will allow the machine administrator to allocate more memory for each VM, and also it will help the machine hold more VMs. Although this approach is straightforward, simply adding more memory brings additional cost. Thus, this paper focuses on reducing memory footprint of the set of VMs sharing a physical machine.

There are several reasons why memory de-duplication will yield benefits. First, the guest operating systems running over the Virtual Machine Monitor (VMM) or the hypervisor generally include one of the standard commodity operating systems. Second, there are multiple instances of any given operating system running on the physical host at the same time. The code sections for different instances of the identical operating system remain the same throughout the lifetime of their execution. These facts provide us the route for de-duplicating and maintaining only a single copy of the memory pages with copy-on-write (COW) mechanism. Thirdly, common application libraries and binaries are also excellent candidates for de-duplication.

We propose VMDedup, an automatic memory de-duplication mechanism within the Xen hypervisor. VMDedup periodically checks for pages present as duplicates in the physical memory and combines them into a single page. This approach allows us to reduce memory footprint in a virtualized environment running various operating systems like Linux and Windows variants. Our mechanism can be easily extended to any other hypervisor running on bare metal or host operating system.

Our contributions are:

1) We design and implement built-in support within the hypervisor to detect duplicate memory pages across its guest systems.
2) We show that the copy-on-write mechanism is able to de-duplicate memory pages and maintain a single copy of the duplicates.
3) Automatic detection and update of memory footprint at runtime in our design does not need any hints from guest operating systems.

The rest of this paper is organized as follows. We first analyze the memory distribution in a virtual system with various numbers of virtual instance in Section II. Next, we show our design in Section III and the implementation in Section IV for this memory de-duplication system. We then evaluate our system through experiments and discuss the results in Section V. In Section VI, we summarize the related work, and finally we conclude our work and provide future approaches in Section VII.

II. How Much Benefit Can We Get?

In this section, we performed experiments to analyze the memory distribution in virtual environments, and the results obtained showed that there was a large mount of duplicated memory in such environment. We believe that a significant amount of memory gains are possible from de-duplication by concentrating just on the base case of kernel code duplication.

Modern operating systems like Linux and Windows are termed memory grabbers [19]. In a typical virtualized environment, the administrator has control over the amount of memory that is assigned to each instance of VMs. However, when the VM boots up, it grabs all of the configured memory from the hypervisor greedily. It is difficult for the administrator to decide the exact memory, which guarantees a good balance between performance and the resource utilization.

One solution for this over-commitment of memory is a ballooning driver [18]. This driver works in conjunction with the VMM to adapt the total memory actually allocated to the virtual machine in a dynamic fashion. However, this solution results in excessive swapping of memory pages by the operating system (OS) and hence it leads to performance degradation.

In cloud computing platforms, both cloud providers and customers prefer deploying applications over a set of virtual machines running similar operating systems [20]. For instance, this makes it easy for cloud providers in the maintenance of their cloud deployments. Further, it is easier for cloud customers to develop applications with a single type of operating system.

In order to understand the amount of duplicated memory in such environments, we performed several experiments using the Xen hypervisor. Xen is chosen because it is a bare-metal hypervisor, such that there is no host OS involved. It also supports both full-virtualization and para-virtualization [6].

We used Ubuntu Linux 12.04 as both host and guest systems on an AMD Phenom$^{\text{TM}}$II X4 955 Quad-core Processor with 4GB of physical memory. The host system kernel is a modified Xen 4.3 kernel with a new hypercall `HYPERCALL_MEM_VM` implemented to interact with handler to calculate SHA-1 hash over the entire range of physical memory on a per-page basis.

When this hypercall is invoked from the user space, the handler calculates hashes over the physical page frames, which are in use by either one of the operating syetsms or the hypervisor. In order to prevent the hypervisor pages from interfering with hash calculations, only the pages owned by the OS on either the host or guest systems, namely the Dom0 or DomUs, were considered. This strategy ensures that a true value of sharing is captured across the VMs. We captured the memory parameters supplied by the newly implemented hypercall for several operating systems running over the Xen hypervisor.

Figure 1 presents the breakup of physical memory into free, non-shared, zero and shared memory pages. We observe that the number of duplicated pages in physical memory increases significantly as more VM instances start running. The amount of shared memory ranges from 10% of the used memory in the case of two VMs to nearly 35% of five VMs. We also find that a large number of zero pages are marked as used. Those pages are occupied by the OS, but contained only zero bytes. It is true that we can use a single page to maintain those zero pages. However, we focus on shared pages only, and hence zero pages are ignored.

Thus, considering a setup with 4GB of physical RAM and five virtual machines, at least 1GB of memory can be saved by de-duplication. This extra memory is sufficient to run about two additional VM instances, or they can be allocated to current running VMs as necessary. Further, these gains will rise, when we consider similar applications and shared libraries.

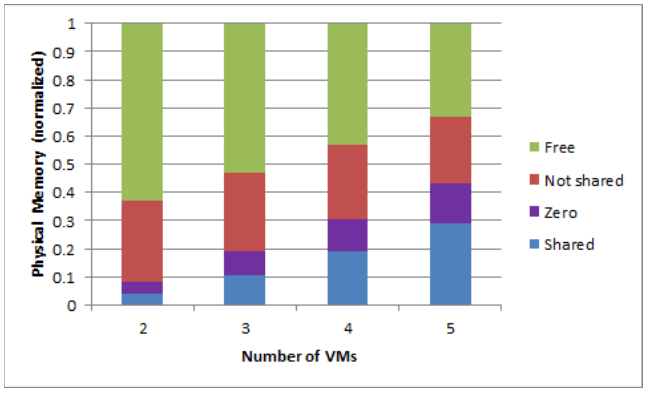

Fig. 1. Distribution of physical memory in virtualized environments

III. Design

The design of VMDedup has three major components. The first component handles identification of suitable candidates for de-duplication. The next component involves strategies for maintaining pages in a de-duplicated fashion. Finally, the last component is the mechanism to check dynamic changes to pages and provide individual copies to their owner. Each of these components is described in the following subsections.

A. *Identification of Candidates*

In a virtualization environment, there are many candidates for de-duplication, such as kernel code, kernel static data, application binaries and application libraries. In addition, some data pages that remain constant for a long time are also good candidates for de-duplication. On the contrary, data pages that keep changing throughout their lifetime are not good candidates.

In order to achieve a good balance between performance and overall memory savings, it is necessary to identify the right candidates for de-duplication and ignore those pages that keep changing constantly. VMDedup uses two different methods, depending upon the type of memory page involved. One of our techniques is for read-only pages in memory, while our second technique is for read-write pages in memory. Kernel code, kernel static data, application binaries and application libraries are contained in pages that are marked as read-only. This read-only attribute exists throughout the execution of the VM. On the other hand, everything other than code and static data is contained in pages, which are marked as read-write, and the contents of the page may change anytime during the execution.

For read-only pages, VMDedup identifies suitable candidates at the time of page-table updates. Whenever a new page is allocated by the operating system, it updates the page tables by making a call into the hypervisor. This call is intercepted in this component of VMDedup to check if the page is read-only and identify if this page is a duplicate of a page that is already in the main memory.

In the case of read-write pages, VMDedup considers only those pages as candidates for de-duplication, which have remained constant for a certain time threshold τ seconds. VMDedup adds the current page to a list of `(page, hash, intervals_unchanged)` tuples for read-write pages upon page-table update for a read-write page by the OS. In order to identify the pages which remain constant after τ seconds, VMDedup periodically utilizes the CPU idle cycles, In order to avoid competing with other processes for system resources, we design this feature as a deferred function with lowest priority.

B. Copy-on-Write Mechanism

This component is responsible for maintaining a single copy of the page in memory for duplicated pages. It is also responsible for ensuring correctness of the system, while multiple owners access the single copy of duplicate pages. Copy-on-write is a common mechanism applied by modern operating systems to share memory regions between different processes [29]. With this technique, an OS maintains a single copy of the shared pages with read-only bit set in the page tables of each of the sharing process. As long as none of the processes attempts to make changes to the contents of the page, the same physical page is used. However, if any of the processes tries to write to the COW page, it results in a page fault. The page fault handler within the operating system then provides a new copy of the physical page to the faulting process. In this way, the OS ensures that one process does not corrupt the address space of any other sharing processes.

We integrate this mechanism within the hypervisor to provide de-duplication of memory across different virtual machines. Once duplicate pages are found, VMDedup updates the address space of the duplicate pages owners, namely the VMs, to point to the same physical page frame and then marks the read-only bit in the page tables of the respective operating systems. VMDedup maintains a list of pages which are present in this de-duplicated fashion. This list is useful for the third component to determine changes to de-duplicated pages.

C. Handling Writes on De-duplicate Pages

This section describes the mechanism used by VMDedup in order to ensure that a write access to a de-duplicated page by one VM does not corrupt the physical memory of other VMs.

As described in the previous subsection, the de-duplicated pages are marked as read-only in the page tables of the relevant operating systems. VMDedup maintains a list of all such pages that are present in de-duplicated fashion. If any VM tries to make changes to a page marked as read-only, the hardware traps into the hypervisor as a page fault. This third component of VMDedup checks to see if the faulting address exists in the de-duplicated pages. The essence is to check if the page fault occurred because of de-duplication mechanism or a memory access violation error. If the page fault occurred as a result of de-duplication, VMDedup copies the contents of the page into a new physical page frame and updates the page table entry of the faulting VM to point to this new page. In this way, VMDedup ensures that a new copy of the page is made for every write access to the de-duplicated page frame in memory.

IV. Implementation

We have implemented VMDedup as a part of the Xen 4.3 hypervisor. Xen was selected as the choice of hypervisor because it is an open-source, bare metal hypervisor that supports both fully-virtualized as well as para-virtualized operating systems to be run as VMs. However, the mechanisms implemented in VMDedup can be easily extended to any other VMM or hypervisor running full-virtualized or para-virtualized operating systems. It can also be used along with shadow page table techniques [24] currently employed by a number of hypervisors, including VMWare.

In this section, we show our implementation details and decisions made for VMDedup.

A. Computing Duplicates by Hashing

Typically, the size of a memory page in most of operating systems is 4KB. Considering that a VM typically has 1GB of memory, it would be expensive to compare the contents of each new page with all the existing pages in the memory to find a duplicate. In order to reduce this overhead, VMDedup uses SHA-1, a standard hashing algorithm [30]. Two pages are considered possible duplicates only if their hash values match. In order to avoid false positives occurring because of hash collisions, VMDedup compares the page contents of possible duplicates to ensure that the page under consideration is actually a duplicate of the other page already present in physical memory. On determination of exact duplicates, the pages are handed over to the second component for actual de-duplication.

B. Red-Black (RB) Tree Based Storage

In VMDedup, all the hashes of pages present in physical memory are maintained in the form of a RB-tree. The reason for choosing RB-tree is two-fold.

First, the RB-tree is a balanced binary tree. Thus, the maximum height of RB-tree at any moment of time is $O(logn)$, and hence the time required to lookup a hash value is $O(logn)$. This decision reduces the overall performance overhead introduced due to the lookup operation.

The second reason for choosing RB-tree is that it is the preferred and standard data structure present in Linux kernel as well as in the Xen kernel. It is used inside of multiple sub-systems of the kernel like process management, disk management to provide an efficient storage and lookup facility. Consequently, standard APIs exist to perform create, insert and search operations on the RB-tree within Xen kernel. Thus, it does not require any additional code to be added into the Xen kernel for the purpose of data structure maintenance, and hence it help the system reduce the overall memory footprint of VMDedup.

C. Code Distribution

The entire code for VMDedup implementation lies within the Xen kernel and it is divided into two parts.

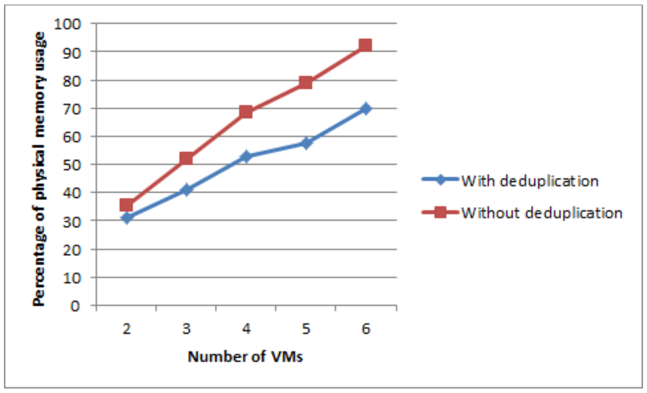

Fig. 2. Amount of physical memory usage with and without de-duplication

The first part of the code is responsible for de-duplicating read-only pages. It is invoked from the page table update handler in the Xen kernel. The page table update handler maps to the `mmu_update` handler function. In case of fully-virtualized operating systems, the hardware traps into the hypervisor and then control is passed to `mmu_update` handler. On the other hand, para-virtualized operating systems make a hypercall `HYPERVISOR_MMU_UPDATE`, which is mapped to the function `mmu_update` handler. Thus, in either case, the control comes to this function, which invokes the test for de-duplication on a read-only page.

The second part of the code is responsible for de-duplicating read-write pages. It executes in the idle cycles of the CPU. This is a deferred function that runs with lowest priority when nothing else is scheduled. Thus, it ensures that it does not use up any cycles useful for work by a VM.

The total codebase for VMDedup is minimal. It occupies fewer than 500 lines of code. This includes the SHA1 functions required for calculation of hash.

V. Evaluation

This section presents the experimental results performed on VMDedup in order to verify the effectiveness of our system. We evaluated VMDedup along two dimensions. One is the overall memory savings achieved by the process of de-duplication. The other is the amount of overhead introduced in the existing system due to the memory optimizations. Each of these results is covered in the following two sub-sections. The experimental setup used for these evaluations is the same as described in Section II.

A. *Memory Savings*

This section describes the overall memory savings achieved in a system using VMDedup. We ran experiments to measure the amount of free memory in the system with and without de-duplication as the number of VMs running over Xen is varied from two to six. Figure 2 shows the results. It depicts the percentage of physical memory in use with and without de-duplication.

We observed that the amount of memory saved by VMDedup varied from 5% for two VMs to around 20% for six VMs.

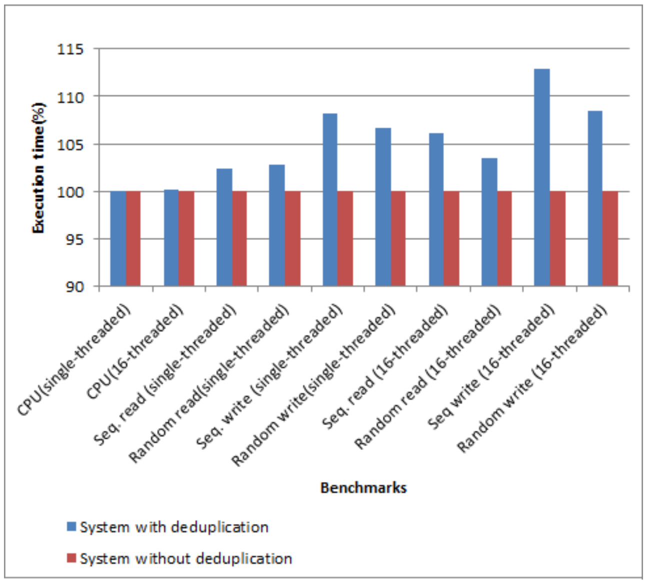

Fig. 3. Performance evaluation of system with and without de-duplication

This saved physical memory could be used to run at least two additional instances of VMs. Thus, VMDedup is able to reduce the overall memory footprint of the virtual machines using the process of de-duplication.

B. *Performance Analysis*

An important dimension for evaluation of VMDedup is the performance overhead introduced by hash calculations, lookup as well as COW technique. It is very important to understand the trade-off between memory savings and the performance loss. This section presents experimental results performed on the same setup as described in the previous section. We used SysBench, a standard multi-threaded application benchmark [32], to identify the performance of the system in two aspects. One is the CPU intensive benchmark and the other is the memory intensive benchmark. All the experiments were run within the unprivileged domain. We conducted experiments using a single-threaded application as well as multi-threaded application. Figure 3 presents the results of all the benchmark test suites on a system with and without de-duplication.

We observed that the overall performance of the system for CPU intensive tasks remains the same. These results reveal the fact that the test suites do not make any significant change in the physical memory sub-system. Thus, the performance for CPU intensive applications is almost the same with or without de-duplication.

The benchmark applications, which perform memory reads and writes, experience 2% to 12% overhead with de-duplication. This overhead is mainly due to the time spent in hash calculations, identification of duplicate pages, as well as the new page allocations and copy of page contents because of COW faults.

An important observation from the above figure is that the overhead for memory read benchmarks is less than the one for memory write benchmarks. This fact arises because memory read benchmarks are subject only to the overhead caused by the page hash calculation and the duplicate page identification. Since the memory pages are only read from, there is no change to the pages during the application execution, and hence no extra overhead. On the other hand, for memory write applications, all the de-duplicated pages that are modified also experience an overhead because of new page allocation and page content copying. Thus, the total overhead for memory write application suites is greater than that for memory read application suites.

VI. Related Work

The current state of art in memory de-duplication for widely used hypervisors or VMM like Xen [4], [6] and KVM [7] is that they do not provide any support for automatic memory de-duplication.

Kernel Samepage Merging (KSM) is a memory de-duplication feature that is built in KVM. It is intended for applications that generate many instances of the same data. However, it requires configurations to be passed from users by calling `madvise` system call to tell the VMM about which address spaces are likely to have duplicate pages [5].

The Xen hypervisor using hardware assisted virtualization allows the OS to indicate what memory regions it wishes to share with some other domain. However, none of the solutions make an attempt to provide an automated support for memory de-duplication.

Previous work proposes the use of a completely new hypercall, a deferrable aggregate hypercall [8], in order to provide support for memory de-duplication within the hypervisor but triggered from user space application. Other work aims at providing memory de-duplication by moving the shared libraries out of the application yet making the application self-contained [9].

Another class of de-duplication effort [10] removes the redundant page caching done at the guest and the host hierarchy in KVM. This solution builds up on top of KSM to provide memory de-duplication. File systems and storage systems have always been a selected target for obtaining optimizations using de-duplication [11], [13]. The same techniques can now be applied to memory de-duplication keeping in mind the issues related to performance and scalability [12], [14], [15], [16].

VII. Conclusions and Future Work

In this paper, we have demonstrated that de-duplication can be used as a powerful tool to achieve significant physical memory savings in cloud computing environments. These memory savings come from duplicate pages that exist in read-only form like kernel code, application binaries and libraries, as well as read-write pages, such as data remaining constant over time.

We have presented a prototype of our system, VMDedup that runs within the Xen hypervisor. This system can be easily extended to any hypervisor managing full-virtualized or para-virtualized VMs. One feature that can be built on top of VMDedup in future is to add support within the block layer of the system in order to identify the pages which are being read from disk and check if the page is already present in physical memory. This feature will save time in reading the duplicate pages from disk into physical memory. Thus, the average time to serve page faults is as good as a system without de-duplication. This method can be used for kernel code as well as data files stored on disk.

VMDedup currently does not include any special code to provide security against attacks to keep the code compact and avoid overhead on the overall system. However, covert channel attacks and defense against such attacks is a well-researched topic. There are many solutions like the BusMonitor [25], partitioned cache architecture [26], virtual firewall [27] and hardware-software integrated approaches [28] that provide security against malicious virtual instances from gaining any information about the co-existing virtual machines. VMDedup can be used with any of these existing solutions to provide de-duplication support along with security against attacks.

Acknowledgments

This material is based on research sponsored by the Air Force Research Laboratory and the Air Force Office of Scientific Research, under agreement number FA8750-11-2-0084.

References

[1] Amazon Web Services, *Elastic Compute Cloud*, 2011 [Online]. Available: http://aws.amazon.com/ec2/

[2] D. Chappell, *Introducing Windows Azure*, 2012 [Online]. Available: http://www.windowsazure.com/en-us/develop/net/fundamentals/intro-to-windows-azure/

[3] T. Burger, *The Advantages of Using Virtualization Technology in the Enterprise*, 2012 [Online]. Available: http://software.intel.com/en-us/articles/the-advantages-of-using-virtualization-technology-in-the-enterprise

[4] S. Spector and Xen.org Community, *Xen Compared to Other Hypervisors*, 2011 [Online]. Available: http://xen.org/

[5] Linux KVM Community, *Kernel Samepage Merging*, 2012 [Online]. Available: http://www.linux-kvm.org/page/KSM

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt and A. Warfield, *Xen and the Art of Virtualization*, ACM Special Interest Group on Operating Systems (SIGOPS) Review. Vol. 37. No. 5. ACM, 2003.

[7] A. Kivity, Y. Kamay, D. Laor, U Lublin and A. Liguori, *kvm: the Linux Virtual Machine Monitor*, Linux Symposium. Vol. 1. 2007.

[8] Y. Pan, J. Chiang, H. Li, P, Tsao, M. Lin and T. Chiueh, *Hypervisor Support for Efficient Memory De-duplication*, IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS). IEEE, 2011.

[9] K. Suzaki, T. Yagi, K. Iijima, N. A. Quynh, C. Artho and Y. Watanebe, *Moving from Logical Sharing of Guest OS to Physical Sharing of Deduplication on Virtual Machine*, 5th USENIX Workshop on Hot Topics in Security (HotSec). USENIX, 2010.

[10] P. Sharma and P. Kulkarni, *Singleton: System-wide Page Deduplication in Virtual Environments*, 21st International Symposium on High-Performance Parallel and Distributed Computing (HDPC). ACM, 2012.

[11] R. Owens and W. Wang, *Non-interactive OS Fingerprinting through Memory De-duplication Technique in Virtual Machines*, IEEE 30th International Performance Computing and Communications Conference (IPCCC). IEEE, 2011.

[12] N. Mandagere, P. Zhou, M. Smith and S. Uttamchandani, *Demystifying Data Deduplication*, ACM/IFIP/USENIX Middleware'08 Conference Companion (Middleware). ACM, 2008.

[13] T. Yang, H. Jiang, D. Feng and Z. Niu, *DEBAR: A Scalable High-Performance De-duplication Storage System for Backup and Archiving*, IEEE 28th International Symposium on Parallel and Distributed Processing (IPDPS). IEEE, 2010.

[14] F. Guo and P. Efstathopoulos, *Building a High-performance Deduplication System*, 2011 USENIX Annual Technical Conference (ATC). USENIX, 2011.

[15] J. Yueh, *De-duplication in a Virtualized Server Environment*, U.S. Patent Application 11/864,756.

[16] P. Efstathopoulos and F. Guo, *Rethinking Deduplication Scalability*, 2nd USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage). USENIX, 2010.

[17] Y. Tsuchiya and T. Watanabe, *DBLK: Deduplication for Primary Block Storage*, IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST). IEEE, 2011.

[18] C. A. Waldspurger, *Memory Resource Management in VMware ESX Server*, 5th Symposium on Operating Systems Design and Implementation (OSDI). USENIX, 2002.

[19] M. Gorman, *Understanding The Linux Virtual Memory Manager*, Prentice Hall, 2004

[20] M. Sheehan, *GoGrid Cloud Survey Report - Operating Systems In The Cloud*, 2011 [Online]. Available: http://blog.gogrid.com/2011/08/01/gogrid-cloud-survey-report-operating-systems-in-the-cloud-part-6/

[21] A. Menon, A. L. Cox and W. Zwaenepoel, *Optimizing Network Virtualization in Xen*, 2006 USENIX Annual Technical Conference (ATC). USENIX, 2006.

[22] S. Govindan, A. R. Nath, A. Das, B. Urgaonkar and A. Sivasubramaniam, *Xen and Co.: Communication-aware CPU Scheduling for Consolidated Xen-based Hosting Platforms*, 3rd International Conference on Virtual Execution Environments (VEE). ACM, 2007.

[23] D. Boutcher and A. Chandra, *Does Virtualization Make Disk Scheduling Passe?*, ACM Special Interest Group on Operating Systems (SIGOPS) Review, Vol. 44. No. 1. ACM, 2010.

[24] K. Adams and O. Agese, *A Comparison of Software and Hardware Techniques for x86 Virtualization*, ACM Special Interest Group on Operating Systems (SIGOPS) Review, Vol. 40, No. 5. ACM, 2006.

[25] B. Saltaformaggio, D Xu and X Zhang, *BusMonitor: A Hypervisor-Based Solution for Memory Bus Covert Channels*, 6th European Workshop on Systems Security (EuroSec), 2013.

[26] D. Page, *Partitioned Cache Architecture as a Side-channel Defense Mechanism*, International Association for Cryptologic Research (IACR), Report 280, 2005.

[27] G. Anthes, *Security in the Cloud*, Communications of the ACM 53, No. 11, 2010.

[28] J. Kong, O. Aciicmez, J. Seifert and H. Zhou, *Hardware-software Integrated Approaches to Defend Against Software Cache-based Side Channel Attacks*, 15th International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2009.

[29] M. J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, 1986.

[30] D. E. 3rd and P. Jones, *US Secure Hash Algorithm 1*, 2001 [Online]. Available: http://www.ietf.org/rfc/rfc3174.txt

[31] The Apache Software Foundation, *Apache Performance Tuning*, 2012 [Online]. Available: http://httpd.apache.org/docs/current/misc/perf-tuning.html

[32] A Kopytov, *SysBench: A System Performance Benchmark*, 2004 [Online]. Available: http://sysbench.sourceforge.net/