IMPACT OF HYPERVISOR CACHE LOCKING ON PERFORMANCE

A Dissertation by

Vidya Suryanarayana

Master of Science, Wichita State University, 2010

Submitted to the Department of Electrical Engineering and Computer Science
and the faculty of the Graduate School of
Wichita State University
in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

December 2013

IMPACT OF HYPERVISOR CACHE LOCKING ON PERFORMANCE

The following faculty members have examined the final copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirement for the degree of Doctor of Philosophy with a major in Electrical Engineering.

_____
Abu Asaduzzaman, Committee Chair

_____
Ravi Pendse, Co - Chair

_____
John Watkins, Committee Member

_____
Mahmoud Sawan, Committee Member

_____
Krishna Krishnan, Committee Member

Accepted for the College of Engineering

_____
Vish Prasad, Interim Dean

Accepted for the Graduate School

_____
Abu Masud, Interim Dean

DEDICATION

To my dearest and loving husband, Raghavendra Salagame, my adorable and precious little son, Prahlad Salagame…I love you both forever and am always indebted to you

ACKNOWLEDGEMENTS

First and foremost, I would like to extend my heartfelt thanks and gratitude to my husband, Raghavendra Salagame, without whom, I would never have been able to start and then be successful in any of my endeavors. His love, patience, optimism, kindness and encouragementhas made my love towards him eternal and I will always be indebted to him. He has always stood by my side in every challenging situation and has held my hand all along the way. I am always grateful to god for giving me another love of my life, my precious little son, Prahlad Salagame, who has made every moment of my life worth living for and every memory worth cherishing. My family has been the driving force behind me to start the doctoral studies and has also been the motivation behind its success. My husband and son are my two eyes and I would never be able to envision myself without either of them.

Next, I want to wholeheartedly thank my parents, for supporting me in every possible way and sculpting me into the person who I am today.I would also like to thank my brother and my extended family for all their help and support.

In my professional life and career, I would like to sincerely thank Dr. Ravi Pendse and Dr. Abu Asaduzzaman. They have always encouraged me along my career and have helped me be a better professional. They have also been great mentors to students and have given me the opportunity to gain research experience and also pursue my doctoral studies with them. They have given me the wonderful opportunity to mentor other Master's students and it has been a very rewarding and satisfying experience for me. I would also like to thank Dr. John Watkins, Dr. Krishna Krishnan and Dr. Mahmoud Sawan for being a part of my committee and steering my

dissertation in the right direction. I am also grateful to them for having reviewed my research and making it more valuable.

I cannot forget to thank another integral part of my doctoral studies, my colleagues and friends at NetApp. NetApp has given me a great opportunity to work in the same field as my research and gain vast knowledge and experience in the new and emerging technology areas. With regards to giving me the above opportunity, I cannot forget to thank Mr. Tony Habashy and Mr. Andrew Lau. They have been great mentors and managers and have always encouraged me in my career. I want to extend my heartfelt thanks to my colleagues and team members in the performance team, who have helped me in gaining more insight into the job and have challenged me with assignments, that has made me think out of the box.

ABSTRACT

Server virtualization has become the modern trend in various industries. Industries are resorting to the deployment of virtualized servers in order to cut the cost of additional, expensive hardware and consolidate the servers on minimal hardware for easier management and maintenance. Virtualized servers are often seen connected to disk arrays in many industries ranging from small to medium business to large data centers. In such a setup, assuring a low latency of data access from the disk array plays an important role in improving the performance and robustness of the overall system. Caching techniques have been researched and used in the past on traditional processors to reduce the number of memory accesses and have proven benefits in alleviating the response times of applications.

The research done in this paper explores caching on the hypervisor and analyzes the performance of data cache locking technique in hypervisor caches. The research aims at reducing the Input / Output (I/O) latency in a server virtualized Storage Area Network (SAN) setup, which thereby increases the performance of applications running on the virtualized servers. The authors introduce a miss table that can determine the blocks of data in the hypervisor cache that need to be locked. Way cache locking method is used for locking, such that only selected lines of cache are locked (not the entire cache). The proposed cache locking technique is later evaluated with the introduction of a small victim buffer cache and probability based cache replacement algorithm. Valgrind simulation tool was used to generate memory traces by virtual machines (VMs). Experimental results show an improved cache hit percentage and a considerable reduction in the I/O response time due to the proposed cache locking technique when compared to the results without hypervisor cache locking.

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

Chapter Page

LIST OF FIGURES

LIST OF FIGURES (continued)

Figure Page

LIST OF FIGURES (continued)

Figure Page

LIST OF TABLES

LIST OF ABBREVIATIONS

I/O	Input/Output
SAN	Storage Area Network
iSCSI	Internet Small Computer System Interface
WCET	Worst Case Execution Time
BAMI	Block Address and Miss Information
CL1	Cache Level 1
CL2	Cache Level 2
IT	Information Technology
CPU	Central Processing Unit
FFT	Fast Fourier Transform
GIF	Graphics Interchange Format
JPEG	Joint Photographic Experts Group

CHAPTER 1

INTRODUCTION

With the arising need to maintain voluminous and ever growing data of modern times, various industries are resorting to use storage area networks (SAN) to store and organize data. SAN provides the servers with access to disk arrays that have terabytes or even petabytes of storage capacity, unlike traditional servers that have directly attached hard disks. Recent advances in computer architecture have paved the way for invention of virtualization techniques like block virtualization, server virtualization, network virtualization, I/O virtualization, storage virtualization and the like. Storage virtualization technology is one of the effective solutions which simplifies Enterprise Data Management with a unified architecture that increases flexibility, scalability and security. Server virtualization has become the foundation of data centers today. Server virtualization enables multiple operating systems also known as virtual machines (VMs), to be run on the same underlying hardware[1]. Hypervisor is the main software component in the virtualized server architecture, which enables server virtualization.

1.1 Motivation

Virtualized servers connected to disk arrays form an integral part of storage area network and are commonly seen these days in a variety of cloud data centers, large organizations, hospitals, social media data centers that demand Terabytes of storage. Disk arrays are isolated from the servers and connected to the servers using protocols such as Fiber Channel (FC), Iscsi, Storage Attached SCSI and the like through which read/write requests to the disk array are transmitted from the applications running on the virtual machines. Applications running on the VMs can be real time or non-real time applications that are time sensitive [2][3]and hence, it is very critical to ensure a low latency of data access from the disk array.

The virtualized servers also known as virtual machines are assigned memory from the Random Access Memory of the physical server. Hence, the greater the RSM capacity of a physical server, the more virtual machines it can support. The virtual machines are allocated a portion of the availablememory for caching purposes. The applications running on the virtual machines constantly access the cache memory to access the data needed by them [1]. Whenever the required data is not found in the cache/memory on the VMs, they read data from the disk array attached to the physical server through protocols such as Fibre Channel, Iscsi, Infiniband and so on. The VMs read and write data to the disk arrays by sending I/Os. Reading data from the disk array is an expensive operation as the request and data has to traverse through the protocol, the spindle and then the disk. In time critical applications, this latency may not be acceptable [4][5]. Therefore, algorithms need to be developed that can help retain the desirable data by the applications in the cache for optimal amounts of time.

Researchers, in the past, have proposed and implemented efficient cache policies in regular, standalone processor systems that can retain important blocks of data in the cache. In those previous researches, the aim was to minimize the number of accesses the processor makes to the internal memory because the data access from the memory was a slower operation when compared to the speed of the processor [6][7][8][9]. In order to bridge the gap between the processor speed and memory speed, researches introduced cache memory, which is typically present on the processor chip. Data is prefetched into this cache from the memory as applications run and access data. Since cache is closer to the processor, time to access data from the cache is much lesser when compared to accessing data from the memory. In this regard, several cache implementations, cache hierarchies, cache replacement algorithms, cache locking techniques were proposed, which could improve the performance of caching as well as minimize the

drawbacks of using the cache.

However, as data storage became more expensive and as industries are resorting to minimize IT expenses, Storage Area Networks, server virtualization, I/O virtualization, cloud computing are becoming more and more popular [10][11]. All these are new technologies, that have gained momentum in the past decade and more research needs to be done in these fields to improve the performance of storage and computing. While previous research has focused on caching in standalone processors, very little work has been done in investigating cache performance in virtual machines [12]. Cache can make a tremendous impact in server virtualized setup and the right usage of cache in VMs can improve the performance of the virtual machines tremendously[13][14].

The authors in this research investigate caching on hypervisors and propose dynamic caching and cache locking techniques that can effectively utilize the hypervisor cache, alleviate the I/O response time, increase the number of cache hits and improve the performance on the whole, in a server virtualized SAN configuration. The authors in their previous research propose caching on the hypervisor instead of caching on the individual virtual servers. In this approach, the hypervisor has control over allocating the cache to the VMs dynamically and on the go. This type of cache allocation makes cache utilization very efficient and allows optimal usage of the available and expensive resource such as cache. In this research, the authors propose algorithms that can improve the performance of such a hypervisor cache.

The authors, in this research, propose cache locking in hypervisor caches, so that, the number of disk accesses is minimized. In order to do this, the authors use a miss table, that can determine the blocks of data in the cache that need to be locked. The authors also propose using a victim buffer cache in order to tight proof the erroneous cache evictions and then propose a probability

based cache replacement algorithm. All these techniques have two main objectives:First, minimize the number of I/Os to the disk array, so that the I/O latency is reduced.

1.2 Research Proposal and Contributions

In this paper, the authors first propose dynamic caching in hypervisor caches and then propose a data cache locking mechanism in the hypervisor cache in order to reduce the number of misses incurred by applications in the hypervisor cache and to reduce the access latency of applications running on the virtual machines. Strategies are proposed on the hypervisor that are used to improve the performance of locking. This paper presents the below major contributions in the field of server virtualization and caching:

1. A hypervisor cache is proposed in which caching on the virtual server is eliminated and caching is at the hypervisor level.
2. Dynamic Caching using thin provisioning is proposed in the afore mentioned hypervisor or cache
3. A miss table is proposed for the locking algorithm. The information in the miss table is used to determine the blocks of data in the hypervisor cache to be locked.
4. A cache locking algorithm for the hypervisor cache is proposed. The locking mechanism proposed is way locking, where certain cache lines are locked and not the entire cache. The cache locking scheme reduces the number of misses incurred by an application running on the VM. Less number of misses results in reduction in the number of disk accesses in a SAN.

1.3 Dissertation Organization

This dissertation is organized as follows. Chapter 2 describes the related work and previous work that is done in the area of cache, cache locking and hypervisor caches. Chapter 3 describes

the cache memory organization in traditional processors and in server virtualized systems. The cache memory organization is considerably different in both the traditional processor and server virtualized processor architectures. The chapter also describes the cache replacement policies and the hypervisor cache architecture. Chapter 4 describesthe preliminary work done by the authors, based on which they proposed the research done in this work. In Chapter 4, the authors propose different types of cache locking in multi core processor caches and evaluate the performance of locking. Chapter 5 explains the proposed cache locking techniques and the best practices in hypervisor caches. The chapter gives details on the proposed hypervisor cache, dynamic caching in the hypervisor cache, hypervisor cache locking mechanism, the victim buffer and also the updated cache replacement algorithm.

Chapter 6gives information on the simulation details, the test bed and workloads used, obtaining memory traces of the VMs and the input/output parameters used. Chapter7 shows the results obtained and discussion of the obtained results. The results are obtained by varying cache parameters such as cache size, associativity in the presence/absence of the victim buffer, cache locking and cache replacement algorithm. The chapter also shows the variation of the total response time when cache locking is enabled and disabled. Chapter 8 gives the conclusions and future work.

1.4 Conference and Journal Publications

The research presented here is based on and built upon many previous research done by the authors in the area of server virtualization and caching. A part of the work was published in the Local Computer Network (LCN) conference in October 2012, which was held in Clearwater, Florida. The work published by the authors in their previous research is given below.

Journal Papers / Submitted/ Under Preparation

- V. Suryanarayana, A.M.Shamasundar, K.M. Balasubramanya, A. Asaduzzaman and R. Pendse, " Performance of Cache Locking in Hypervisor Caches, " In preparation
- A. Asaduzzaman, V. Suryanarayana and F. N. Sibai, " Innovative Level-1 Cache Locking Strategies for Multi-Core Architectures, " Journal of Computers and Electrical Engineering, Apr 2013

Conference Papers

- V. Suryanarayana, A. Jasti and R. Pendse, " Credit Scheduling and Prefetching in Hypervisors using Hidden Markov Models, " LCN Conference, Oct 2010
- V. Suryanarayana, K.M. Balasubramanya and R. Pendse, " Cache Isolation and Thin Provisioning of Hypervisor Caches, " LCN Conference, Oct 2012
- V. Suryanarayana, H. Dhanekula, A. Asaduzzaman and R. Pendse, " Desktop Virtualization in Cloud and BWT Compression, " PDCS Conference, Nov 2012
- A. Asaduzzaman, V. Suryanarayana and M. Rahman, "Performance Power Analysis of H.265/HEVC and H.264/AVC running on Multi-Core Cache Systems, " 2013 Intl Symposium on Intelligent Signal Processing and Communication Systems

CHAPTER 2

LITERATURE SURVEY

B. Singh [3], studies various caching schemes on the KVM hypervisor and discusses the advantages and disadvantages of double caching in virtualized environments. The paper proposes some solutions to overcome the disadvantages of caching on both the hypervisor and virtual machines. The author first proposes a mixed caching approach where caching is pushed on the hypervisor side and the cache on the VMs is monitored and shrunk frequently. Another algorithm is proposed wherein a ballooning driver is cooperatively used to control the page cache.

H. Chen et al [4] identify memory as a bottleneck of application performance in virtualized setup. They propose a scheme called REMOCA (Hypervisor Remote Disk Cache) in order to reduce the number of disk accesses, thereby reducing the average disk I/O latency. Caching in traditional processors has been researched extensively in the past and has proven benefits in saving the CPU cycles by caching the frequently accessed data. In many high performance computer architectures, accesses to the memory are cached in an attempt to compensate for the relatively slow speed of main memory .

Caching on hypervisors is relatively new and very little research has been done in this area of server virtualization. P.Luet al [5] mention that prediction of miss rate at the VMs is very beneficial for efficient memory assignment to the VMs. They also mention that such a prediction is very challenging because the hypervisor does not contain any knowledge about the VM memory access pattern. The authors propose a technique that the hypervisor uses to manage the part of VM memory so that all accesses that miss the remaining part of the VM memory can be traced by the hypervisor. In this mechanism, the hypervisor manages its memory for exclusive

caching, i.e., the pages that are evicted from the VMs are cached. The pages entering from the VM and not from the secondary storage are cached. They use the ghost buffer technique to predict the page miss rate on VMs where the memory is beyond its current allocation.

X. Vera et al [6] have done a compile time cache analysis with data cache locking to examine the worst case execution time analysis. According to the authors, in real time systems, cache locking trades the performance of the cache for predictability of the cache. They mention that cache behavior is very unpredictable in real time systems and are effective only when the programs exhibit sufficient data locality in their memory accesses. The analysis done at the compile time assists in increasing the predictability of caching in real time systems and the approach can be effectively applied in actual architectures.

V. Suhendra and T. Mitra [7] have conducted research on the caching schemes for shared memory multi cores for real time systems. They have developed and evaluated various design policies for the shared L2 cache by means of static/dynamic locking and task/core based partitioning. They indicate that having a shared L2 cache in multi core systems is very advantageous because each core can be assigned the required amount of cache as per the core's requirement and multiple cores can quickly access the shared data. However, when real time applications are running on the cores, it is very critical to efficiently schedule and share the L2 cache because of strict timing requirements. The authors use cache locking and cache partitioning mechanisms to perform a static analysis on the shared cache. They use cache locking to load selected contents into the cache and prevent those contents to be replaced at runtime. They use cache partitioning to assign a portion of cache to each task and restrict cache replacement on each individual partition.

Abu Asaduzzamanet al [8] have studied cache performance in an embedded system running

real time applications. They indicate that modern computing architectures running real time applications should provide high system performance and high timing predictability. They further mention that most of the unpredictability in the modern computing systems comes from caching and is a critical entity that needs to be examined for running real time applications. The authors propose a miss table based cache locking scheme at level 2 (L2) cache in modern day embedded systems that can improve the timing predictability and the power ratio of the processor. Information in the miss table is used by the authors to selectively lock blocks of data that could improve the timing predictability and reduce the power consumed by the system. The authors mainly focus on embedded systems because they identify the power dissipation on the die to be a critical component that needs attention.

AbhikSarkaret al [9] have studied the effects of cache locking in multi core systems running real time applications. They mention in their research that locking the cache improves the predictability of cache access behavior of a hard real time task. However, real time tasks have small cache footprints and low intra task conflicts and they obtain a shorter worst case execution time by using locks. In cache locking, execution of a task assumes cache hits for locked lines, but when there are a lot of task migrations, locking the cache does not do any good. The authors identified locked cache line mobility as a hindrance to task migration in real time systems. The authors, in their work, have focused on migrating the locked cache lines and providing a predictable task migration scheme to the scheduler.

M.Juet al [10] have done a performance analysis of caching technique and analyzed the effectiveness of caching in single core multi-threaded processors. The cache was used for IP route caching. Their simulation and results show that, when the miss rate is less, caching can be very effective. But as the miss rate increases, the efficiency of caching decreases.

J.Irwinet al [11] have studied the performance of instruction cache in multimedia devices. They state that if instructions are evicted from the cache by competing blocks of code, the running application will take significantly longer to execute than is the instructions were present. In the paper, the authors let the compiler allocate the cache and also automatically configure instruction caches that can improve the overall predictability of multimedia systems. The authors propose to use a partitioned cache for this purpose that guarantees determinism in the instruction cache.

V. Suryanarayana et al [13] propose a dynamic caching algorithm for the VMs based on thin provisioning technique. In their work, the authors propose isolating the cache from the individual VMs and allowing the hypervisor allocate cache to the VMs on-the-go based on the needs of the VM. This technique makes efficient utilization of an expensive resource such as a cache and also improves the response times of the VMs in processing the application requests.

V. Suryanarayana et al in [18] have proposed a shared cache on the hypervisor and a prefetching algorithm to prefetch the blocks of data into the shared hypervisor cache. The authors implemented a Hidden Markov Model to predict the data blocks that have a high probability of access after the current I/O. The blocks with the highest probability of read are brought into the shared hypervisor cache. Their results show that prefetching of data blocks into the hypervisor cache improves the response time of the VMs. The authors considered a para virtualized setup in their work and propose changes to the credit scheduling algorithm. In summary, the authors try to alleviate the response time involved in a server virtualized setup by decreasing the scheduling latency when credit scheduler is involved and also the data fetch latency by proposing prefetching.

H. Kim et al [19] propose a cooperative caching mechanism known as “XHive”inservervirtualized setup. In their proposed technique, the virtual machines collaborate

with each other and maintain a shared copy of the data blocks, which prevents the need to fetch redundant blocks of data from the disk array over and over again. In a server virtualized setup in SAN, the I/O operations are more expensive than the native system due to virtualization overheads. According to their scheme, a singlet, which is a block cached solely by the virtual machine, is preferentially given more chances to be cached in the machine memory. Although the authors demonstrate through experiments that, XHive considerably reduces the number of I/Os and improves the read performance, XHive is limited in usage to the following assumptions: 1) The VMs share the underlying storage and 2) The number of cooperative cache hits are way higher than the storage accesses. Sharing the storage is usually seen in clustered environments and is not very widely deployed. The cooperative caching strategy is expected to efficiently operate for read intensive workloads and the authors have evaluated their strategy on mostly read only/ read intensive workloads.

While the above researches have mainly focused on investigating cache performance in virtualized servers, not much research has been done in improving the cache in virtual machines by implementing locking. Most of the work that have analyzed cache locking performance are implemented on traditional processor caches and have shown significant benefits. Some of the major contributions in cache locking in traditional processor caches is outlined below.

P. Thierry et al [21] investigate the impact of Worst Case Execution Time analysis of caches on security. The authors mention that such a WCET analysis leads to security breaches in task partitioning and can result in huge overheads, making the caches useless. In order to address the security concerns, the authors propose a cross layer approach for a secure management of L1 cache by the scheduler. In their paper, the authors have not considered shared data caches or multi-level caches.

J.Liedtke et al in [25] describe an OS-controlled application-transparent cache-partitioning technique that can transparently assign tasks to partitions. The authors also study the worst case impact of asynchronous cache activities on tasks whose ideal execution times are known. In order to overcome the limitations of traditional cache partitioning techniques, the authors use memory coloring techniques.

While the above research on cache and locking has been done and tested on traditional processor caches, there has been no significant work done in investigating the performance of cache locking in server virtualized environments, especially hypervisor caches. In this paper, the authors study the performance of data cache locking when implemented on hypervisor caches. A miss table is introduced on the hypervisor in order to determine the cache blocks that need to be locked. Cache locking is a technique in which certain cache lines are locked, so that they cannot be replaced from the cache. The data blocks that are locked can still be accessed by applications. Cache locking in hypervisor caches locks those blocks of data that incur the most number of misses by the applications running directly on top of the hypervisor cache. Hence, cache locking is intended to decrease the miss percentage of cache blocks and increase the number of cache hits. The authors later evaluate the cache locking technique in the presence of a victim buffer cache and a probabilistic cache replacement algorithm when compared to the traditional LRU. Experimental results show that cache locking in hypervisor cache improves the performance of caching by decreasing the miss percentage and reducing the total I/O response time.

CHAPTER 3

CACHE MEMORY ORGANIZATION

3.1 Cache Mapping

In traditional processors, the data is prefetched from the main memory into the processor cache, so that applications can quickly access the important data. The data from memory is mapped and stored in certain lines in the cache. This mapping is done in 3 ways: Direct mapping, Fully Associative Mapping and Set Associative Mapping. In direct mapping, a memory block with address '1' is mapped to cache line '1'. Whenever, the cache line '1' is full, a block of data needs to be evicted from the cache in order to hold the new data in cache line '1'. Data eviction from a cache follows the rules of the cache replacement policy in place. Direct mapping takes place as shown in Figure 1.

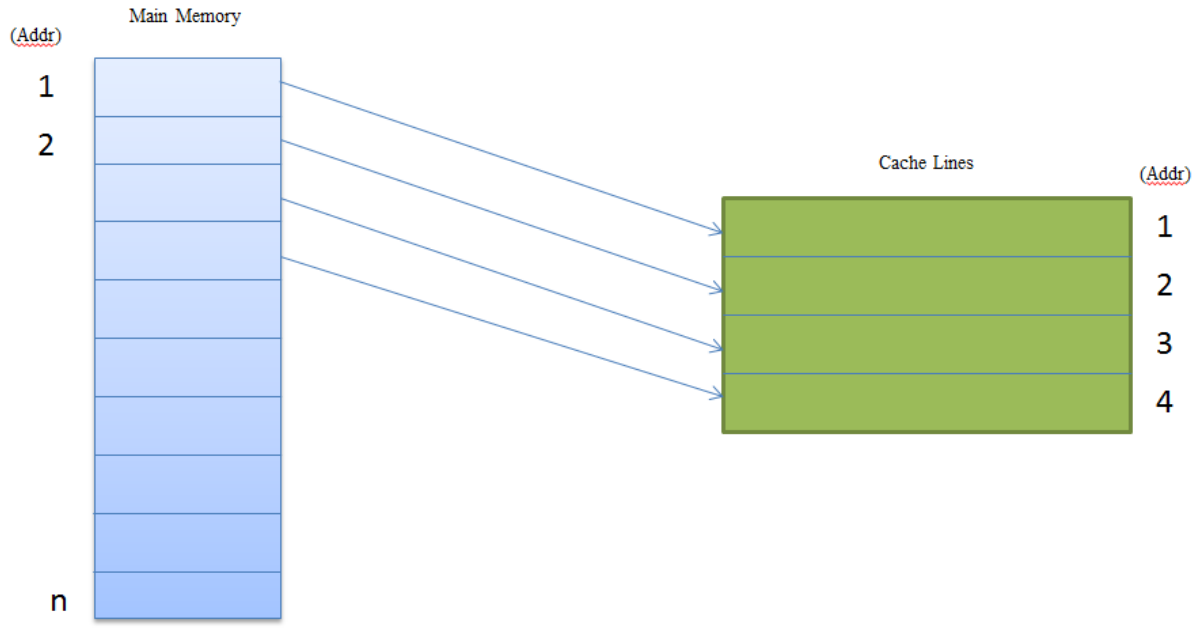

Figure 1: Direct Mapped Cache

In fully associative cache mapping, a memory block can be stored in any cache line. When a cache block is requested, the entire cache must be searched in order to return the requested cache block. This is as shown in Figure 2.

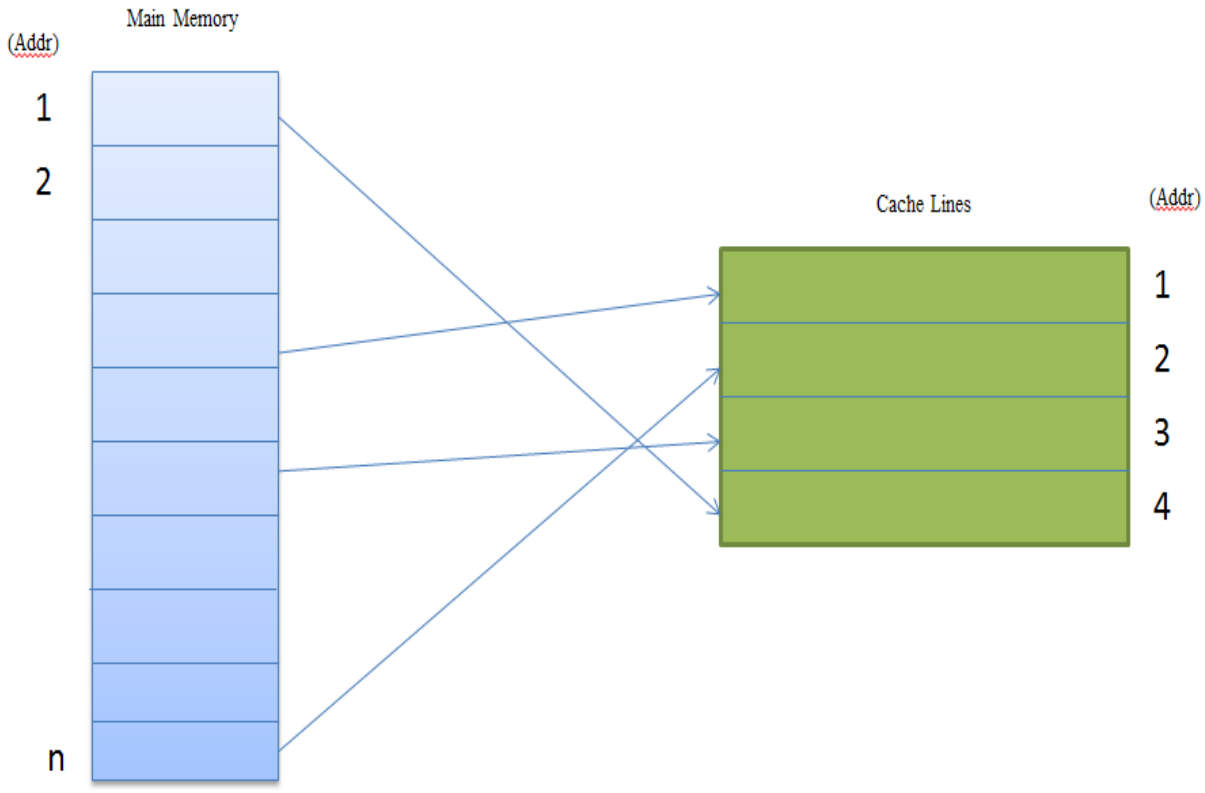

Figure 2: Fully Associative Cache

In a set associative mapped cache, a cache line is divided into many sets. A block from memory can reside in any of the sets in a cache line. Hence, when a block of data is requested from a set associative cache, an entire set needs to be searched to find the cache block. A set associative cache results in lesser seek time when compared to a fully associative cache, but results in a longer seek time when compared to a direct mapped cache.

3.2 Cache Memory Organization

The memory and cache organization in virtualized servers is slightly different from the traditional server memory organization. In the former, a lot of virtualization is enabled, which would make a lot of physical server resources such as memory, cache, CPU virtualized as well. In traditional servers, the amount of memory in the processor is dedicated to the entire physical server. The number of CPU cores in the physical processor is assigned entirely to the server [28]. However, in a server virtualized setup, where there are many virtual servers running on a single physical server, the underlying physical resources such as memory, CPU.etc..have to be shared by the virtual servers. The hypervisor, which is the important entity that enables server virtualization, is responsible for allocating the resources to each of the virtual servers.

In the non-virtualized setup, another important memory organization to consider is the cache organization [31][33]. Depending on the processor architecture, a physical processor may have two or higher number of cache levels, such as L1, L2 and L3 cache. The cache is mainly used by the processor for fetching and storing blocks of data that are frequently accessed by applications running on the physical server. The cache in a non-virtualized system was mainly designed to bridge the speed gap between the memory and processor. A data fetch from the memory is considered more expensive in terms of CPU cycles, when compared to the data fetched from processor cache. The data is prefetched from the memory and stored in cache for the processor to access. Figure 3 shows the memory organization in a non-server virtualized setup.

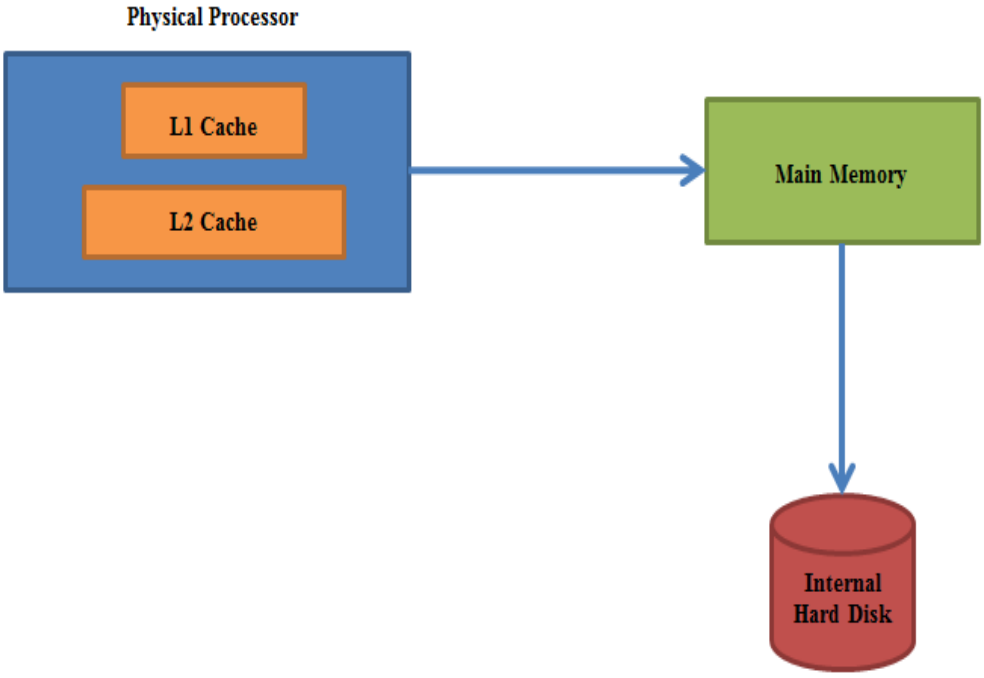

Figure 3: Memory Organization in Non-Virtualized Setup

In a server virtualized setup, the cache organization is very different. Here, the virtual machines are assigned memory from the available RAM on the physical server. A part of memory that is assigned to the VMs is used for caching by the applications running on the VM. The physical cores are virtualized and assigned to each VM. The research conducted in this work is related to the cache that is used by the virtual machines and not the physical cache that is present on the physical server. Figure 4 shows the memory organization in a server virtualized setup.

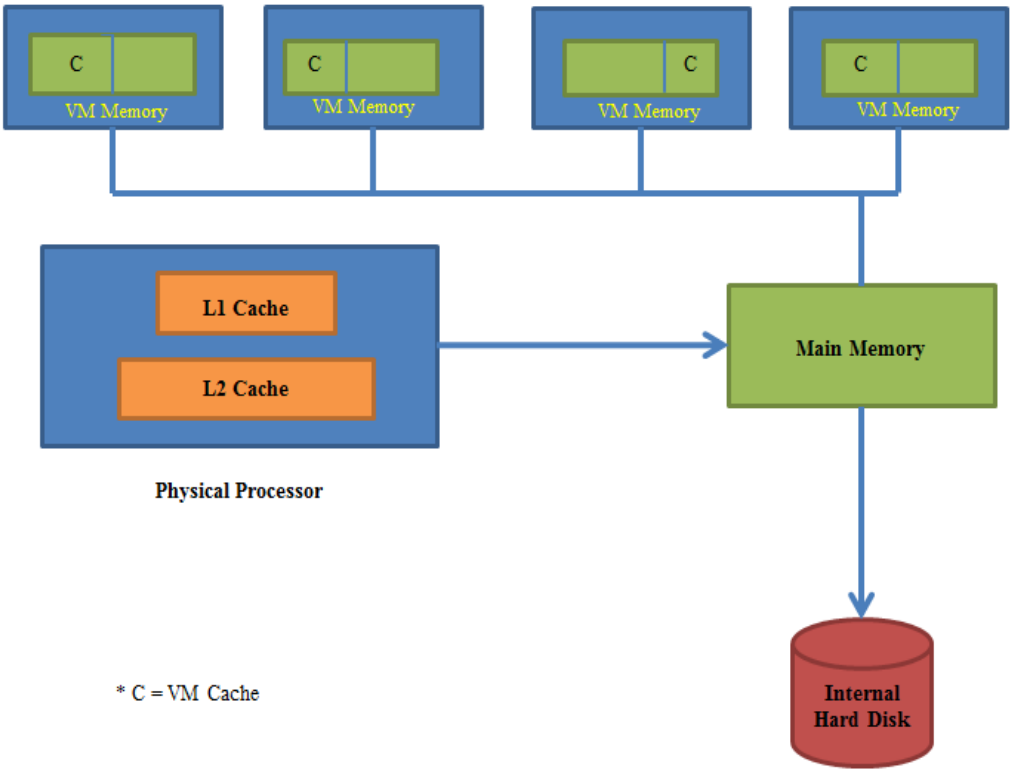

Figure 4: Memory Organization in Server Virtualized Setup

3.3 Cache Replacement Policies

The cache is a very small, expensive piece of memory. Because of its size, it cannot accommodate all the blocks of data that applications require. Once the cache becomes fully populated, certain blocks of data need to be evicted from the cache. Usually, the unwanted blocks of data are evicted so that useful blocks can be brought into the cache. It is very important that the right decision be made about what data is unwanted and what data is useful. A cache replacement algorithm or the policies set on the cache for replacement determine the above criteria. The commonly used cache replacement algorithms are:

1) LRU - where the least recently used block in the cache is evicted
2) MRU – where the most recently block is evicted

3) LFU – where the least frequently used block is evicted

4) Random – Where a block is randomly evicted out of the cache

The cache replacement policies set and the cache eviction algorithms determine the performance of caching.

3.4 Cache Locking

The performance of caching is measured by the number of cache hits an application encounters. When the right data blocks are cached, the number of cache hits increase and the effectiveness of caching increases. Hence, techniques and algorithms need to be applied on the cache, that can increase the number of cache hits. Various such techniques and algorithms have been proposed for a traditional processor cache [32][34]. Cache locking is one of the techniques used to increase the number of cache hits. Cache locking refers to locking certain portions of the cache so that the locked cache lines are not replaced from the cache. However, the locked blocks of data can still be accessed by applications running on the server. A technique similar to cache locking is used in Oracle databases. This technique known as "pinning" refers to keeping database packages in the shared pool of Oracle database's System Global Area [16].

There are two main types of cache locking namely, entire locking and way locking. In the entire locking technique, the entire cache is locked and the blocks of data in the cache cannot be replaced. In the way locking technique, not the entire cache is locked, but some lines in the cache are locked. Entire locking provides performance benefits for small applications while way locking provides performance benefits to large applications. The objective behind cache locking is that, by locking the most needed data blocks or the frequently accessed data blocks, there is a high probability of finding the required blocks of data in the cache and the number of cache hits

can be increased. In other words, cache locking tries to achieve a balance between the cache size and the number of cache hits an application encounters.

3.5 Hypervisor Cache

In the present day, caching in SAN is done on the disk array controller and on the VMs [3][17]. The physical machine hosting the VMs is connected to the disk array using protocols such as Fiber Channel, SAS, Infiniband and the like. The disk arrays are basically a set of hard disk drives and array controllers. Logical Units (LUNs) are created on the disks in the array that are presented to the physical host as virtual storage volumes. The disks from the array are abstracted to form a virtual disk pool and each VM is allocated storage from the virtual disk pool [14]. The VMs use a portion of the allocated storage space for caching [13][14]. The cache on the VMs is first searched by the applications running on the VMs. If the required data is not present in the VM cache, then, the disk array controller cache is searched for the required data. If a cache miss occurs even in the controller cache, then the data block is fetched from the disk.

Accessing the data from external disk array introduces latency for the applications running on the VMs. On the other hand, caching on the VMs can result in inefficient cache utilization [13]. In order to make efficient utilization of the available cache, the authors proposed isolating the cache from the VMs and instead caching on the hypervisor. The authors in [1] showed that sharing the cache among the VMs deteriorates the performance of the VMs by nearly 30%. Memory allocation policies in technologies like VMware, XEN partition the physical memory among the different VMs and allow performance isolation [15]. Furthermore, research done by the authors in [24] shows that, sharing on-chip resources such as caches results in challenges to guaranteeing predictable application performance.

Every VM is allocated its private share of cache on-the-go based on its requirements by the

hypervisor. In this research, the authors propose cache locking on such a hypervisor cache that will improve the performance of caching. The secondary level cache of the physical server is used for hypervisor caching and the proposed hypervisor cache is partitioned, so that every virtual server gets its own share of the hypervisor cache. In a non-virtualized setup, the secondary cache of a physical server is a large, shared cache. In this research, the proposed hypervisor cache is not shared by the VMs and locking is implemented on each partition that is private to the virtual servers.

The main design features of the hypervisor cache are:

1) It should be near to all the VMs and dynamic in nature
2) The hypervisor cache proposed is not shared by the VMs and each VM has its own private share of cache on the hypervisor
3) The proposed dynamic hypervisor caching scheme efficiently utilizes the available cache to cache the VMs blocks of data
4) It should cache the most frequently accessed data blocks by the VM hence reducing the number of disk accesses, thereby reducing the latency.

The architecture of the proposed hypervisor cache is as shown in Figure 5. As seen in the figure, there are 4 VMs running different operating systems such as Windows 2003, Windows 2008, Red Hat Linux 5 and the Cent OS. The physical disks are presented to the VMs as virtual disks due to the abstraction layer. The miss table is controlled by the hypervisor and is used to determine the cache lines that need to be locked. The physical server hosting the VMs is connected to the storage subsystem through protocols such as Fibre Channel, Infiniband and so on.

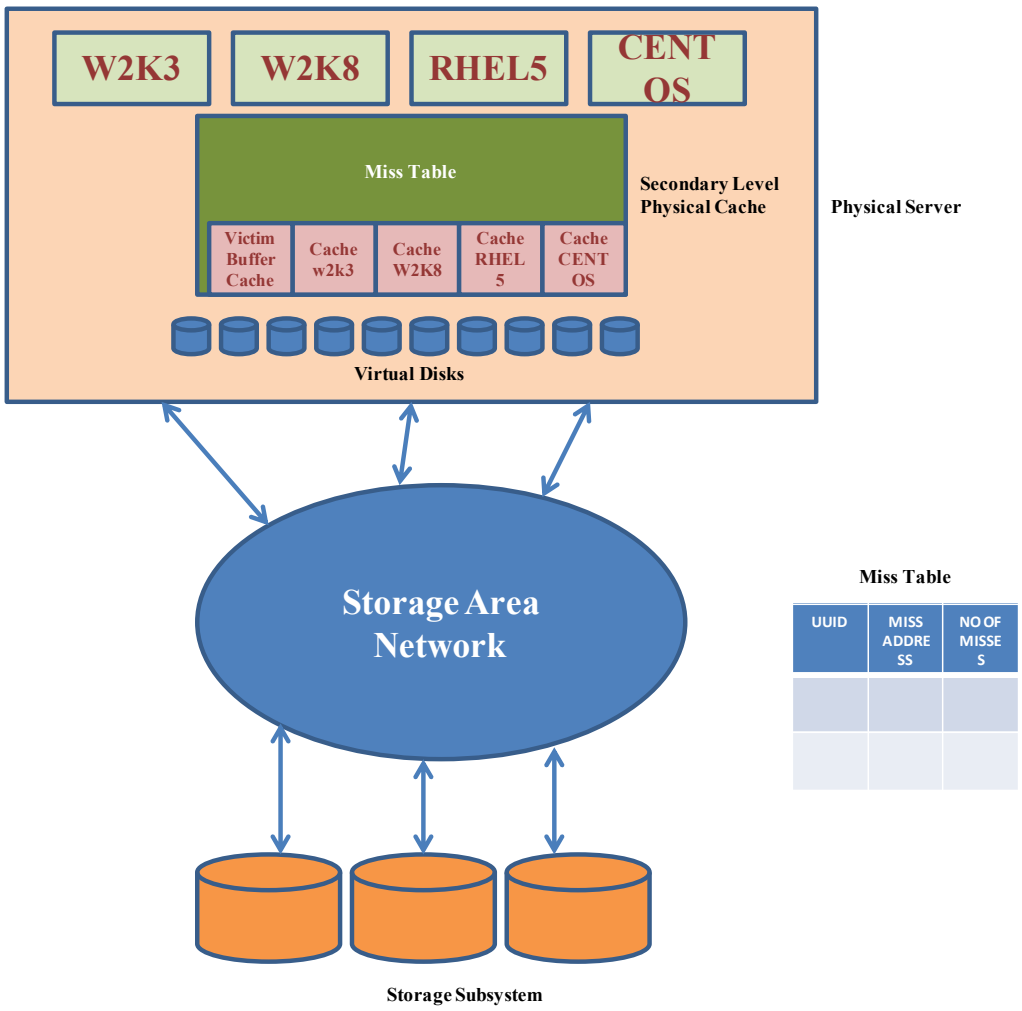

Figure 5: Architecture of hypervisor cache

3.6 Dynamic Caching using Thin Provisioning

Dynamic caching of the hypervisor cache refers to allocating cache to the virtual servers on the go, on a need per basis. Dynamic cache allocation leads to thin provisioning of cache resources and enables efficient utilization of an expensive piece of memory like cache. In a paravirtualized setup, the hypervisor is aware of all the IO requests that are being made by the VMs [35][37][40] and can hence manage the cache allocation to the VMs effectively. Since cache is one of the most crucial resources which influences the performance of a VM, ample care has to be taken on the placement of a cache. Having it at the hypervisor level and allowing the hypervisor to take a

wise decision on the allocation of resources augurs well with the needs the posed by the VMs. The following sections explain the model used for thin provisioning of cache resources on the hypervisor and the proposed algorithm.

3.6.1 Issues with Static Caching

The current caching mechanism used in server virtualization is based on caching on virtual machines. The physical disks attached to the physical server are abstracted to form a virtual disk pool. The virtual disks are allocated to the VMs and a part of the allocated virtual disk space on each VM is used for caching purposes. This method of static allocation of cache on the VMs has several disadvantages:

1) Not every virtual server is active all the time. For example, a backup server is active during the night because there will not be many active users. Game servers are active only for some time during the day. Database servers are active when employees are at work and so on. Static allocation allocates cache to the servers even when they are not active resulting in cache wastage.
2) When a server has a high I/O rate, a statically allocated cache space might not be sufficient to store the data and will result in high numbers of cache misses. When the cache miss rate increases, the response time of the virtual server also increases. This could have very detrimental consequences on the performance of the virtual server.
3) Static allocation results in under utilization of the cache.

CHAPTER 4

PRELIMINARY WORK: CACHE LOCKING IN MULTICORE

This section describes the preliminary work done by the authors before proposing the hypervisor cache solutions in this research. In their preliminary work, the authors study and analyze the effect of cache locking in multi core processor caches and then apply the technique to a new area of research emerging in the storage industry, namely, server virtualization.

Multiple caches in multicore architecture increase power consumption and timing unpredictability. Although cache locking in single-core systems shows improvement for large multithreaded applications, there is no such effective strategy for multicore systems. In this work, the authors propose three level-1 cache locking strategies for multicore systems – static, random, and dynamic. In the random strategy, blocks are randomly selected for locking. The static and dynamic schemes are based on the analysis of applications' worst case execution time (WCET). The static scheme does not allow unlocking blocks during runtime, but the dynamic scheme does. Using VisualSim and Heptane tools, the authors simulate a system with four cores and two levels of caches. According to the simulation results, the dynamic cache locking strategy outperforms the static and random strategies by up to 35% in mean delay per task for the workloads used (e.g., MPEG3 and MPEG4).

Multicore architectures are more suitable for multithreaded, parallel, and distributed processing, because concurrent execution of tasks on a single processor, in many respects including execution time and thermal constraints, is inadequate for achieving the required level of performance and reliability. Startingwith two cores, the number of cores in multicore processor is increasing. According to recently published articles, IBM POWER7 has up to 8 cores, Intel Xeon has up to 12 cores, AMD Opteron has up to 12 cores, MIT RAW has 16 cores,

and Tilera TILE-Gx has 100 cores. As the number of cores in the processors is increasing, software applications are having more and more threads to take advantage of the available cores [63][67][68]. Based on currently available processors and future design trends, the first level caches inside the cores and the second level caches outside of the cores are common in most multicore processors. In a multicore processor, two or more independent cores are combined into a die. Each core has its own level-1 cache (CL1); in most cases, CL1 is split into instruction cache (I1) and data cache (D1). Multicore processor may have one shared level-2 cache (CL2) or multiple distributed CL2s [69][70][71]. It is established that cache parameters may improve system performance [66]. However, cache consumes considerable amount of power and introduces execution time unpredictability.Real-time applications demand timing predictability and cannot afford to miss deadlines. Therefore, it becomes a great challenge to support real-time applications on power-aware multicore systems.

Studies show that cache locking improves predictability in single-core systems [63][64]. Cache locking is the ability to prevent part of the instruction or data cache from being overwritten. A locked cache block is excluded by the cache replacement policy. However, the content of the locked cache block can be updated according to the cache write policy. Cache entries can be locked for either an entire cache or for individual ways within the cache. Entire cache locking is inefficient if the number of instructions or the size of data to be locked is small compared to the cache size. In way cache locking, only a portion of the cache is locked by locking ways within the cache. Unlocked ways of the cache behave normally. Way locking is an important alternative of entire locking. Using way locking, Intel Xeon processor achieves the effect of using local storage by Synergistic Processing Elements (SPEs) in IBM Cell processor architecture [71][72]. Way cache locking at level-1 cache in multicore systems should improve

the overall performance of large (and multithreaded) real-time applications. To the best of the author's knowledge, no effective level-1 cache locking strategy is available for multicore systems.

The authors propose three level-1 cache locking strategies for multicore systems in this work. First, the authors introduce random strategy. According to this strategy, some blocks should be randomly selected, preloaded, and locked in level-1 cache. The random strategy is simple to implement (no pre-processing is needed), but performance improvement cannot be guaranteed. Then, the authors introduce static and dynamic strategies. Blocks are preselected for locking based on WCET analysis of applications in both (static and dynamic) schemes. In the static scheme, a locked block cannot be unlocked during the runtime. In the dynamic scheme, a core is allowed to lock, un-lock, and re-lock its cache blocks, during runtime, at the beginning of each code segment. Static locking may be a better option over random strategy when all or most applications are known and large in code-size. However, static locking may decrease overall performance if most applications are small. Dynamic locking strategy provides the flexibility so that each core can decide whether to apply cache locking or not for a particular code segment at run-time. In the dynamic strategy, a core can adjust its locked cache size during run-time to excel performance. The well-known MESI protocol can be used to deal with cache inconsistency. In case of multiple cached copies of the same memory block, (i) all blocks can be locked or (ii) only the first cached block can be locked and other cores should refer the locked copy (not the main memory copy) for reading/writing.

4.1 Proposed Cache Locking Strategies for Multi Core Systems

In this section, the authors present our proposed level-1 cache locking strategies for multicore systems. The authors propose three strategies – random, static, and dynamic CL1 cache locking.

In random strategy, blocks are randomly selected for locking. In the static and dynamic strategies, blocks are preselected based on WCET analysis – cache miss information is used knowledgeably with cache locking algorithm to determine the memory blocks to be locked. Information about the blocks that cause cache misses are collected together and called block address and miss information (BAMI). One BAMI is generated for each application after post-processing the tree-graph generated by Heptane. Static scheme does not allow unlocking blocks during runtime; however, dynamic scheme allows changes in the locked cache size at runtime. Each job (like a thread) consists of a number of tasks. Each task can be considered as a single instruction or a set of instructions. Finally, cache locking issue with multiple cached copies is also addressed in this section.

4.1.1 Random CL1 Cache Locking

In random CL1 cache locking, block selection for cache locking is done randomly. In this strategy, jobs are selected from the job queue depending on the job properties and available cores for processing. Selected jobs are assigned among the cores. Based on the selected jobs, CL1s and CL2 are preloaded using randomly selected blocks. Any additional spaces in CL2 are also preloaded using randomly selected blocks. Random CL1 cache locking is illustrated in Figure 6. After being assigned a job, each core decides (at cache level) if it should apply cache locking or not. A core may decide not to apply cache locking for a small job, because small jobs may entirely fit in the core's cache. If the core decides to apply cache locking, it may lock any number of ways it wants. Different core may lock different cache size (number of ways). The overhead due to the fact that each core makes its own decision about locking is negligible, because each core has the required information (e.g., cache size, job size, and BAMI). The locked cache remains locked until the core is done with its assigned job.

Each core completes its job independently. To calculate average delay per task, the maximum delay is considered for each batch of jobs. *The maximum delays fromall batches of jobs are added together; then the sum of the maximum delays is divided by the total number of tasks. The cores re-decide whether they should apply cache locking or not depending on the newly assigned jobs. For large or new applications (i.e., the code does not fit in the cache or miss information for block address is not known, respectively), random CL1 cache locking may be beneficial.For smallorknown applications(i.e., the code fits in the cache or miss information is known, respectively), random cache locking is not favorable.In addition to cache locking, block selection for preloading and block selection for cache replacement are also done randomly in this strategy.

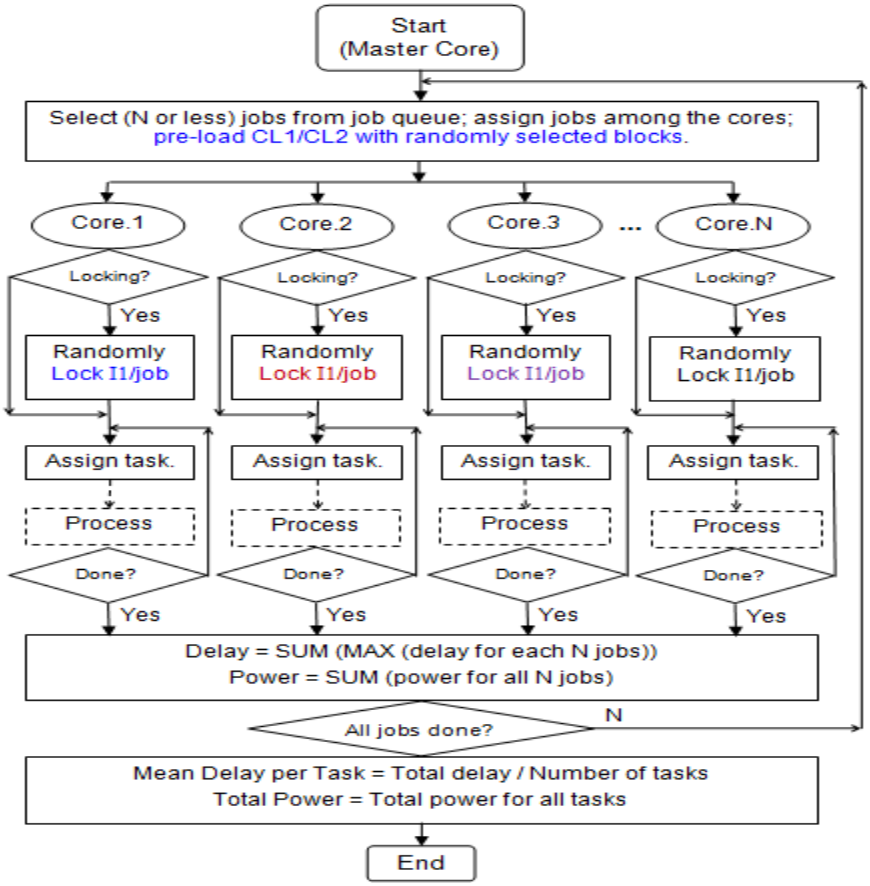

Figure 6: Random CL1 Cache Locking

4.1.2 Static CL1 Cache Locking

In the static CL1 cache locking, block selection for cache locking is done using WCET analysis based on BAMIs of the applications. Unlike random CL1 cache locking, block selection for cache locking is done first using the BAMIs of all jobs in the job queue. Then the master core (Core.0 at CPU level) makes the decision if static CL1 cache locking should be applied, before the jobs are actually assigned to the cores. If Core.0 decides not to apply cache locking, no core locks any of its caches. If Core.0 decides to apply cache locking, each core should lock its similar ways. Locked caches remain locked until all the jobs are completed. Figure 7 illustrates static CL1 cache locking strategy.

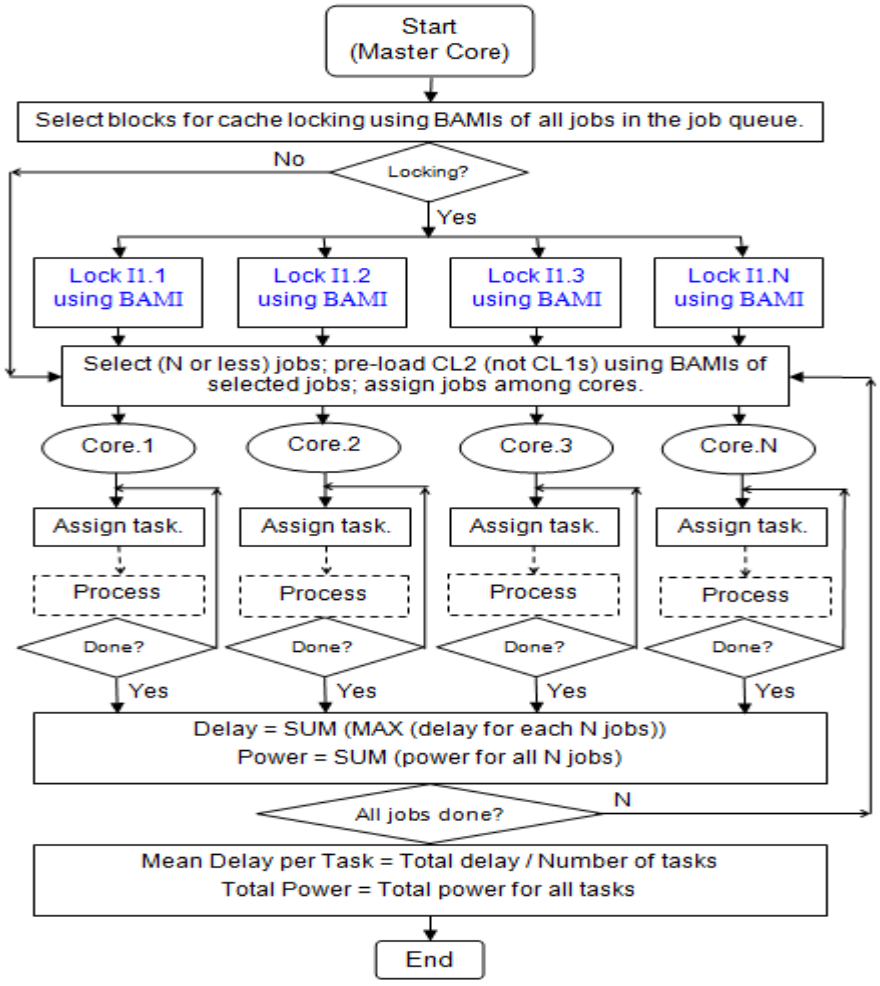

Figure 7: Static CL1 Cache Locking

For processing, jobs are selected from the job queue depending on the job properties and available cores. Based on the selected jobs, only CL2 is preloaded using the BAMIs of the jobs. A core cannot lock any ways it wants (it is decided by the master core, Core.0, of the CPU). Each core completes its job independently. Mean delay per task is calculated using similar methods as used in random CL1 cache locking. To calculate mean delay per task, the maximum delay is considered for each batch of jobs. After completion of a batch of jobs, another batch is selected. However unlike random CL1 cache locking, the CL1s are not re-preloaded according to the newly selected jobs (only CL2 is re-preloaded). Also, unlike random CL1 cache locking, the cores cannot re-decide whether it should apply cache locking or not (cache locking decision is made only once, by the master core, Core.0). For large applications (when the code does not fit in the cache but miss information for block address is known), static CL1 cache locking may be beneficial.

In this strategy, the BAMI entries are used to determine the block with the maximum number of misses for preloading and the block with the minimum number of misses is selected as the victim block for cache replacement.

4.1.3 Dynamic CL1 Cache Locking

Like static CL1 cache locking, block selection for cache locking, preloading, and cache replacement in the dynamic CL1 cache locking is done using the BAMIs of the applications. Like random CL1 cache locking, each core decides if it should apply CL1 cache locking or not for the assigned job. As a result, dynamic CL1 cache locking has the flexibility and merits of making the cache locking decision at each core using the BAMI. However, this strategy is complicated when compared with the other two strategies. Figure 8 depicts our proposed dynamic CL1 cache locking strategy.

Like random CL1 cache locking strategies, jobs are selected from job queue first. Then CL1s and CL2 are preloaded using the BAMIs of the selected jobs, like static CL1 cache locking. Selected jobs are then assigned among the cores. Like random CL1 cache locking (and unlike static CL1 cache locking), each core decides if it should apply cache locking or not after a job is assigned to it. If a core decides to apply cache locking, it may lock any number of ways it wants. Different cores may lock different cache sizes. The locked cache remains locked until the core is done with its assigned job. Each core completes its job independently. Mean delay per task is calculated using similar methods as used in random and static CL1 cache locking. To calculate mean delay per task, the maximum delay is considered for each batch of jobs. After completion of a batch of jobs, another batch is selected.

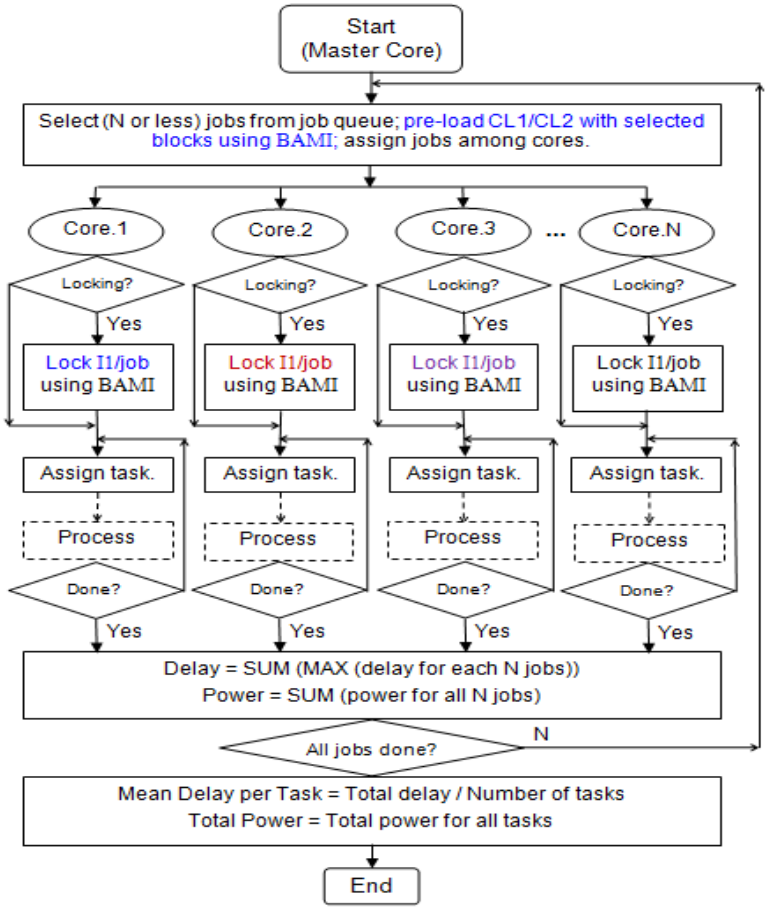

Figure 8: Dynamic CL1 Cache Locking

Like random CL1 cache locking (and unlike static CL1 cache locking), both CL1s and CL2 are re-preloaded according to the newly selected jobs. The cores re-decide whether they should apply cache locking or not depending on the new jobs. Dynamic CL1 cache locking strategy has the advantages of both random and static CL1 cache locking strategies to improve overall system performance. For any applications (i.e., the code may or may not fit in the caches) when block address and miss information are known, dynamic CL1 cache locking may be beneficial.

Comparing with the proposed random and static CL1 locking strategies, it is complicated but feasible to implement the proposed dynamic CL1 locking strategy. The complexity is two-fold: CL1 cache locking is asynchronous across the cores (like random CL1 locking strategy) and BAMIs are needed (like static CL1 locking strategy).

4.2 Simulations

The authors evaluate our proposed CL1 cache locking strategies for multicore systems using simulation technique. In this section, the authors discuss the simulation details and present the simulation results. The authors use Heptane [71] and VisualSim (short for VisualSim Architect) [72] tools to model and simulate a multicore embedded system. Heptane is used for WCET analysis of application on a single-core system with one-level cache and to generate workloads for VisualSim. VisualSim is used to simulate multicore systems with multi-level caches and to collect the results. In the following subsections, the authors briefly discuss the assumptions, applications, simulated architecture, important parameters, and tools.

4.3 Assumptions

Important assumptions include the following,

- A homogenous multicore is simulated where all cores are identical. Cache parameters used are the same for all cores.

- In random and dynamic cache locking, each core can implement cache locking independently as needed. However in the static cache locking, the master core(Core.0 of the CPU) decides whether cache locking should be implemented or not and all cores either locks their similar ways or not, respectively.
- Write-back memory update policy is used for all strategies.
- In random cache locking, victim block is selected randomly. However in the static and dynamic cache locking, victim block is the block with the minimum number of misses(which is determined by using a BAMI).
- The delay introduced by the bus that connects CL2 with the memory is 15 times longer than the delay introduced by the bus that connects CL1 and CL2.

4.4 Applications Used

To overcome the limitations of simulation tools and applications, the authors choose widely used Heptane and VisualSim simulators and consider five popular applications that are widely used in math/science, Internet, and multimedia to run the simulation programs. These diverse applications are: Fast Fourier Transform (FFT), Graphics Interchange Format (GIF), Joint Photographic Experts Group (JPEG), Moving Picture Experts Group's MPEG3, and MPEG4 (part-2). Selected applications have various code sizes. The code size for FFT, GIF, JPEG, MPEG3, and MPEG4 are 2.34 KB, 35.17 KB, 41.90 KB, 87.51 KB, and 91.83 KB, respectively. Heptane takes C code as the input application and generates tree-graph. Tree-graph shows which memory blocks caused cache misses and how many (if any). This tree-graph information is used to create the BAMI.

4.5 Simulated Architecture

The authors model and simulate a four-core system which depicts popular Intel Xeon quad-core architecture. Each core has level-1 private cache which is split into I1 and D1 and unified CL2 is shared. As shown in Figure 9, BAMI is implemented inside the cores and BALCI is implemented in CL2.

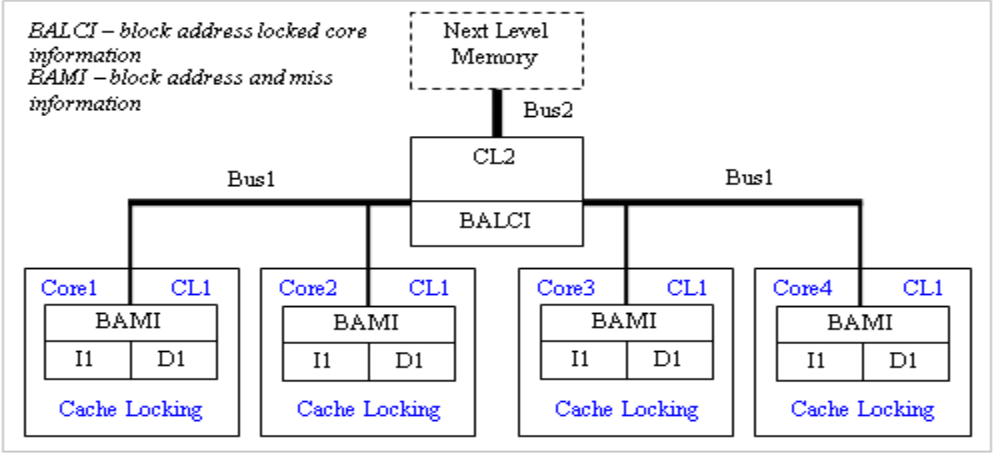

Figure 9:Simulated Architecture

4.6 Important Parameters

The authors keep number of cores fixed at 4, CL2 cache size fixed at 1024 KB, associativity level fixed at 8-way, and line size fixed at 128 Byte. The authors vary I1 (and D1) cache size from 4 KB to 64 KB (start at 4 KB, double every time). Finally, the authors vary the percentage of locked I1 cache size from 0.0% to50.0%; start with no locking, lock one extra way (starting with way-0) in every successive attempt.Time to load CL1 from CL2 is 1 unit and to load CL2 from the main memory is 15 units.

Output parameters include mean delay per task. Delay is the time between the start of execution of a task and the end. Mean delay per task is defined as the average delay of all the tasks.

4.7 Tools Used

Simulation tools are very important for precise assessment of the systems under consideration. the authors did not find any single tool available for modeling and simulating the proposed CL1 cache locking in multicore architecture. Hence, the authors use two effective and famous simulation tools, Heptane [21] and VisualSim [22], in this study. In addition, the authors develop C programs to pre-process the application code and post-process the Heptane generated tree-graphs.

Heptane is used to characterize the applications and generate the workloads. Heptane is a well-known WCET analyzer from IRISA, a research unit in the forefront of information and communication science and technology. Heptane simulates a processing core, takes C code as the input application, and generates a tree-graph that shows the blocks that cause misses. After post-processing the tree-graph, block addresses are selected for cache locking.

VisualSim (short for VisualSim Architect) is used to model the multicore architectures, simulate cache locking at shared cache, and collect simulation results. Heptane-generated workloads are used to run VisualSim simulation programs. VisualSim from Mirabilis Design is a graphical simulation tool to build models. VisualSim contains a complete suite of modeling libraries, simulation engines, report generators, and debugging tools. Using VisualSim, the authors model the abstracted multicore architectures. VisualSim provides a simulation cockpit that has functionalities to run the VisualSim model and to collect simulation results. Simulation cockpit can be used to change the values of the input parameters and store the results as text and/or graph files.

4.8 Results and Discussion

The authors model a four-core system with two-level cache memory subsystem and simulate our proposed random, static, and dynamic CL1 cache locking schemes using FFT, GIF, JPEG, MPEG3, and MPEG4 applications. The authors obtain results by varying the amount of locked cache size and I1 (and D1) cache size. In the following subsections, the authors first discuss how execution time predictability is impacted by cache locking, followed by the impact of the percentage of locked I1 cache size and the total I1 cache size. Then (finally), the authors compare the proposed CL1 cache locking strategies: random, static, and dynamic.

4.8.1 Average Cache Hit Rate

The average cache hit rate can be improved by applying proposed cache locking strategies, because cache locking helps hold the blocks in the cache during execution that might generate more misses if not locked. The authors conduct this experiment using the Heptane simulation package. For a system with 2 KB cache size, 128 Byte line size, and 8-way associativity, the total number of blocks in cache is 16 (2048/128) and total number of set is 2 (16/8). If 2 blocks from each set is locked, 4 (2x2) blocks out of 16 blocks (i.e., 25% of I1) is locked. From Heptane WCET analysis, FFT code generates 246 cache misses in the cache subsystem mentioned above. It is also found that the maximum number of misses from any 4 blocks is 128. By locking those 4 blocks that create the maximum number of misses (i.e., 128), predictability can be enhanced (as more than 50% cache misses are avoided) by locking only 25% of the cache size. These results are summarized in Table 1. When multiple applications are being executed in a multicore CPU, dynamic CL1 cache locking should provide the best predictability as each core can implement cache locking as/if needed. In a multicore system, random and static CL1 cache locking may not provide the best predictability because randomly selected blocks may not be the blocks with

maximum cache misses and static cache locking doesnot have the flexibility to change locked cache during runtime.

Table 1:Average Cache Hit Rate

Parameter	**Value**
Cache size	2 KB
Line size	128 Byte
Associativity level	8-way
Total number of blocks in cache	16 (2048 / 128)
Number of blocks for 25% cache locking	4 (16 * 0.25)
Total number of misses for FFT	246
Maximum misses from any 4 blocks	128
Maximum cache misses possible by 25% cache	50.41%
Cache misses saved by 25% random cache locking	< 50.41%
Cache misses saved by 25% static cache locking	< 50.41%
Cache misses saved by 25% dynamic cache locking	Always 50.41%

4.8.2 Impact of Locked Cache Size

As the amount of locked cache size increases – on the one hand, more cache blocks that cause most of the misses are locked; but on the other hand, the effective cache size decreases. In this subsection, we explore the impact of cache locking on mean delay per task using proposed random CL1 cache locking strategy. To run the simulation program, the authors send similar workload through all four cores and vary the locked cache size and obtain the Figure 10 illustrates mean delay per task for all five workloads with various locked I1 cache size (0% to 50% locking). Simulation results show that random CL1 cache locking has significant impact on large applications (like MPEG3 and MPEG4) than small applications (like FFT). For MPEG4, mean delay per task decreases as we move from no locking to 25% locked I1 cache size; mean delay per task starts increasing as we move beyond 25% I1 locking as compulsory cache misses increase. For GIF and JPEG, random CL1 cache locking has some (but not very significant)

positive impact on mean delay per task. For FFT, there is no positive impact of random CL1 cache locking. This is because FFT code entirely fits inside I1 but other codes do not.

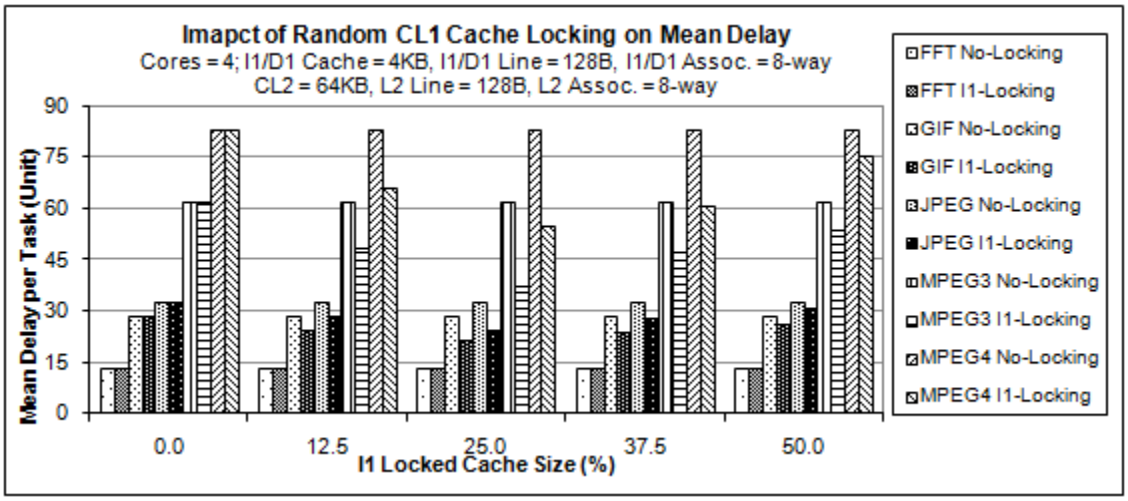

Figure 10: Impact of Random CL1 Cache Locking on Mean Delay

4.8.3 Impact of Entire Cache Size

We know that capacity misses decrease as cache size increases. In this subsection, we examine how random CL1 cache locking impact on mean delay per task when I1 cache size changes. Figures 11 illustrate the average delay per task for various I1 cache size starting at 4 KB. Results for no locking and 25% random I1 cache locking using FFT, GIF, and MPEG4 are considered.

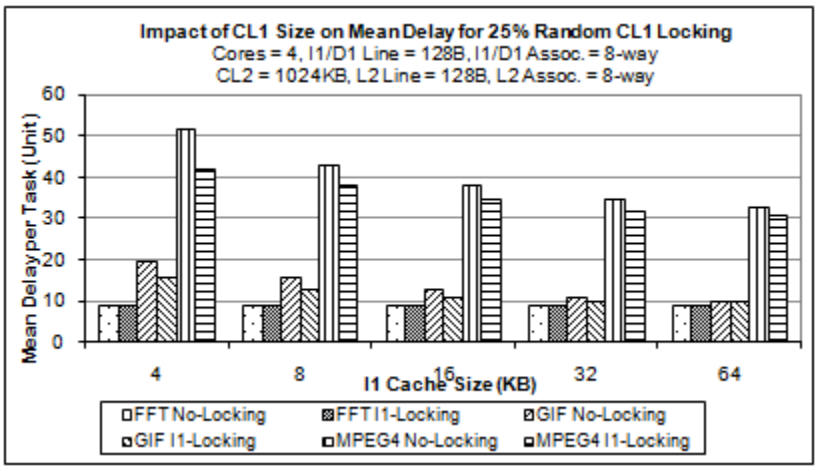

Figure 11: Impact of CL1 Size on Mean Delay for 25% Random CL1 Locking

4.9 Conclusions

This work presents three level-1 cache locking strategies for multicore systems: static, random, and dynamic. The random strategy is easy to implement as blocks are randomly selected for locking by each core (at core level). The static and dynamic schemes are based on WCET analysis. For static locking scheme, blocks are preselected by the master core (Core.0 at CPU level) using BAMIs of all applications and cannot be unlocked during the runtime. For the dynamic locking scheme, blocks are preselected by each core (at core level) using BAMI of the assigned job and the locked cache size can be changed during runtime for each job. A system with four cores and two levels of caches is simulated using FFT, GIF, JPEG, MPEG3, and MPEG4 applications. According to the simulation results, timing predictability can be enhanced by avoiding more than 50% cache misses (see Table 1). Simulation results also show that the static and random schemes may offer good improvement for large applications. Dynamic CL1 cache locking is difficult to implement as each core determines how many ways to lock for optimal outcomes.However, the dynamic cache locking strategy outperforms the static and random strategies by up to 35% and 22%, respectively.

Based on the simulation results, the proposed CL1 cache locking should have significant performance impact on commercially available embedded multicore processors (like Intel Atom D510 and ARM Cortex-A9) and on multicore server processors (like Intel quad-core Xeon DP and AMD quad-core Opteron) for real-time applications.

A hybrid cache locking strategy (no locking on some cores for small applications, static locking on some cores for large applications, random locking on some cores for new/unknown applications, and dynamic locking on the rest of the cores) may be a better choice for multicore architecture.

As a future work to this research, the authors wanted to investigate the impact of cache locking in hypervisors. Server virtualization is a relatively new technology and very little research has been done in this area with respect to studying the performance of cache. Hence, the authors extend the preliminary work described in this chapter to study the performance of cache and cache locking in hypervisors. This is explained in the next chapter.

CHAPTER 5

PROPOSED IMPROVEMENT TECHNIQUES FOR HYPERVISOR CACHE

5.1 Proposed Dynamic Cache Allocation Model

This section describes the model for dynamically allocating the cache resources among the VMs. As explained earlier, the hypervisor manages the cache and is the central entity which decides cache allocation to the VMs. The model and the algorithm used by the hypervisor to decide the cache allocation is as explained below. Table 2 represents the parameters used in the model. The authors used the following parameters as indicators to allocate and de-allocate hypervisor cache to the VMs dynamically:

1) Number of IO requests a VM receives in a particular time interval, $[t\ ,\ t+\ \Delta t]$
2) Cache Miss percentage of a VM in the time interval, $[t\ ,\ t+\ \Delta t]$

Table 2:Parameter used in Dynamic Cache Model

Parameter	Description
$\mathbf{C}$	Total available cache size on the hypervisor
$\mathbf{N}$	Number of active Virtual Machines
$\mathbf{C_{Alloc}}$	Cache allocated to each VM
$\mathbf{C_{Threshold}}$	Threshold cache usage for each VM
$\mathbf{R_{Threshold}}$	Threshold of number of I/O requests on the VM
$\mathbf{C_{MissTh}}$	Threshold of number of cache misses for the VM
$\mathbf{R_n}$	Number of I/O requests on the VM in $[t, t+\ \Delta t]$
$\mathbf{C_{Used}}$	Amount of cache used by a VM in $[t\ ,\ t+\ \Delta t]$
$\mathbf{C_{Miss}}$	Number of cache misses on the VM in$[t, t+\ \Delta t]$
$\mathbf{C_{Remain}}$	Amount of cache remaining on each VM

Initially, the total available cache *C* is divided and equally allocated to each of the active virtual servers. The hypervisor owns a small portion of the cache space for itself and is not allocated to the VMs [3]. If the portion of cache owned by the hypervisor is represented by C_h,

then the available cache to allocate to the VMs can be given by:

$$C_{vm} = C - C_h \quad \text{.....(1)}$$

The cache space initially allocated to the VMs is given by:

$$C_{alloc} = \frac{C_{vm}}{N} \quad \text{.....(2)}$$

The model developed in this research is based on the following assumptions:

1) Each virtual machine plays the role of a server. Example, a VM can be a database server, web server and so on
2) The workload characteristics of each VM are known apriori
3) The values of $C_{Threshold}$, $R_{Threshold}$ and C_{MissTh} are calculated when the virtual server has heavy workloads running on it. Hence, these values are taken as a reference to make decisions about allocating and de-allocating cache space

The dynamic caching algorithm proposed in the research is based on thin provisioning technique. A virtual server which owns excess cache than the threshold, in time interval $[t\ ,\ t + \Delta t]$ acts as a cache donor and a virtual server which has less cache than the threshold borrows the cache space from the donor. The virtual server cache allocation and de-allocation are decided based on the following conditions in time interval $[t\ ,\ t + \Delta t]$:

- If $R_n > R_{Threshold}$ and $C_{Miss} > C_{MissTh}$ then cache needs to be allocated to the virtual server and the virtual server acts as a borrower
- If $R_n < R_{Threshold}$ and $C_{Miss} < C_{MissTh}$ then, cache can be de-allocated from the virtual server and the VM acts as a donor
- If $R_n \geq R_{Threshold}$ and $C_{Miss} \leq C_{MissTh}$ then no action will be taken because it indicates that the virtual server still has cache remaining

- If $R_n < R_{Threshold}$ and $C_{Miss} > C_{MissTh}$ then, the cache replacement policy has to be looked into as this scenario is independent of cache allocation.
- If the $C_{Used} > 2C_{Threshold}$ for two consecutive time intervals $2[t, t + \Delta t]$in a virtual server and $C_{Remain} < 2C_{Threshold}$ then the condition calls for a cache allocation and cache is borrowed from the donor which has a greater cache remaining than required
- If $R_n < R_{Threshold}$ and $C_{Remain} > 2C_{Threshold}$ then the VM with the greatest C_{Remain} will be the first to be de-allocated of cache equivalent to $C_{Threshold}$

The evaluation of the above proposed algorithm is provided in the simulation and results section.

5.2 Proposed Miss Table for Cache Misses

The miss table is a data structure that holds the addresses of data blocks that incur the most number of misses by the applications running on the virtual machines. The idea behind using the miss table on the hypervisor is, to make available those blocks of data in the hypervisor cache that are missed greater number of times than the threshold, because, the data addresses that incur large number of misses are the data blocks that are frequently accessed by applications on the VM. The miss table is hence used to identify the frequently accessed data blocks by applications running on the VMs and making them available to the VMs. A single miss table is proposed on the hypervisor that can store the data block addresses incurring misses by the VMs.

Although the miss table is transparent to the VMs, the hypervisor uses the information in the miss table for cache locking purposes. The hypervisor sees all the VM data accesses that miss the VM memory in the form of I/O requests [5]. The authors in [2] have developed a tool, which they call Geiger, that can accurately determine the VM data access pattern on the hypervisor. From the above mentioned researches, one can infer that the VM cache misses are visible to the

hypervisor and hence, the hypervisor can store the addresses of the data blocks that incur misses. In this research the authors use the Valgrind tool to trace the data access by VMs. The miss table proposed in this research consists of three fields namely, unique identifier of the VM, address of the blocks missed by the VM and the number of times that address incurs a miss as shown in Table 3. For the sake of simplicity, the miss table is made to hold twenty addresses that incur maximum number of misses for each VM. The addresses stored in the miss table are the ones that are most needed by the VM and the data blocks corresponding to those addresses need to be cached and locked, so that they are not evicted. Since the miss table is populated over a period of time, cache locking can be done dynamically once the application on the VM starts execution.

Table 3: Miss Table

UID of theVM	**Address of the block missed**	**Number of misses**
1	**000A0800**	**400**
1	**000A0543**	**239**
2	**E51B5378**	**536**
3	**B538C12**	**220**
.	.	.
.	.	.
.	.	.

5.3 Proposed Hypervisor Cache Locking

The hypervisor cache proposed in the research aims at reducing the number of data cache misses on the hypervisor cache and thereby improve the hit rate of data blocks in the cache. The miss table explained in the previous section is used to determine which data blocks in the hypervisor cache need to be locked. Hence the authors propose a locking mechanism on the hypervisor cache that retains the frequently accessed data block with the help of the miss table. Cache locking is mostly used in processor caches in order to improve the cache hit percentage.

When the data blocks in the cache are locked, they cannot be replaced from the cache. They can however, be read and written, but not evicted from the cache until the cache line is unlocked. The cache locking mechanism proposed on the hypervisor cache is way locking where certain lines in the cache are locked.

By locking the cache lines, it can be made sure that the frequently accessed blocks are present in the hypervisor cache and further memory and disk accesses can be avoided. The cache locking algorithm is applied to each share of the cache that is private to the VMs. In a real time SAN, accessing data blocks from the disk array is very expensive and any reduction in the number of I/O requests is very significant. This is because, the disk array is connected via protocols such as FC, Iscsi, SAS and I/O requests have to traverse through them, which can be very time consuming if the number of I/O is large. A further question that arises is the time for which the cache lines need to be locked. Since the requirements of the VMs change dynamically with respect to the data blocks that incur misses, it is desirable to have an algorithm that dynamically determines the data addresses that are frequently missed and an algorithm that locks the cache accordingly and dynamically. The design of such an algorithm is a work in progress by the authors of this paper. In this research, the cache lines selected are locked for the entire execution of the application on the VM, for the sake of simplicity.

5.4 Victim Buffer

A victim buffer cache is a small cache used to hold the recently evicted cache lines. The victim buffer cache proposed in this research is a small piece of cache that stores blocks of data that are evicted from the main hypervisor cache. The victim buffer cache is usually very small when compared to the main cache. The main purpose of using a VB cache is to cache the blocks of data that are evicted from the main cache, so that, next time the evicted blocks are needed by an

application, they are moved into the main cache faster and a disk access can be avoided. In this research, every active VM's hypervisor cache assigned to a VM is given a small victim buffer cache. In this research, the victim buffer cache is abstracted from the secondary level cache. This means that, a small part of the hypervisor cache is used as a victim buffer cache. Hence, the victim buffer cache is still a part of the secondary cache of the physical server.

5.5 Updated Cache Replacement Algorithm

Cache replacement policies play an important role in the performance of caching. A good cache replacement policy preserves the frequently accessed blocks in the cache, while evicting the unnecessary blocks. In this research, the authors propose a probability based cache replacement algorithm that preserves those blocks of data, whose probability of access by applications running on the VMs is higher than the threshold and evicts those blocks of data whose probability of access is lesser than the threshold. The probability based cache replacement algorithm proposed in the research is effective and results show that the performance is better than the LRU replacement algorithm. The algorithm and its working are described below.

The authors implemented a probability based cache replacement policy and analyzed the performance of hypervisor caching. In the probability based cache replacement, when an application is run on the VM, an access history table is generated for the blocks of data in the cache. The history table contains the block number and the number of accesses to that block in the hypervisor cache. Based on the number of accesses, a threshold value is calculated for the probability of access of the block. Hence, once the workload starts running, the cache is monitored for some time to generate the access history table. Once the table is generated, the probability of current access to the cache block is calculated. The current probability is compared with the history probability and if the current probability of access is lesser than the history

probability of access to the cache block, then the block is evicted from the cache. If not, the cache block is not evicted. The advantage of using probability based cache replacement is that the blocks that have a high probability of access are retained in the cache and hence, the number of cache hits increase. This in turn increases the effectiveness of cache locking.

CHAPTER 6

SIMULATION DETAILS

In this section, the authors describe the simulation details and implementation of the dynamic caching and locking mechanism on the hypervisor cache. The section also describes how the miss table and victim buffer cache are implemented.

6.1 Dynamic Caching Simulation

The model developed in this research was used to evaluate the performance of virtual servers when dynamic caching using thin provisioning is used. The model was evaluated with the dynamic caching and without dynamic caching and the experimental results show that dynamic caching using thin provisioning reduces the response time of the server and also leads to efficient cache utilization. In order to evaluate the dynamic cache allocation model, virtual box hypervisor was installed on a host PC and 3 virtual servers were created. The specifications of the host PC are as follows:

- Sony Vaio with Intel® Core™ i5 M460
- 2.53GHz processor
- 4GB RAM
- Windows 7 Operating System (OS) and will be the Host for the three Virtual Machines

The specifications of the three virtual machines created are as follows:

- 20GB of assigned hard disk
- 1 GB of RAM allocated for each VM
- Windows Vista Professional OS

Each of the VMs act as dedicated servers. The servers in consideration are File server, Web server and Database server. FileZilla software has been used in both the server and the client for the file server VM. MySQL software was used as the database management tool on the database server. Due to the constraints of creating a web server for a localized environment, the authors had to resort to IOmeter software to generate the required load for a web server.

The reasons for considering these servers as candidates for the experiments is because, the workloads on them depict varying percentages of reads and writes and different levels of cache utilization. Hence, the proposed model can be better analyzed for different workloads. Cache performance of each VM is measured for different amounts of loads and parameters. Generally caching is done at various levels, be it the processor cache, RAM or the database buffer. For the sake of experiments in this research, the authors consider caching on the VMs and see how the performance varies for different servers.

Windows Task Manager and performance monitor are the tools being used for the measurement of cache utilization by a process at any given point in time. The cache utilization and response time for the servers are recorded for the three servers. Cache performance for a file server is measured while there is a file transaction happening between the server and the client. Similarly, for a database server, cache performance is measured when the server is processing database queries for different request sizes of data. For a web server, the cache performance is measured for different IOmeter loads. The test files that are being used are of uniform sizes across the three servers.

The requests from each of the servers go through the hypervisor through which the cache is controlled. The algorithm proposed in this research works on controlling the dynamic provisioning of the cache while maintaining the optimum performance metrics which are pre-

determined and are as per the industry standards. The proposed model is evaluated for the workloads on different servers and the response time and cache utilization are evaluated. The results are promising and are evaluated for each parameter separately.

6.2 Installation of Hypervisor and Virtual Machines

For the purpose of this research, Oracle's virtual box hypervisor was used to host the virtual machines. Three virtual machines were created and CentOS operating system was installed on them. Four different workloads were run on the virtual machines to collect traces of the memory loads and stores. The details on the workloads used and memory trace generation are explained in the subsequent sections.

6.3 Workloads Used

Workloads used in the research are primarily used to generate memory traces on the VM, which consists of details of the memory addresses where the data is read from and stored into. Four different types of workloads were run on the VMs and they are as follows:

1) Array Sort: An array sort program written in C consisting of lots of sorting operations constituted the first type of workload. The array sort program involves a lot of memory swap in and swap out, which further aids in generating a good memory trace and can help evaluate the locking mechanism well.

2) Gzip: TheGzip utility is used to compress the files and is used as a second workload on the virtual machine. The compression operation mainly involves a lot of write operations because compression converts the original file to a compressed version of the file (file.gz). This workload was chosen because it consists of more writes than reads and the proposed locking algorithm can be evaluated for similar such workloads.

3) Fast Fourier Transforms: Fast Fourier Transforms are used widely in the area of mathematics and involves computations. FFT is used to compute discrete Fourier transform and its inverse. The advantage of using the FFT is, the same mathematical computation can be done in lesser time which otherwise would have taken more time. The Fast Fourier Transform was chosen so that the locking mechanism can be evaluated for computational workload.

4) MP3: The MP3 workloadrunning on the VMs is a streaming feature where a music file is continuously read from the memory. The MP3 workload involves a lot of I/Os and a lot of read operations. The locking algorithm is evaluated for the streaming workload involving a lot of reads.

6.4 Obtaining Memory Traces of the VMs

Since the data block addresses that incur cache misses need to be recorded on the miss table, it is necessary to obtain the memory traces of the virtual machines with the above workloads running on them. Valgrind and Lackey tool was used to obtain memory traces of the applications running on the VMs [22]. The output of Valgrind is a file that contains a list of memory data

references made by the application running on the VM. The output of Valgrind consists of all the data addresses that are loaded, stored and modified. The miss table consists of data addresses that incur misses and hence, the authors had to filter the load values since load values correspond to addresses that incur misses. A Perl script was written to filter the load values from the output of Valgrind.

The cache performance for the three types of workloads discussed above with and without cache locking, with and without a victim buffer cache were evaluated. Experimental results show that cache locking can provide benefits in reducing the percentage of hypervisor cache misses and the response time for the applications running on the VMs.

6.5 Assumptions

This research makes the following assumptions:

1) The hypervisor used for server virtualization is not a bare metal hypervisor, but, a para virtualized one. In a para virtualized setup, the VMs are unaware that they are virtualized and assume that they are not sharing the underlying resources.
2) The workloads evaluated were run on a single VM and not on multiple VMs simultaneously.
3) The traces obtained were for the duration of running any type of workload on a single VM.
4) The cache locking proposed is for data cache and not the instruction cache
5) All the VMs are assumed to have same amounts of memory and cache
6) Simulations were conducted on a 64 bit machine. Results may vary for a 32 bit architecture
7) The VB cache is designed to hold 12 blocks of data (maximum).

6.6 Input/Output Parameters

The simulations were conducted for different cache sizes with several configuration features such as: Cache locking enabled, cache locking disabled, VB Cache enabled, VB cache disabled, probability based cache replacement enabled/disabled and so on. The input parameters were the cache size, associativity and the output parameter was the miss ratio. Tables4,5,6 show some of these parameters and their output. Table 4 shows the miss percentage with and without cache locking for different cache sizes.

Table 4: Varying Cache Size with and w/o Locking

Cache Size (KB)	Miss Percentage with Locking	Miss Percentage w/o Locking	% Reduction in Miss Rate
2	15.76	17.37	1.61
4	10.5	12.02	1.52
8	4.9	5.7	0.8
16	3	3.8	0.8
32	1.7	2	0.3
64	1.3	1.5	0.2

Table 5: Varying Cache Size with and w/o Locking in the presence of a Victim Buffer

Cache Size (KB)	Miss Percentage with Locking	Miss Percentage w/o Locking	% Reduction in Miss Rate
2	14.39	17.37	**2.**98
4	11	12.02	1.02
8	5.3	5.7	0.4
16	3.3	3.8	0.5
32	1.6	2	0.4
64	1.3	1.5	0.2

Table 5 shows the miss percentage with and without cache locking in the presence of a victim buffer and Table 6 shows the miss percentage with different associativities.

Table 6: Varying Associativity with and w/o Locking

Associativity	Miss Percentage with Locking	Miss Percentage w/o Locking	% Reduction in Miss Rate
2	15.76	17.37	1.61
4	10.5	12.02	1.52
8	5.5	5.7	0.2
16	4.4	5.1	0.7

The values in the above tables are used during simulations for the array sort workload. Similarly, the cache sizes and associativities were varied for different workloads and the results were recorded.

CHAPTER 7

RESULTS AND DISCUSSION

This section explains the results obtained as a result of the proposed dynamic caching, cache locking mechanism and implementation of victim buffer cache and the probability based hypervisor cache replacement algorithm. The proposed techniques are evaluated for different workloads with different cache configurations and the results look promising in terms of reducing the number of cache misses and reducing the data access latency by the VMs. The proposed techniques in this research are evaluated with cache parameters such as cache size, associativity. Cache size and associativity are the two main parameters that directly affect the performance of caching.

7.1 Evaluation of Cache Utilization with respect toFileSize

The thin provision model is designed to dynamically allocate the cache and also use the cache efficiently and effectively among the VMs. The authors do not consider the caching policy in this aspect of their calculation. Most caches usually have the LRU cache replacement policy in place. For the first part of the experimentation, the authors consider the case of static cache allocation. In the static allocation technique, every VM gets a definite and fixed amount of cache irrespective of its workload. If the cache becomes full, evictions start taking place based on the cache replacement policy and new blocks are brought into the cache.

Figure 12 shows the utilization of the cache as the request block size varies. Eventually the cache in all the three servers reaches its maximum limit. We can see that the database server takes the longest to process the workload, which is the same across the three servers in an equally distributed, statically allocated cache scenario. The database server, web server and the file server take 80, 28 and 8.5 minutes respectively to process a gradually incrementing workload

for a cache capacity of 825 MB. The drawback lies in the fact that, as the workload increased, the cache reached its maximum capacity. This leads to a common cache miss called capacity miss, where in, the cache miss percentage increases due to the fact that a lot of data has to be swapped in and swapped out. In such a case, the cache becomes conservative, thereby decreasing the cache performance and affecting the overall throughput.

The thin provisioning model was applied to the virtual servers and the cache was allocated dynamically to the VMs according to the model. With the thin provisioning model in place, it can be observed that the cache utilization is optimal and for the same file size, the cache is not limited. Another interesting thing to observe is that, there is no converging point for the thin provisioning model. This is because, cache is allocated on the go, on a need per basis and the VM does not run out of cache space quickly. Fig 13 shows the improvement achieved by this algorithm.

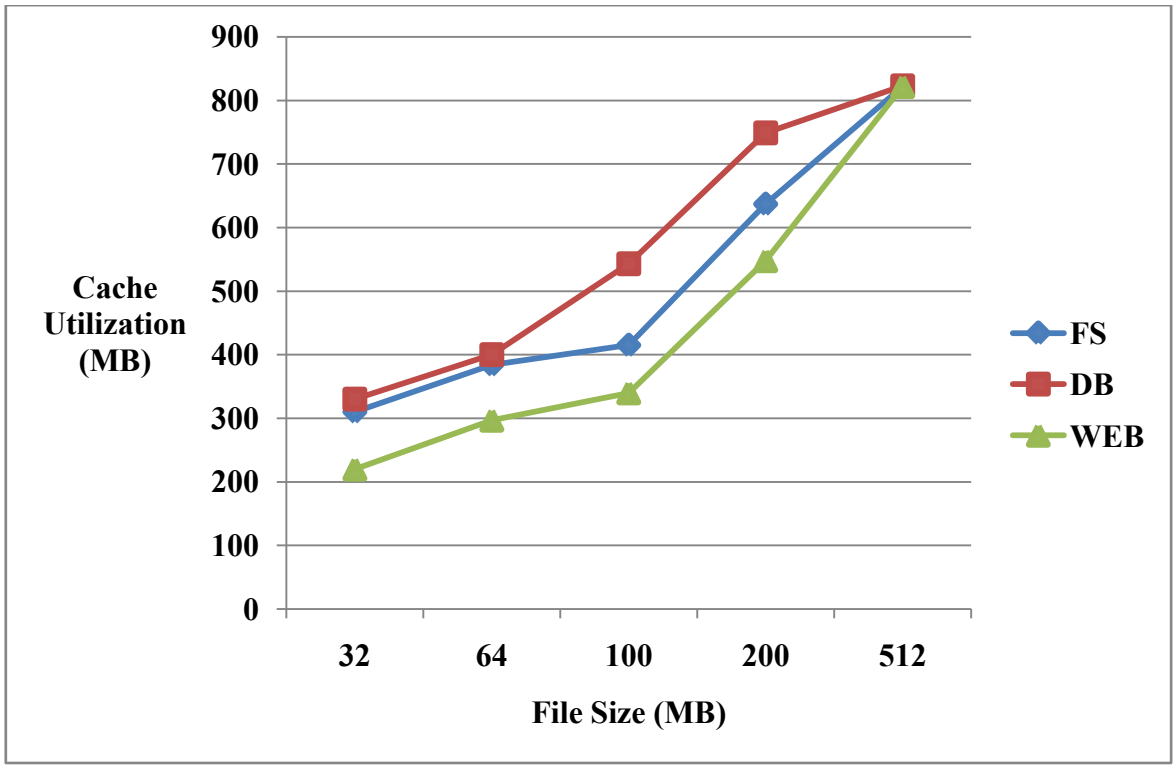

Figure 12: Cache Utilization with Static Caching

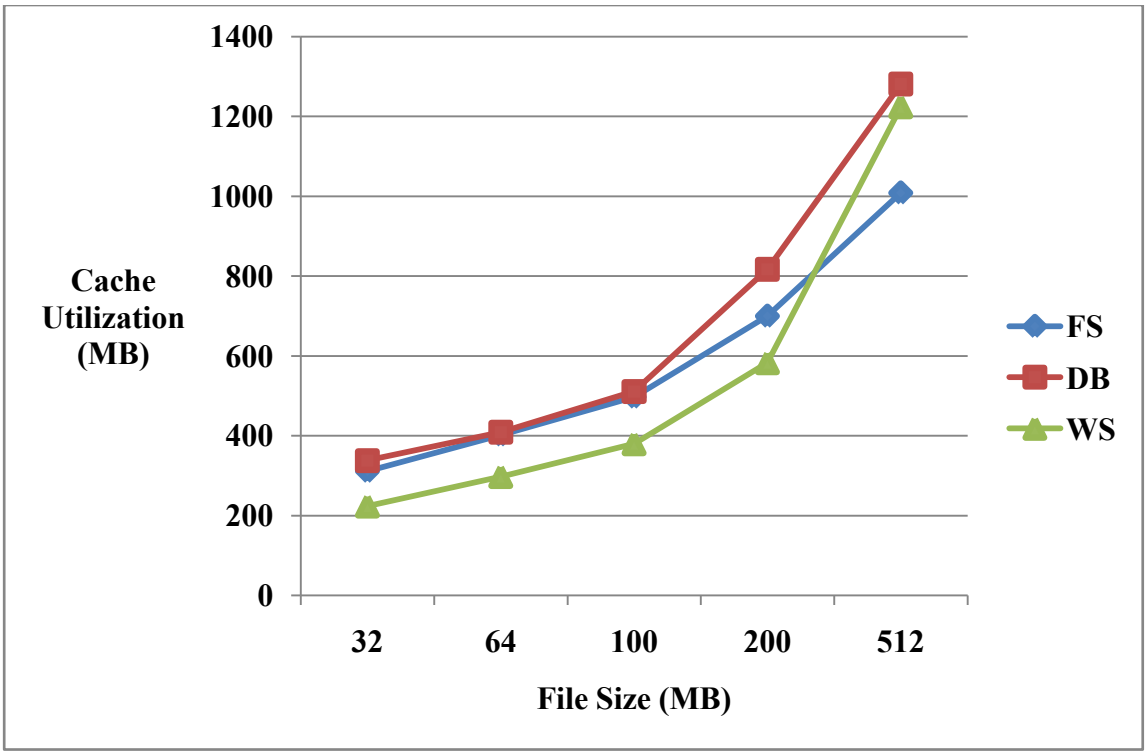

Figure 13: Cache Utilization with Dynamic Caching

7.2 Evaluation of Response Times

Response time can be considered as the time interval between the initiation of request by the user and the time at which it will be serviced. Response time is a major factor that decides the performance of any system. A lot of parameters are responsible for affecting the response time, of which, the cache and processor speed are the most significant ones. When cache is statically allocated, the file server took only 8.5 minutes to process the workload, whereas, the database server took 80 minutes for the same. This clearly indicates that the cache at the file server is lying dormant while that of the database server was exhausted. Similarly while the database server was processing a smaller block of data, file server was almost exhausted of its cache resources. Therefore, a balance has to be achieved among the three servers so that the overall performance of caching and the performance of the entire system improves.

Figure14 shows that the dynamic allocation of the cache improves the response time of the server, by adequately balancing the cache among the three servers. Dynamic allocation also helps reduce the number of capacity misses because, cache is allocated to the VM when there is a need for it, unless there is no cache space to allocate. Hence, the chance that a VM might run out of cache quickly is thinner than otherwise.

Database server with thin provisioning has shown a remarkable decrease of 25% in the response time, file server has shown a 17% decrease and the web server has shown a 14% reduction in the response time when dynamic caching was in place.

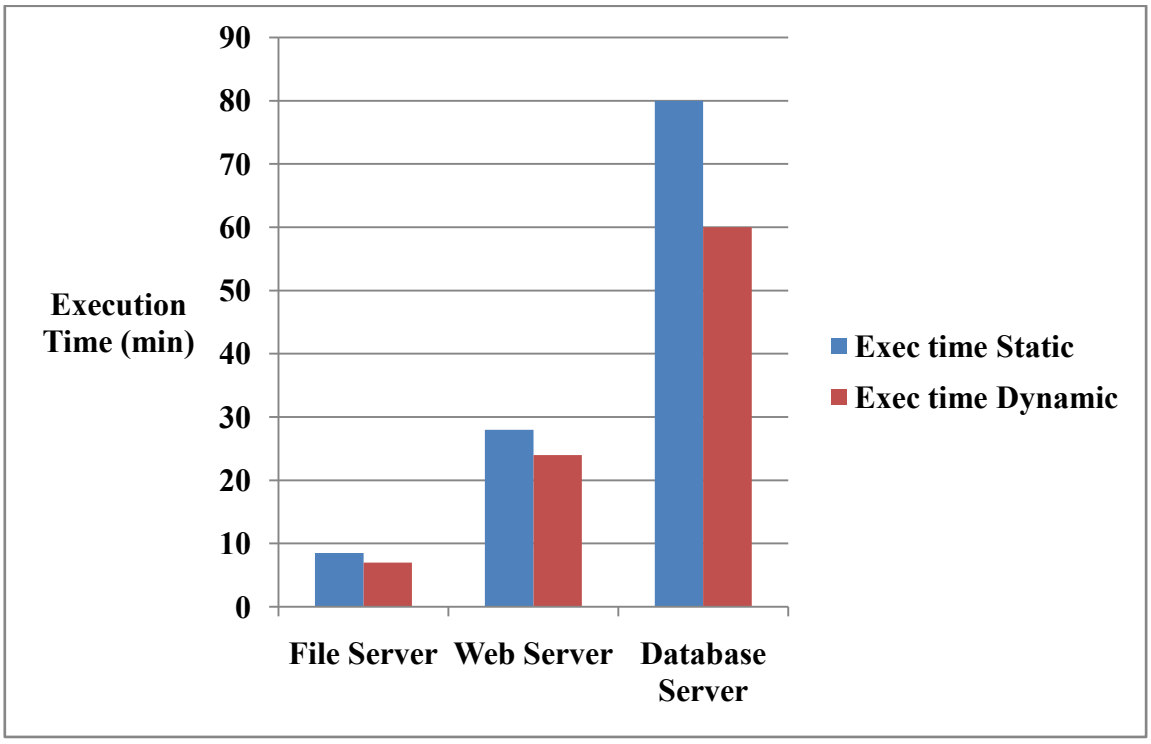

Figure 14: Response Time Comparison with Static and Dynamic Caching

7.3 Evaluation of Cache Utilization with respect to Time

The performance of the three servers considered varied considerably when they were tested with the thin provision cache allocation model. Figures 15,16 and 17 show the cache utilization with respect to time for statically allocated and dynamically allocated caching mechanisms for

the three servers. It can be seen that the file server, database server and the web server reach the maximum cache capacity much sooner in the static cache technique when compared to the dynamic allocation model. This is because, when the cache becomes full in the dynamic model, more cache can be allocated based on the availability of extra cache space. Hence, the dynamic caching using thin provisioning can help improve the performance of caching and the entire system.

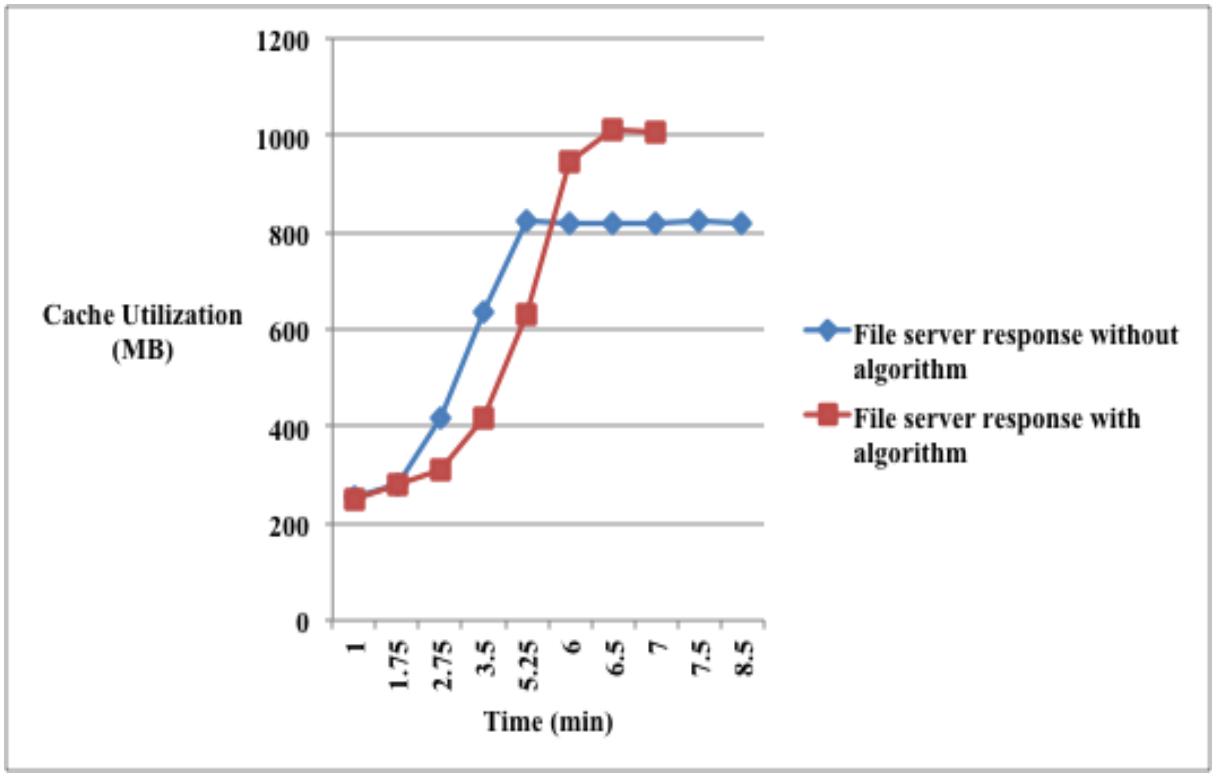

Figure 15: Cache Utilization w.r.t Time in File Server

As seen in the above Figure 15, the cache utilization for the file server increases with the dynamic caching in place, when compared to the cache utilization with static caching. The web server and database server show similar trends and are as shown in Figures 16 and 17 respectively.

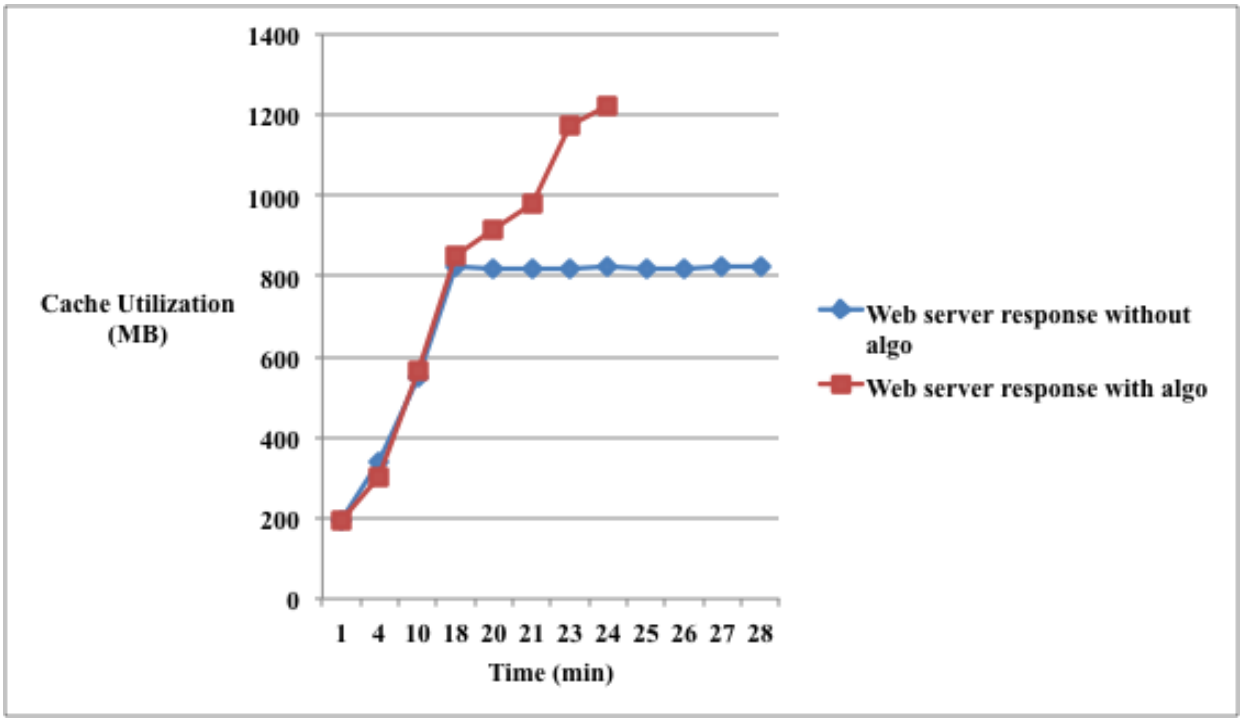

Figure 16: Utilization w.r.t Time in Web Server

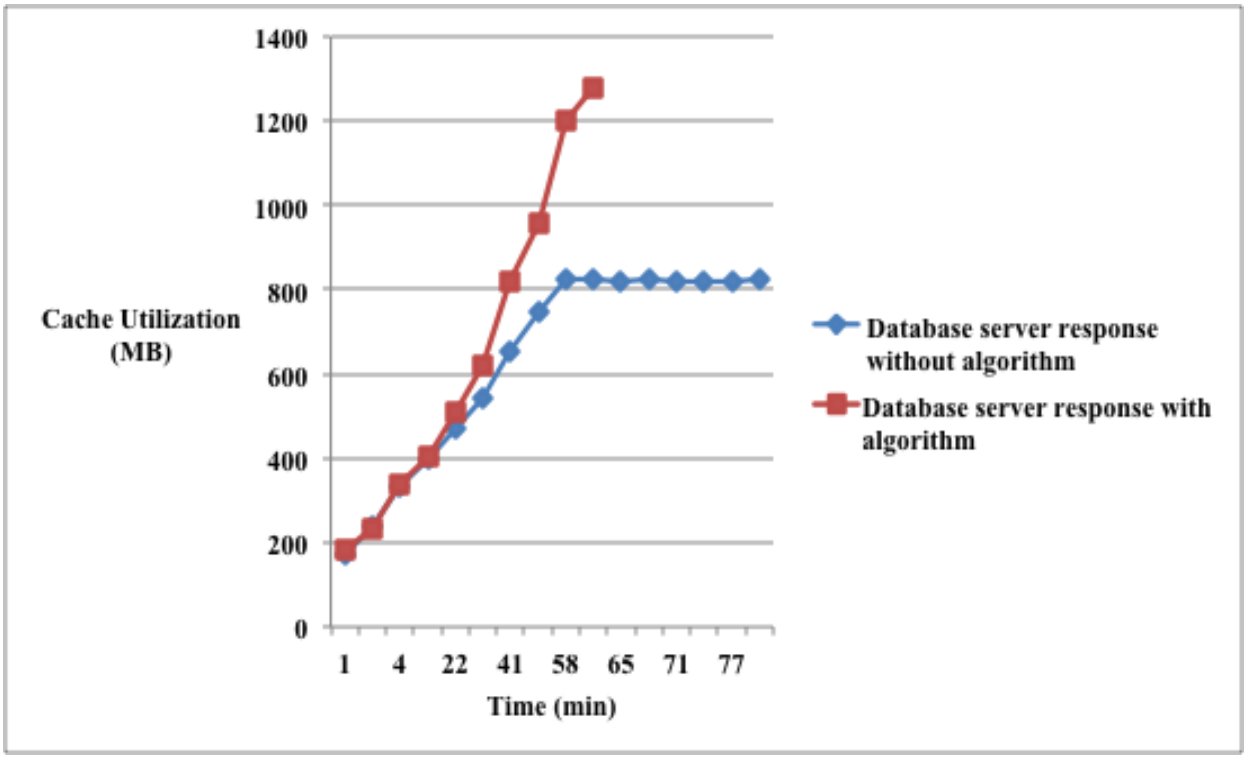

Figure 17: Utilization w.r.t Time in Database Server

As seen above, the dynamic cache allocation to virtual servers are very beneficial. In that, they not only help to improve the performance of the servers, but also make efficient utilization of an expensive resource such as a cache.

7.4 Evaluation of Cache Miss Ratio With and Without Hypervisor Cache Locking

7.4.1 Cache Miss Percentage with respect to Cache Size

The 4 workloads considered, namely, Array sort, FFT, Gzip and MP3 were evaluated for the percentage of cache misses when locking was in place and without cache locking in place. A 5% cache locking was initially used to evaluate the strategy. Later, the percentage of cache locking was varied from 5% upto 30% and the cache performance was recorded and the results of this are shown later. Figures 18, 19, 20 and 21 show the miss percentages for the 4 workloads when the cache size was varied from 2KB to 64 KB. The authors observed that implementation of cache locking in the hypervisor cache reduced the miss rate by about 1.7% for the array sort workload and by about 0.8% for the Gzip workloads when the cache sizes were 2 KB and 4 KB respectively, for a 5% cache lock. However, as the cache size increased, cache locking did not show considerable improvement in performance.

7.4.2 Cache Miss Percentage with respect to Associativity

The performance of the four workloads was evaluated with and without cache locking by varying the associativity of the hypervisor cache. A cache size of 4KB was used while varying the associativity, since a cache size of 4KB resulted in optimal performance as shown in the previous section. Associativity refers to the way the main memory blocks are mapped to the cache. A set associative cache was used for the purpose of this research. The associativity of the hypervisor cache was varied in terms of 2,4,8 and16 and the miss percentage for the various workloads was recorded. Figures 22, 23, 24 and 25 show the performance of the hypervisor

cache with respect to associativity for the four workloads used. As seen from the Figure 18, the array sort application resulted in 1.6% reduction in the number of cache misses for an associativity of 2, FFT showed a 0.6% reduction for an associativity of 4, Gzip showed a reduction of 0.8% in the miss percentage for an associativity of 4 and MP3 showed a 0.6% reduction in the percentage of cache misses with an associativity of 2 when cache locking was implemented. As the associativity increased, cache locking showed the same performance as no locking. An associativity of 4 for the locked hypervisor cache resulted in lesser number of cache misses for a majority of the workloads used.

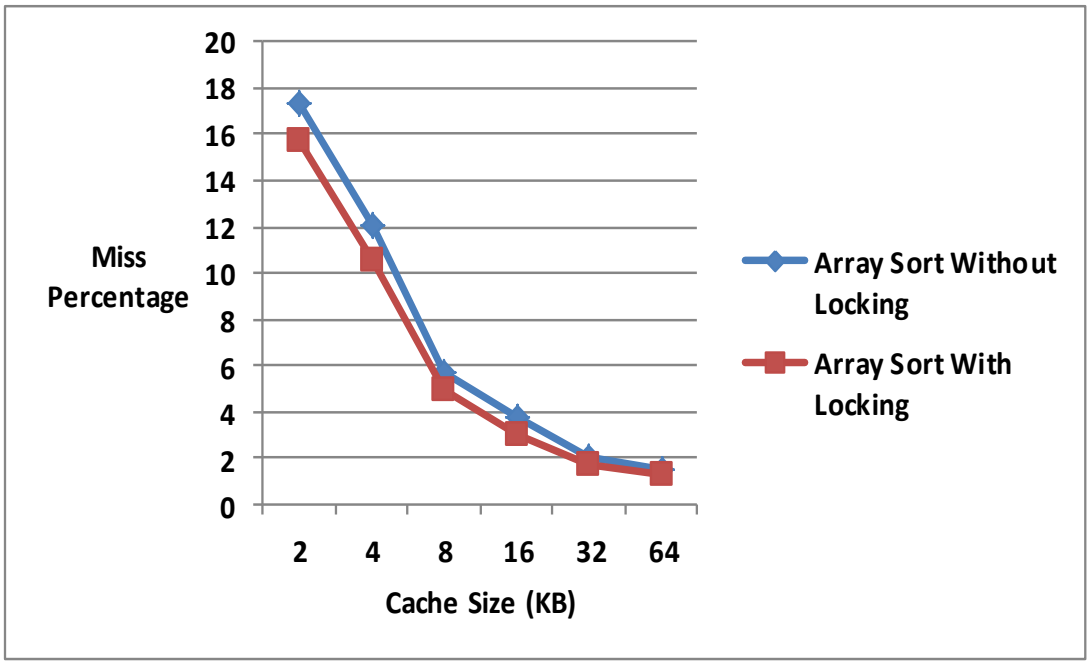

Figure 18:Miss Percentage for Array Sort

Since array sort consists of a lot of eviction operations, we can see from Figure 18 that cache locking does help in decreasing the number of cache misses.

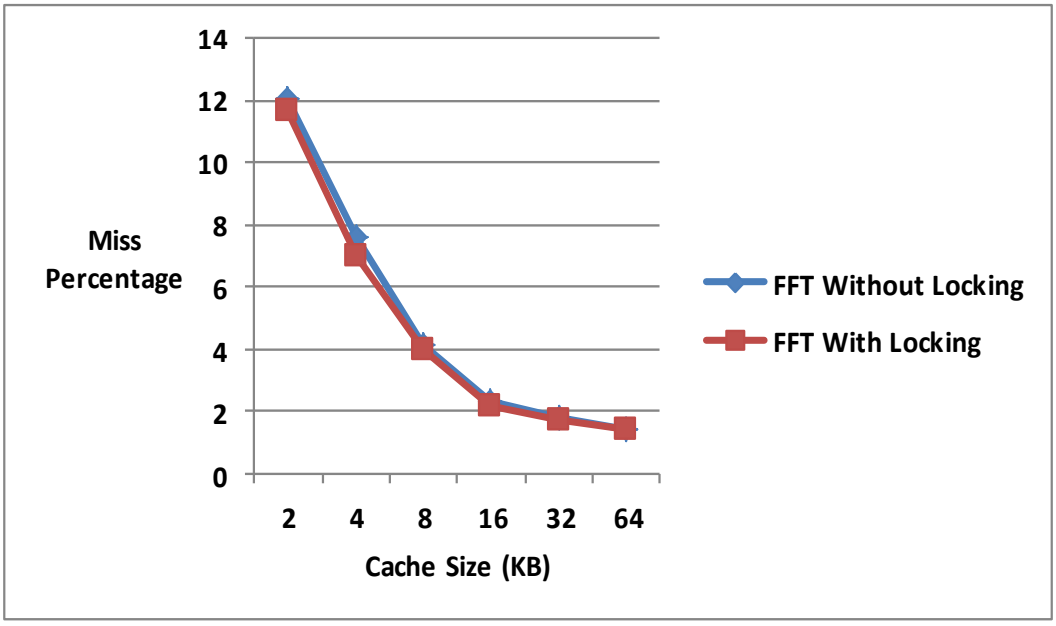

Figure 19:Miss Percentage for FFT

FFT workload is compute intensive and hence shows very little improvement in cache misses with cache locking. This is as shown in Figure 19.

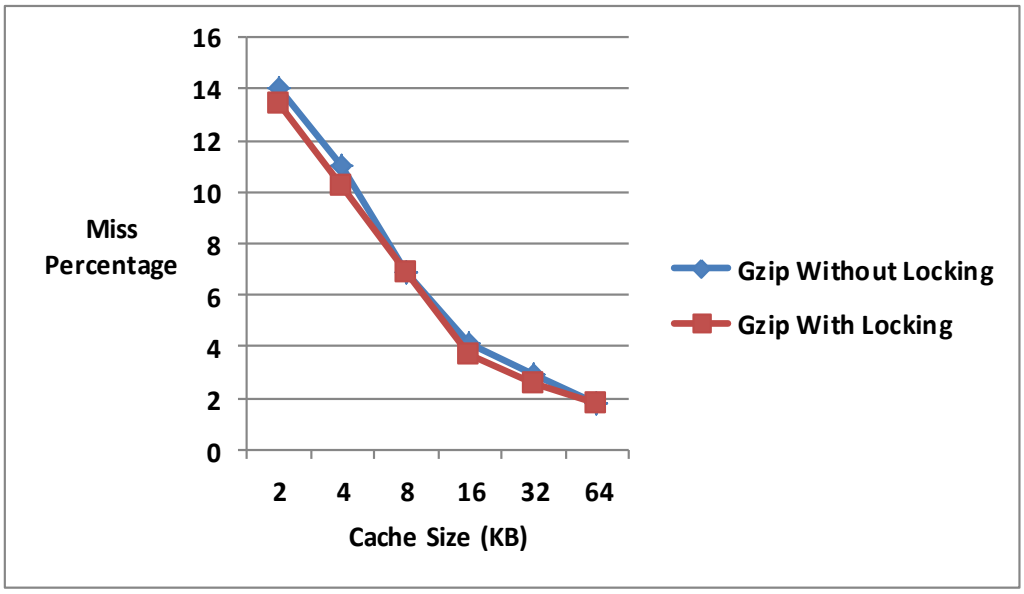

Figure 20:Miss Percentage for Gzip

Gzip workload is write intensive and shows some improvement with cache locking, but not considerable. This is because, the cache locking technique proposed in this research is implemented in the read cache, not in the write cache. Figure 20 shows this impact.

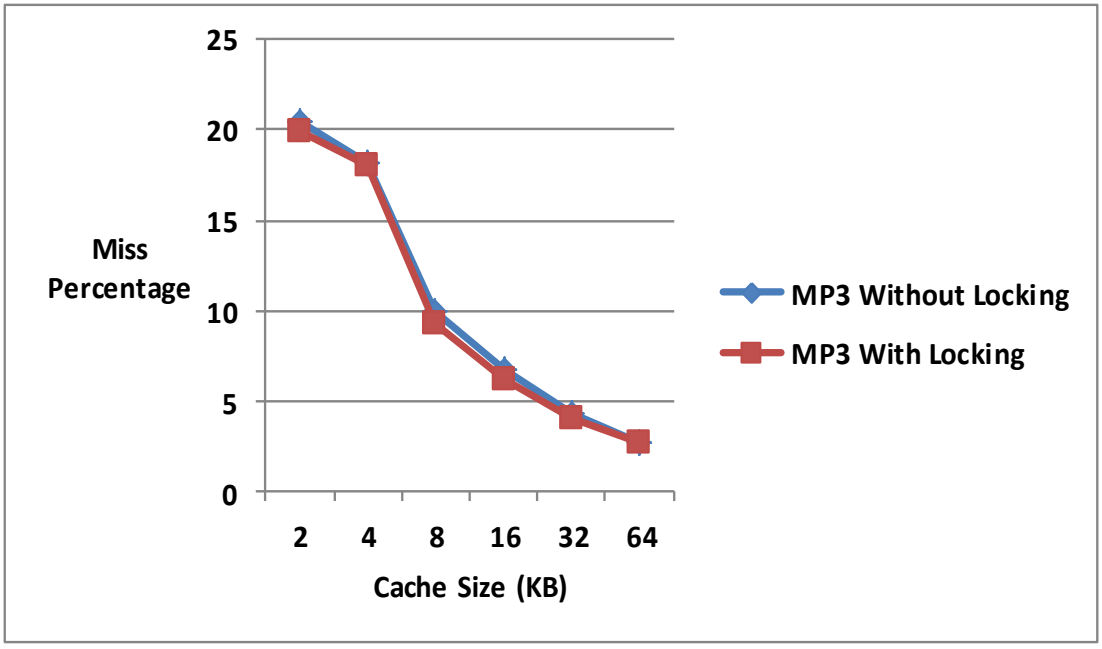

Figure 21:Miss Percentage for MP3

As seen in Figure 21, MP3 workload did not show considerable improvement with the cache locking technique. This is because, MP3 is a sequential workload, hence, the probability that a previous data block again in future is very less.

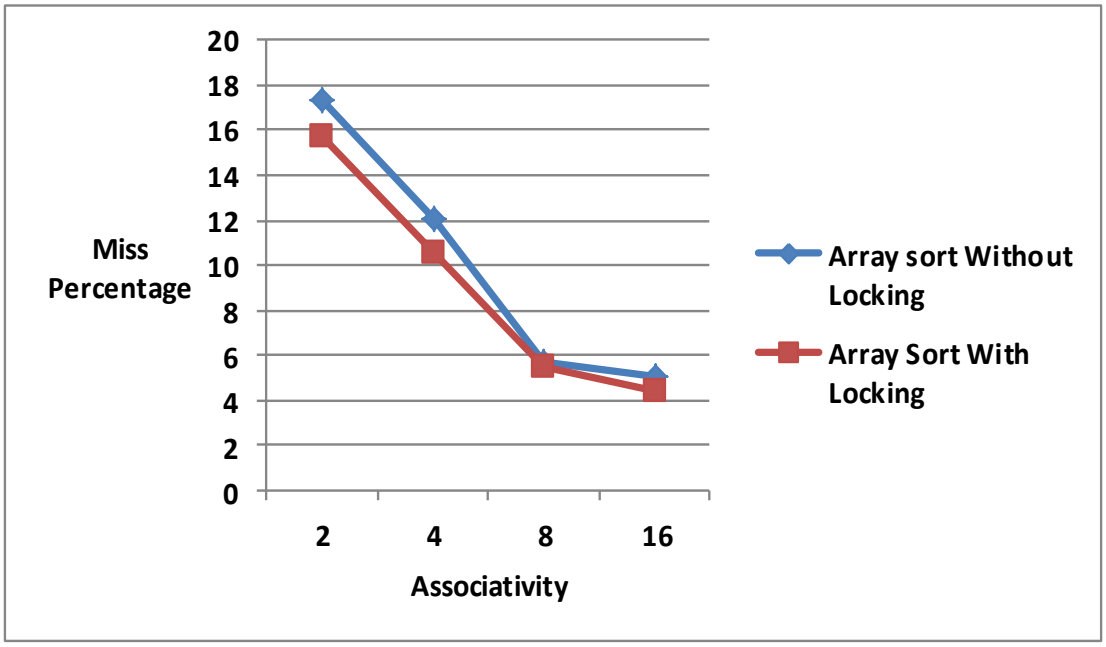

Figure 22: Miss Percentage for Array Sortwrt Associativity

A seen in the above Figure 22, an associativity of 2 benefits the cache locking technique.

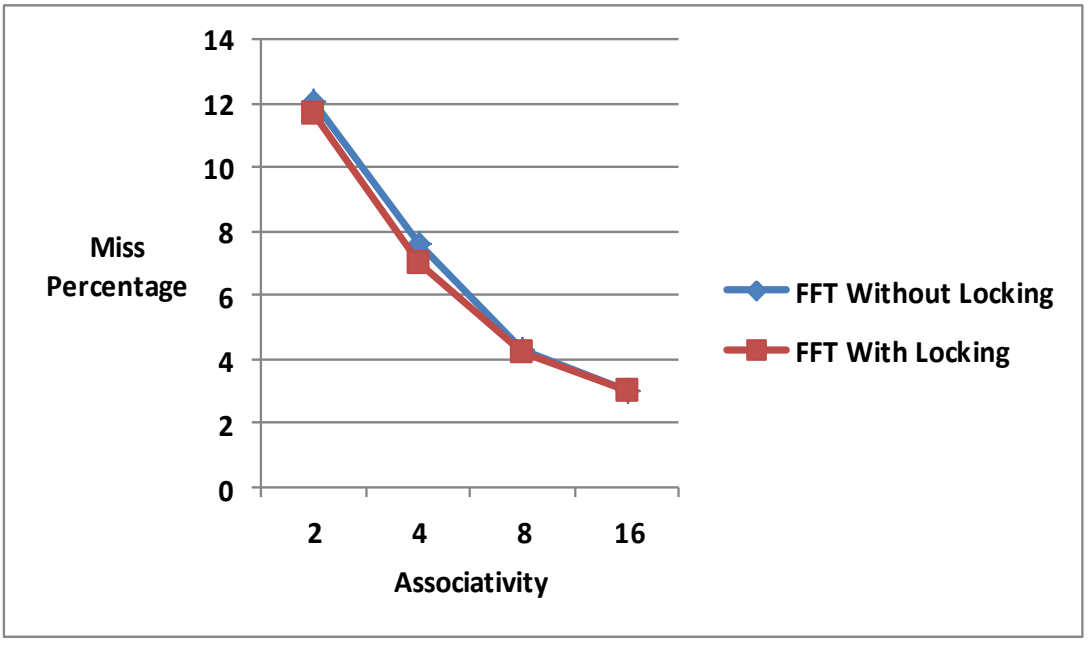

Figure 23: Miss Percentage for FFT

Unlike array sort, the FFT workload shows an advantage with the cache locking technique when the associativity is 4. This is seen in Figure 23.

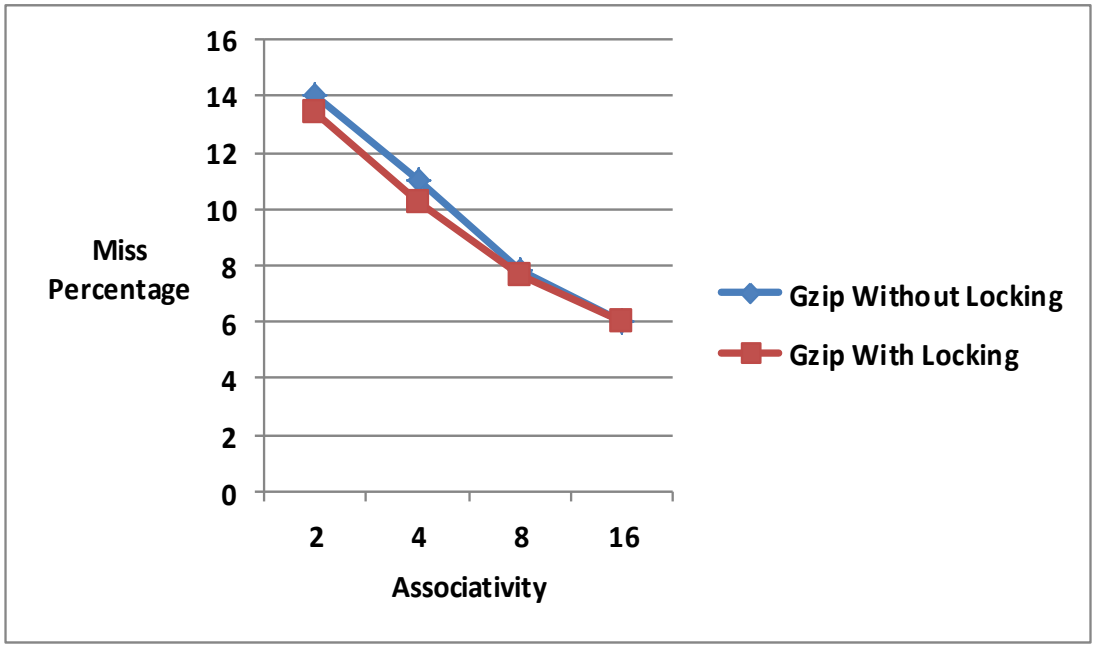

Figure 24: Miss Percentage for Gzip

As seen in Figure 24, Gzip workload shows a decrease in the cache miss percentage when the associativity is set to 8 or 16, but, when the associativity is 2 or 4, cache locking does not show any improvement.

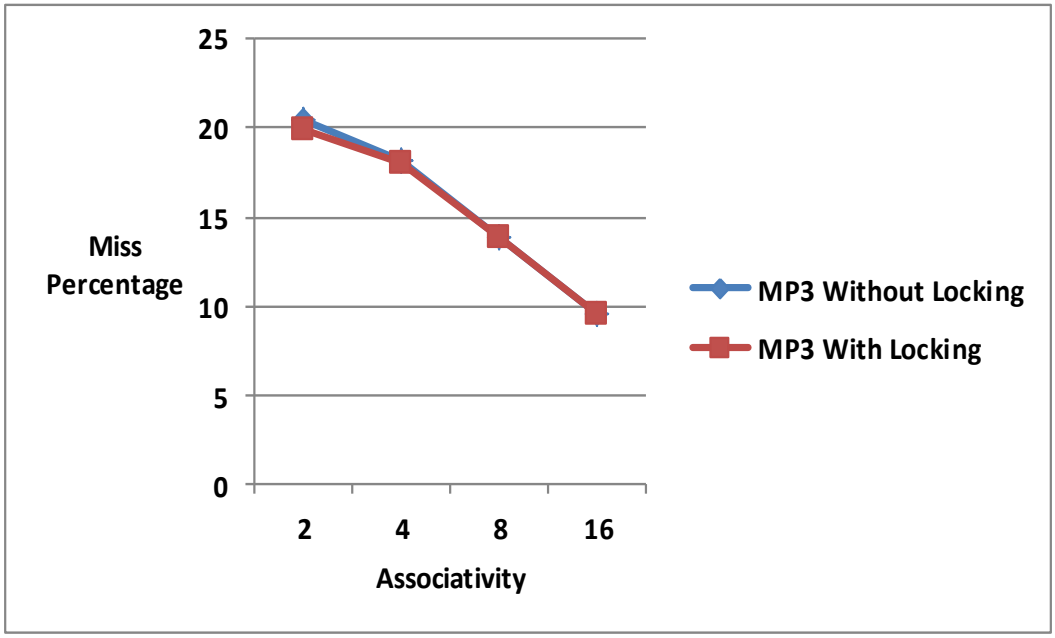

Figure 25: Miss Percentage for MP3

As seen in Figure 25, MP3 workload shows an improvement in the cache miss percentage when the associativity is 4.

As seen in the above figures, different workloads show benefits with the cache locking technique in place with different associativities. One needs to used an optimal value of associativity while designing the cache, depending on the types of workloads used.

7.5 Evaluation of Cache Miss Percentage with and without Cache Locking and VB Cache

In this section, the authors introduce a victim buffer cache on the hypervisor in addition to the main hypervisor cache. The victim buffer cache is a very small cache and holds the blocks of data that are evicted from the main cache. The cache miss percentage is recorded for the hypervisor with cache locking and without cache locking in the presence of a victim buffer cache.

7.5.1 Cache miss percentage with respect to Cache Size

When the victim buffer cache is added, the probability of finding a block of data in the cache increases. The workloads used in this research were evaluated with the caching locking technique and the presence of a victim buffer cache. When cache locking was in place and in the presence of a victim buffer cache, the array sort application showed a 3% decrease in the number of cache misses for a 2KB cache, the FFT showed a 0.9% reduction in the number of cache misses for a 4 KB cache, Gzip showed a 1.65% reduction for a 4 KB cache and MP3 showed a 2.3% reduction in the number of cache misses for a 4KB cache. As the cache size increased, cache locking did not show much of a performance improvement when compared to no locking. Figures 26, 27, 28 and 29 show this variation.

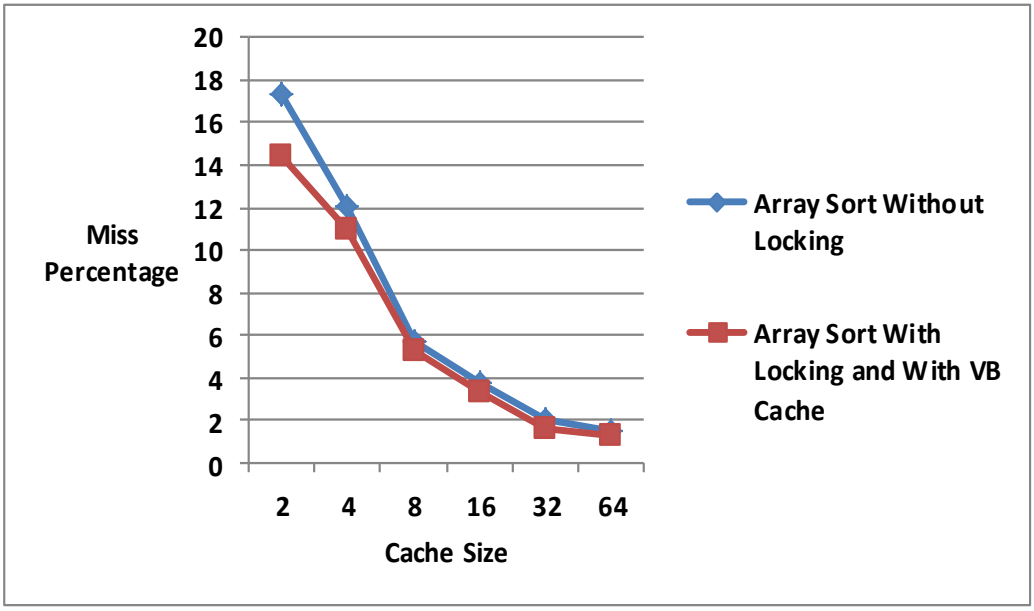

Figure 26:Miss Percentage for Array Sort

As seen from Figure 26, using a VB cache further improves the cache miss percentage in the array sort workload.

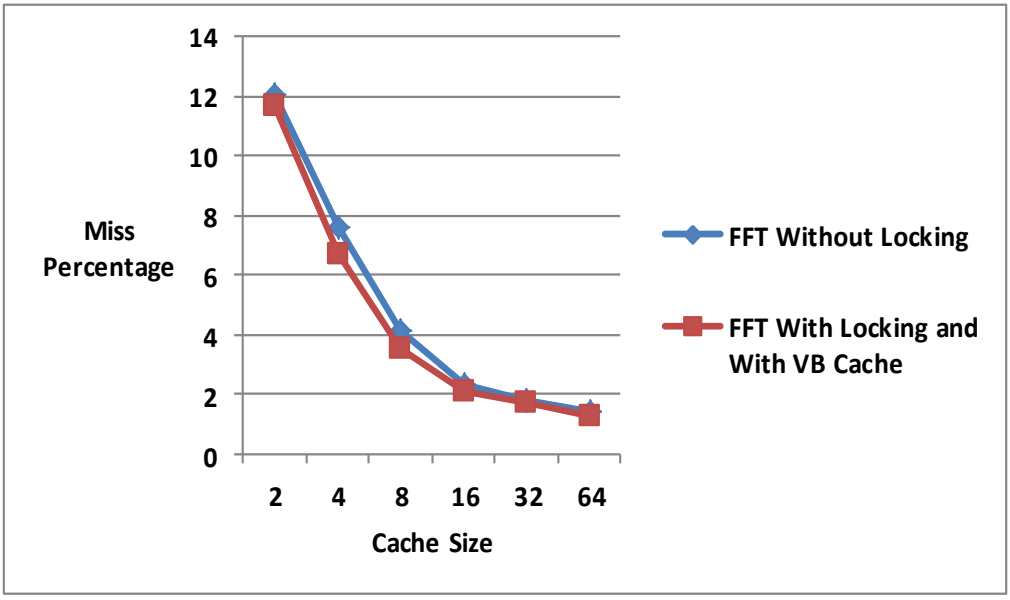

Figure 27:Miss Percentage for FFT

From Figure 27, we can see that the FFT workload does show a little bit of improvement in the presence of a VB cache when compared to no VB cache, for the same reasons stated before.

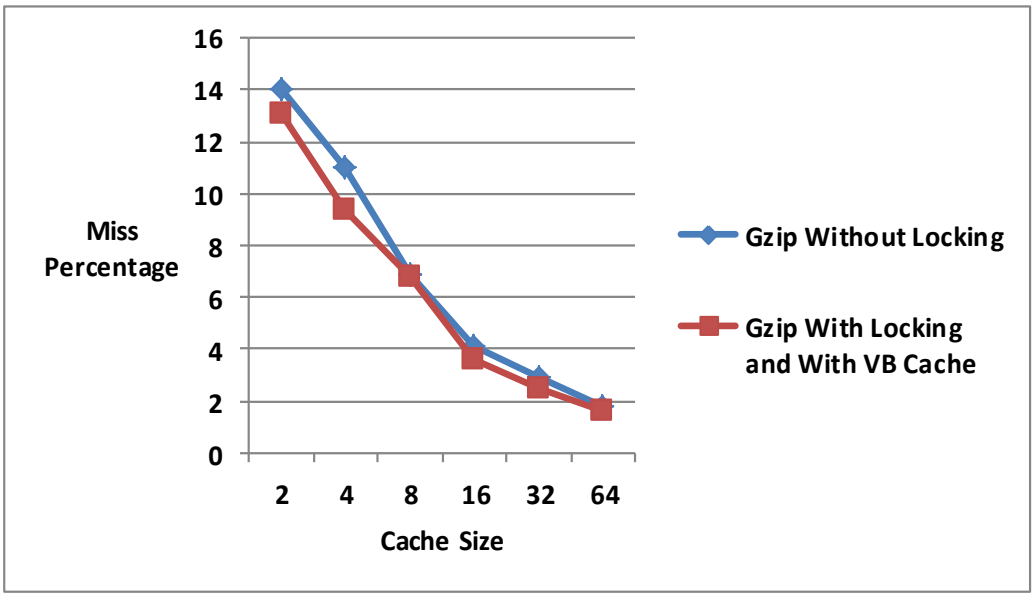

Figure 28 :Miss Percentage for Gzip

As seen in Figure 28, Gzip workload decreases the percentage of cache miss by 2% when the VB cache is used.

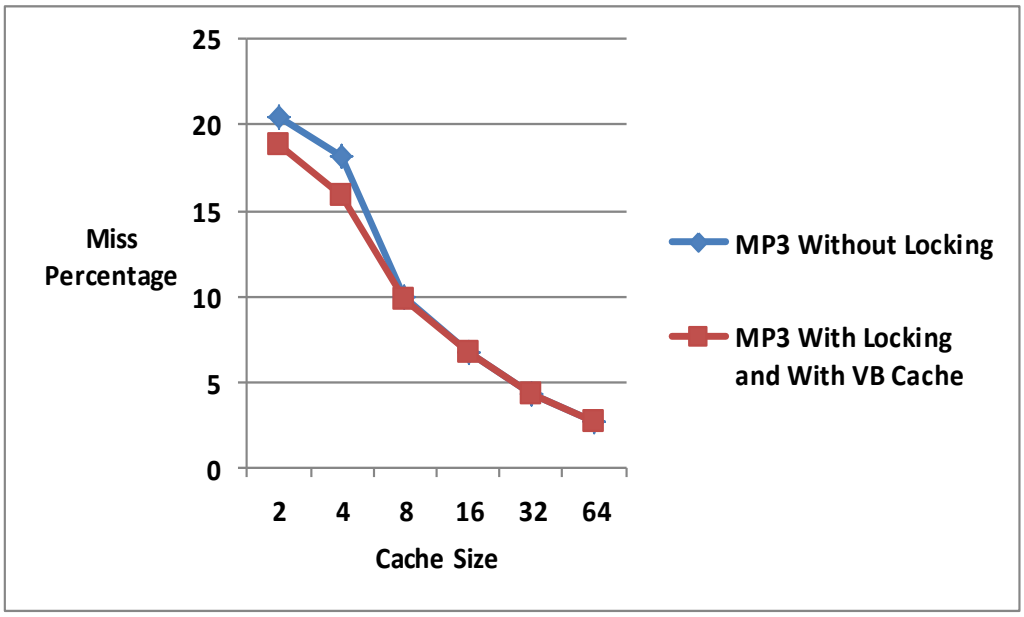

Figure 29:Miss Percentage for MP3

MP3 workload shows a small improvement in the cache miss percentage in the presence of a VB cache. This is shown in the Figure 29.

7.5.2 Cache Miss Percentage with respect to Associativity

When the victim buffer cache was present, the associativity of the hypervisor cache was varied to see the effect on cache locking. A cache size of 4KB was used for the experimentation. Figures 30, 31, 32 and 33 show the effect of varying associativity on the miss percentage. As seen from the figure 5, the presence of a victim buffer cache and an associativity of 2, 4, reduces the miss percentage by 3% in the array sort workload, 1% in the FFT, 1.7% in Gzip and 2.3% in the MP3 workload.

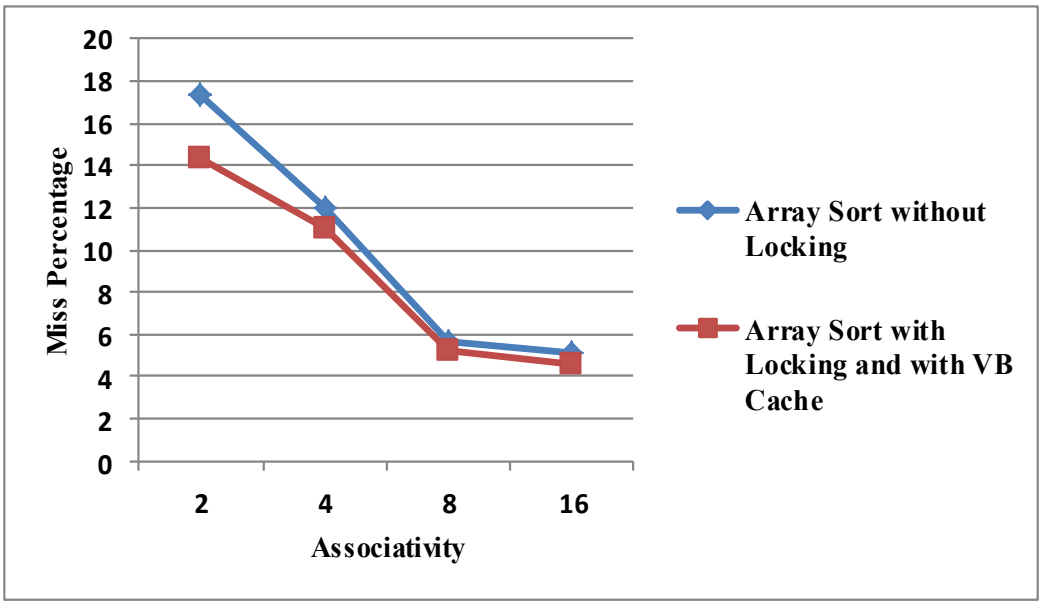

Figure 30:Miss Percentage for Array Sort

The authors of this research wanted to see what effect associativity has in the presence of a VB cache. As seen from Figure 30, the array sort workload showed benefits with cache locking when the associativity was set to 2.

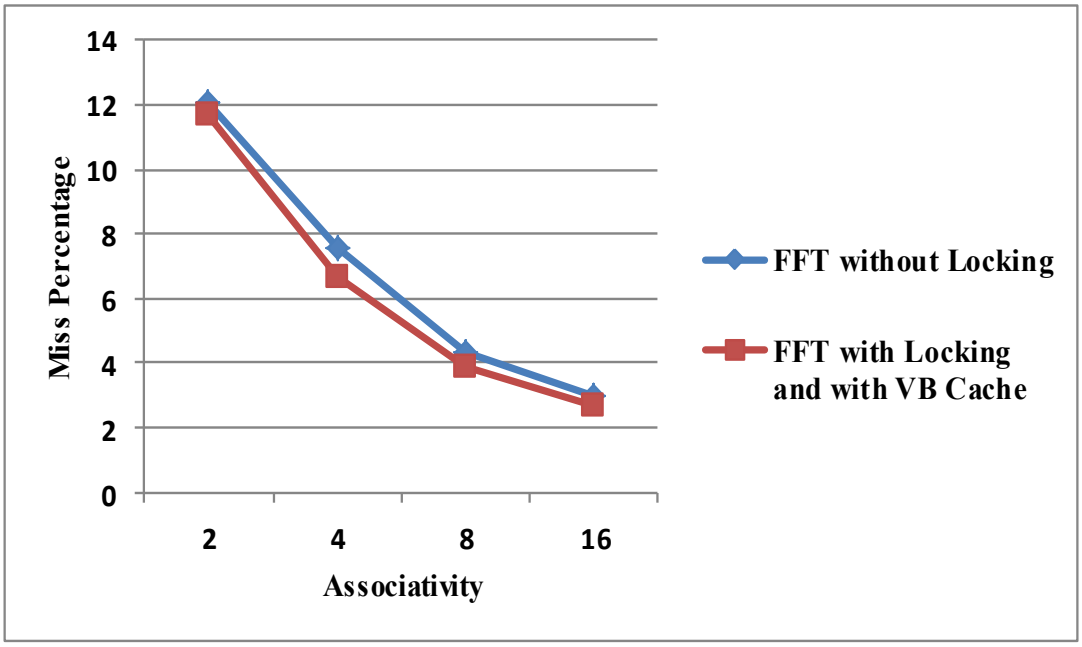

Figure 31: Miss Percentage for FFT

The FFT workload showed an improvement when the associativity was set to 4 as seen in Figure 31.

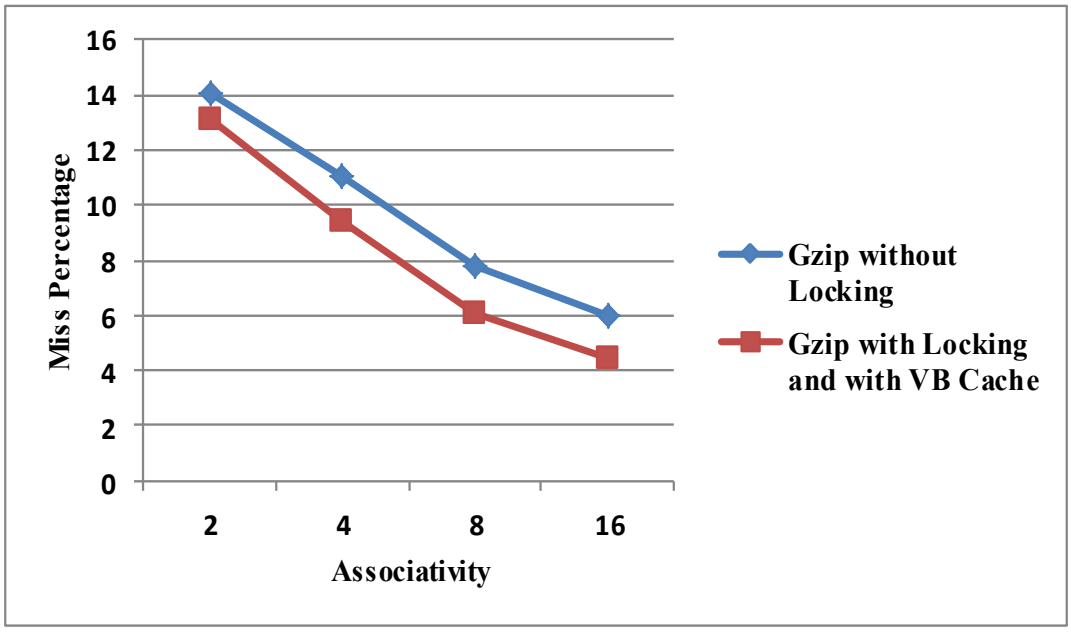

Figure 32: Miss Percentage for Gzip

Gzip workload showed the same performance as not using a VB cache even when VB cache was used. An associativity of 8 and 16 showed a considerable decrease in the number of cache misses.

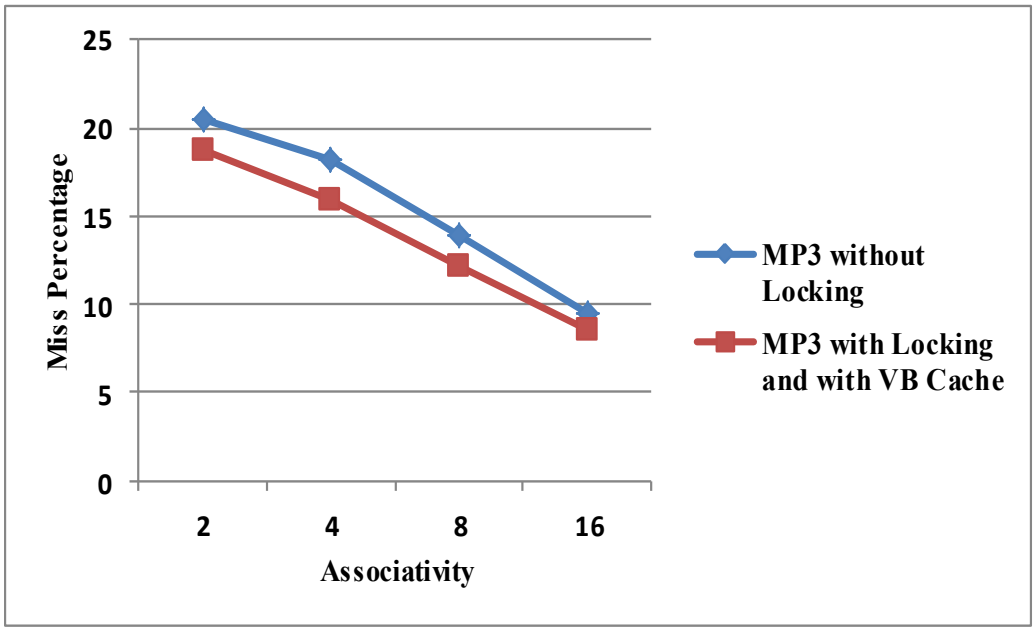

Figure 33:Miss Percentage for MP3

The performance showed by MP3 workload with an associativity of 4 and the presence of a VB cache is the same as the performance in the absence of the VB cache.This is shown in Figure 33.

7.6 Evaluation of Response Times with and without Cache Locking

When the application running on a virtual server encounters a cache hit, the required data block is fetched from the cache. Hence, by increasing the number of cache hits, the application response time can be reduced. If the required data block is not found in the cache, then cache miss occurs and results in an I/O request. I/O requests fetch the desired data from an external disk array connected to the physical server through protocols such as Fibre Channel (FC), SAS, iSCSI etc. The time needed for an I/O request is given by equation 3 [12]:

$$T_{I/O} = MC_M + (1 - M)C_H \approx MC_M \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (3)$$

Where 'M' is the hypervisor cache miss ratio, C_M is the cost of a cache miss, C_H is the cost of a cache hit. The cost of a cache miss is the time taken to fetch the desired block of data from the disk array. Hence, C_H can be written as shown in equation 4:

$$C_H = 2T_{FC} + T_{SEEK} + T_{ROT} + T_{cc} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (4)$$

Where T_{FC} is the transmission delay on the Fiber Channel link, T_{SEEK} is the seek time on the disk, T_{ROT} is the rotational latency and T_{cc} is the time to search the controller cache on the disk array for the required data block. If the block is found in the controller cache, then the seek time and rotational latency are not involved and is equal to 0.

Figure 34 shows the total I/O response time without cache locking, with cache locking and with cache locking in the presence of a VB cache. Total I/O response time is the total time taken to access the desired data blocks from the disk array when the data request encounters a cache miss on the VM. As seen from Figures 34, 35,36 and 37, cache locking with VB cache decreases the total response time when compared to an unlocked cache and cache locking without a VB cache. The response time saving vary for the different workloads for different cache sizes.

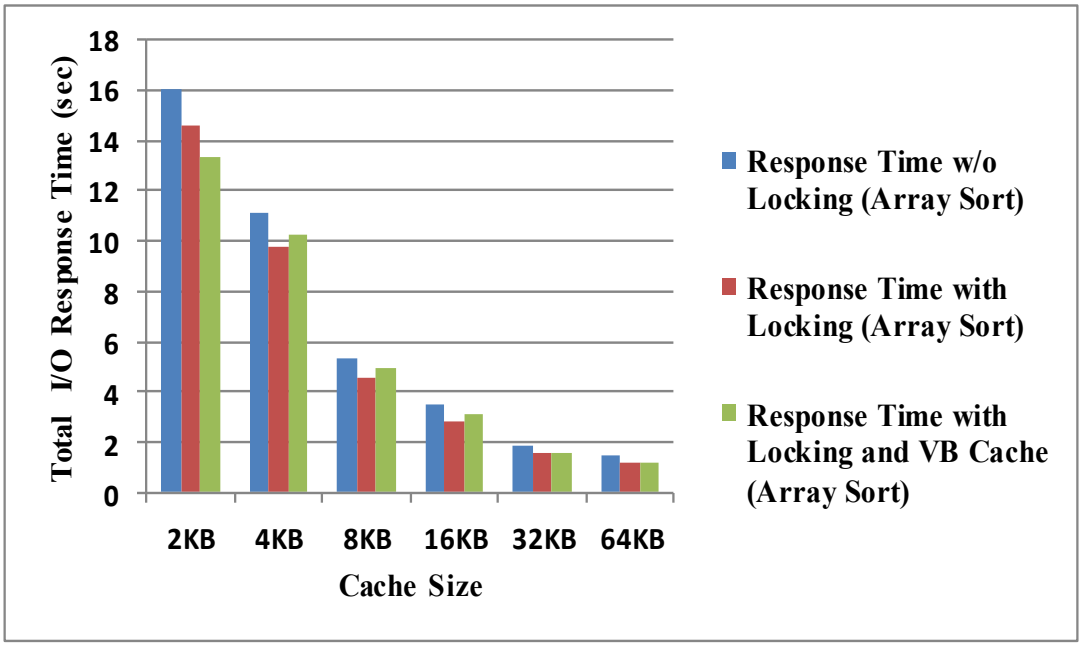

Figure 34:I/O Response time for Array Sort

As seen from Figure 34, the response time varies with the cache size. For the array sort workload, a cache size of 2 KB shows the greatest benefit of using cache locking and VB cache in terms of reducing the I/O response time.

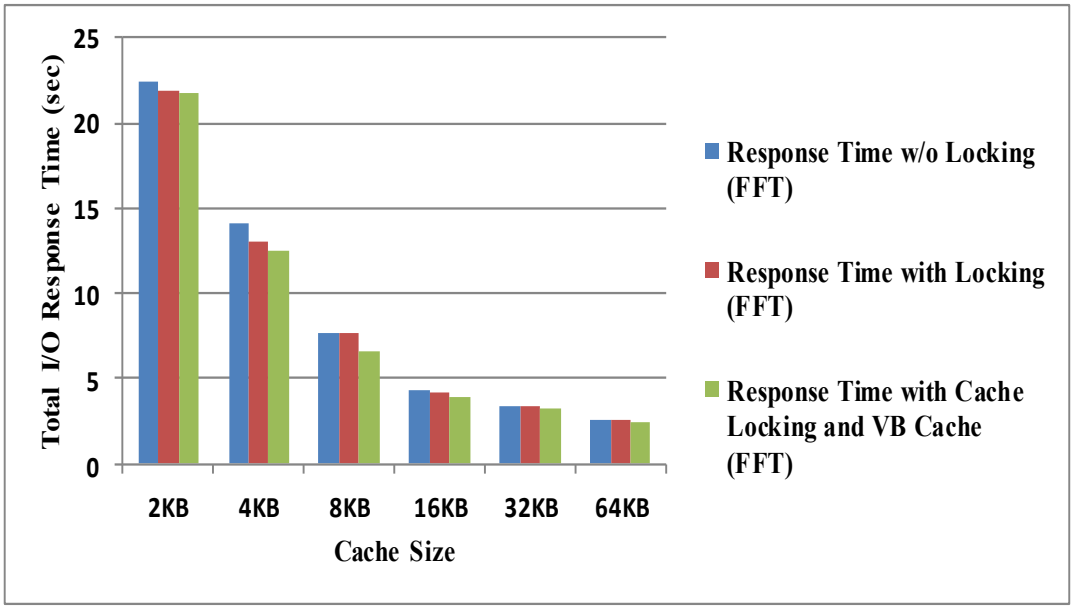

Figure 35:I/O Response time for FFT

As seen from the above Figure 35, for the FFT workload, a cache size of 4KB shows the greatest benefit of reducing the I/O response time in the presence of a VB cache and cache locking.

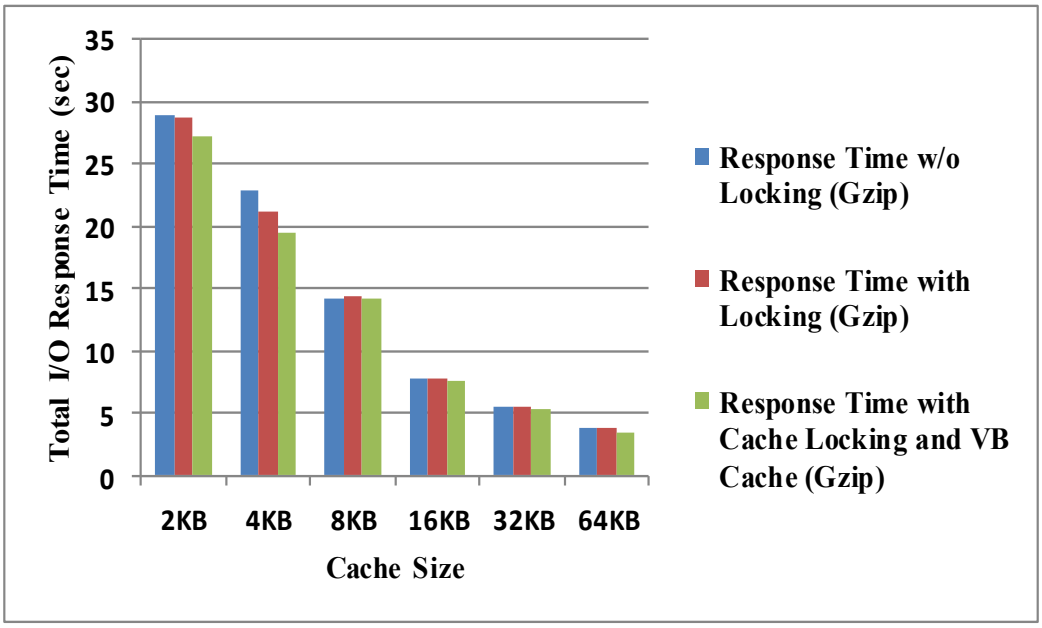

Figure 36:I/O Response time for Gzip

For the gzip workload, a cache size of 4KB proves beneficial in alleviating the I/O response time. This is seen in Figure 36.

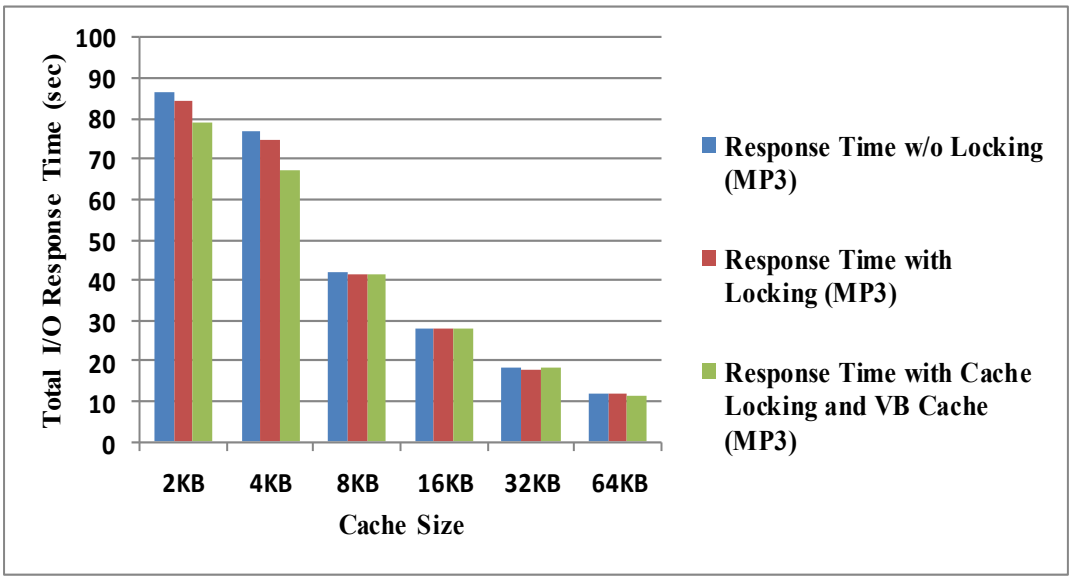

Figure 37:I/O Response time for MP3

As seen in Figure 37, a cache size of 4KB helps the MP3 workload as well, in reducing the I/O response time.

7.7 Cache Hit Percentage for Varying Percentages of Cache Lock

The percentage of cache locked plays an important role in determining the cache performance, in addition to the other factors mentioned above. For some workloads, a higher percentage of cache lock leads to better performance, while in some other workloads, a lower percentage of cache lock leads to better performance. A hypervisor cache size of 4KB was used and the percentage of cache lock was varied for the four workloads from 5%-30% with and without a VB cache and the results were analyzed. The results obtained in the previous sections were obtained with a 5% cache lock and hence, the benefits were not clearly visible in terms of the cache hit percentage. In this section, the percentage of locking was varied and the cache hit percentage for varying percentages of cache lock was recorded as shown in Figures 38, 39, 40 and 41.

A cache size of 4KB was found to be optimal and gave good performance results. Hence, a 4KB cache was selected for locking purposes. In the presence of a VB cache, the array sort workload showed a highest number of hits when the cache is 15% locked, FFT and Gzip showed the highest number of hits when the cache is 30% locked and MP3 showed the highest number of hits when the cache is 5% locked. This analysis helps to further develop a dynamic locking strategy for the hypervisor caches. As mentioned earlier, a static locking strategy was implemented in this research where certain cache lines are locked for the entire execution of the workload.

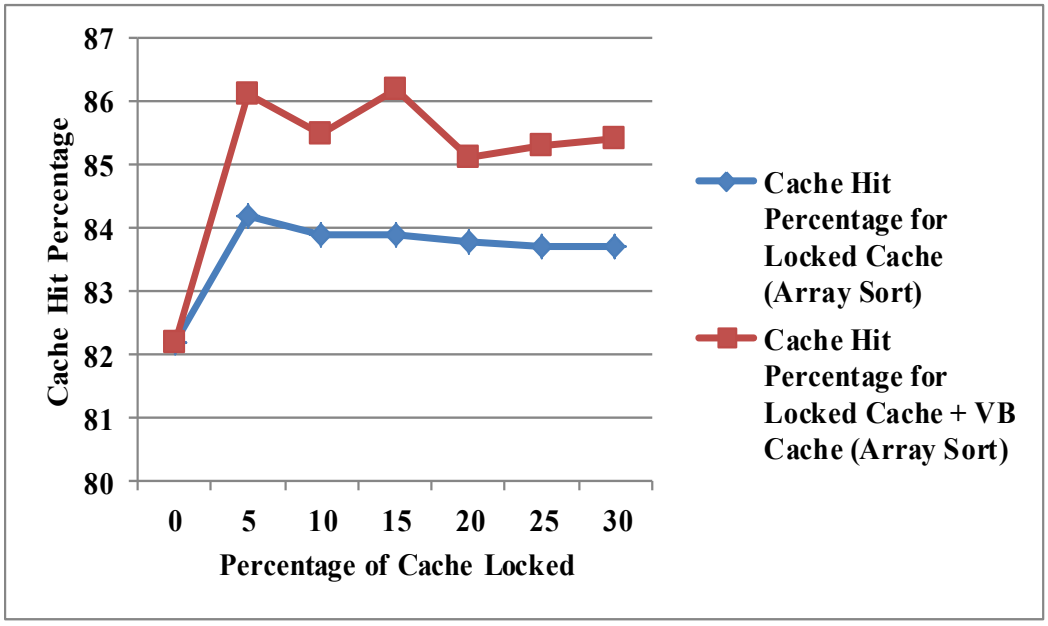

Figure38:Cache Hit Percentage for Array Sort

When 15% of the cache is locked, the array sort shows an increase in the number of cache hits, as seen in Figure 38.

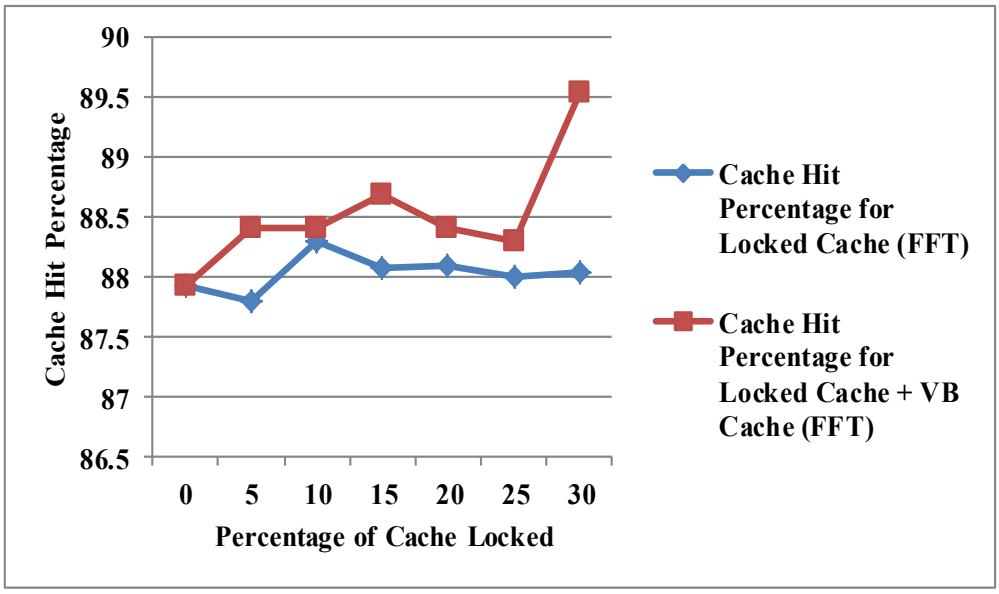

Figure 39: Cache Hit Percentage for FFT

A 30% cache lock, shows an improvement in the number of cache hits for the FFT workload as seen in Figure 39.

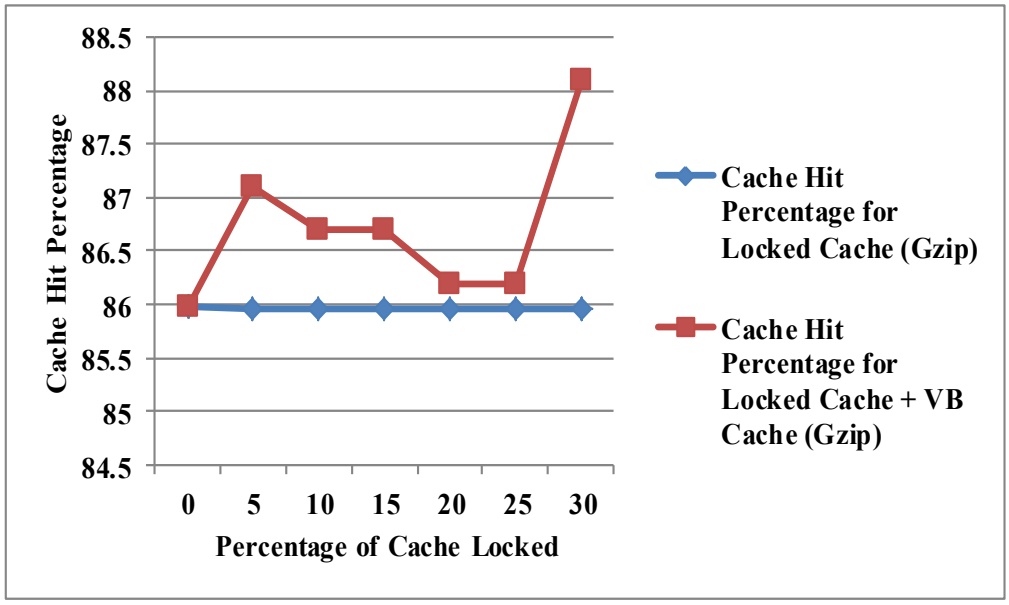

Figure 40:Cache Hit Percentage for Gzip

As seen in Figure 40, a 30% cache lock shows the greatest benefit of increasing the number of cache hits for the Gzip workload.

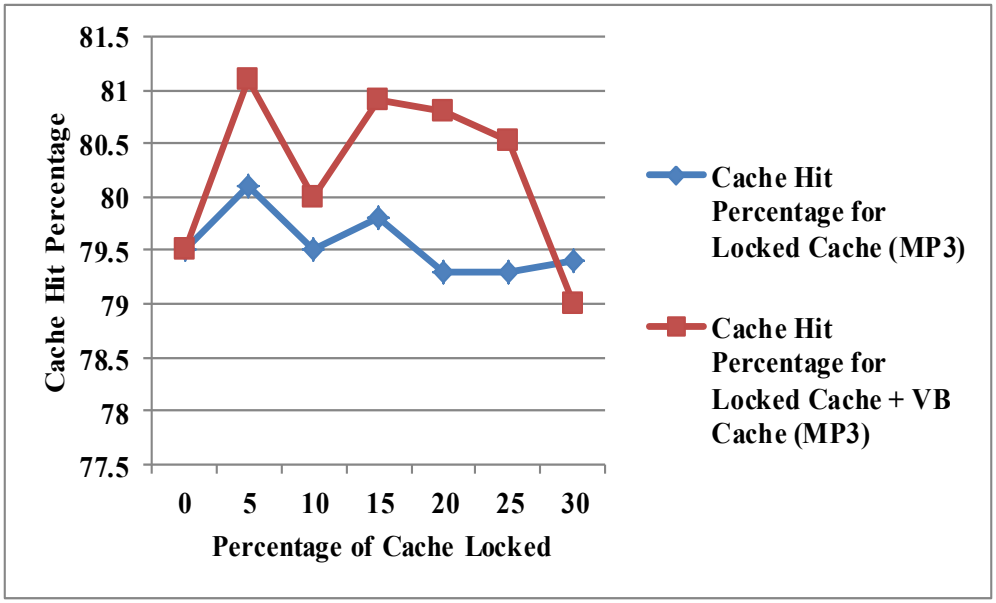

Figure 41:Cache Hit Percentage for MP3

A 5% cache lock has showed an increase in the number of cache hits for the MP3 workload as seen in Figure 41.

7.8 Probability Based Cache Replacement

In the results obtained in the previous sections, the Least Recently Used (LRU) cache replacement policy was used. In the LRU scheme, a block of data in the cache that was used well past in time is evicted out of the cache. In order to determine which blocks are recently used, counters are maintained and are updated frequently. Experimental results showed that, a probability based cache replacement in the hypervisor caches reduces the number of misses further, for optimal cache size of 2KB and 4KB. Figures42, 43, 44 and 45 show the number of misses when LRU is used when compared with the probability based cache replacement algorithm. As seen from the results, a probability based cache replacement algorithm outperforms the traditional LRU algorithm since, it can predict a block that will be less frequently used. Such a block with a low probability of access is not required to be cached and is evicted from the hypervisor cache.

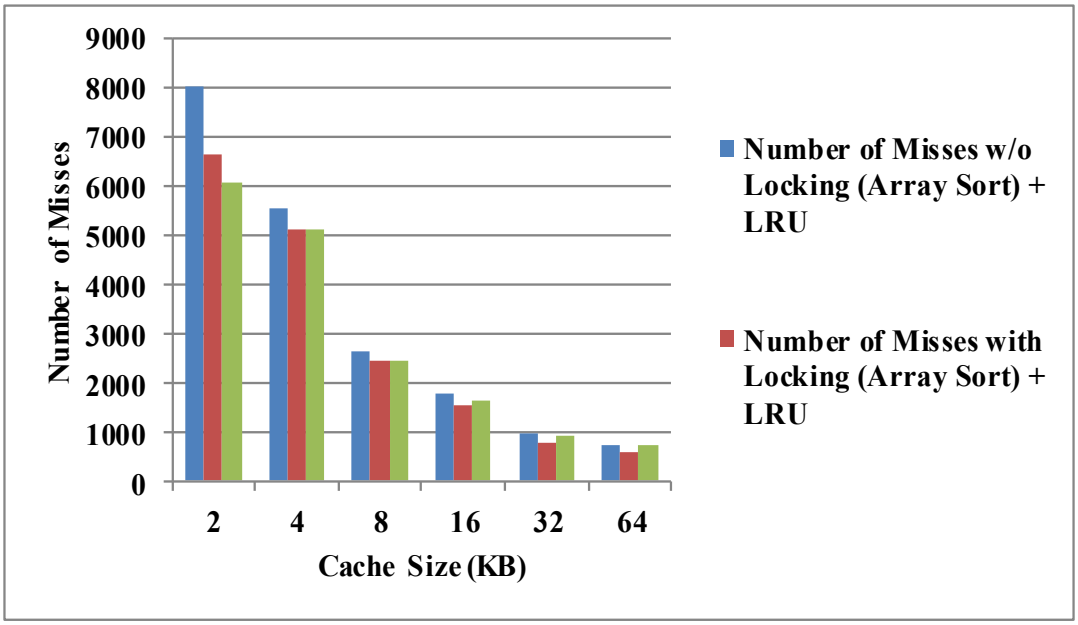

Figure 42:Number of Misses for Array Sort

As seen from Figure 42, a 2KB cache shows a reduction in the number of cache misses in the presence of a VB cache, cache locking and probability based cache replacement algorithm.

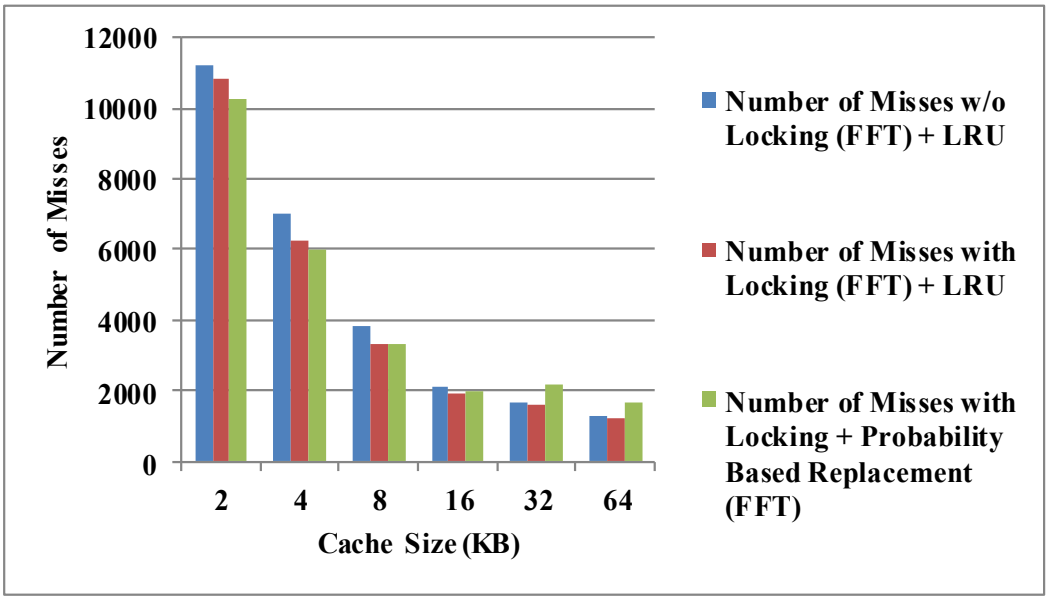

Figure 43:Number of Misses for FFT

A cache size of 4 KB helps with the FFT workload in the presence of a VB cache, probability based cache replacement algorithm and cache locking. This is seen in Figure 43.

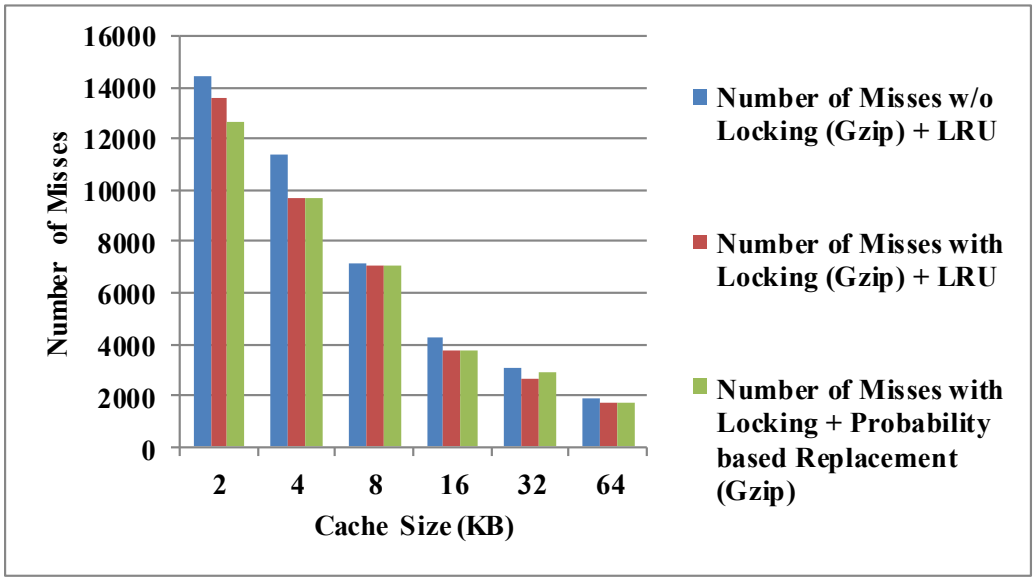

Figure 44:Number of Misses for Gzip

A 2KB cache works best for the Gzip workload as seen in Figure 44.

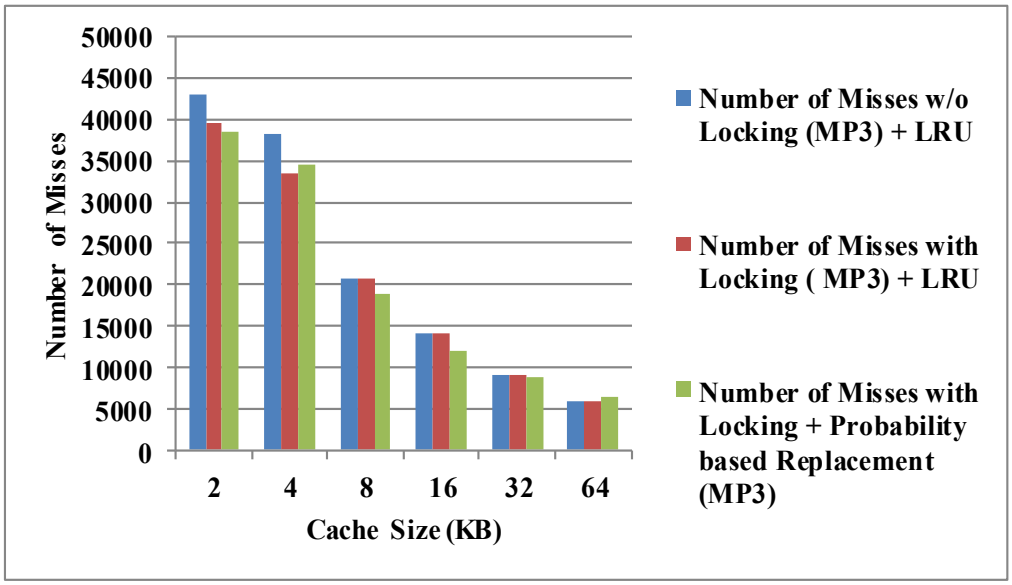

Figure 45:Number of Misses for MP3

A cache size of 2KB has proven beneficial for the MP3 workload as seen in Figure 45.

7.9 Discussion of Results

This section discusses the cache locking performance results obtained for the hypervisor cache when the four different workloads were run on the virtual servers. Apart from what is obvious and already discussed in the previous sections, there are a few general and interesting observations that need to be noted. The hypervisor cache locking technique proposed in this research tries to achieve a balance in optimizing the cache size while trying to improve the performance of caching using a relatively smaller cache size. A smaller cache is less expensive when compared to a larger cache and is ideal to be used on hypervisors. An optimal sized cache and lower associativity reduces the overhead on the hypervisor in maintaining the cache and the search time inside the cache is minimized. Experimental results show that cache locking produces optimal results when a hypervisor cache size of 2 KB - 4 KB is used with an associativity of 2, 4 for the system in place. Furthermore, the experiments performed in this

research show that different workloads show variation in performance for different percentages of cache locked. For most or all of the workloads considered in the research, a cache lock percentage in the range of 5% - 30% shows a drastic improvement in the cache hit rate.

It can also be observed that, the presence of a VB cache further improves the performance of the virtualized system in terms of reducing the number of misses and decreasing the I/O response time. Hence, having a VB cache on the hypervisor can result in good and noticeable performance improvements.

Overall, hypervisor cache locking technique has shown good results in reducing the miss rate when an optimal percentage of cache is locked. Yet another important benefit of cache locking lies in the fact that the total I/O response time for an application running on the virtual server is reduced. Even a small reduction in the I/O response time is very significant in a storage area network, since external disk accesses can be very expensive and may degrade the performance if not addressed. Hence, an optimal hypervisor cache size of 2 KB – 4KB, an associativity of 2, 4 and a cache lock percentage between 5% - 30% can improve the performance of hypervisor cache in the presence of a VB cache for the configurations used in this research. Results also show that when a probability based cache replacement algorithm is used as opposed to the traditional LRU, the number of misses can be further reduced, resulting in a better cache performance. The above results were recorded on a 64-bit processor system. The results may vary to some extent on a 32-bit machine.

CHAPTER 8

CONCLUSION AND FUTURE WORK

8.1 Conclusion

In this research, the authors first proposed a hypervisor level cache in a virtual machine environment and later proposed dynamic caching in hypervisor caches. The authors then studied the performance of cache locking in hypervisor caches. Static caching in virtual machines can lead to poor utilization of the cache. Dynamic caching proposed in this technique has shown great improvements in the cache efficiency. Cache locking is a technique used to improve the performance of caching by locking some cache lines that could potentially decrease the number of cache misses. A miss table was implemented to determine the cache lines that need to be locked. The locked cache lines are essentially those blocks of data that incur the most number of misses. This is because, the cache lines that incur the most number of misses are the blocks that are most needed by the applications running on the VMs. The performance of caching when locking is implemented is studied for the parameters – miss percentage and I/O response time. Later, a victim buffer cache is introduced and the performance of cache locking is studied. Results show that cache locking results in a marginal decrease in the miss ratio when about 5% of the cache is locked. However, as the percentage of locking increases to about 30%, the number of cache misses with locking and in the presence of VB cache is considerably decreased.

In the presence of VB cache, cache locking results in a considerable decrease in the total I/O response time. A decrease in the I/O response time is of a lot of significance in a Storage Area Network setup where data blocks are accessed from external disk arrays. Also, when a probability based cache replacement algorithm is used as opposed to the traditional LRU, the number of misses can be further reduced, resulting in a better cache performance. The authors conclude by saying that cache locking in hypervisor caches aims to improve the performance of

caching by using an optimal cache size. A smaller cache is favorable and less expensive than a larger cache. Hence, cache locking with an optimal percentage of lock is very effective in improving the performance of caching and also alleviating the total I/O response time.

8.2 Future Work

The cache locking percentage used in this research is set manually to analyze the effect of the percentage of the cache lock on the workloads. However, it is desirable to decipher the optimal percentage of cache lock automatically using an algorithm. This is left as a future work and needs further investigation. The probability based cache replacement algorithm could lead to erroneous results and is chance based. It is also an overhead to keep track of the number of accesses of a data block. Hence, a better cache replacement algorithm needs to be developed. This is also left as a future work.

REFERENCES

REFERENCES

[1] O. Tickoo, R. Iyer, Ramesh Illikkal, Don Newell, "Modeling virtual machine performance: Challenges and Approaches", ACM SIGMETRICS Performance Evaluation Review, 2009, pp. 55 – 60.

[2] W.C. Ku, S.H. Chou, J.C. Chu, C.L. Liu, T.F. Chen, J.I. Guo, and J.S. Wang, "VisoMT: A Collaborative Multi-threading Multicore Processor for Multimedia Applications with a Fast Data Switching Mechanism," IEEE Transactions on Circuits and Systems for Video Technology, vol. 19, no. 11, pp. 1633-1645, 2009.

[3] B. Chapman, G. Jost, R.V.D. Pas, "Using OpenMP – Portable Shared Memory Parallel Programming," The MIT Press, 2008.

[4] A. Asaduzzaman, I. Mahgoub, "Cache Modeling and Optimization for Portable Devices Running MPEG-4 Video Decoder," in MTAP Journal, 2006.

[5] A. Asaduzzaman, F.N. Sibai, M. Rani, "Improving Cache Locking Performance of Modern Embedded Systems via the Addition of a Miss Table at the L2 Cache Level," in JSA Journal, 2010. Stephen T. Jones, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, "Geiger: Monitoring the buffer cache in a virtual machine environment," ACM Sigmetrics, 2006, pp. 14- 24.

[6] Balbir Singh, "Page/ Slab cache control in a virtualized environment," Proceedings of the Linux Symposium, 2010.

[7] H. Chen, X. Wang, Z. Wang, X. Wen, X. Jin, Y. Luo, X. Li, "REMOCA: Hypervisor Remote Disk Cache, " IEEE International Symposium on Parallel and Distributed Processing with Applications," 2009, pp. 161-169.

[8] Pin Lu, Kai Shen, "Virtual Machine Memory Access Tracing with Hypervisor Exclusive Cache," Proceedings of the USENIX Annual Technical Conference, 2007.

[9] Xavier Vera, Bjom Lisper, Jingling Xue, "Data Cache Locking for Higher Cache Predictability," Proceedings of the 2003 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems, 2003.

[10] VivySuhendra, TulikaMitra, "Exploring Locking and Partitioning for Predictable Shared Caches on Multi Cores," Proceedings of the 45th Annual Design Automation Conference, 2008.

[11] Abu Asaduzzaman, FadiSibai, Manira Rani, "Improving the Cache Locking Performance of Modern Embedded Systems via the Addition of a Miss Table at the L2 Cache Level," Journal of Systems Architecture: the EUROMICRO journal, 2010.

[12] AbhikSarkar, Frank Mueller, HariniRamaprasad, "Static Task Partitioning for Locked Caches in Multi Core Real Time Systems".

[13] M.Ju, H. Che and Z.Wang, " Performance Analysis of Caching Effect on Packet Processing in a Multi-Threaded Processor,"*In Proc.* of the 2009, WRI International Conference on Communications and Mobile Computing, 2009.

[14] J. Irwin, M.D. May, H.L. Muller, D. Page, "Predictable Instruction Caching for Media Processors,"*In Proc.*of the IEEE International Conference on Application-Specific Systems, Architectures and Processors, 2002.

[15] R.H. Patterson, G.A. Gibson, M. Satyanarayanan, "Using Transparent Informed Prefetching (TIP) to Reduce File Read Latency," Goddard Conference on Mass Storage Systems and Technologies, 1992, pp. 329 – 342.

[16] V.Suryanarayana, K.M.Balasubramanya, R. Pendse, "Cache Isolation and Thin Provisioning of Hypervisor Caches,"Accepted for publication in the 37th IEEE LCN Conference, 2012.

[17] 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,"In Proc. 19th ACM Symposium on Operating System Principles, 2003, pp. 164-177.

[18] C. A. Waldspurger., "Memory resource management in vmwareesx server," InProc. 5th USENIX Symp.on Operating Systems Design and Implementation, 2002, pp. 181–194.

[19] AvinashSrivastav, "Pinning Objects to Improve Apps Performance, " https://blogs.oracle.com/stevenChan/entry/pinning_objects_to_improve_app, 2007, last accessed on 09/15/2012.

[20] SNIA Tutorials, https://www.snia.org, last accessed 08/31/2012.

[21] V. Suryanarayana, A. Jasti, R. Pendse, "Credit Scheduling and Prefetching in Hypervisor Caches using Hidden Markov Models," 35th IEEE International Conference on Local Computer Networks, 2010, pp. 224-227.

[22] H. Kim, H. Jo, J. Lee, "XHive: Efficient Cooperative Caching for Virtual Machines," IEEE Transactions on Computers, Vol 60, January 2011, pp. 106-119.

[23] M.R.Aliabadi, M.R.Ahmadi, "Proposing a Comprehensive Storage Virtualization Architecture with Related Verfication for Data Center Application," In Proc of Advanced Information Sciences and Service Sciences, Volume 2, September 2010, pp. 68-75.

[24] P. Thierry, L. George, J. Hermant, F. Germain, D. Ragot, J. Lacroix, "Toward a Predictable and Secure Data Cache Algorithm: A Cross-Layer Approach,"10th International Symposium on Programming and Systems, April 2011, pp. 148-155.

[25] Valgrind Tool Suite, http://valgrind.org/info/tools.html, last accessed on 07/24/2012.

[26] J. Reineke, D. Grund, C. Berg, R. Wilhelm, "Timing Predictability of Cache Replacement Policies," In Proc. Of Automatic Verification and Analysis of Complex Systems, September 2006, pp. 99-122.

[27] S. Cho, L. Jin, K. Lee, " Achieving Predictable Performance with On-Chip Shared L2 Caches for Manycore-Based Real-Time Systems," In Proc of the 13th IEEE International Conference on Embedded and Real Time Computing Systems and Applications, 2007, pp. 3-11.

[28] J. Liedtke, H. Hartig, M. Hohmuth, "OS- Controlled Cache Predictability for Real-Time Systems," In Proc. Of 3rd IEEE Real-Time Technology and Applications Symposium, 1997, pp. 213.

[29] A. Landau, M. Yehuda, A. Gordon, "SplitX: Split Guest/Hypervisor Execution on Multi-Core," In Proc. Of 3rd Conference on I/O Virtualization, 2011, pp. 1.

[30] Z. Yang, H. Fang, Y. Wu, C. Li, B. Zhao, H. Huang, "Understanding the Effects of Hypervisor I/O Scheduling for Virtual Machine Performance Interference," 4th Intl Conference on Cloud Computing Technology and Science, 2012, pp.34 – 41.

[31] K. Ye, X. Jiang, D. Ye, D. Huang, "Two Optimization Mechanisms to Improve the Isolation Property of Server Consolidation in Virtualized Multi-Core Server,"In Proc. Of the 2010 IEEE 12th Intl. Conference on High Performance Computing and Communications, 2010, pp. 281 – 288.

[32] T. Luo, S. Ma, R. Lee, X. Zhang, D. Liu, L. Zhou, "S-CAVE: Effective SSD Caching to Improve Virtual Machine Storage Performance," In Proc. Of the 22nd Intl. Conference on Parallel Architectures and Compilation Techniques, 2013, pp.103 – 112.

[33] J. Feng, J. Schindler, "A De-duplication Study for Host Side Caches in Virtualized Data Center Environments," IEEE 29th Symposium on Mass Storage Systems and Technologies, 2010.

[34] V. Kazempour, A. Kamali, A. Fedorova, "AASH: Asymmetry Aware Scheduler for Hypervisors," In Proc. Of the 6th ACM SIGPLAN/SIGOPS Intl. Conference on Virtual Execution Environments, pp. 85 – 96.

[35] J. Ahn, C. Kim, J. Han, Y. Choi, J. Huh, "Dynamic Virtual Machine Scheduling in Clouds for Architectural Shared Resources," In Proc. Of the 4th USENIX Conference on Hot Topics in Cloud Computing, 2012, pp. 19-19.

[36] L. Ye, C. Gniady, "Improving Energy Efficiency of Buffer Cache in Virtual Machines," Intl. Green Computing Conference, 2012, pp. 1-10.

[37] P. De, M. Gupta, M. Soni, A, Thatte, "Caching VM Instances for Fast VM Provisioning: A Comparitive Evaluation," In Proc. Of the 18th Intl. Conference on Parallel Processing, 2012, pp. 325-336.

[38] VMware Whitepaper, “Understanding Full Virtualization, Paravirtualization and Hardware Assist,” 2007.

[39] P. Goyal, D. Jadav, D. Modha, and R. Tewari, “CacheCOW: QoS for Storage System Caches,” InProc. 11th Intl Conf. on Quality of Service, 2003, pp. 498-515.

[40] 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,”In Proc. 19th ACM Symposium on Operating System Principles, 2003, pp. 164-177.

[41] R. Prabhakar, S. Srikantaiah, C. Patrick and M. Kandemir, “Dynamic Storage Cache Allocation in Multi-Server Architectures,”In Proc. Conf on High Perf. Computing Networking, Storage and Analysis, 2009.

[42] O. Tickoo, R. Iyer, R. Illikkal and D. Newell, “Modeling virtual machine performance: Challenges and Approaches,” ACM SIGMETRICS Performance Evaluation Review, 2009, 55-60.

[43] S. T. Jones, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, “Geiger : Monitoring the buffer cache in a virtual machine environment,”In Proc. 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, 2006, pp. 14-24.

[44] G. W. Juette and L. E. Zeffanella, “Radio noise currents n short sections on bundle conductors (Presented Conference Paper style),” presented at the IEEE Summer power Meeting, Dallas, TX, Jun. 22–27, 1990, Paper 90 SM 690-0 PWRS.

[45] H. Chen, X. Wang, Z. Wang, X. Wen, X. Jin, Y. Luo, X. Li, “ REMOCA: Hypervisor Remote Disk Cache ,“ IEEE International Symposium on Parallel and Distributed Processing with Applications, 2009, pp. 161-169.

[46] R. Goldberg. Survey of Virtual Machine Research. IEEE Computer, 1974, pp. 34–45.

[47] Intel Virtualization Technology Specification,ftp://download.intel.com/ technology/computing/vptech/C97063.pdf, 2005.

[48] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI ’02), Boston,Massachusetts, December 2002.

[49] T. M. Wong and J. Wilkes. My Cache or Yours? Making Storage More Exclusive. In Proceedings of the USENIX Annual Technical Conference (USENIX ’02), Monterey, California, June 2002.

[50] D. Muntz and P. Honeyman, “Multi-Level Caching in Distributed File Systems - or - Your Cache Ain’tNuthin’ But Trash,” In Proc of the USENIX Winter Conference, January 1992, pp.305–313.

[51] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP ’95), December 1995, pp. 79–95.

[52] A. Gulati, G. Shanmuganathan, X. Zhang, and P. Varman, “Demand Based Hierarchical QoS Using Storage Resource Pools,” in USENIX ATC, 2012.

[53] A. Gulati, G. Shanmuganathan, X. Zhang, and P. Varman, “Demand Based Hierarchical QoS Using Storage Resource Pools,” in USENIX ATC, 2012.

[54] M. K. Qureshi and Y. N. Patt, “Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches,” in MICRO, 2006, pp. 423–432.

[55] R. Lee, X. Ding, F. Chen, Q. Lu, and X. Zhang, “MCC-DB: Minimizing Cache Conflicts in Multi-core Processors for Databases,” PVLDB, vol. 2, no. 1, 2009, pp. 373–384

[56] D. Chisnall, The Definitive Guide to the Xen Hypervisor, Upper Saddle River, NJ, USA: Prentice Hall PTR, 2007.

[57] L. Ye, G. Lu, S. Kumar, C. Gniady, and J. H. Hartman, “Energy-efficient storage in virtual machine environments,” in VEE, 2010.

[58] R.F.Cmelik and D. Keppel, Shade: A Fast Instruction Set Simulator for Execution Profiling, Sun Microsystems Inc., Technical Report SMLI TR-93-12, 1993.

[59] V. Chadha and R. J. Figueiredo. ROW-FS: a user-level virtualized redirect-on-write distributed system for wide area applications. In Proceedings of the 14th international conference on High performance computing, HiPC '07, 2007, pp. 21-34.

[60] S. Osman, D.Subhraveti, G. Su, and J. Nieh,“The design and implementation of Zap: a system for migrating computing environments,” SIGOPS OS Review,2002,pp.361-376.

[61] C. Tang,”FVD: a high-performance virtual machine image format for cloud,” In Proc of the 2011 USENIX conference on USENIX annual technical conference, USENIXATC '11, 2011, pages 18-18.

[62] W. Zhao and Z. Wang, "Dynamic Memory Balancing for Virtual Machines," In Proc. Int'l Conf. Virtual Execution Environments (VEE), 2009.

[63] D.T. Meyer, G. Aggarwal, B. Cully, G. Lefebvre, M.J. Feeley, N.C. Hutchinson, and A. Warfield, "Parallax: Virtual Disks for Virtual Machines," In Proc. European Conf. Computer Systems (Eurosys), 2008.

[64] J. Sugerman, G. Venkitachalam, and B.-H. Lim, "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor," In Proc. USENIX Ann. Technical Conf., 2001, pp. 1-14.

[65] S. Govindan, J. Choi, A.R. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam, "Xen and Co.: Communication-Aware cpu Management in Consolidated Xen-Based Hosting Platforms," IEEE Trans. Computers, vol. 58, no. 8,Aug. 2009, pp. 1111-1125.

[66] H.V. Caprita, and M. Popa, "Design methods of multi-threaded architectures for multicore microcontrollers," 2011 6th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI-2011), pp. 427-432, 2011.

[67] W.C. Ku, S.H. Chou, J.C. Chu, C.L. Liu, T.F. Chen, J.I. Guo, and J.S. Wang, "VisoMT: A Collaborative Multi-threading Multicore Processor for Multimedia Applications with a Fast Data Switching Mechanism

[68] I. Puaut, "Cache Analysis Vs Static Cache Locking for Schedulability Analysis in Multitasking Real-Time Systems," http://www.cs.york.ac.uk/rts/wcet2002/papers/puaut.pdf. 2006.

[69] E. Tamura, J.V. Busquets-Mataix, J.J.S. Martin, A.M. Campoy, "A Comparison of Three Genetic Algorithms for Locking-Cache Contents Selection in Real-Time Systems," in the Proceedings of the Int'l Conference in Coimbra, Portugal. 2005.

[70] W. Tang, R. Gupta, and A. Nicolau, "Power Savings in Embedded Processors through Decode Filter Cache" in Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE'02), pp.1-6, 2002.

[71] Heptane, "Heptane – a WCET analysis tool," http://www.irisa.fr/alf/index.php?option=com_content&view=article&id=29. 2013.

[72] VisualSim, "VisualSim – a system-level simulator," http://www.mirabilisdesign.com/. 2013.

APPENDIX

APPENDIX

KEY TERMINOLOGIES

Storage Area Network:A network of servers and storage where the storage is arranged at the back end as an array and connected to the servers in the front end.

Virtualization: The method of abstracting an underlying entity.

Server Virtualization: A method of virtualization wherein the server OS is abstracted and installed as many virtual servers on a physical server.

Cache: A piece of hardware/software that is closer to the applications that the memory and stores the frequently accessed data.

Instruction Cache: A cache that stores frequently accessed instructions.

Data Cache: A cache that stores the frequently accessed data blocks.

Cache Hits: The process of finding the data required by the applications in the cache.

Cache Misses: The process of not being able to find the required data by applications in the cache.

Hypervisor: The software that sits on top of the server OS and makes server virtualization possible.

Dynamic Caching: The technique of allocating cache dynamically and on-the-go.

Thin Provisioning: The technique of allocating less cache than needed and later allocate cache on a need basis.

I/O Response Time: The time needed to fetch the required data by applications from the cache/memory/hard disk.

APPENDIX (continued)

Cache Locking: The technique of locking certain cache lines, so that, those lines cannot be replaced.

Associativity: The technique that decides how the data blocks are mapped from the memory to the cache.