# Hypervisor Cache Provisioning with Cognizance of Application Behaviour in Derivatve Cloud

**M. Tech Project Report**

Submitted in partial fulfillment of the requirements
for the degree of

**Master of Technology**

by

**Kanika Pant**
Roll No: 153050042

under the guidance of

**Prof. Purushottam Kulkarni**


Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai

Approval sheet

This dissertation entitled **Hypervisor Cache Provisioning with Cognizance of Application Behaviour in Derivatve Cloud** by **Kanika Pant** is approved for the degree of Master of Technology in Computer Science and Engineering from IIT Bombay.

**Examiners**

Mythili Vutukuru V. Mythili

Varsha Apte

**Supervisor**

Purushottam Kulkarni

**Chairman**

Date: _____

Place: _____

Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Kanika

(Signature)

Kanika Pant

(Name of the student)

153050042

(Roll No.)

Date: 27/June/2017

# Acknowledgement

I would like to thank my guide Prof. Purushottam Kulkarni for his support and motivation throughout my project. I would also take this opportunity to thank Debadatta Mishra for his valuable inputs which made it possible for me to proceed in my project. Lastly I would like to thank my colleagues Prashanth, Shyamli, Bhavesh and Pijush for their constant motivation all along towards finishing this project.

# Abstract

Derivative Cloud, a new emerging platform with light weight containers provisioned inside virtual machines can be used for providing IaaS and PaaS services. It is quite challenging to make hypervisor and isolation framework for containers to go hand in hand. This is quite important while resource provisioning for containers since hypervisor is agnostic to nesting and guest has no defined method for providing hypervisor with resource provisioning information for the containers. Having differentiated resource provisioning for containers will help manage resources and satisfy higher level application objectives.

In this work we are going to define challenges with the differentiated provisioning of hypervisor managed second chance cache. Hypervisor caching provides a way for memory management for improving resource utilization and application performance. Differentiated resource provisioning would provide basic framework which can enable developing memory and cache management policies depending on the application behaviour. Empirically knowing the application behaviour and using it for making provision decisions can help managing cache efficiently and satisfy application needs. This empirical study provided with different application characteristics. We have application's cache characteristics and system resource utilization provisioned with different amount of cache provisioned for them. Static provisioning of cache among the containers which failed due to failed hypothesis, not taking into account disk I/O contention.

# Contents

# List of Tables

# List of Figures

# 1. Introduction

Server consolidation is the most important use case of virtualisation[9]. It enables to fit in multiple virtual machines inside a physical server such that all the resources get efficiently utilised. Without this setup all the resources of physical server were underutilized. This efficient utilisation comes with its own challenges. Since we are over-provisioning and sharing resources among multiple virtual machine, some management is required.

Among all the resources the vital enabler for high consolidation is memory overcommitment. There is always a tradeoff between efficiently using the resources and dynamically handling over-provisioning and performance implications. Whatever techniques we use for handling overcommitment should exploit the above tradeoff.

There is a lot of challenge in memory management since there is a opaqueness between resource manager which is hypervisor and guest operating system which uses the resources. They work independently and manage resources individually. Many techniques are being used for memory management. The memory management techniques being mentioned have different use-cases. We have dynamic ballooning[21] which is to bridge the opaqueness gap such that right decisions are made. Then we have memory content deduplication[21][16][8][10] for efficiently using memory. Lastly we have hypervisor caching[13] which is both for efficient memory utilization and reducing disk access latency.

The techniques mentioned above are in context single level virtualized setup. But since there is always a possibility for enhancement and improvement, cloud computing has also taken a step forward and introduced another setup which has its own use case. This new setup is a nested setup called as - derivative cloud. In this nested setup, there is an intermediate service provider provides his/her own service on the top of the service he/she has rented. The services provided here are managed at two levels, one at the hypervisor level and another from within the virtual machines. In this nested setup we can have both container based as well as virtual machine derivative cloud. Resource management which earlier talked about is inadequate in the nested setup where we have multi level control.

## 1.1 Derivative Cloud & Resource Management

Derivative cloud present a situation where we have nested virtualization[11]. In the current era derivative clouds use containers inside the VMs because as we all know containers are lightweight and overheads are significantly low in using them for nested virtualization. Resource management in case of container based derivative clouds requires container based services to be provisioned and managed from within the VMs and VM level services to be provisioned and managed by hypervisor. If the resource management techniques which we are using for single level virtualization are used here, they will be inadequate for derivative clouds because hypervisor is agnostic to nesting present and guest has no way to pass that

information to the hypervisor so that it can provision resources according to the application need.

## 1.2 Hypervisor caching for single level virtualized setup

Hypervisor caching helps in disk cache memory(page cache) management in virtualized setup. The name comes from the controlling entity which is the hypervisor. Hypervisor cache can be configured in different ways. One way is to collect the free system memory and have it controlled by hypervisor as a second chance cache. Another way is to use techniques like ballooning which manages system wide memory allocations and offloading the disk caching to the hypervisor fully or partially. This cache can be configured as differentiated cache among the virtual machines by the hypervisor for efficient memory management such as compression deduplication etc. Now the gap here is single level management. The hypervisor is agnostic about the applications running inside the VMs. The derivative cloud setup introduces the management gap between the hypervisor and applications running inside the VMs.

## 1.3 Differentiated Cache Provisioning Framework in derivative cloud setup

Hypervisor caching for virtualized setups can be extended to fit to a situation of derivative clouds. But in derivative cloud scenario we will have differentiated cache provisioning at two levels. One as per-VM cache provisioning and this per-VM cache should be provisioned on a per container basis. In this setup there will be cache provisioning policies at two levels. These policies at each level are mutually exclusive but the service hosting entities i.e. hypervisor and virtual machines will be influencing cache provisioning decisions. The differentiated cache provisioning framework provides with the following things-

- Differentiated partitioning of cache among the containers running applications as well as VM level cache provisioning.
- Providing with configurable cache storage options for execution entities running inside VMs.
- Supporting two level of disk cache management which can adapt to changing demands in derivative cloud setup.

This framework enables managing cache specification at two levels dynamically. Policies for cache management which depends on application behaviour is still left out of this work.

## 1.4 Contribution

- Application Characterization for differentiated cache in a derivative cloud setup.
- Cache provisioning for applications using the application characteristics.

## 1.5 Problem Statement

Having a differentiated cache provisioning framework, characterizing different applications such that we can come up with a procedure such that we can provision hypervisor cache for different applications to satisfy their higher level objectives.

# 2. Background

This chapter includes the necessary information required to understand the purpose of the project in a better way. This includes very fundamental things which need attention for differentiated cache provisioning. The sections following this chapter gives the brief idea about the following things:

- Second Chance Cache:This part basics about second chance cache. Key points about how it functions and what are its benefits. Different implemented and usable forms of the second chance cache.
- Different Cache Configuration:Provided with a second chance cache how differently can it be configured and used. There will be discussion about usefulness of different configuration.
- Last section includes some basic information about containers. Resource management in containers.

## 2.1 Second chance Cache

Second chance cache has been setup with exclusivity in its nature. It has been integrated with the page cache such that it provides a second chance for the pages getting evicted from the page cache. This is to ensure that if recently evicted page is again required by the applications then it may save a disk read, thus reducing I/O latency and improving the performance of an application. Exclusivity is ensured between the page cache and second chance cache as when page is evicted from the page cache it is being stored in the second chance and cache and page requested from the second chance

cache is removed from there and placed into the page cache.

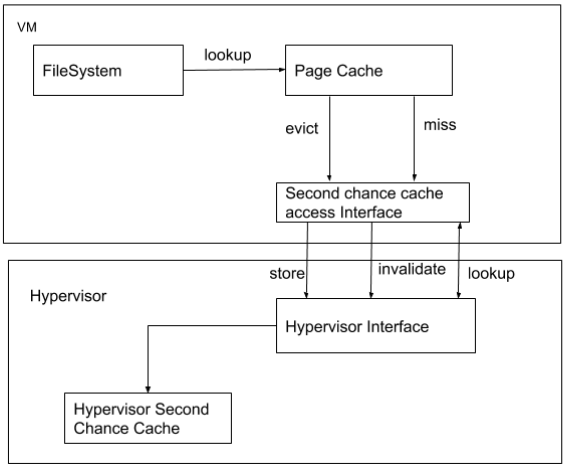


Figure 2.1: Second chance cache in action with system page cache in an exclusive manner

### 2.1.1 Transcedent Memory

One such implementation of second chance cache is Transcedent Memory[13][15][13]. It exploits the second chance cache interface being provided in the linux kernel. The interface being provided by the kernel deals with only clean pages. There are two cache interfaces one with the guest and the other with the host. The interface present with the guest is called tmem cleancache[3] frontend and the one with host is called tmem backend. The two interfaces communicate with the help of hypercall. The basic key operations performed by the interface are:-retrieve(get) a disk block from the cache, store a block into the cache store and invalidating objects from the cache. This system is indexed based on file-system mount points, file information in the form of inode number and offset(block number). This cache is partitioned based on the VMs and the partitions are called pools.

### 2.1.2 Differentiated Hypervisor Cache in Derivative Clouds

As we have already discussed in the previous sections that derivative cloud setup is a nested setup and in this work we are mainly focused on application containers running inside VMs. For this nested setup there are two resource managers one the hypervisor and other would be the VM administrator. Since we are providing service on the top of a service there will be multi-level policies. Therefore the second chance cache we are discussing about should also be multilayered. This multilayered should be differentiated among the VMs at the first layer and among the containers within the VMs at the second layer. We have this kind of framework where we can have differentiated cache partitioning at both the levels. Design is same as single level cache management framework. Here also we have two interfaces one at VM level which is called as the second chance cache interface and other at the hypervisor level. Communication happens with the help of hypercall. We have two types of cache backing stores one

is SSD and the other is memory. Cache can be provisioned at two levels depending on some policy as there is a facility to give different amount of share for VMs and containers. There are two policy controllers one at the hypervisor level and other at VM level. The cache allocations are in the terms of weights in percentage form. There is also a facility to specify cache store type with the help of policy controller at VM level. The figure below will explain this setup more appropriately.

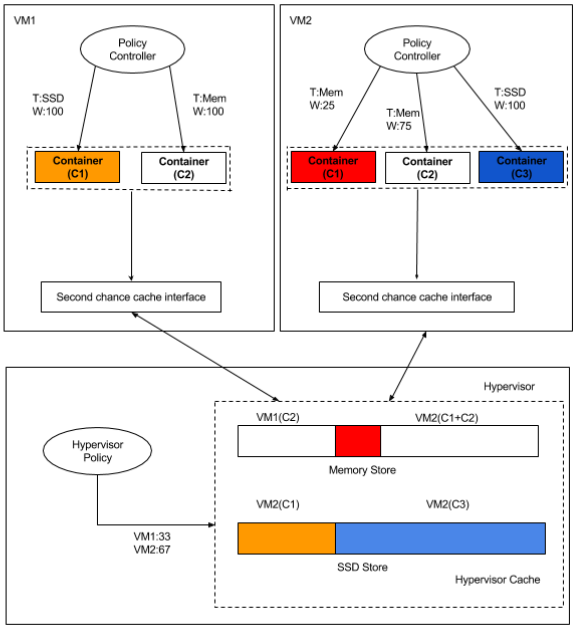


Figure 2.2: Second chance cache in action with system page cache in an exclusive manner

The diagram above explains the framework with two VMs and each containing 2 and 3 containers respectively. Hypervisor policy controller assigns weight to the VMs and partition the cache in two with share of 33 to VM1 and 67 share to VM2. The policy controller at VM level will provision the cache among the containers from the share provided to the VM. In the first VM C1 uses a 100 share on SSD store and C2 takes a 100 share in memory. While in case of VM2 C1 and C2 takes a 25:75 share in memory and C3 takes a 100 share in SSD store.

For having this policy controller extension to cgroup resource control framework is required. With cgroup resource control framework we can specify allocation limits for different resources. Since for our framework it is required to additionally specify the cache type and weight for the cache allocation therefore the extension should be able handle all this. The interface at the guest is not designed to work well with cgroups therefore the second chance access interface also needs to be modified such that it integrates well with the updated cgroup resource control framework.

The problem which exists with the available second chance cache interface is that presently it assigns a unique pool-id to a new file system registered to the second chance interface by providing that information to the cache store implementation. Now in this nested derivative cloud we have containers within the VMs such that each VM can have multiple containers running inside them. This allows to have a single identifier for a single filesystem which makes second chance interface inapplicable for this derivative cloud setup. These identifiers are quite useful for cache operations like lookup, store and invalidate. So both cgroup and second chance cache extensions are required and they should be integrated such that

there is always a new unique identifier for each application container being instantiated inside a VM.

The cache store to be implemented for this nested setup requires to provide with container level policies. The hypervisor level policy is enforced with the help oh hypervisor policy controller. The aplication container level policy needs to be transmitted to the hypervisor cache manager via second chance cache interface. There is a backend storage implementation for enforcing the policies at both levels. It also has a feature for dynamic reconfiguration of the policies. This feature of the backend is required for dynamic provisioning of cache with changing application requirements and removal or instantiation of VMs. The backend gives the flexibility to application containers to have their objects in either SSD cache store or memory store.

Memory pressure is also handled with the help of eviction policy at the backend. The eviction policy is simple FIFO.

## 2.2 Different cache configuration

We can have different cache configurations: unified, statically partitioned or dynamically partitioned.

### 2.2.1 Unified vs Static Partitioning

We can do the comparison with the help of an example. The example setup is such that there are 2 VMs with different type of workloads.
VM1 - workload which has less temporal locality but high IO request rates (100 blocks per sec).
VM2 - workload which has high temporal locality but IO request rate is moderate (50 blocks per sec).
Assume scheduling time is 1 sec for each VM and cache size is 100 blocks.
**Case 1** : **Unified Cache** - When VM1 is scheduled it will cache 100 blocks occupying full cache. Next VM2 is scheduled which will replace 50 blocks of data of VM1 with its own data which is fine since its data blocks have better temporal locality than that of VM1. Next when VM1 is scheduled it will replace all the blocks in cache with its own data and the blocks with lesser locality replace blocks with higher locality. Next time when VM2 is scheduled it will get miss instead of hit for those replaced blocks in cache layer. This leads to higher IO latency since it decreases hit rate.
**Case 2** : **Partitioned cache** - If cache is partitioned, for example 50 blocks to VM1 and 50 blocks to VM2 then performance will be better since VM1s blocks wont replace VM2s blocks. So we can say that cache partitioning is required for better hit rate. Also partitioning the cache gives hypervisors control to prioritize the VMs. For example if VM is of high priority then hypervisor can statically allocate more cache to it.

### 2.2.2 Static Partitioning vs Dynamic Partitioning

In the previous example we took into account only one aspect of the workload. But workload is associated with working set size. If a VM get a static cache allocation more than its working set size then there wont be any significant performance benefit as compared to cache size equal to working set size and there will be inefficient cache utilization. So if we partition the cache dynamically with changing working set size then it would give efficient cache utilization.
Let us consider similar example as above.

Case 1 : static partitioning with VM1 - 50 blocks and VM2 - 50 blocks
Case 2 : dynamic partitioning which is based on temporal locality so VM1 - 5 blocks and VM2 - 95 blocks.
In case 1, VM1 wont be utilizing the whole cache since its working set size is only 5 blocks so no point in giving 50 blocks to it. So there is cache wastage of 45 blocks.
In case 2, VM1 gets sufficient amount of cache while VM2 can utilize the cache better since it has better locality of reference.

## 2.3 Container Basics

### 2.3.1 Containers

Containers act as an alternative to virtual machines. Virtual machine comparison is given to the containers because they provide OS based virtualization. This means from a single OS kernel, resources can be shared among multiple separate execution environments. Apart from resources, binaries and libraries may also be shared but if required each individual container can have them exclusively for their use. Because of the shared kernel and other resources the size of the software stack gets reduced. That is why we call containers as light-weight virtual machines. Thus these light-weight virtual machines require less processing, memory and storage resources and thousands of containers can be hosted on the single physical system. Another advantage would be rapid start-up and shut-down capabilities of containers. Containers can be better suited than virtual machines in various scenarios of:

- Low latency and lossless networking usage.
- Applications with fewer security requirements
- Implementations that involve only limited groups of users

### 2.3.2 Linux Containers(lxc)

Linux container[2][12] or simply lxc is an open source project which work with Linux kernel to provide multiple containers on the same physical host. There are two features included in linux kernel which provide support for workload isolation in linux containers. Since these features are included in linux kernel therefore we call these mechanisms as core software based. Those two features are:

- Control Groups
  Cgroups[1] are collection of processes such that kernel gets a software based facility to control the subsystems such as memory, processor usage and disk I/O. Thus cgroupsgives the facility of resource control.this property can be exploited by the LXC.
- Namespace
  Linux Namespaces provides process groups their own isolated system view. Namespaces exist for different types of resources as in the case of cgroups. This allows individual containers to have their own IP addresses and interfaces. It also enables limitation to each control groups view for the physical or virtual resources such as details about file systems, processes, network interfaces, inter-process communications, host names, and user information.

# 3. Related Work

## 3.1 Hypervisor Caching

Caching is always there whether we are in native setup or virtualized setup. KVM provides with hosted virtualization and by default supports inclusive caching. Inclusive caching gives a possibility for content duplication as both VM and host can have same pages in their respective page cache. Singleton[18] provides with a way of deduplicating KSM scanned pages[8] and the inclusive cache. Thus translating the inclusive nature of cache to exclusive.

Another work named Transcedent Memory[13] which is associated with Xen which supports exclusive caching framework. This exclusive framework is guest supported as the memory for cache is collected from self ballooning feature present with Xen.

Transcedent Memory extended to KVM setup[17] enabled many caching combinations and also provided comparative analysis for proving effectiveness of hypervisor caching.

The above literature about caching is confined to single virtualized setup. There is no support for such setting in nested setup. This discussion is related to infrastructure related issues for hypervisor caching where we want a framework which supports second chance hypervisor caching.

There is other literature which deals with cache management in different ways. The first worth mention would be mortar[14] . In this work applications like memcached which support distributed key value store are modified and hypervisor cache is being exposed to it.

The second worth mentionable work is Software defined caching [20] where hypervisor cache is being talked about in multi tenant setup. Cache can be configured anywhere in the storage path. Cache provisioning policies are application SLA driven.

## 3.2 Derivative Clouds

There has been literature talking about nested virtualization[11]. Nested virtualization provides hosting VMs inside VMs. The motivation behind nested virtualization is addressing several issues which include cloud security where the provider controls both layers, facilitating homogenization of clouds by users themselves instead of VM standardisation or homogenization by cloud providers[22] and hypervisor troubleshooting. Literature exists about resource management in nested setup. Derivative cloud being the newest terminology for nested setup has a very little literature associated with it which includes spotcheck[19] which is entirely based on business idea. The nested setup contains containers inside

the VMs.

Our work enhances the existing work with cache provisioning policies using application characteristics.

# 4. Experimental Evaluation of Application Behaviour

Cloud computing is a new way of offering services. This has completely changed the way businesses and consumers operate. A variety of applications are being hosted by cloud providers for different purposes. Each application has higher level objectives which is promised by the provider depending on the resource availability. The underlying technology used by cloud providers is virtualization. High server consolidation ratios is the primary objective of virtualization from which the cloud providers will get benefited. This objective requires very efficient resource management policies so as to meet the application objectives promised by the cloud providers. Memory overcommitment is most important enabler for high consolidation ratios. Therefore memory management should be done in most efficient way. A lot of work has been done in this area. There are many techniques for managing memory in virtualized setup. To list down a few includes-dynamic ballooning,content deduplication,hypervisor caching etc. A recent new addition to cloud computing world is derivative cloud. Derivative cloud uses nested virtualization. The name suggests that something is derived and in this case it signifies services are derived from another set of services. Derivative cloud platform repackages and resells resources purchased from native IaaS platform. First level resource manager is unaware of the services being promised by the second level resource manager.

Let us talk in terms of some example where we say that suppose there is a native service provider and a derivative cloud service provider. For our case we will consider containers running inside virtual machines. Applications are running inside the containers. Each application has some higher level objective which can be in terms of performance. Derivative cloud service provider, if possible promises to fulfill that objective. This is one part of the story, let us think more about the nested setup of derivative cloud. Resource management techniques in single level virtualization setup are inadequate in this derivative cloud setup. Since hypervisor is unaware of the nested applications and there is no defined mechanism in the form of interface for explicitly providing inputs for application oriented resource provisioning.

Providing this type of framework will give the flexibility to the resource manager to explicitly have application characteristics. These characteristics will provide insight as to how application behave. Having all this knowledge would help provisioning resources among the applications.

This work deals with memory management in derivative cloud setup. The area of focus in memory management would be hypervisor caching. This is one way of efficiently managing memory. We have a framework which enables differentiated hypervisor caching for derivative clouds. There are two types of hypervisor cache stores, one is SSD backed and other is memory backed. We can configure any cache for applications. This gives the flexibility of having a choice between cache stores.

This chapter deals with experimental evaluation of application characteristics. We will be performing experiments varying different parameters and measuring different performance metrics for applications.

## 4.1 Objective

As discussed in the beginning of the chapter that hypervisor cache provisioning is a non trivial task. In derivative cloud setup satisfying application objectives require some knowledge as to how an application behave. Once the behaviour is known we can make an attempt towards provisioning cache among the applications. Therefore the objective of the experiments is to possibly provide some understanding about the application behaviour.

## 4.2 Experimental setup and requirements

Experiments should always answer some question and should be based on some hypothesis. If these things are not known then it is useless to perform experiments. After having a question and hypothesis for experiments we can come up with an objective function. Knowing the objective function we can design the experiment. The objective function requires some parameters and varying those parameters we measure some metrics of importance.

Experiments which are going to perform are related to knowing the application behaviour. We are giving the hypothesis that application behaviour changes with hypervisor caching. We want to quantify how that behaviour changes. This quantification may give some peculiarities about the application characteristics.

### 4.2.1 Parameters which will be varied during the experiments

There is only one parameter of interest for these experiments i.e. cache size. We are going to vary cache size for individual applications and measure some performance metrics.

### 4.2.2 Metrics of importance for satisfying experimental objective

The metrics which we are going to measure for these experiments consider application, cache, and the whole system resources. All of them together will give the way application behave under different conditions. Metrics for different category are:

- Application Performance
  - Throughput:Unit of throughput can be ops/sec or mb/sec
  - Latency:Unit of latency can be in ms or us
- Cache Characteristics
  - Hits in the cache
  - Cache Used
- System Resources Utilisation
  - Page cache memory used by the application
  - Anonymous memory used by the application
  - Swap memory used by the application

### 4.2.3 Workloads which are being characterized

In this section we will be discussing about the applications we are using for experiments. For experimentation we call these applications as workloads since they generate load for the setup which we are testing. These workloads try to emulate application behaviour and throttle our system for worst case conditions so that we can prove correctness of our system. These workloads are chosen keeping in mind the nature of experiments we want to perform and the framework against which we are using them. In our case clearly we want to know behaviour of the applications with a framework providing differentiated cache to the applications running inside the containers. The workloads chosen are both a mix of I/O intensive as well as memory intensive. There are two categories of the workloads which we are empirically analysing.

- Synthetic workload:We are using filebench[6], which is a synthetic workload benchmark. There are multiple workload options provided by filebench which try to behave like real world applications.
  - Webserver: From all the workloads provided by filebench we have chosen webserver for our purpose. It tries to emulate the requests which are possible with actual webserver application. It does this with dummy files, opening and reading them. Files reading index use uniform distribution.
- Real Workloads:Following is the list of real workloads used for our experiments. They will work the same way actual applications work.
  - MongoDB: MongoDB[4] is an open-source NoSQL database. It does not have table based relational database structure. It has a dynamic schema which uses JSON like documents.
  - Redis: Redis[5] is a in-memory data store for storing key value pairs. It can be used as database, cache and message broker. Its in memory usage makes it important to show that it does not require cache for its working.

For real workloads we have used YCSB(Yahoo Cloud Server Benchmark)[7] benchmark. YCSB is used to generate requests for our database servers like mongodb, redis and mysql. It acts like clients for database servers so that we can emulate a real world situation and test our system. It is a framework which provides a set of common workloads.

### 4.2.4 Experimental environment

The system specification where experiments will be performed. **Host**

- Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz
- 16 cores of CPU (with hyper threading support)
- 1 TB of hard disk space
- 32 GB RAM
- Ubuntu 14.04 LTS desktop, 64 bit
- Kernel version 4.1.5
- KVM Hypervisor

**Guest**

- 13 cores of CPU (with hyper threading support)
- 128 GB of virtual disk space
- 16GB RAM (based on experimental configuration)
- Ubuntu 14.04 LTS desktop, 64 bit
- Kernel version 4.1.5
- Container technology: lxc

## 4.3 Analysing empirical results for different applications individually

Having a system with above mentioned specification and a framework for differentiated cache provisioning we will perform experiments. Following the objective for the experiments along with parameters and measurable metrics we will have 2 types of experiments.

- Base Case: In this experiment we run different workloads individually without any differentiated caching framework. This is to have an idea how workloads perform if there is no cache in the system.
- Differentiated Cache provisioning framework: In this case we will perform a series of experiments. For each workload we will have a series of experiments. In each experiment cache size will be varied and also the cache specific metrics, application metrics and system utilization metrics will be logged. We will stop the cache variation when the performance of the application in invariant with further increase of cache size. These set of experiments will be performed for both SSD and in-memory cache for each application.

The above discussed sections provide with answers for design aspects of experimental setup. Following that design, experiments were performed with different workloads. In the next few sections we are going to discuss about the results we got from the experiments being done. We try to analyse the results which we log for each run of the experiments for different applications.

### 4.3.1 Application Performance Variation with Cache Size

This section provides with how the performance varies with variable cache sizes. Since the performance metrics which we have chosen for application are throughput and latency we will be talking about them. As the hypothesis says that the performance should improve with increase in cache size if at all the application is using the page cache. Results are being logged and shown as graphs further in this section. We are using five applications and each application is being experimented both with SSD and in memory cache for varying cache sizes. Results are shown below:

- Webserver provided with memory cache

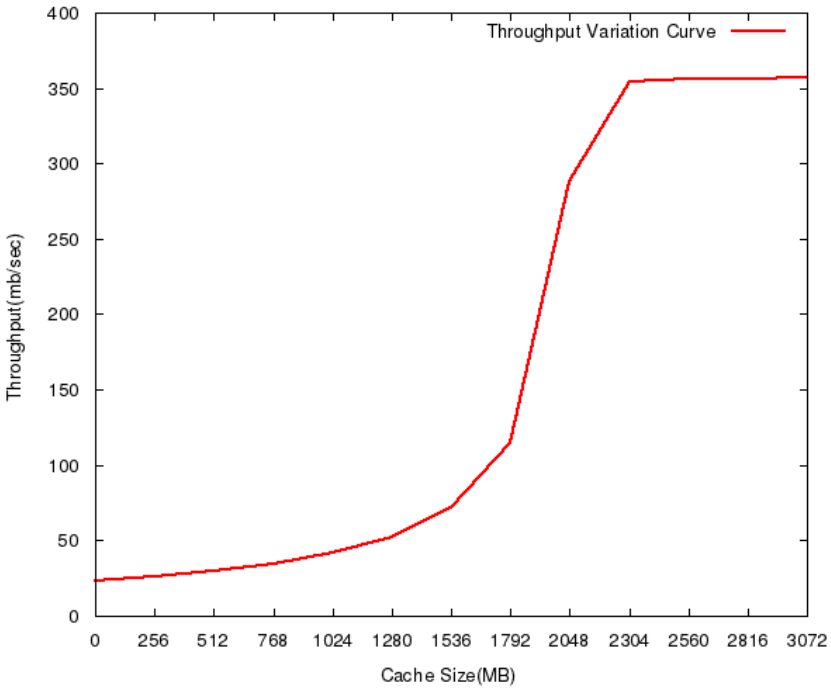


Figure 4.1: Throughput vs Memory Cache Size

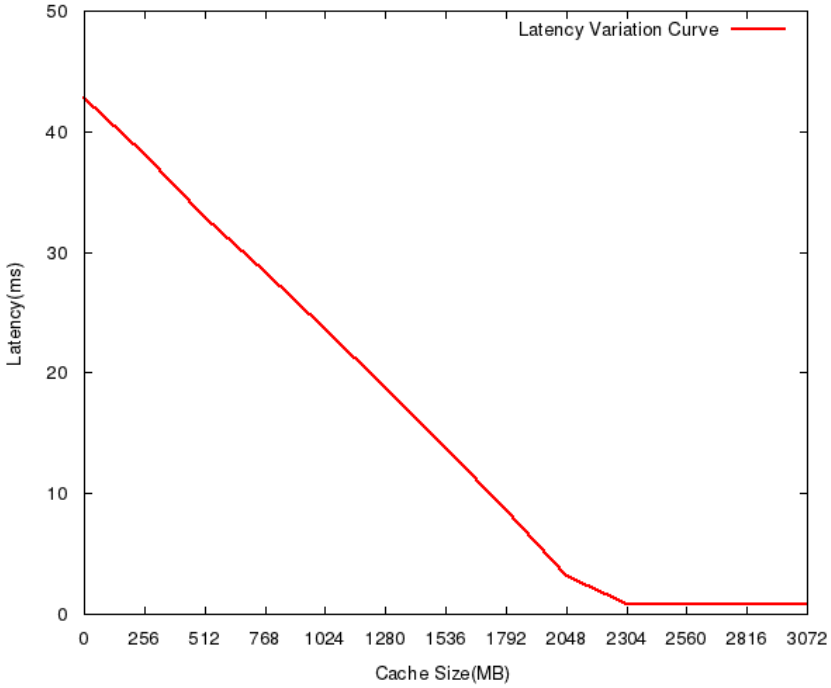


Figure 4.2: Latency vs Memory Cache Size

- Webserver provided with SSD cache

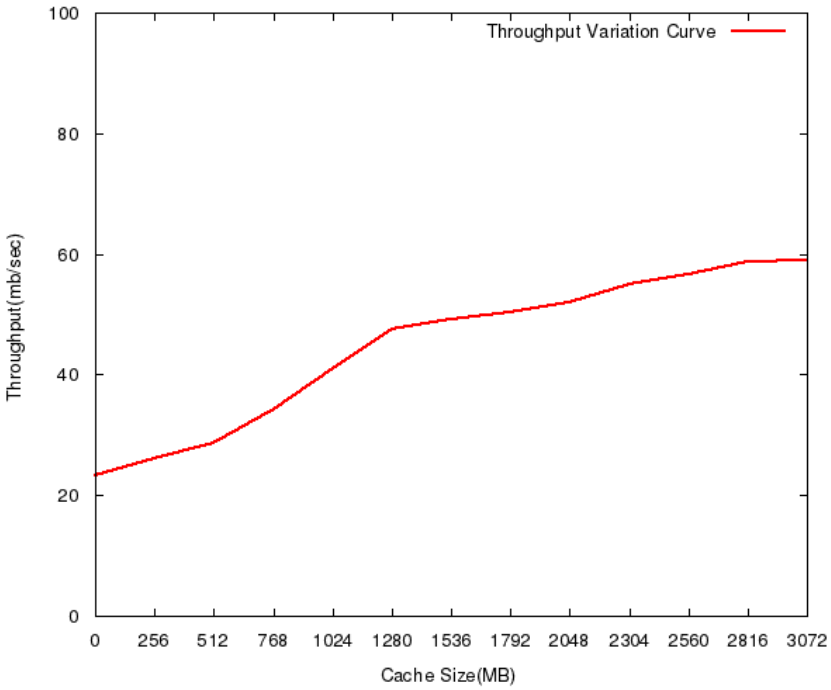


Figure 4.3: Throughput vs SSD Cache Size

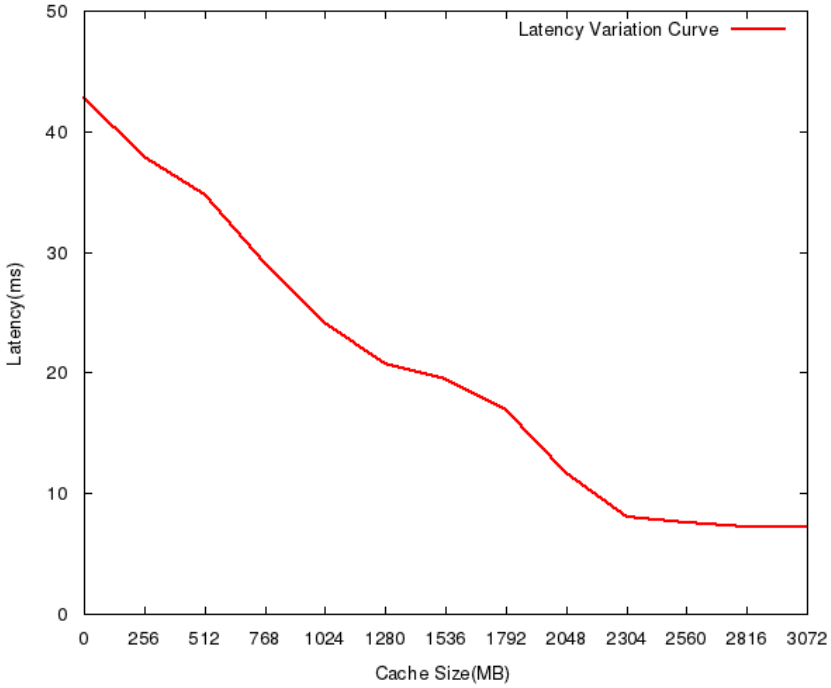


Figure 4.4: Latency vs SSD Cache Size

- MongoDB provided with memory cache

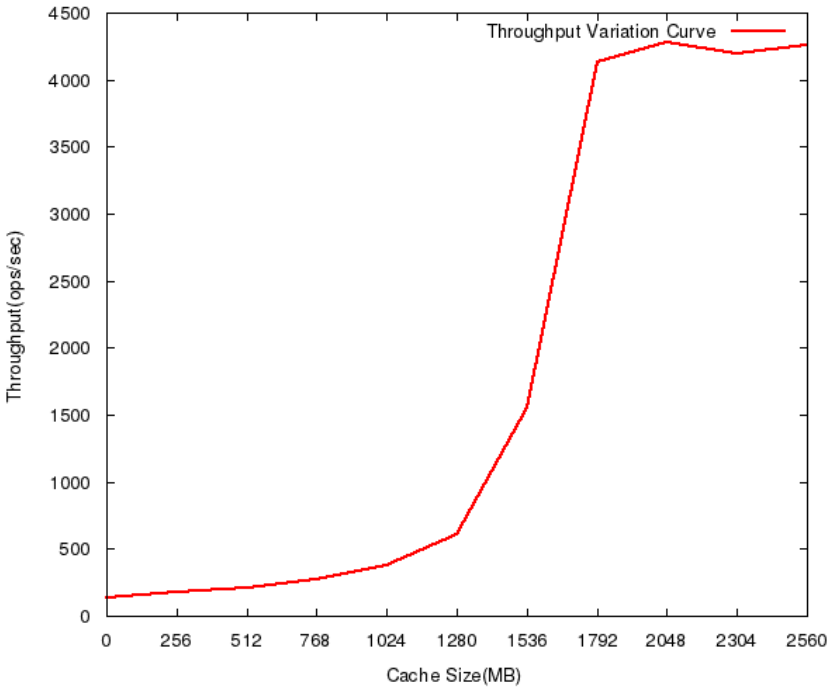


Figure 4.5: Throughput vs Memory Cache Size

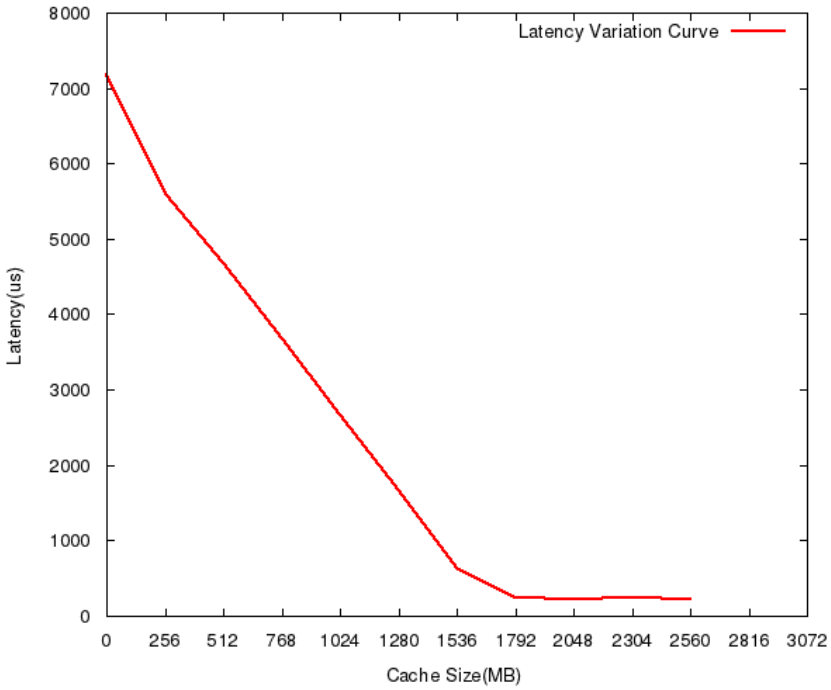


Figure 4.6: Latency vs Memory Cache Size

- MongoDB provided with SSD cache

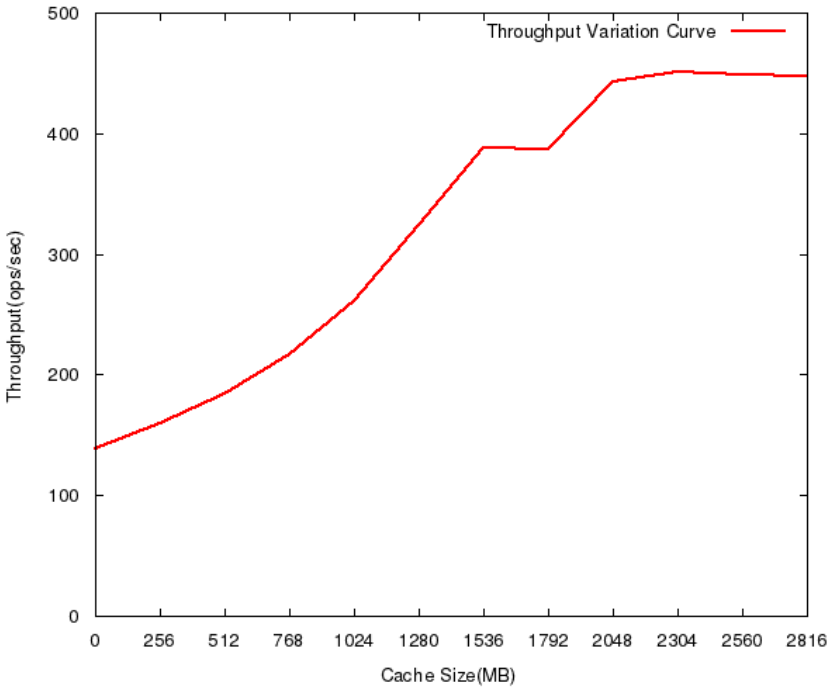


Figure 4.7: Throughput vs SSD Cache Size

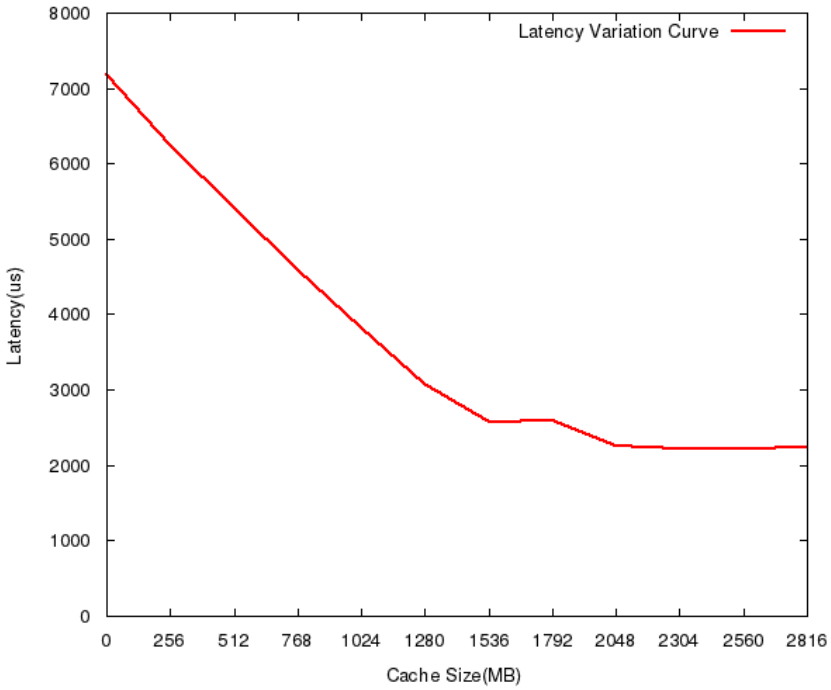


Figure 4.8: Latency vs SSD Cache Size

- Mysql provided with memory cache

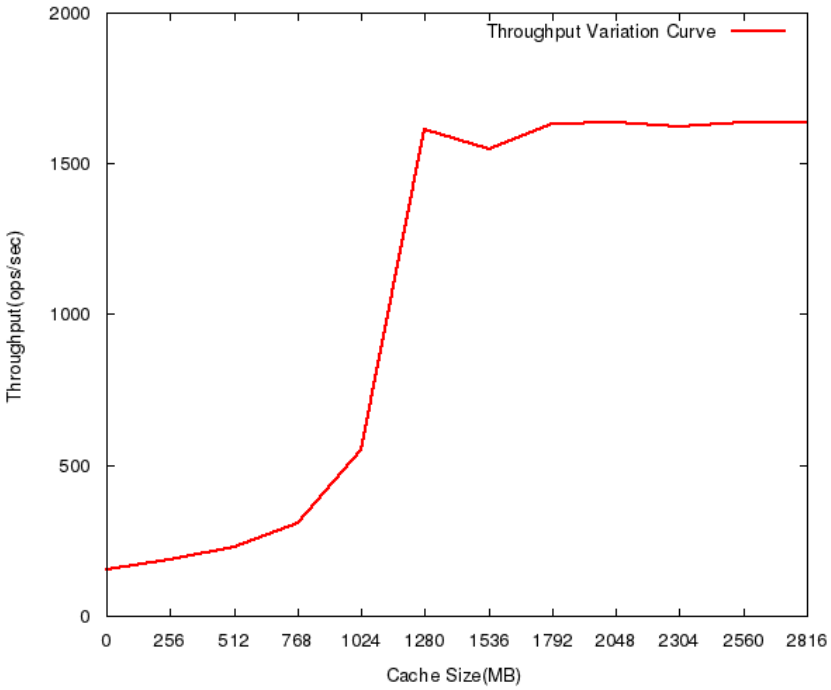


Figure 4.9: Throughput vs Memory Cache Size

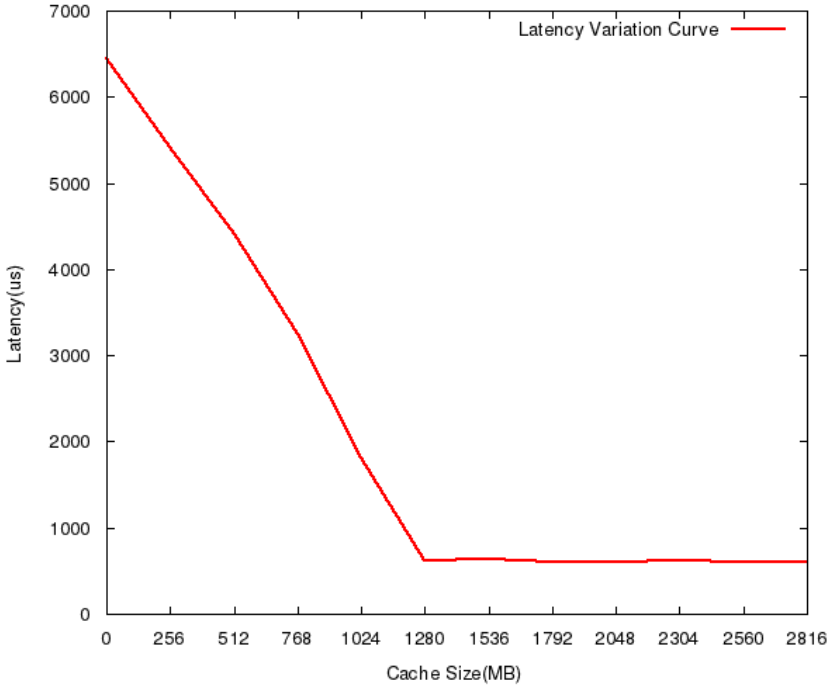


Figure 4.10: Latency vs Memory Cache Size

- Mysql provided with SSD cache

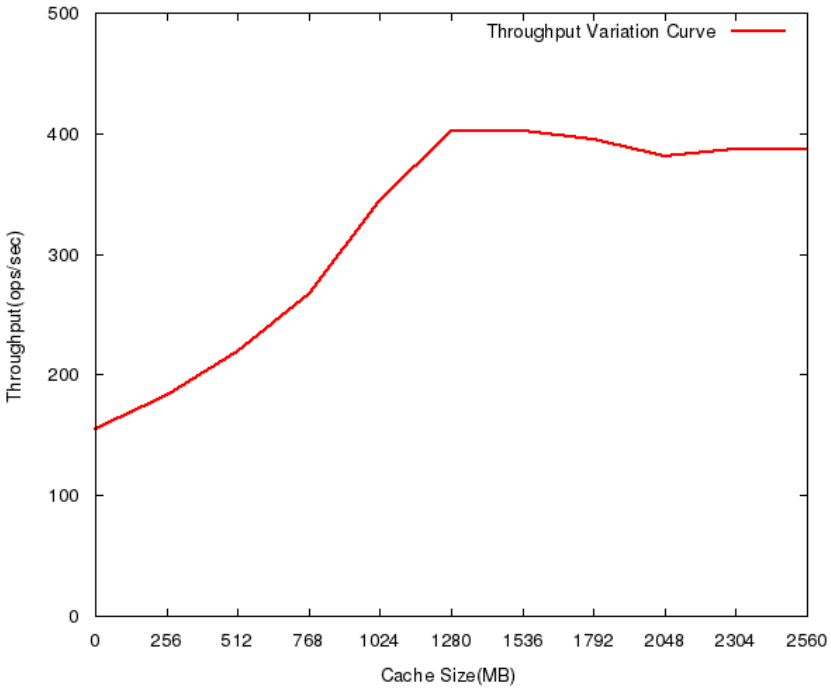


Figure 4.11: Throughput vs SSD Cache Size

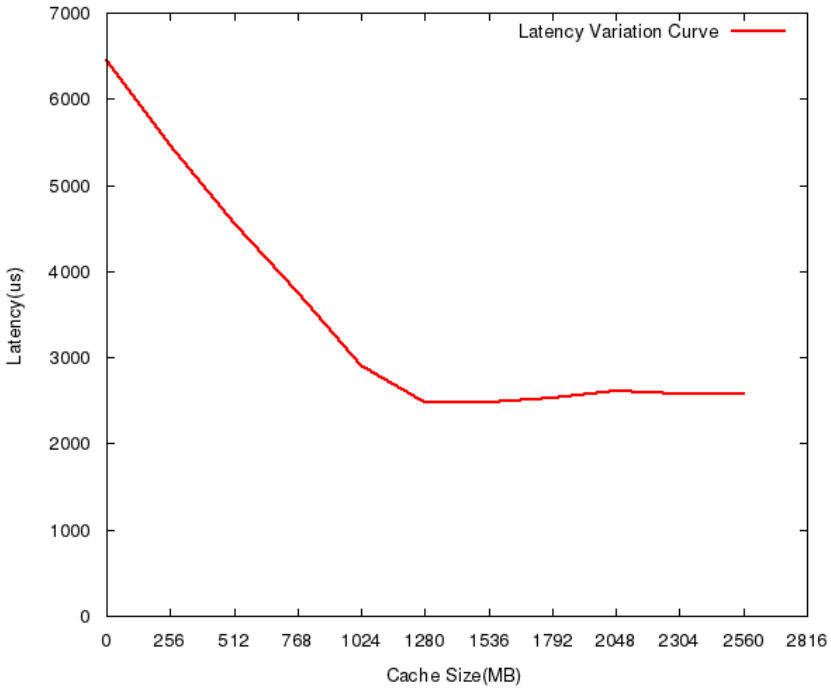


Figure 4.12: Latency vs SSD Cache Size

- Webproxy provided with memory cache

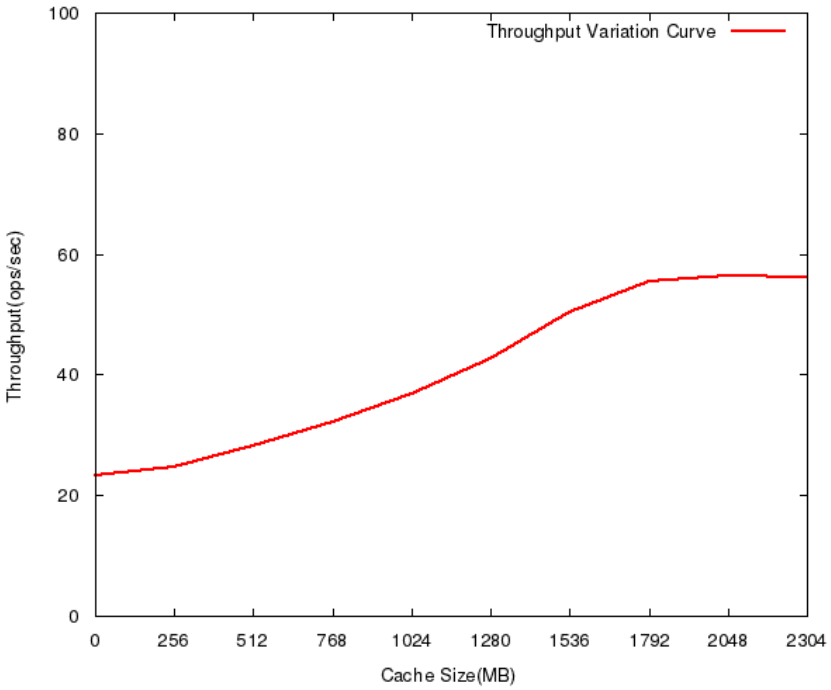


Figure 4.13: Throughput vs Memory Cache Size

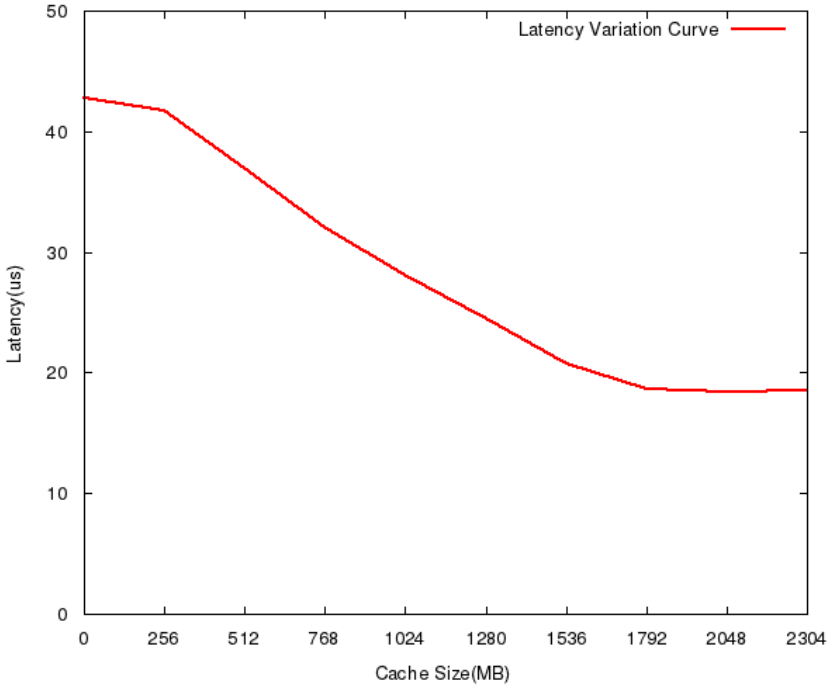


Figure 4.14: Latency vs Memory Cache Size

- Webproxy provided with SSD cache

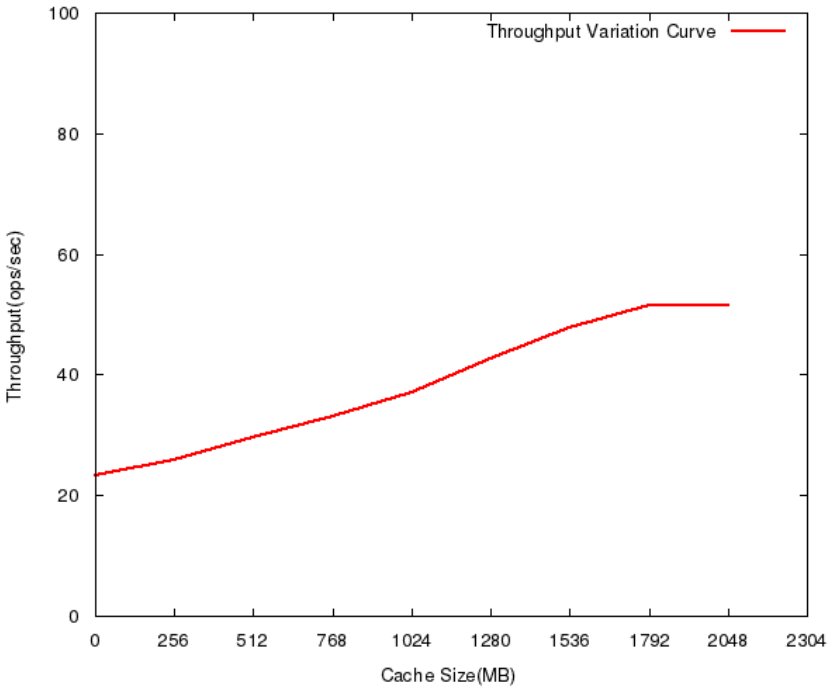


Figure 4.15: Throughput vs SSD Cache Size

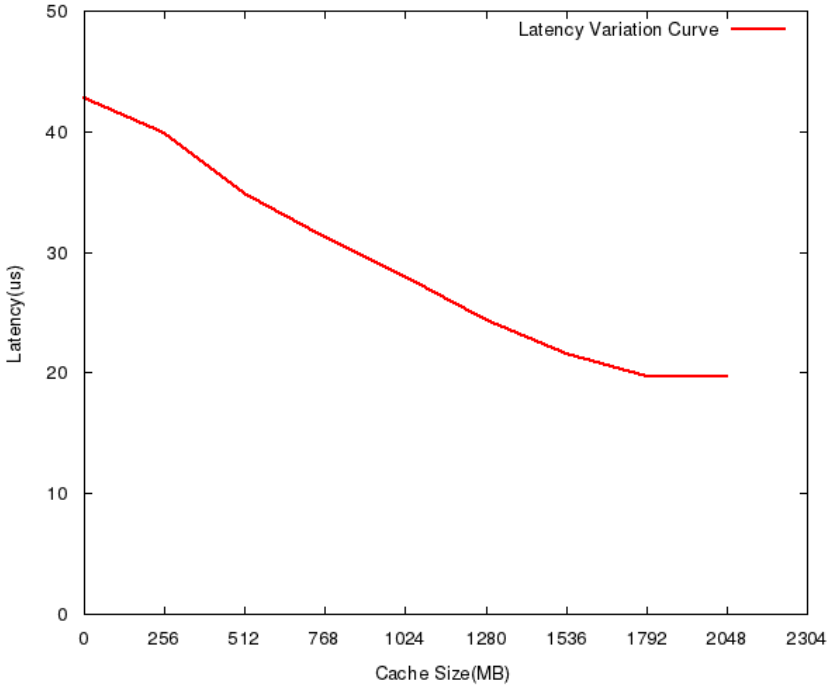


Figure 4.16: Latency vs SSD Cache Size

- Redis with Memory Cache

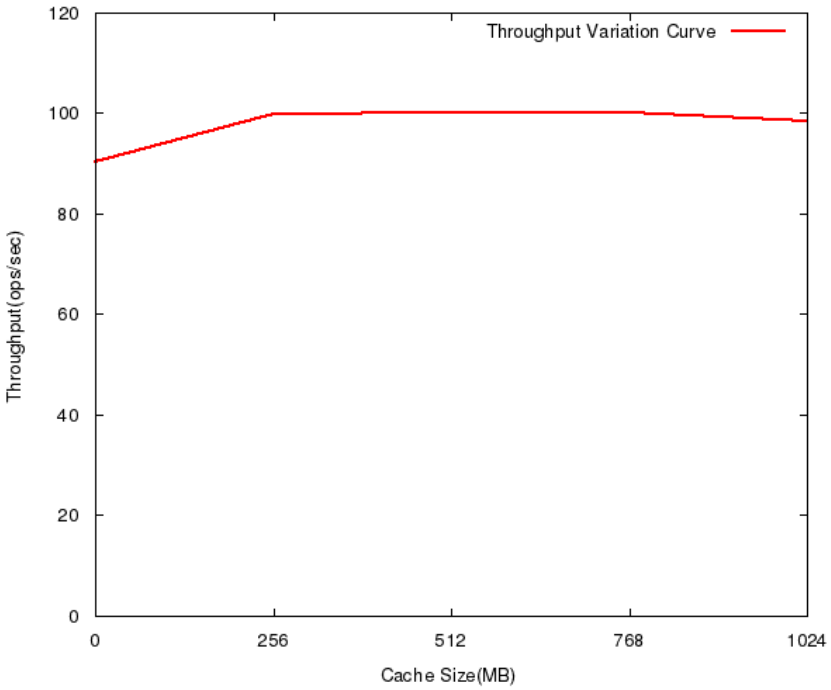


Figure 4.17: Throughput vs Memory Cache Size

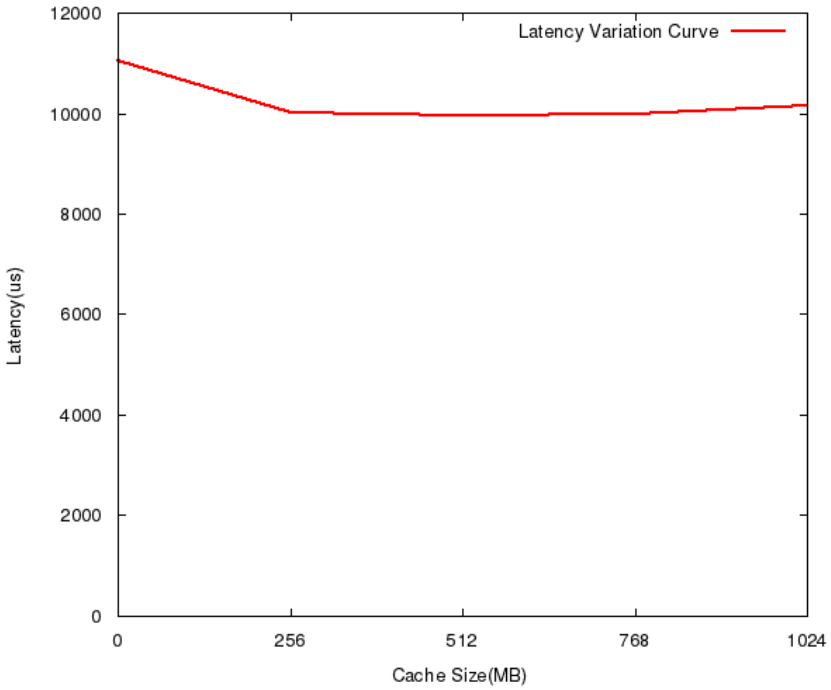


Figure 4.18: Latency vs Memory Cache Size

### 4.3.2 Cache Behaviour for the Applications

These are the results which the guest is sending to the host so as to reduce some opaqueness between the containers and the resource manager. This may help in cache provisioning decisions for the containers. Basically these results include IO requests which needs to be either accommodated into the cache or

retrieved from the cache. A successful retrieval counts to a hit in the cache. This hit would save us from going to the disk and reduce the disk I/O latency and improve the performance for a workload. Since hits are the most important metric for knowing the application behaviour in terms of performance, we are trying to show the cache hit variation of each application with varying cache size. Below are the plots shown for each application.

- Webserver provided with memory cache

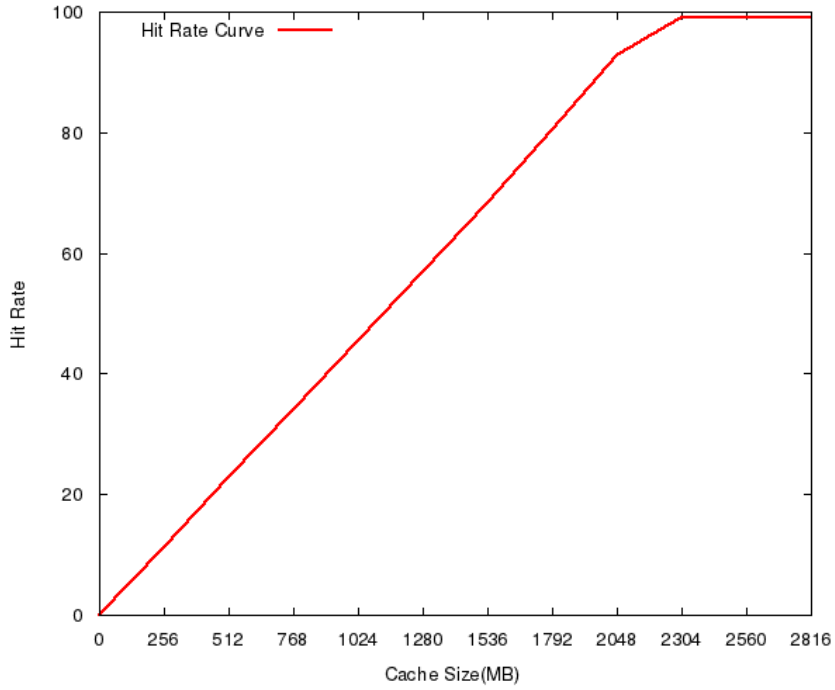


Figure 4.19: Hit Rate vs Memory Cache Size

- Webserver provided with SSD cache

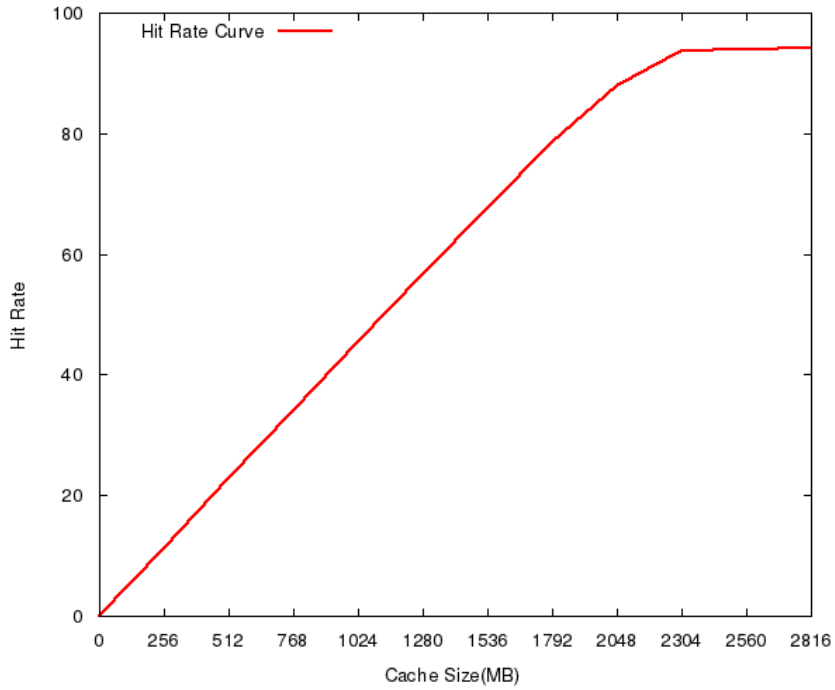


Figure 4.20: Hit Rate vs SSD Cache Size

- MongoDB provided with memory cache

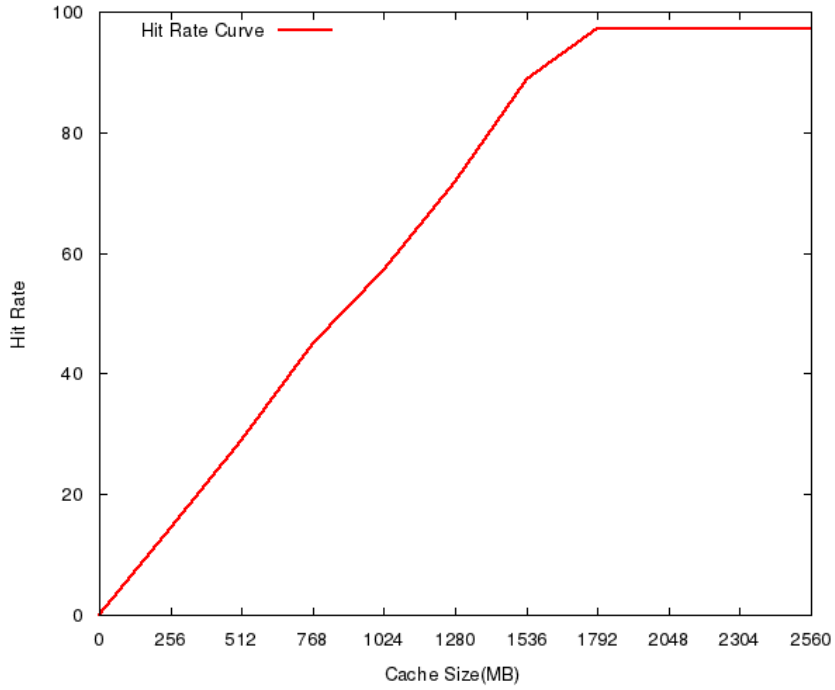


Figure 4.21: Hit Rate vs Memory Cache Size

- MongoDB provided with SSD cache

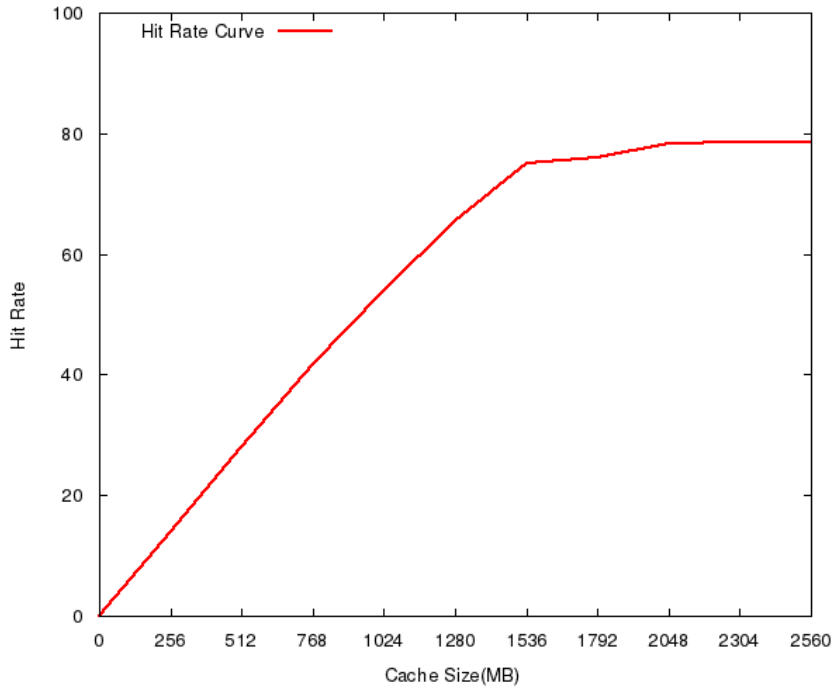


Figure 4.22: Hit Rate vs SSD Cache Size

- Mysql provided with memory cache

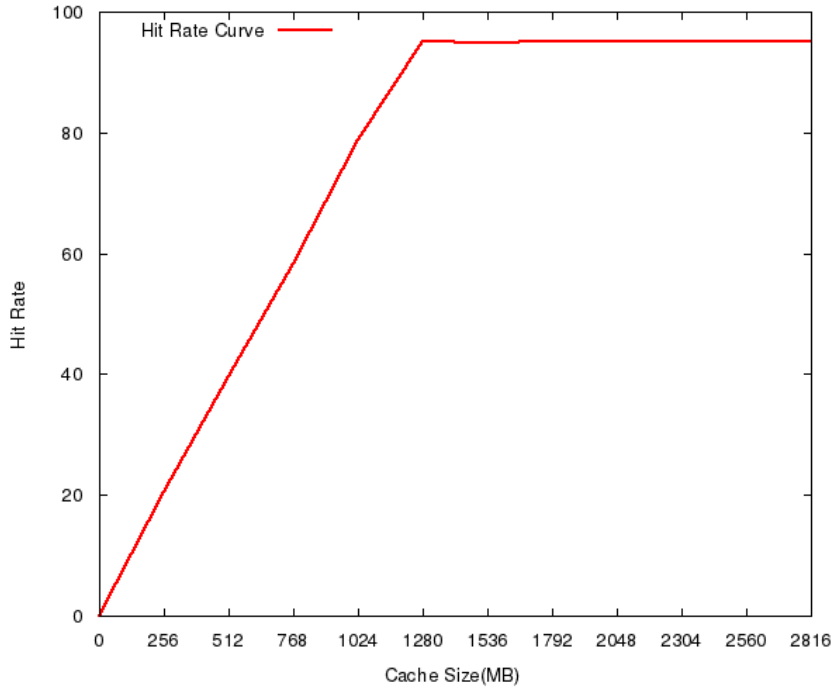


Figure 4.23: Hit Rate vs Memory Cache Size

- Mysql provided with SSD cache

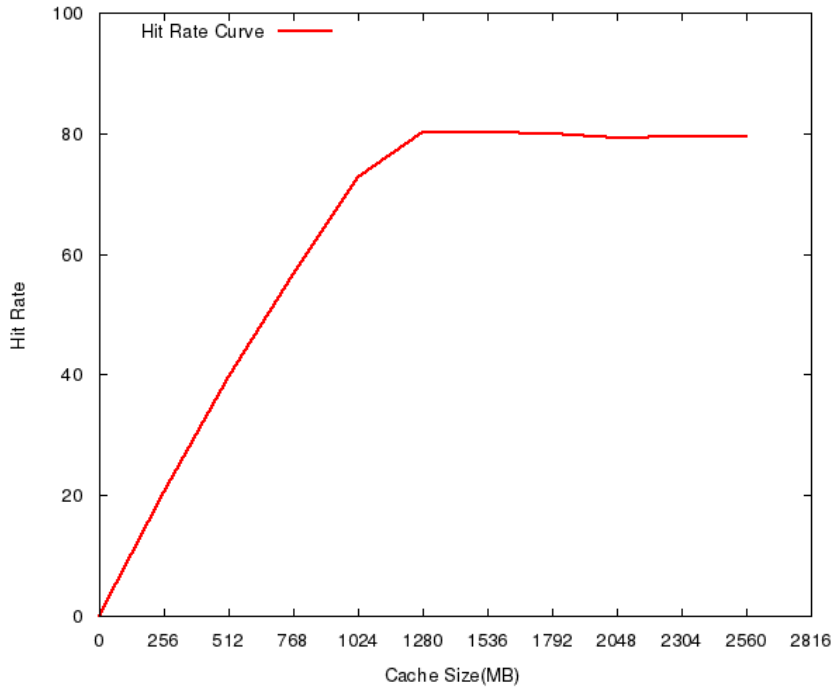


Figure 4.24: Hit Rate vs SSD Cache Size

- Webproxy provided with memory cache

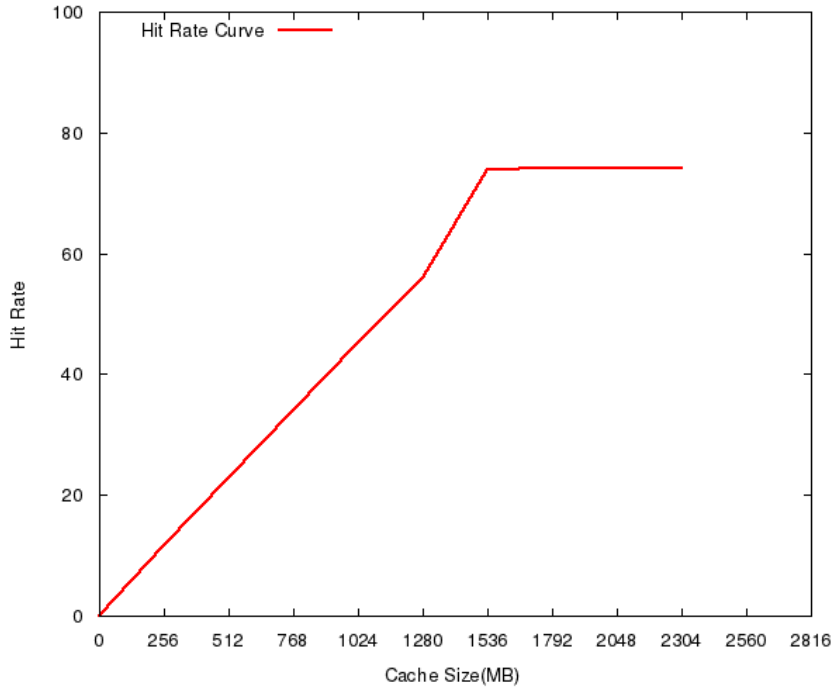


Figure 4.25: Hit Rate vs Memory Cache Size

- Webproxy provided with SSD cache

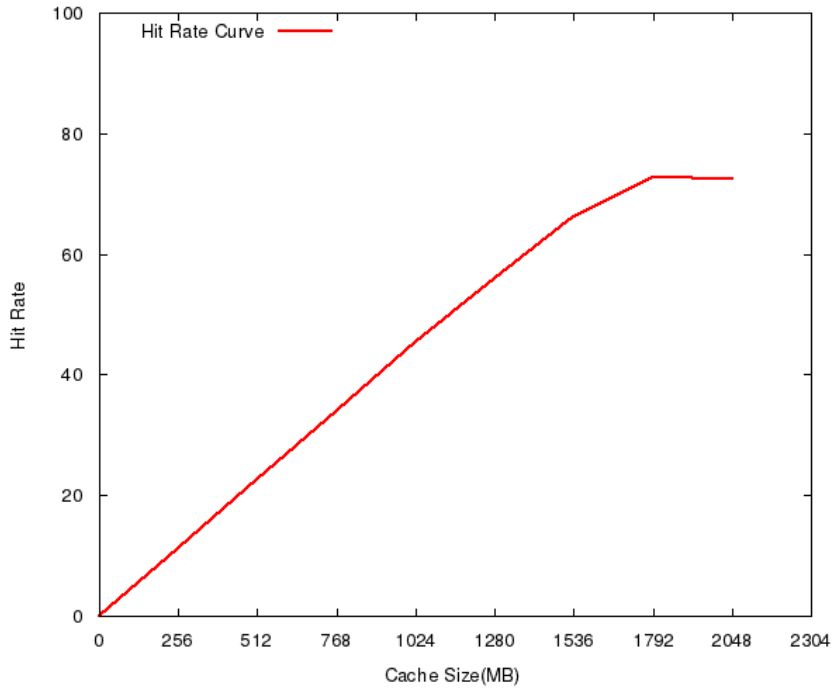


Figure 4.26: Hit Rate vs SSD Cache Size

- Redis with memory cache

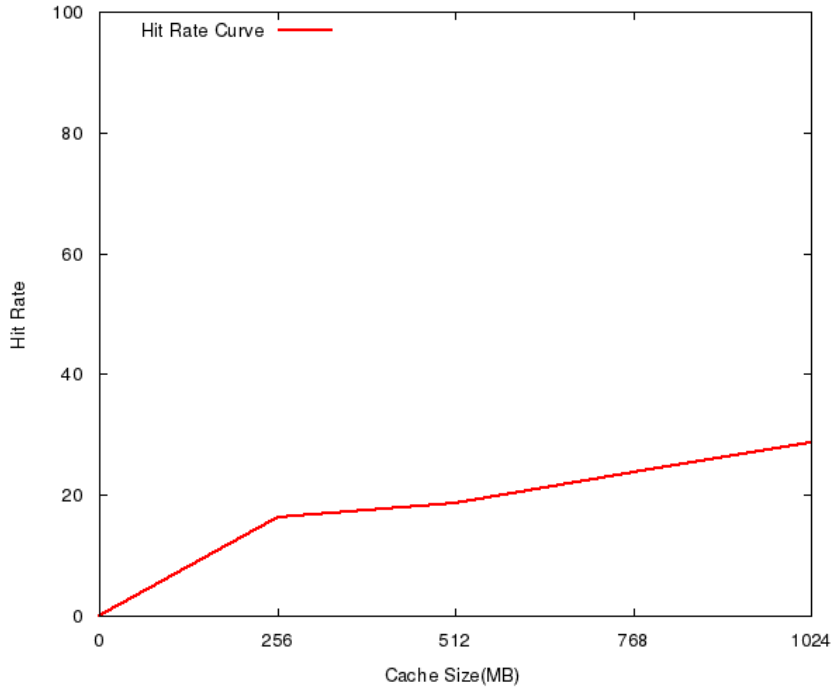


Figure 4.27: Hit Rate vs Memory Cache

## 4.4 Comparative Analysis

### 4.4.1 Comparing cache characteristics for different Applications

This section comprises of the results which we have for how cache characteristics changes with time for different applications. Application behaviour varies from each other in terms of cache usage over the period of time. This aspect is quite important as it can give an insight how requests are coming to the cache over the period of time. This is not important for our current work but could be useful with hybrid setup where we say an application can use both SSD and memory cache together. Then these results allow us to know and to make a decision as how much to keep in SSD cache and memory cache for an application. This may improve application performance as well as efficiently utilising memory. Since this is something related to application behaviour as application request pattern remain same whether we use SSD cache or memory cache we have one result for each application experimented with any of the cache store. The graphs for different applications are shown below.

- Webserver

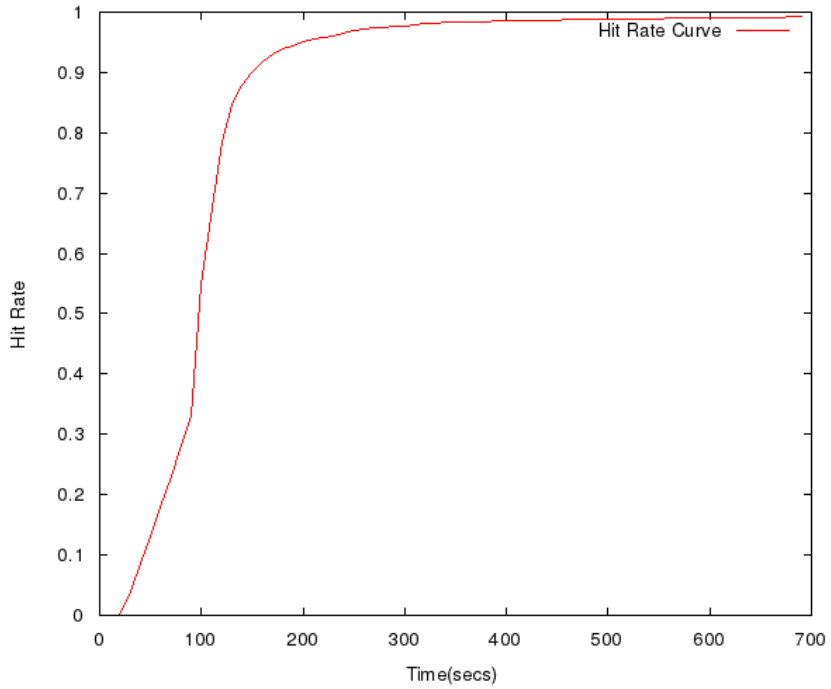


Figure 4.28: Hit Rate vs Time

- Mongodb

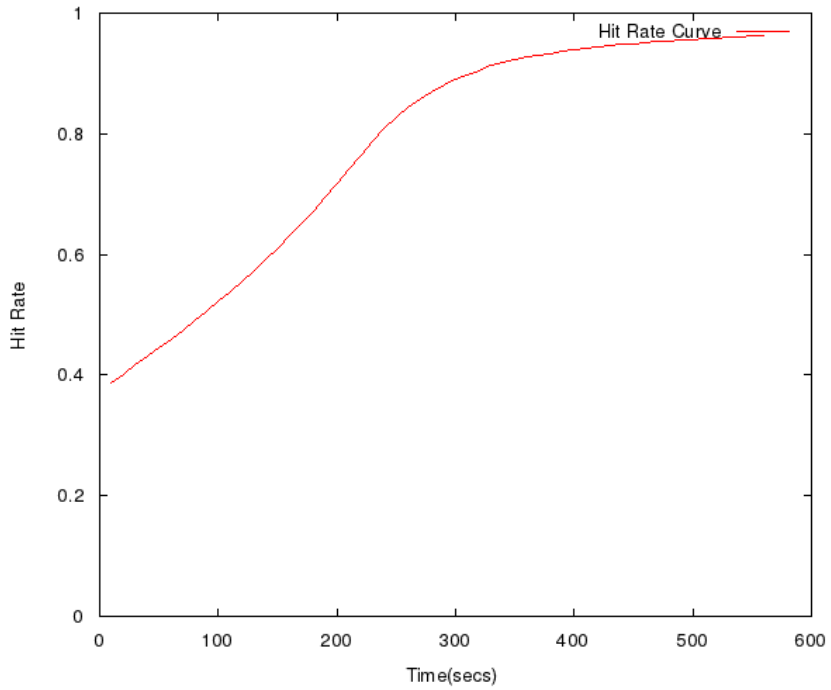


Figure 4.29: Hit Rate vs Time

- Mysql

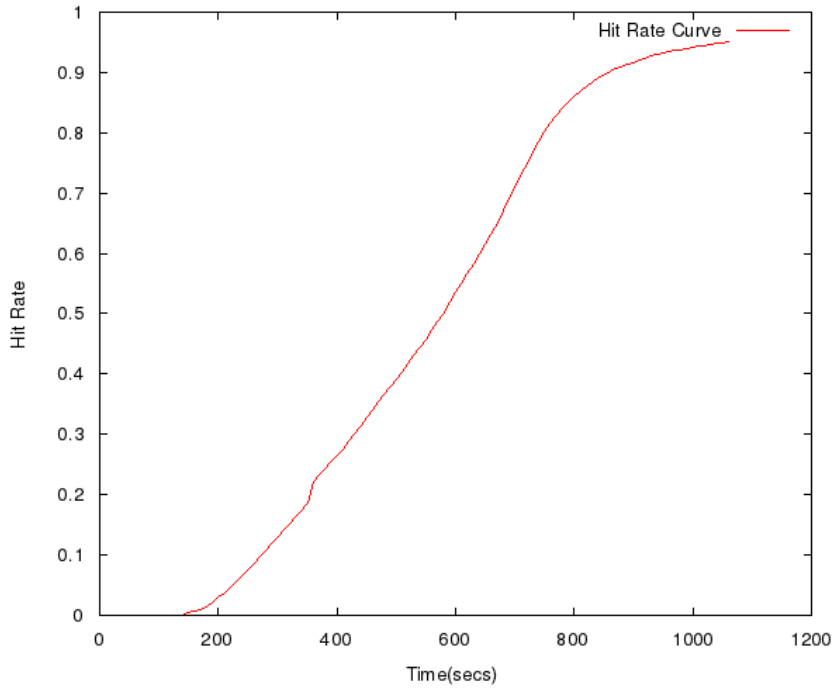


Figure 4.30: Hit Rate vs Time

- Redis

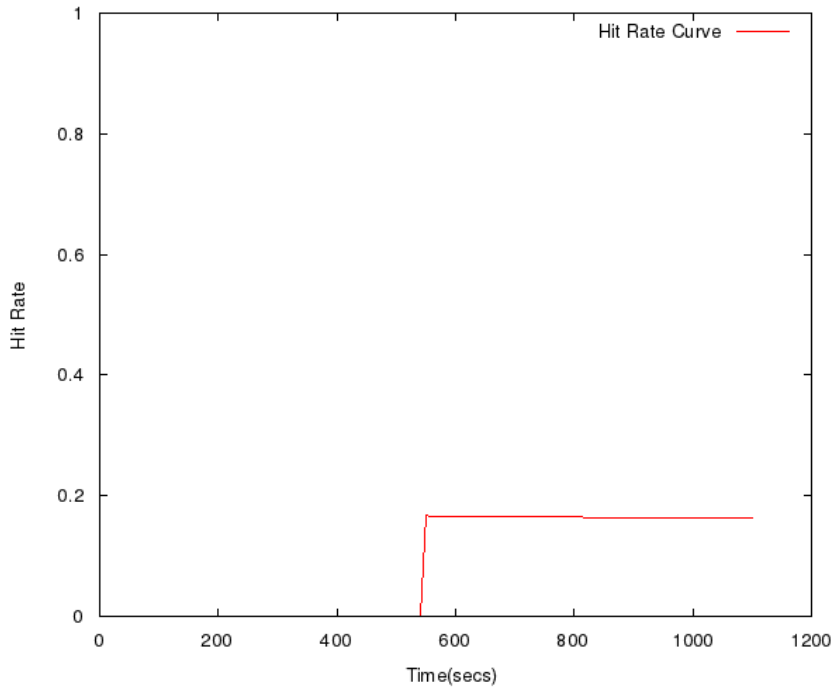


Figure 4.31: Hit Rate vs Time

### 4.4.2 Comparing different forms of memory usage of Applications

There are two types of memory used by applications in a system. One is anonymous memory and another is page cache memory, latter being file backed. Applications which are doing disk I/O will use most of the system memory as page cache while applications which are not doing disk I/O will use mostly anonymous memory. We have more variety here which is direct I/O, aaplications performing direct I/O will use only anonymous memory but no page cache memory.

This section shows the results as to how different applications use system memory in the form of anonymous memory and page cache memory. These results will again categorize an application into highly I/O intensive,moderately I/O intensive, memory intensive. Following table shows the results for different applications.

| Application | Page Cache Memory | Anonymous Memory |
|---|---|---|
| Webserver | 838.78 | 88.14 |
| MongoDB | 705.98 | 259.65 |
| Mysql | 387 | 566.61 |
| Webproxy | 765 | 77.34 |
| Redis | 1.57 | 952.1 |

Table 4.1: Cache Memory and Anonymous Memory usage of different applications

# 5. Cache Provisioning with Application Characteristics

Now we have seen behaviour of five different applications. Finding a use-case for the results is important. The problem statement given in the introduction section says that we need to characterize the applications and know their behaviour with the differentiated cache provisioning framework. This part has been done with the assumption that applications would behave approximately same when running individually as well as when running with multiple other applications. Keeping this in mind we can use the results to come up with a method to provision cache for different applications running together which satisfies their individual requirements. The next few sections are trying to address this problem.

## 5.1 Cache provisioning decision with the help of empirical results

For just to test whether our idea works, we have given a simple method which take application results and application requirements as input and return cache allocation weights for those applications.

The method is very simple and uses slope method to get the cache allocation weight for the application. Further in this section we will define the application requirements and show the cache allocation weights given by the method. In the next section we will experimentally verify whether the results obtained are satisfying the individual requirements of the applications. Following are the different tables representing the requirements and corresponding cache allocation.

- First set of requirements and cache required to satisfy those requirements

| Application | Performance(ops/sec) |
|---|---|
| mongo_medium | 1500 |
| mongo_low | 400 |
| mongo_high | 3000 |

Table 5.1: Application requirements

| Application | Memory Cache | SSD Cache |
|---|---|---|
| mongo_medium | 1522MB | 0MB |
| mongo_low | 1053MB | 1821MB |
| mongo_high | 1680MB | 0MB |

Table 5.2: Cache required for the performance fulfillment

Total amount various cache is shown by the following table:

| Memory Cache | 4GB |
|---|---|
| SSD Cache | 1821MB |

Table 5.3: Cache available in the system

- Second set of requirements and cache required to satisfy those requirements

| Application | Performance(ops/sec) |
|---|---|
| Webserver | 2000 |
| MondoDB | 600 |
| Mysql | 300 |
| Webproxy | 1000 |

Table 5.4: Application requirements

| Application | Memory Cache | SSD Cache |
|---|---|---|
| Webserver | 1605MB | 0 |
| MongoDB | 1275MB | 0 |
| Mysql | 743MB | 1000MB |
| Webproxy | 1227MB | 1227MB |

Table 5.5: Cache requirement for the performance

Total amount various cache is shown by the following table:

| Memory Cache | 4GB |
|---|---|
| SSD Cache | 1228MB |

Table 5.6: Cache available in the system

- Third set of requirements and cache required to satisfy those requirements

| Application | Performance(ops/sec) |
|---|---|
| Webserver | 2000 |
| MondoDB | 600 |
| Mysql | 300 |

Table 5.7: Application requirements

| Application | Memory Cache | SSD Cache |
|---|---|---|
| Webserver | 1605MB | 0MB |
| MongoDB | 1275MB | 0MB |
| Mysql | 743MB | 1000MB |

Table 5.8: Cache requirement for the performance

Total amount various cache is shown by the following table:

| Memory Cache | 3GB |
|---|---|
| SSD Cache | 1000MB |

Table 5.9: Cache available in the system

- Fourth set of requirements and cache required to satisfy those requirements

| Application | Performance(ops/sec) |
|---|---|
| Webserver | 7000 |
| MondoDB | 4000 |
| Mysql | 1500 |

Table 5.10: Application requirements

| Application | Memory Cache | SSD Cache |
|---|---|---|
| Webserver | 2060MB | 0 |
| MongoDB | 1779MB | 0 |
| Mysql | 1253MB | 0 |

Table 5.11: Cache requirement for the performance

Total amount various cache is shown by the following table:

| Memory Cache | 5GB |
|---|---|
| SSD Cache | 0MB |

Table 5.12: Cache available in the system

## 5.2 Experimental Verification of the results given by the procedure

Now after using the method we have cache allocations for different applications in different setups. The reason for having multiple setups is to have good combination of different requirements which can cover as many possibilities as possible. In this section we will have results after allocating desired cache to different applications in different setups.

- Results for the first set of applications chosen for experiment

| Application | Performance(ops/sec) |
|---|---|
| mongo_medium | 485.19 |
| mongo_low | 224.38 |
| mongo_high | 1167.31 |

Table 5.13: Application performance after allocating required amount of cache

The following graph represents the performance comparison

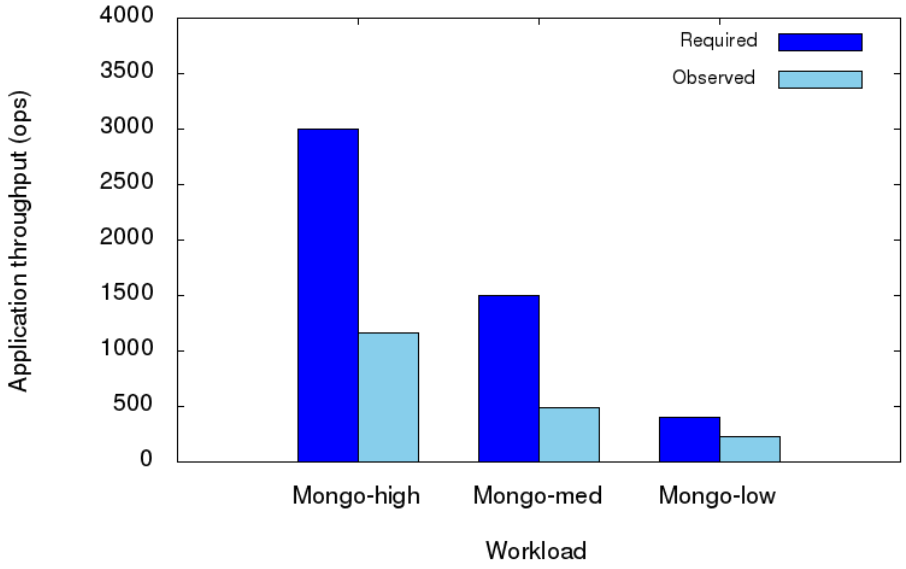


Figure 5.1: Application performance after provisioning

- Results for the second set of applications chosen for experiment

| Application | Performance(ops/sec) |
|---|---|
| Webserver | 827.17 |
| MondoDB | 66.82 |
| Mysql | 69.81 |
| Webproxy | 175.91 |

Table 5.14: Application performance after allocating required amount of cache

The following graph represents the performance comparison

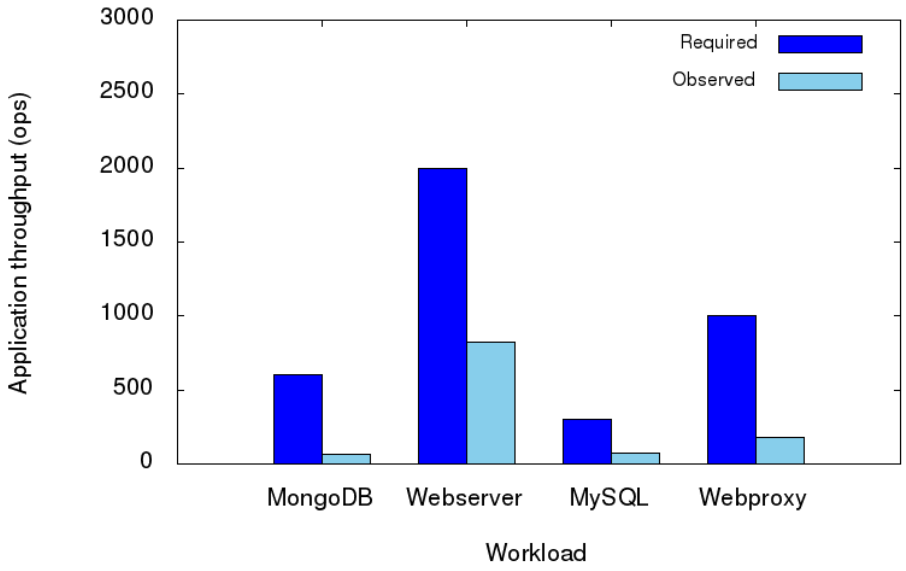


Figure 5.2: Application performance after provisioning

- Results for the third set of applications chosen for experiment

| Application | Performance(ops/sec) |
|---|---|
| Webserver | 2769.76 |
| MondoDB | 92.75 |
| Mysql | 93.46 |

Table 5.15: Application performance after allocating required amount of cache

The following graph represents the performance comparison

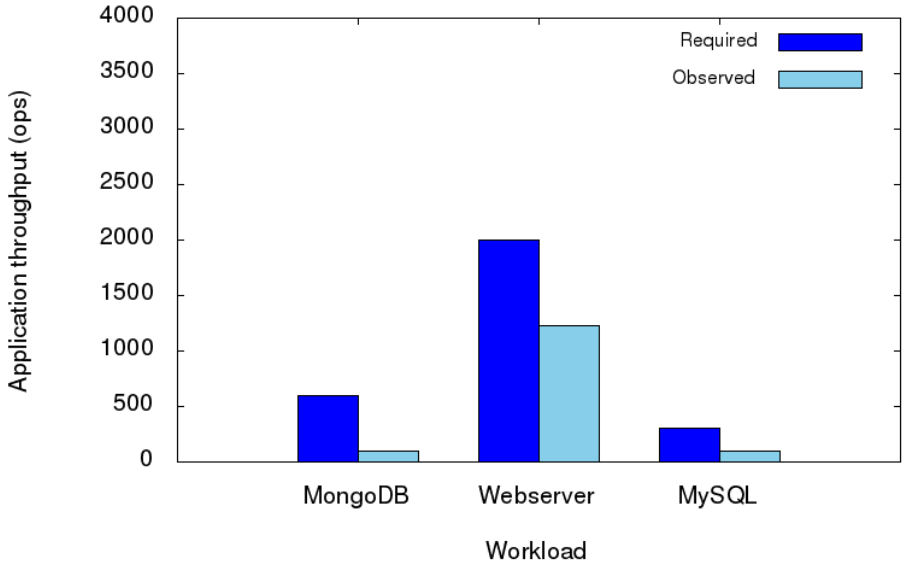


Figure 5.3: Application performance after provisioning

- Results for the fourth set of applications chosen for experiment

| Application | Performance(ops/sec) |
|---|---|
| Webserver | 6186.08 |
| MondoDB | 1625.50 |
| Mysql | 1349.70 |

Table 5.16: Application performance after allocating required amount of cache

The following graph represents the performance comparison

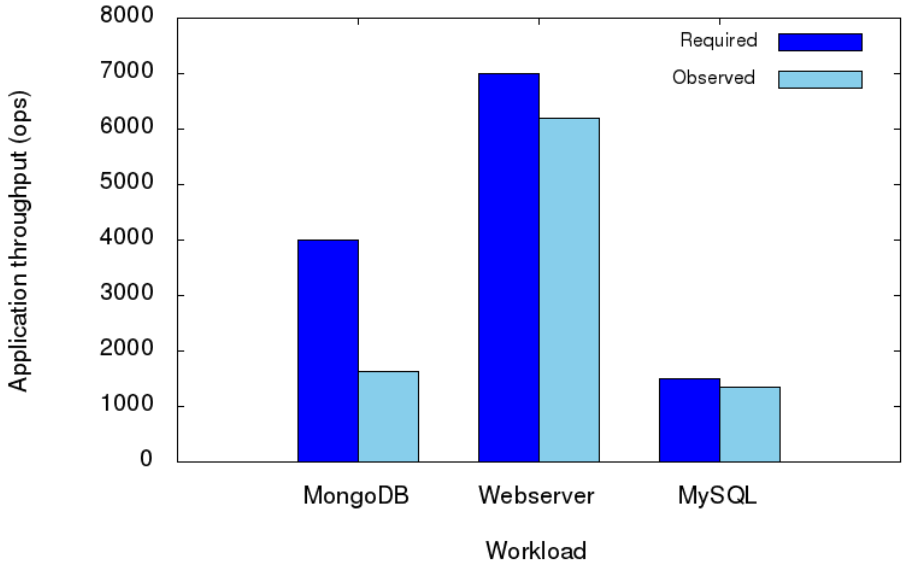


Figure 5.4: Application performance after provisioning

## 5.3 Studying application performance with desired cache allocation

Before we analyse results and come to some conclusion, we should have information about the working set size of the different applications we have. Therefore following table would give the working set size of the different applications under observation.

| Application | WSS(GB) |
|---|---|
| Webserver | 2.82 |
| MondoDB | 2.18 |
| Mysql | 1.37 |
| Webproxy | 2.50 |

Table 5.17: Working Set size of different application

This WSS information include page cache as well as the second chance cache.

Looking at the results for different setups, we can see that results are not as expected. Even after providing the desired cache sizes to various applications we are not able to achieve the performance which

is expected. We tried to figure out what is reason for such behaviour. But before having a discussion about it we should discuss following things about different application combination and requirements.

- **First setup**: Here we have chosen three same applications. Their requirements are such that they cover high, low, medium performance for the applications. In this two of the application with high and moderate performance requirement approached working set size for themselves.
- **Second setup**: In this setup we have four applications where the requirements are moderate depending on the application. Only Mysql has low requirements. Webserver and Webproxy are highly I/O intensive. Cache requirement of the applications was quite less than the working set size of the application.
- **Third setup**: Since the results for second setup were bad we tried to have to have only one highly I/O intensive. There were only 3 applications in this setup.
- **Fourth Setup**: In this setup all the applications have high performance requirements. And their cache requirements are approaching their working set size.

Since results are not as expected for all the four setups. We will look for a possible reason with only one setup. We have chosen fourth setup for this discussion.

## 5.4 Comparison of collective performance with isolated performance

- Cache Behaviour

| Application | Isolated | Simultaneous |
|---|---|---|
| Webserver | 98.55 | 98.43 |
| MongoDB | 97 | 93.01 |
| Mysql | 93 | 89.52 |

Table 5.18: Cache behaviour comparison of applications running in isolation vs simultaneous

Though we can see that cache behaviour is almost same for isolated and simultaneous case, but still the performance of mongodb is quite less than what we expected. Since all the resources are same when running individually or simultaneously, there should be some bottleneck for such performance degradation.

Since memory usage depends on the application behaviour and it is not changing in both the cases, therefore there is no difference in memory usage in both the case. Cache behaviour we have just seen which is also same. Since we have done CPU pinning and same cores are being given, therefore here also there is no discrepancy.

Last resource left to be looked upon is disk. As we all know it is shared resource and a very slow resource as compared to memory and CPU. The crux of this project is that we are using caching for improving performance, since I/O is quite costly with disk. If we have I/O intensive applications and a cache other than page cache we can improve disk latency and application performance.

Now in the above setup we are running 3 applications together. All three of them are contending for the same disk. What we have observed is a lot I/O waiting on the cores being given to the application. We

have a fixed load phase for application and during that load phase all the requests are not in the cache and we are still going to the disk for some requests. Therefore applications has to wait for the input and output requests to complete, because other applications are in queue for its input output operations to complete.

This can be shown with a log which has I/O stats including I/O wait for each core. Since we know the core being allocated to the application we can verify the results. The results which we are showing consider only the run phase for an application. I have verified it for only mongodb application just to provide it for evidence.

- Average I/O wait is above 90% for most run of the experiment in case of simultaneous execution of mongodb.
- Average I/O wait is around 70% in the beginning for 1-2 mins and reduces to 2-3% for rest of the run in isolated execution of mongodb.

## 5.5 Reasons for failure of the hypothesis and solutions for getting it right

**Hypothesis**: Knowing the individual behaviour of the applications we can use it to provision cache to the applications and expect them to behave the same when running them together.

### 5.5.1 Disk I/O contention and less CPU utilization

This hypothesis misses out disk contention. Since disk is a very slow resource, therefore when one application was running that time there was very less time spent on I/O wait but with multiple applications running together depending on the requesting rate of the applications some CPU utilisation of some applications is very less due to I/O wait.

But there is a case where having a cache can improve this situation. When the applications are running with high performance requirement and cache required for it nearly equals working set size then performance is not hampered due to I/O wait.

# 6. Conclusion And Future Work

We conclude this thesis with an empirical study of the framework with differentiated cache provisioning for derivative clouds, behavioural analysis of different applications and finally using those application characteristics for having a way to determine cache allocation required for meeting individual requirements of applications.

There is always a scope for improvement. As mentioned in the previous chapter that the hypothesis we used to perform experiment was partially satisfied therefore it is required to redesign the experiments which take into account the I/O wait of the CPU for the applications. Future extensions can be:

- With redesigned experiment results, provide with an algorithm which tries to satisfy individual application requirements as well as maximize throughput.
- Since the current work was done statically, something can thought about doing it dynamically.

# Bibliography


[1] LINUX cgroup. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[2] LINUX containers. https://linuxcontainers.org/.

[3] LINUX kernel documentaion: vm/cleancache. www.kernel.org/doc/Documentation/vm/cleancache.txt.

[4] MongoDB database description. https://docs.mongodb.com/.

[5] Redis dataset description. https://redis.io/documentation.

[6] SUN filebench. www.filebench.sourceforge.net/wiki/index.php/Main_Page.

[7] YCSB benchmark description. https://redis.io/documentation.

[8] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using ksm. In *Proceedings of the linux symposium*, pages 19–28. Citeseer, 2009.

[9] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM, 2003.

[10] Sean Kenneth Barker, Timothy Wood, Prashant J Shenoy, and Ramesh K Sitaraman. An empirical study of memory sharing in virtual machines. In *USENIX Annual Technical Conference*, pages 273–284, 2012.

[11] Muli Ben-Yehuda, Michael D Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The turtles project: Design and implementation of nested virtualization. In *OSDI*, volume 10, pages 423–436, 2010.

[12] Neil Brown. Control groups series. 2014.

[13] Dan Magenheimer. Chris Mason. Dave McCracken. Kurt Hackel. Comparative analysis of page cache provisioning in virtualized environmentstranscendent memory and linux. In *Ottawa Linux Symposium (OLS)*, 2009.

[14] Jinho Hwang, Ahsen Uppal, Timothy Wood, and Howie Huang. Mortar: Filling the gaps in data center memory. *ACM SIGPLAN Notices*, 49(7):53–64, 2014.

[15] Dan Magenheimer. Transcendent memory in a nutshell. 2011.

[16] Grzegorz Miłós, Derek G Murray, Steven Hand, and Michael A Fetterman. Satori: Enlightened page sharing. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, pages 1–1, 2009.

[17] D. Mishra and P. Kulkarni. Comparative analysis of page cache provisioning in virtualized environments. In *2014 IEEE 22nd International Symposium on Modelling, Analysis Simulation of Computer and Telecommunication Systems*, pages 213–222, 2014.

[18] Prateek Sharma and Purushottam Kulkarni. Singleton: system-wide page deduplication in virtual environments. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 15–26. ACM, 2012.

[19] Prateek Sharma, Stephen Lee, Tian Guo, David Irwin, and Prashant Shenoy. Spotcheck: Designing a derivative iaas cloud on the spot market. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, 2015.

[20] Ioan Stefanovici, Eno Thereska, Greg OShea, Bianca Schroeder, Hitesh Ballani, Thomas Karagiannis, Antony Rowstron, and Tom Talpey. Software-defined caching. Technical report, Tech. Rep, University of Toronto.

[21] Carl A Waldspurger. Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[22] Dan Williams, Hani Jamjoom, and Hakim Weatherspoon. The xen-blanket: Virtualize once, run everywhere. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 113–126, 2012.