Adaptive memory management frameworks for derivative clouds

Master's Thesis Report

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Technology

by

Prashanth
Roll No: 153050095

under the guidance of

Prof. Purushottam Kulkarni

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

Dissertation Approval

This dissertation entitled **Adapative memory management frameworks for derivative clouds** by **Prashanth** is approved for the degree of Master of Technology in Computer Science and Engineering from IIT Bombay.

Examiners

Myttkin Vutukuru V. Mythkin

Varsha Apte

Supervisor

Purushottam Kulkarni

Chairman

Varsha Apte

Date: 28-06-2017

Place: IIT-BOMBAY, MUMBAI

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.

Prashanth

(Signature)

PRASHANTH

(Name of the student)

153050095

(Roll No.)

Date: 26-06-2017

Abstract

Derivative clouds—nesting of virtualization entities to provision cloud platforms, has gained popularity with the emergence of container based virtualization techniques. Several large cloud providers have adopted derivative clouds by provisioning containers within virtual machines to provide cloud services to their customers and to stay at par with the industry standards, we also have followed a similar setup to emulate our derivative cloud setup.

As a part of our initial work, we have made an attempt to understand memory management in containers in a native and derivative cloud setup. We have identified drawbacks in the memory management controller provided by Linux, thereafter which we proposed updates to the existing controller and evaluated the same. Our controller was able to differentially provision memory as a resource among several container executing both in a native and derivative setup. The effects of such differentially provisioned containers proportionally impacted application throughputs.

We then moved to analyzing cache partitioning frameworks and how they impact in achieving application performance objectives. We identified that existing cache partitioning frameworks had no support for derivative clouds and hence made changes to it. However we established that mere cache partitioning frameworks fail to support applications having in-VM memory requirements. Hence we moved to a decentralized multi-level memory management framework that supports provisioning both memory and cache as a resource at the derivative provider. With our memory management framework we were able to provision for a wider range of applications. .

Acknowledgements

I would like to thank my advisor Prof. Purushottam Kulkarni for his constant support and advice which has helped me take this thesis to its completion. I would also like to thank Prof. Umesh Bellur for his guidance during the Phase-1 of my M.Tech. thesis project. Lastly, I would like to thank Debadatta Mishra and Chandra Prakash with whom I have collaborated by research works with for being as supportive as possible.

Prashanth

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The term computing is pervasive today. In additional to this, the past couple of decades has witnessed an hostile take over by *The Internet*, which has been helpful in connecting systems over the globe. Overtime, several businesses have started offering computational resources as a service, over the Internet. This has lead to the establishment of a new paradigm of computation known as *Cloud Computing* or simply the Cloud and such businesses are called cloud service providers.

The objective of a customer is to run his application on the cloud without affecting the performance of his application. On the other hand, the objective of a cloud provider is to minimize running costs when serving multiple such customers with promised guarantees to make their business profitable. This leads to conflicting goals between the provider and the customer. How this is handled would be discussed in the next section.

Providers today offer various kinds of cloud services like Software as a service (SaaS), Platform as a service (PaaS) and Infrastructure as a service (IaaS). SaaS provides software applications being run on a cloud server. PaaS supports the complete life cycle of building and delivering applications. IaaS provides basic compute resources as a service, and is considered as the most primitive form of providing cloud services. Most of our discussions would be centered with having IaaS in mind although the findings could be extrapolated to either of the services types mentioned here. A recent study [3] suggests that 75% of the corporates are migrating to using the cloud to run their businesses, and also that by 2020 all corporates would be using cloud services similar to how the Internet is used today.

The emergence of cloud computing paradigm has opened gates to a new direction for flow of Systems research. Traditional systems were developed to only serve a single or a group of trusted users. Now with multiple untrusted users existing on a single system has lead to changing the focus of improving efficiency, manageability, service guarantees over a group of isolated customers, who have to be protected from being affected by other customers running on the same system. One of the common ways to achieve this is using *Virtualization*. Virtualization seems to be the most effective and secure way of achieving this. Virtualization has several techniques used but the two techniques we would be focusing are Hardware level virtualization using *Virtual Machines* (VM) and Operating System level virtualization using *Containers*.

1.1 Deterministic resource provisioning for cloud services

Most commonly provisioned resources are compute, storage, network and memory. Most cloud services make use of hardware level virtualization techniques that use Virtual Machines to provision compute resources as per client requirements. Such use of virtual machines has been done for over a decade now

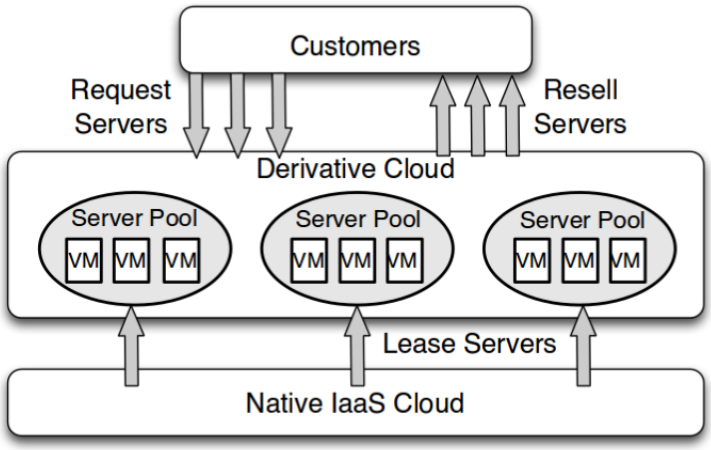


Figure 1.1: Depiction of a derivative IaaS cloud platform, Source:[1]

and have gotten relatively stable and secure. Earlier resources were provisioned by cloud providers were static, however static provisioning leaves less room for server consolidation hence providers are moving to a more elastic on-demand model where resources provisioned are over-committed to a group of customer and servers are provisioned as per actual resource needs. A popular example for the same is Amazon's EC2 [4] that offers elastic web services which expand or compress as per actual needs. Elastic provisioning of resources in a virtualized environment is more tricky task and leads to a lot of non-determinism.

There are several advanced resource provisioning techniques [5] [6] [7] [8] etc. using virtual machines that make use of horizontal/vertical provisioning techniques to satisfy client QOS (quality of service) requirements at the same time perform server consolidations to reduce operating costs. However, hardware level virtualization layer induces overheads that are caused by dual control loop while scheduling resources, complete hardware stack emulation for each VM, resources used by the hypervisor (entity that manages VMs) etc. These overheads lead to bad cost-benefit ratios which adversely affects customers by overpricing services offer by cloud provider.

A recent trend in virtualization has been towards OS-virtualization that makes use of lightweight containers to provision resources. Several researches [9] [10] [11] [12] [13] have shown that containers provide provide near about the same features (with a few limitations) as that of virtual machines but with much lesser overheads. Static provisioning using containers can be done easily today, however deterministic provisioning of containers is still to be explored in depth specially in situations of overcommitment. Containers are relatively a young technology that needs further refinement to be used in deployment. Several enterprises are hesitant to move towards containers due to the existing security issues. More about containers shall be provided in the coming chapters.

This idea of deterministic provisioning can be expanded to a *derivative cloud* environment as purposed by P.Sharma in his recent work [1] which repackages and resells resources purchased from native IaaS platforms. A derivative cloud can offer resources to customers with different pricing models and availability guarantees not provided by native platforms using a mix of resources purchased under different contract. Derivative cloud providers rent resources from native cloud providers to resell services to customers as shown in Fig 1.1.

Although containers support has been provided in most major operating systems today, it is most popular and widely used in operating systems running on a Linux kernel. Our entire work would focus on elastic provisioning of containers in a native Linux environment and extend its implications to the derivative setup. However at this stage, we focus on deterministic provisioning of memory and disk-caching as a resource in this thesis.

1.2 Memory management in clouds

Memory as a resource has gained popularity recently with emergence of more memory intensive applications in various fields of computing like data analytics, caching that have a very strong correlation with application performance that is dependent on the memory available in the system. Most memory sensitive applications constantly use any unused memory available in the system to benefit them. A simple example is an Key-value used to cache frequent key's accessed by a web application.

Currently available provisioning knobs in the Linux container framework are quite effective while provisioning for applications when the host system isn't under any memory pressure and overcommitment (When resources promised on a system is more than resources available). However overcommitment is a fundamental requirement to provision cloud servers to maximize provider running costs as discussed earlier. In a native Linux system, memory pressure might be generated due to the following

1. Additional memory required by processes of other containers
2. More memory required by processes/services running on host Linux OS
3. Memory pressure generated by kernel threads/processes

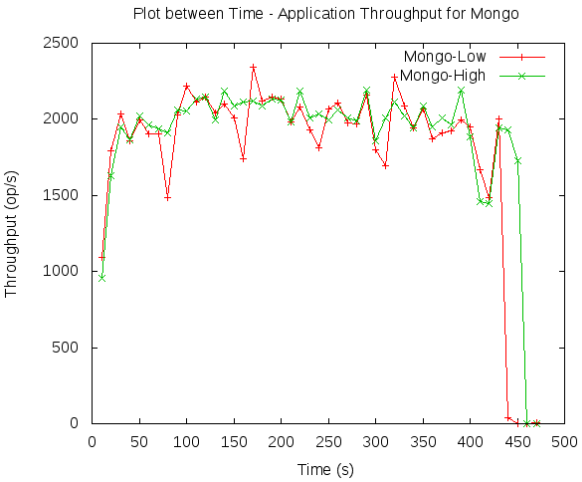
Let's see how memory overcommitment and pressure can disrupt desired functionality of the existing container framework provided by Linux containers.

1.2.1 Issues in native container environment

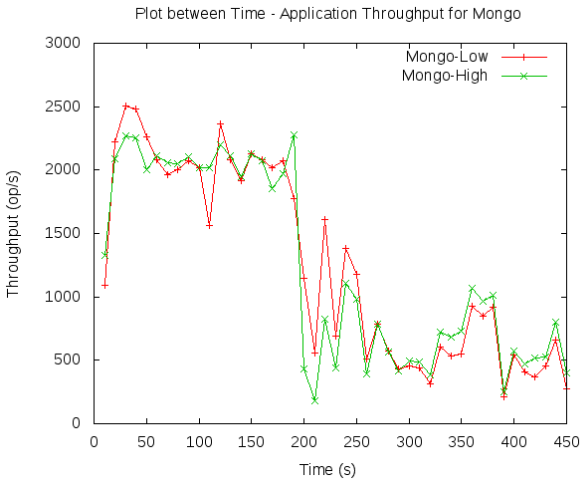
Consider two containers provisioned for running Mongo-DB containers from 2 different customers on a same host machines. Now that average memory used by the 2 containers are in the ratio of 1:2 and the customers for the 2 containers are also paying for their services in the same ratio. Let's call container with 1x workload usage as Mongo-Low and that of with 2x usage as Mongo-High. Now assume the customers have been provisioned using existing memory knobs with the same ratio as shown in Tab:1.1. For the sake of simplicity assume that all the containers over-provisioned for all other resources and aren't throttled by any other resource. Low and High can be thought of relative priorities of each of the containers.

	Average Memory Usage	Cost Paid for Service	Memory Provisioned	Desired Throughput
Mongo-Low	1x	1x	1x	1x
Mongo-High	2x	2x	2x	1x

Table 1.1: Memory Provisioned for the Containers using existing memory provisioning knobs



(a) Running applications without containers



(b) Observed after provisioning containers using existing knobs

(c) Desired after provisioning containers using knobs

Figure 1.2: Application throughputs for problem establishment

Consider 3 cases as described below. In cases 2 and 3, containers are allowed to run normally for about 100s and there after which free memory in the system gradually reduces by generating of memory pressure by an external entity.

- Case-1: Running applications natively in a system with 1:2 memory assignments and no pressure
- Case-2: Observed throughputs after provisioning with existing knobs under pressure
- Case-3: Desired throughputs under pressure

Fig 1.2a shows the simple case where both the containers achieve desired throughputs when running on a system with no container specific provisioning. The containers start execution with no memory pressure and hence must be able to each equal throughputs initially in all 3 cases although. Case-2 still shows lower levels of application performance for higher provisioned containers even when the system is under no pressure. In observed throughputs under pressure Fig 1.2b, we observe that throughputs of Mongo-High (Container with higher priority) is negatively affected at some point or the other, when in reality its throughput had to be better if not same as shown in Fig 1.2c considering its higher resource allocations.

	No Pressure	Observed	Desired
Mongo-Low	1825	1268	1255−
Mongo-High	1972	1242	1255+

Table 1.2: Average throughput in each case (op/s)

Table 1.2 shows how average throughputs vary in each case. It can be seen that the average throughput in case-2 which is the provisioning of containers using existing knobs may negatively impact containers with higher allocations. By looking at the example here, we can conclude by saying that

1. Native memory allocations work well with containers where there doesn't exist any memory overcommitment and pressure
2. However when the two occur, memory reclamation may adversely affects containers which are better provisioned (since they were promised higher QOS) than those which aren't.

1.2.2 Amplification of issues in derivative cloud environment

Considering the above described setup to derivative cloud where the native cloud provider is using VMs to provision customer demands. This VM acquired from the native cloud provider is again repacked and resold by the derivative cloud provider to specific customers. In this case, this situation further complicated due to two reasons,

In the native case, memory overcommitment was a required condition for the previously described situation to arise. Consider the case where all containers were assigned memory considering the available system memory in such a way that there is no overcommitment, however now the native cloud provider (host system) could reduce the memory available to the system using different memory reclamation policies at the host.

1. Memory overcommitment is not a required condition
2. Memory pressure maybe introduced by three factors described earlier or an additional factor like an external host system driver (eg: Balloon Driver)

The reclamation could be trigger by a host driver like the *Balloon Driver* that is widely used by the *Hypervisor* (Entity that manages VMs) by cloud providers. This leads to further discrepancies in the memory management at the container level.

1.3 Caching in the cloud

Caching of data has played a crucial role in provisioning for applications on the cloud. Caching provides a faster mechanism to access frequently accessed data. There are several web services these days that make use of CDNs (content distribution networks) to cache their frequently accessed to minimize response time by spreading these CDNs servers across the globe. This is one form of commonly used caching frameworks. Another form of caching occurs at the application level where frequently accessed key-value pairs are stored in a key-value store like Redis [14], Memcached [15]. However our work deals with cloud frameworks where the cloud provider caches client data to improve client performance. Let's begin with how caching occurs in a traditional cloud setup.

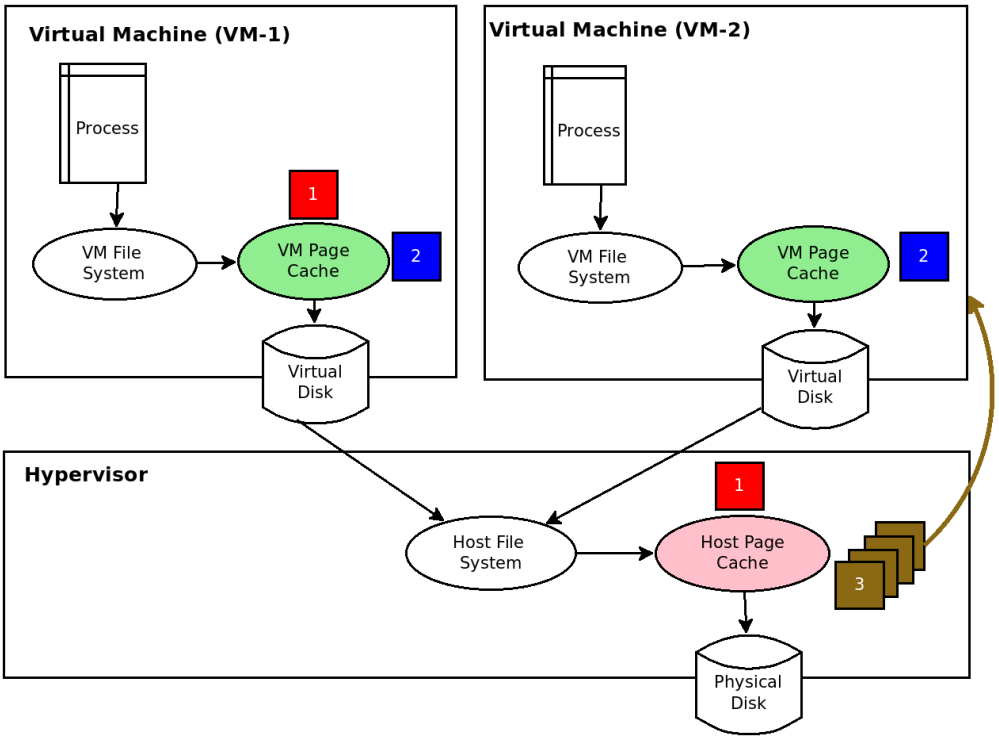


Figure 1.3: Drawbacks of caching in traditional VM cloud setup

1.3.1 Drawbacks of caching in traditional (VM) cloud setup

Fig 1.3 depicts a traditional host page caching at occurs in a virtualized setup. There are two-levels of caching occurring here, one at the guest and the other at the host level. Both these caches are controlled by their respective operating systems and are unaware of one another. This leads to improved performance but will also lead to wastage of resource. The drawbacks of such a setup are (as illustrated in Fig 1.3 listed below,

1. A copy of the same page is present at the hypervisor page cache and the VM page cache.
2. A copy of the same page maybe present across VMs.
3. The host page cache could be flooded with page caches from a single or a set of VMs.

1.3.2 Hypervisor managed caching

To overcome above drawbacks, several works[16, 17] have proposed the use of a more controlled caching framework called the *Hypervisor managed caches* as depicted in Fig 1.4. Hypervisor managed caches are caching frameworks whose control lies in the hands of the native cloud provider. The native cloud provider cloud provision these caches to satisfy an application level objective which could be exposed as a service to their clients or also to configured based on a global provider level policy to maximize throughput.

Hypervisor managed caches eliminates the drawbacks previously discussed by providing an exclusive cache there by removing any redundant pages. The cache size can be configured there by eliminating flooding of page cache by a single VM. Typically host-side page cache is turned off when hypervisor managed cache is enabled.

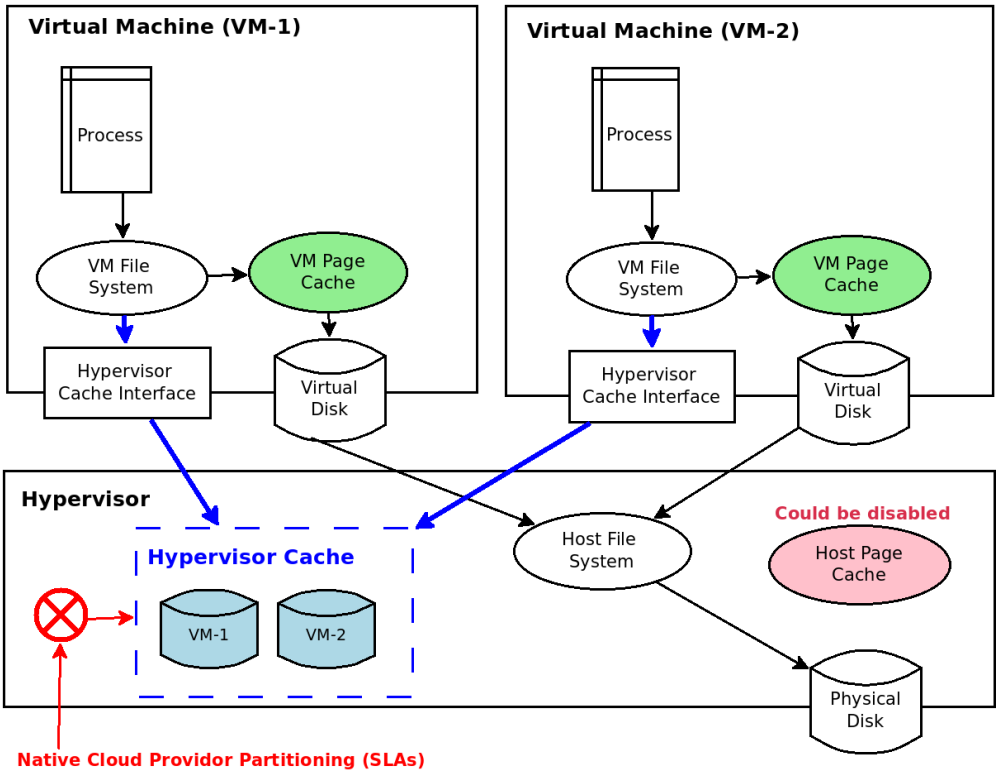


Figure 1.4: Hypervisor managed cache

1.3.3 Issues of caching frameworks in derivative clouds

Hypervisor managed caches have shown significant improvements in performance improvements in a native cloud environment, but however when it comes to derivative clouds the existing frameworks aren't able to differentiate the nested levels of virtualization entities. Existing cache partitioning frameworks[17, 18] that partition caches for each VM assumed a single application running and hence successfully partitions the same, however with multiple applications running in an derivative setup, cache partition to achieve application objectives (SLAs) becomes challenging as shown by DoubleDecker[19]. DoubleDecker was the preliminary work done based on which this work is built upon, however even with DoubleDecker like other caching frameworks there exist drawbacks as discussed below.

Lack of framework support in derivative clouds

Caching frameworks existing lack support for derivative clouds. They lack the engineering components required to exposed container as an individual provisioning entity. DoubleDecker addressed this by exposing cache provisioning to a derivative cloud provider. A similar approach could be used for other partitioning frameworks as well.

Dual layers of isolated control

With a caching framework like mentioned above incorporated into a derivative cloud setup makes it have two isolated control centers - one at the native cloud provider level (hypervisor) and the other at the derivative cloud provider (VM) level. The native cloud provider is only responsible for partitioning the cache, where as the derivative cloud provider is responsible for distributing the VM memory among the

container. Such isolation makes it difficult for a derivative cloud provider to get a holistic view while provisioning for application objectives.

Application cache sensitivity is unaccounted

With two levels of memory provisioning - In-VM and hypervisor cache makes it challenging to provision applications has some applications have specific needs of In-VM requirements whereas the others are more flexible about it. Caching frameworks fail to address this issue.

1.4 Problem description

Keeping the above drawbacks in mind, the following are the detailed problem descriptions of our work. In the initial phase of our work we wished to

1. Understand the existing memory management policies used by Linux to manage containers.
2. Identify issues with existing policies and purpose new policies to be supported for.
3. Do the same for both native and derivative cloud environments.
4. Design a solution to support purposed policies and empirically evaluate its correctness.

In this phase of our work we wish to

1. Understand existing cache partitioning frameworks.
2. Identify drawbacks with the existing frameworks in achieving application objectives.
3. Come up with solution to fix identified issues and empirically evaluate its correctness.

1.5 Contributions

The following are the contributions of our work.

1. Built a differentiated memory management controller for containers. This work[20] has been accepted in *IEEE ICDCS '17* for the poster track.
2. Designed and implemented a holistic decentralized memory (and cache) management framework for containers. This work is in submission at *ACM Middleware '17*.

Chapter 2

Background

This chapter provides a background to our works that exists. These background are either derived from looking into kernel code or are parts of existing design details.

2.1 Memory management between processes in Linux

Memory is allocated/deallocated in terms of pages in a Linux operating system. Memory management in Linux is done using techniques like virtual memory, demand paging, swapping caching etc. They separate between the memory needed by a process and the memory physically allocated on the RAM. The OS creates a large virtual address space for each process. In this section we focus on how memory is managed between processes or a group of processes. We mainly focus on how memory is assigned and reclaimed between them.

2.1.1 Memory pages used by a process

Memory used by processes are divided into 3 types of pages

1. Anonymous Pages: Pages those which are not associated with any files on disk. They are process memory pages and have a virtual address (VA) associated with it.
2. Page cache pages: Are an in-memory representation of a files on the disks but don't have a VA associated with it.
3. Mapped pages: Are also an in-memory representation of a file on the disk but also have a VA associated with it.

2.1.2 Memory allocation

When the process needs memory to be allocated, Linux decides the how this memory is going to be allocated physically on the RAM. The process/ application does not see in physical RAM addresses. It only sees virtual addresses from the virtual space assigned to each process. The OS uses a page file located on the disk to assist with memory requests in addition to the RAM. Less RAM means more pressure on the Page file. When the OS tries to find a piece of memory that's not in the RAM, it will try to find in the page file, and in this case they call it a page miss. The actual physical memory allocated (RSS) to a process depends on how much free memory is available in the system. On free memory becoming freshly available in the system, the

OS tries to equally distribute the available memory to all processes that are demanding for more memory.

2.1.3 Memory reclamation without container support

When the system memory starts to get tight, the kernel can free memory by cleaning up its own internal data structures - reducing the size of the inode and dentry caches however most pages in the system are user process pages. Hence the kernel, in order to accommodate current demands for user pages, must find some existing pages to toss out. A proper balance between anonymous and page cache pages must be maintained for the system to perform well. Kernel offers a knob called swappiness, that specifies how much favor anonymous versus page cache pages while reclamation. The default value for swappiness favors the eviction of page cache pages.

The system maintains two LRU lists to evict pages in the system commonly referred as *LRU/2 algorithm*, one active list containing all the pages that were recently used and another inactive list which contains all the pages that weren't used recently. One pair (active and inactive) for anonymous pages and one pair for page cache pages. The kernel favors reclamation from page cache pages over anonymous pages and inactive pages over active pages and iterates these lists to satisfy reclamation requests there by trying to maximize application performance while satisfying requests.

2.2 Containers

Container in simple terms can be defined as,

"Container is a process or set of processes grouped together along with its dependent resources into a single logical OS entity. It enables multiple isolated user-space instances on a host machine."

Containers [2] are built as an extension to the existing operating system and not as an independent system. Container provides virtualization of isolated user spaces at an OS-level and hence containers executing on a host machine reuse the functionalities of the host kernel. This makes it better by reducing redundant kernel pages as used in VMs but comes at the cost of containers only of host OS type to execute on a system.

A high level difference between a VM and containers can be seen in Fig 2.1. The biggest advantage of using containers over virtual machines is that they provide much lesser performance overheads. Containers are usually managed by container managers, which are entities similar to how Hypervisors are to VMs. Container managers are shipped by different organizations like Docker [21], LXD [22], OpenVZ [23] etc. All container managers makes use of 3 Linux kernel components and combine them to form the building structure. The deploy their own controllers on top of this.

1. Control Groups: Used for resource accounting and control
2. Namespaces: Resource isolation among resources provisioned to different users on the same system
3. Disk Images: The disk image which provides the ROOTFS for a container to execute. It contains the distribution related packages, libraries, and application programs.

For the purpose of this discussion, we would focus on control Cgroups (Cgroups) as this provides the mechanism to control resources which includes performing memory management.

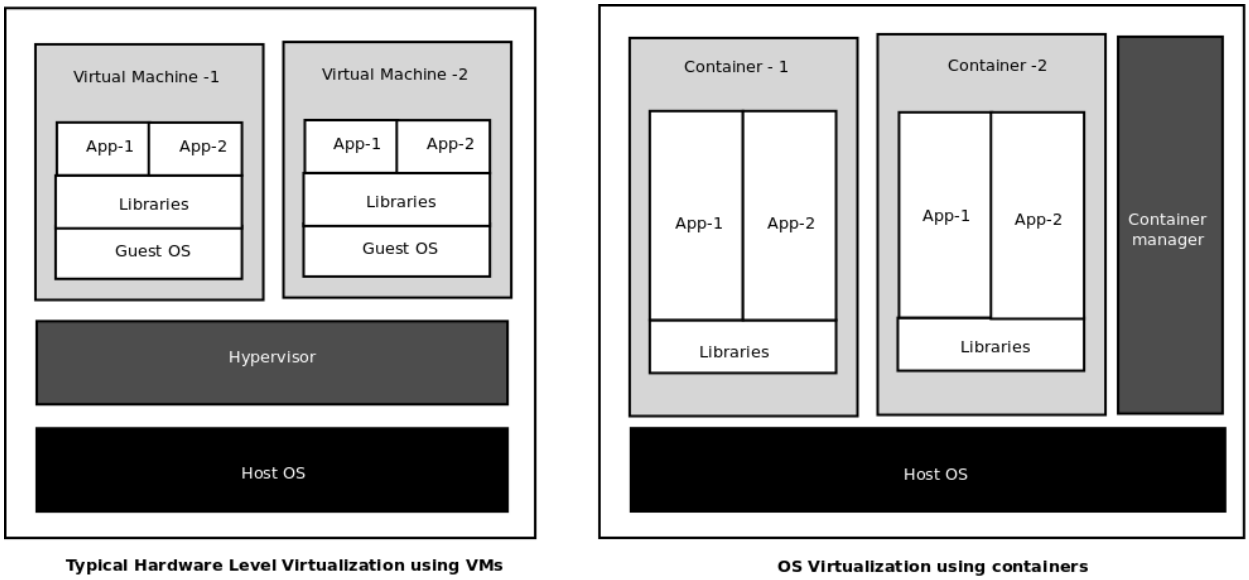


Figure 2.1: Difference between a VM and Container

2.2.1 Control groups

A solution to process group control and accounting was proposed by Google in 2007 which was originally called Generic Process Containers [24] and was later renamed to Control Groups (Cgroups), to avoid confusion with the term Containers. A cgroup/subsystem refers to a resource controller for a certain type of CPU resource. Eg- Memory cgroup, Network cgroup etc. It is derives ideas and extends the process-tracking design used for cpusets system present in the Linux kernel. There are 12 different Cgroups/subsystems, one for each resource type classified.

For the purpose of our discussion we will stick to subsystem as the terminology referring to individual resource control and cgroup to refer a cgroup node in hierarchy. The Linux kernel by default enables most subsystems. The overheads introduced by Cgroups are negligible. Most subsystems follow their own hierarchy for their individual resource. The Linux exposes Pseudo file systems as userspace APIs to interact with them.

Fig 2.2 illustrates a minimalistic outline of a Cgroups hierarchy with 3 subsystems mounted in a system onto their own hierarchies. The three subsystem mounted are - memory, cpuset and blkio and are mounted at `/sys/fs/Cgroups/`. Memory root cgroup of 8GB is divided into two Cgroups M1 and M2 of 4GB each. cpuset root cgroup of 4CPUs is divided into two Cgroups C1 and C2 of 3CPUs and 1CPU respectively. blkio root cgroup of is divided into two Cgroups B1 and B2 of 1000 and 500 as relative weights respectively. Every process which attaches itself to the same set of subsystems are referred by a single `css_set` which in turn points to the cgroup node the process is attached to. In the Fig, processes 1,2 attach itself to the blue `css_set` and 3,4,5 to the red one. The `css_set` in turn has pointers to `container_subsys_state` that is one for each cgroup. Notice how the blue `css_set` points to the root cpuset cgroup there by assigning it all the CPUs in the system which is also a valid and default value to attach processes.

Memory Cgroups

Memory subsystem use a common data structure and support library for tracking usage and imposing limits using the "resource counter". Resource controller is an existing Linux implementation for tracking resource usage. Memory cgroup subsystem allocates three `res_counters`. The three of them are described below.

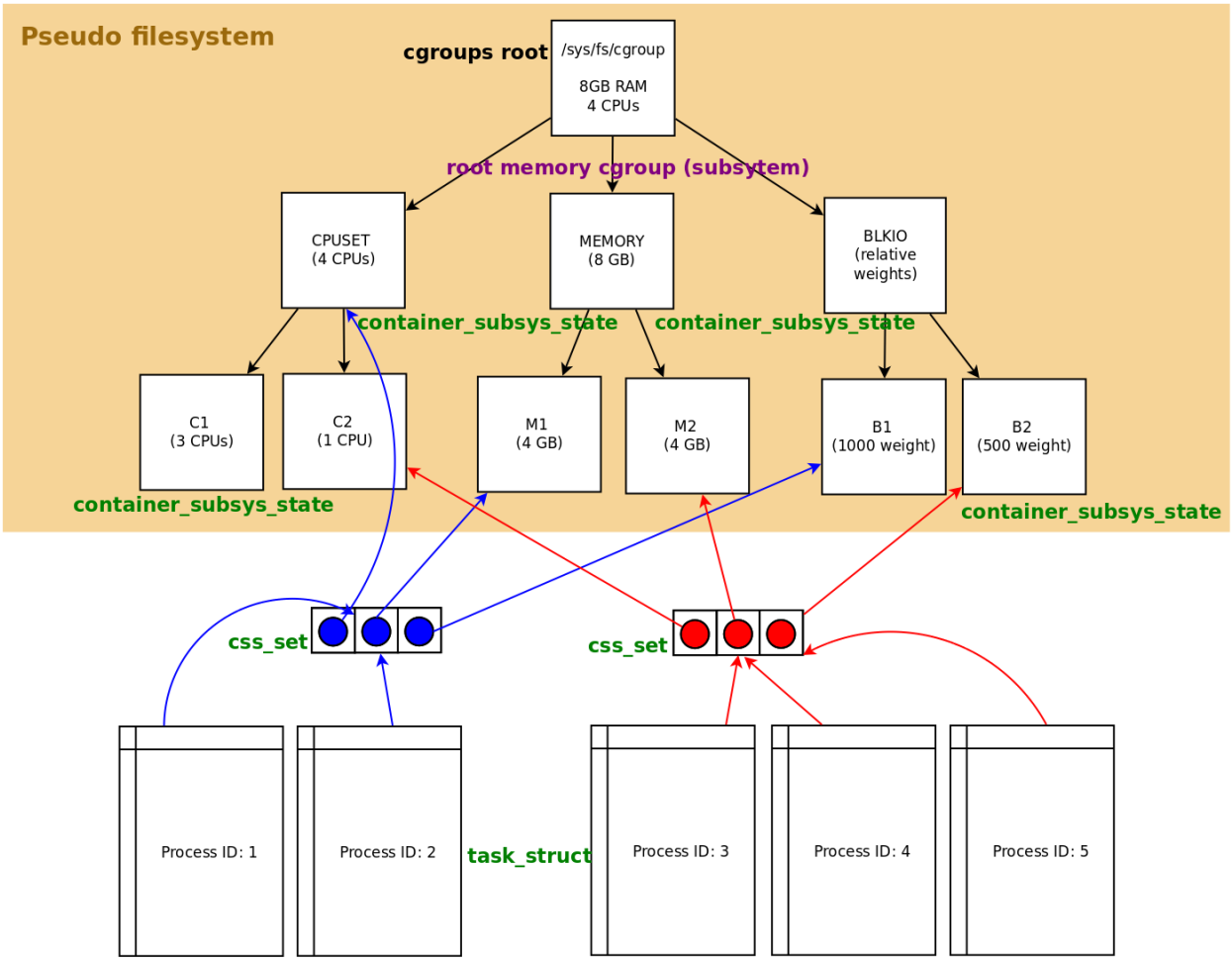


Figure 2.2: Control groups illustration using 3 controllers, Source:[2]

i. Accounting: Accounting memory for each process group. Keeps track of pages used by each group. Pages can be classified into four types.

- Anonymous: Stack, heap etc.
- Active: Recently used pages
- Inactive: Pages read for eviction
- File: Reads/Writes/mmap from block devices

ii. Limits: Limits can be set on each Cgroups. Limits are of two types - soft and hard. Soft limit is the limit up to which the system guarantees availability. Hard limit is the limit up to which the system tries to accommodate, but cannot guaranty this if system is under memory pressure. Limits can be set in terms of byte for,

- Physical memory
- Kernel memory
- Total memory (Physical + Swap)

iii. OOM: Out Of Memory killers are used to kill processes or trigger any other such event on reaching hard limit by a process group.

More about memory management using memory subsystem in the Linux kernel shall be described in the coming section.

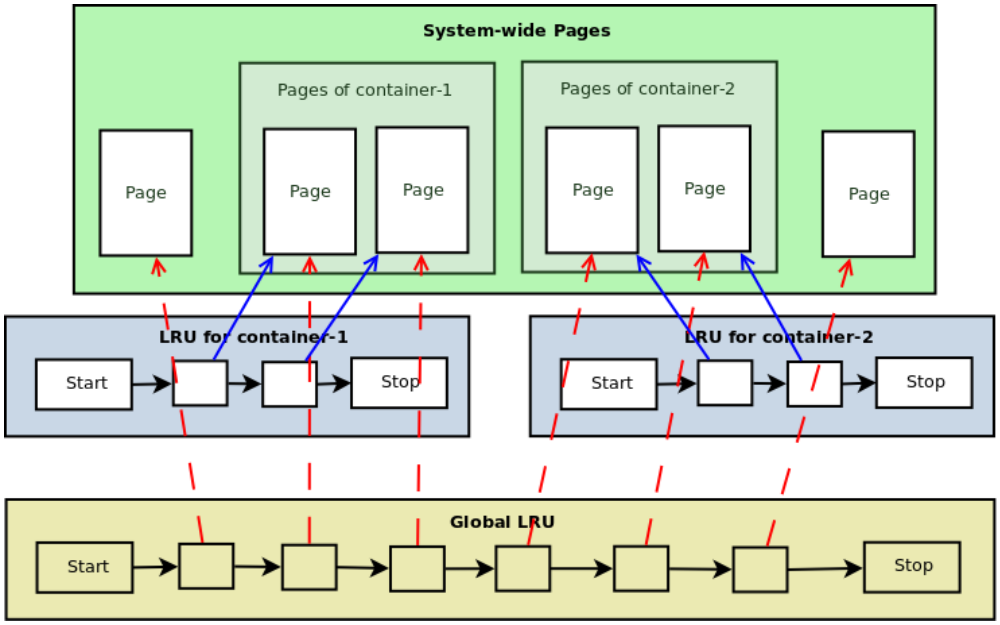


Figure 2.3: Mapping of pages to LRU lists

Memory reclamation with Cgroups Most recent Linux kernels that are even skipped through stable Linux distribution support Linux containers. They provide the following set of knobs to provision containers - Hard limit, Soft Limit, OOM Control, Swappiness to name a few important ones. Hard limits can further be specified in terms of process, kernel and tcp memory utilization. However, with respect to reclamation we focus on soft limits here.

The following section describes the existing policy based on theoretical readings and looking upon Linux kernel code. The current system-wide policy incorporates memory reclamation keeping in mind the memory Cgroups. Reclamation can broadly occur in two situations,

1. **System-Wide (Global) Reclamation:** When the system is under memory pressure when all/most of its pages are occupied
2. **Container Specific (Local) Reclamation:** When only a particular of the container is under pressure due to exceeding its hard limit

For the purposes of our problem, we focus on System-Wide Reclamation. It must be remembered that System-wide reclamation can again broadly occur in two situations,

1. **Synchronous:** When system is under memory pressure due to new page requests and not enough free pages available
2. **Asynchronous:** System clears up memory routinely when free

Both Synchronous and Asynchronous global reclamation ultimately end up taking a similar path for memory reclamation, with little differences when it comes to regions to reclaim from and how much to reclaim. A kernel function call trace for system-wide reclamation is described in Fig 2.4. It shows how sync and async requests are ultimately mapped to the same set of function calls.

As shown in Fig 2.3, in recent Linux kernels a LRU (LRU/2, but LRU used for simplicity) list is stored for every container created also there is a Global LRU list which contains the pages of all processes in the system (including in the ones in a new container). All processes are by default put into the default container and hence its pages are a part of the global LRU. Once a process is moved to a specific container, its pages also become a part of its local per container LRU list also.

Figure 2.4: Kernel Function Call Trace for System-Wide Reclamation

An container (Memory cgroup) when has its soft limit set, has a value called excess computed for it at every container node. The system internally makes use of a RBTree to store the exceeds. The exceed is computed at `mem_cgroup_update_tree()` at regular intervals using the formula,

$$Excess = Usage - SoftLimit \tag{2.1}$$

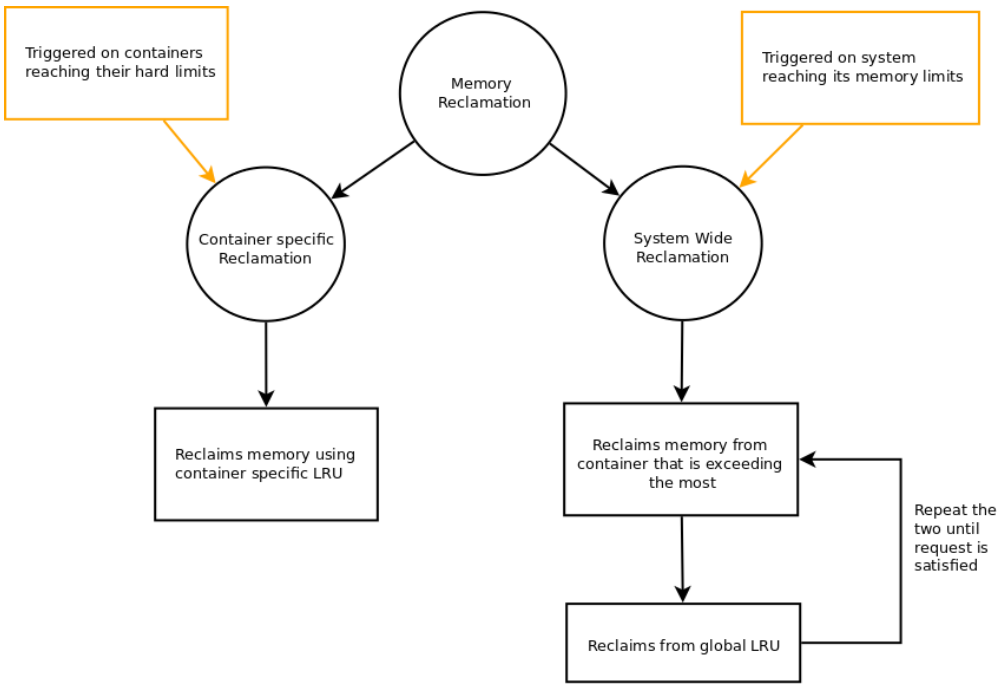


Figure 2.5: Existing policy for Memory Reclamation

Two types of reclamations after the global reclamation policy used currently,

- **Soft Memory Reclamation (SMR):** is the one specific to Cgroups where container which exceeds its soft limits by the most is reclaimed from using the per container based LRU. It internally also reclaims from all its child containers.
- **Global LRU reclamation policy (GLR):** reclaims based on the global LRU list in which all processes in the system are a part of

The existing reclamation policy is illustrated as a flow chart in Fig 2.5. In simple terms the global reclamation policy is a combination of both Soft Memory Reclamation and the native global LRU based reclamation. The reclamation algorithm tries to maximize most of the requested reclamation from SMR and prefers GLR only when requests aren't being satisfied by SMR.

All above container specific memory reclamation patterns were derived based on theoretical readings and looking upon Linux kernel code. We need to establish the correctness our hypotheses, and understand the existing system and how it impacts applications running inside containers in a native container setup and derivative cloud setup. Hence the coming sections describes the empirical analysis to do the same.

2.3 Second chance cache for derivative clouds

DoubleDecker[19] is a second cache caching framework developed to support derivative clouds. It's initial intent was to tackle the problem of configuring hypervisor caches in a derivative cloud environment.

DoubleDecker provides a second chance caching framework for Linux+KVM+LXC. DoubleDecker is built upon existing the existing T-Mem second cache caching framework for Linux. There have been implementations[25] of T-Mem to support a per-VM partitioning schema. DoubleDecker builds upon this framework by providing a per-container partitioning framework.

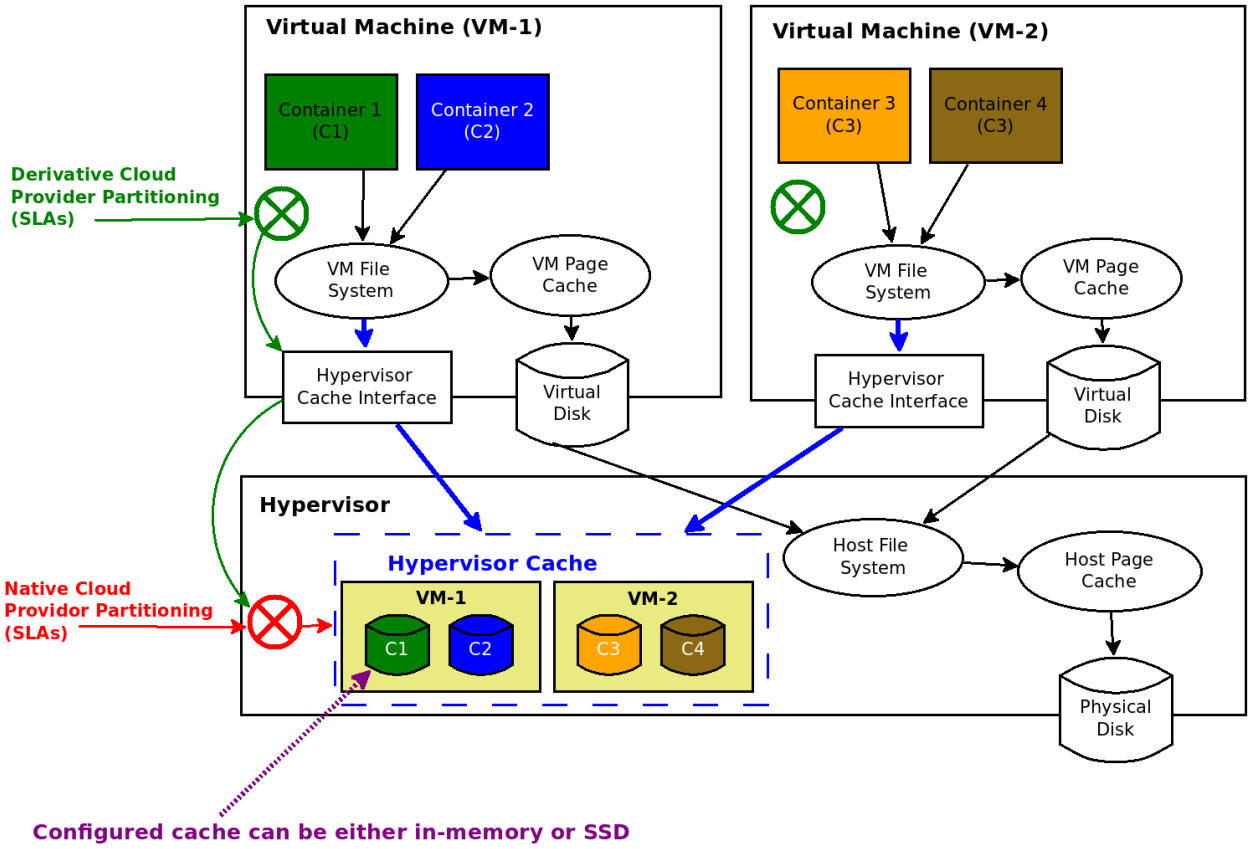


Figure 2.6: DoubleDecker architecture: second cache caching for derivative clouds

The following are a comprehensive list of features supported by DoubleDecker,

1. Supports a **two level configurable caches**. These partitions are either in-memory or at the SSD.
2. A cache partition can be configured on a per-VM basis by the native cloud provider at the hypervisor.
3. The cache partitioned at the VM-level can further be partitioned by an derivative cloud provider inside the VM on a **per-container basis**.
4. Cache supports a **resource conserving nature**, where any cache that is underutilized is provisioned to other applications at need more cache.
5. Exclusive caching at all levels.

Architecture The architecture for DoubleDecker is illustrated in Fig 2.6. The hypervisor caches are initially partitioned on a per-VM basis by the native cloud provider at the hypervisor. Now, each VM cache partition can be further subdivided by the derivative cloud provider on a per container basis. Provisioning is done in-terms of *Relative Weights*. The relative weight for an container would map to the actual cache partition as given below,

$$ContainerCacheAllocation = (\frac{RelativeWeightofContainer}{TotalWeightofallContainers}) * CacheAllocatedforVM \quad (2.2)$$

Similarly is the cache partition for each VM calculated but instead of using the VM cache allocation, we would be using the entire host cache configuration. The cache is partitioned based on a higher level SLA requirements at each of the cloud provider. This SLA could be fixed for a particular container using a higher level policy.

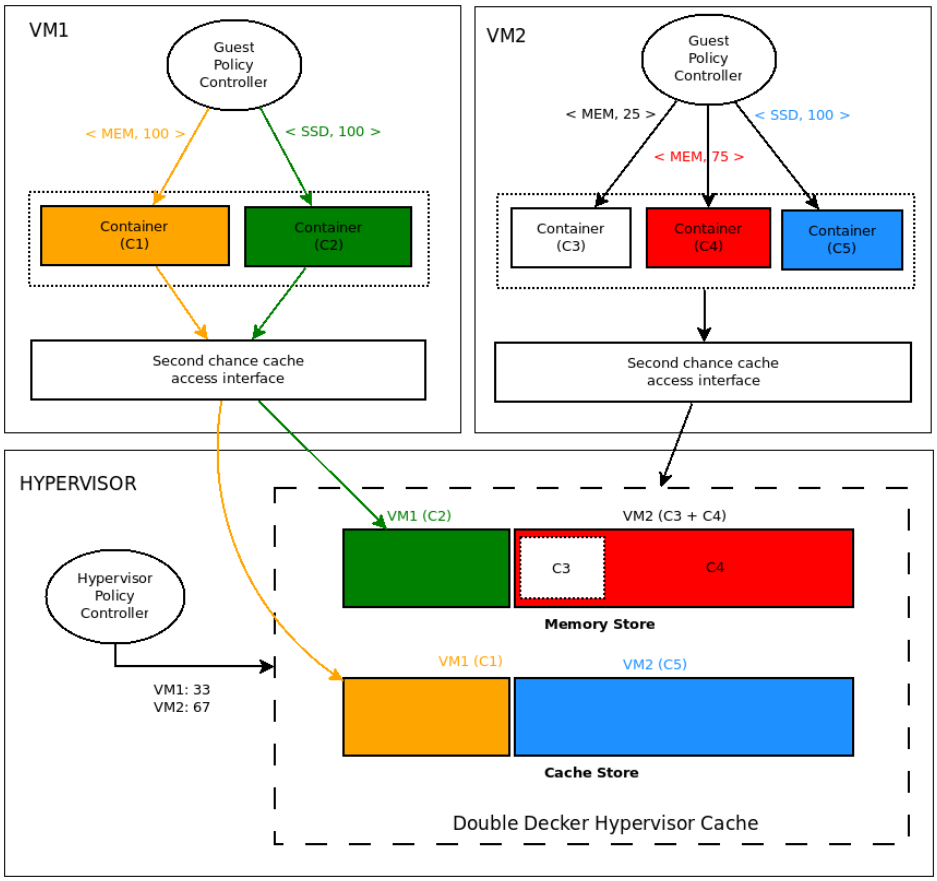


Figure 2.7: DoubleDecker internals

Internals The Fig 2.7 describes the T-Mem implementation overview of double decker. Each container is configured using a relative weight parameter as described in Section 2.3. There is pool created for each configured container cache. This pool represents an abstract design component which contains all the pages for a particular container are present in. There are several data structures which make up for the T-Memimplementation, however the most crucial one is a per pool tree structure that keeps track of all put objects.

DoubleDecker is built upon existing the existing T-Mem second cache caching framework for Linux as mentioned earlier. In specific it only works as a second cache only clean (not dirty) disk backed pages. It doesn't cache anonymous (pages with VA mappings) or dirty disk backed pages. This ensures a simple form of consistency in case of system failure.

Inserts into the cache are referred to as *PUTS* and accesses to these objects are referred to as *GETS*. Cleaning up of pages are referred to as *FLUSHES*. Evictions in this triggered when the cache is full, and the application (VM followed by container) that is most exceeding it's limits are targeted first. Evictions are based on a simple LRU eviction algorithm for now. However any eviction algorithm is pluggable with this implementation.

Chapter 3

Related work

The following is a comprehensive list of related works. Although most of these related works are motivating factors to that drive our work, none of them are similar works.

3.1 Memory management in virtualized environments

There have been several attempts to provide efficient memory management for virtual machines. The most prominent used approach is of that Ballooning [26]. Ballooning is an mechanism that reclaims pages considered least valuable by the guest OS running inside the virtual machine. This allows the VM to decide which memory pages to release instead of the host trying to determine this. German Molto [27] expands the idea of Ballooning to provide a system to monitor the VM memory and apply vertical elasticity rules in order to dynamically change its memory size by using the memory ballooning technique provided the KVM hypervisor. Overdriver [28] on the other hand presents a system that handles all durations of memory overload. It adapts its mitigation strategy to balance the trade offs between migration and cooperative swap to handle memory overcommitments. Ex-Tmem [25] stores clean pages in a two-level buffering hierarchy with locality-aware data placement and replacement. It enables memory-to-memory swapping by using non-volatile memory and eliminates expensive I/O caused by swapping.

3.2 Resource provisioning in virtualized environments

Looking at researches that have looked at resource provisioning, CloudScale [6] can resolve scaling conflicts between applications using migration, and integrates dynamic CPU voltage/frequency scaling to achieve energy savings with minimal effect on application SLOs. Tim Dornemanna[5] proposed a solution that automatically schedules workflow steps to underutilized hosts and provides new hosts using cloud computing infrastructures in peak-load. The system was based on BPEL to support on-demand resource provisioning. Aneka [8], is a platform for developing scalable applications on the Cloud, that supports provisioning resources from different sources and supporting different application models. It support the integration between Desktop Grids and Clouds. Elastic Application Container (EAC) [29] is a virtual resource unit for delivering better resource efficiency and more scalable cloud applications.

3.3 Nested virtualization

A derivative cloud is a nested setup, virtual machines nested in virtual machines[30] or containers deployed in virtual machine [1, 31], the latter being the focus of this work.

3.4 Hypervisor managed caches

There has been extensive literature centered around hypervisor caching [32, 16, 33]. There are several types of hypervisor caching addressed in various works, however the framework we have used to build upon in our implementation is described below.

3.4.1 Transcendent Memory

Transcendent Memory (Tmem)[34] is an approach make proper utilization of underutilized memory present in Linux based system by provisioning an in-memory cache. This was then extended to virtualized setups to be able to build caches for individual VMs. However there have been attempts [25] to make this single level Tmem cache to an hybrid multi-level setup in a non-virtualized by environments. Tmem caches however only cache clean disk pages. They don't cache dirty disk or anonymous pages.

3.4.2 Hypervisor cache partitioning

On top of this there has been work carried out to partition hypervisor caches based on application SLAs[33, 17]. In these works, each application is treated as a VM and provisioned accordingly. Centuar[17] proposes a single level cache partitioned on a per VM bases by satisfying individual application objects or a global hypervisor level policy. SDC[33] follows a similar approach but provides heuristics for even a multi-level cache and how application requirements could be satisfied. Both There approaches make use of MRC construction techniques[35, 36, 37].

3.5 Conclusions

Although there are several existing works in elastic resource provisioning (including memory) for virtual machines as listed above. There has been no attempt to provide an deterministic memory management policy for containers. To our knowledge this is our first attempt to do so.

Applicability of hypervisor caching in a derivative cloud setup is hindered due to inability of existing frameworks to support this sort of provisioning, and also we would like to look at the overall picture of memory management at all levels of the hierarchy along with the cache partitioning to satisfy application SLA.

Chapter 4

Differentiated memory management controller for containers

The existing memory management controller posses a few issues when it comes to managing memory among containers in an over-committed scenario. We initially bring out these issues, and then propose a solution and evaluate our solution with respect to the existing controller.

4.1 Drawbacks of existing memory management for containers

We have used empirical evaluations to verify and demonstrate our hypotheses. These hypotheses bring out the issues existing container memory management controller—Memory Cgroups. We have used both a native cloud testbed and an derivative cloud testbed to establish our issues.

4.1.1 Experimental setup

The following are the experimental setup configurations, metrics and workloads used to establish the correctness of our hypotheses in a native cloud environment.

Experimental configurations

The set of configurations used for an analysis of memory management techniques in a container environment must be relevant, and easy to apply. The following configurations fit this criteria, and have been used for the evaluation.

- **Number of containers:** The number of containers that are currently executing in the system.
- **Memory soft limit of container:** The minimum promised memory to a given container by the system on which the container is executing.
- **Memory hard limit of container:** The maximum memory that can be assigned to a container by the system on which the container is executing.
- **Memory usage of each container:** The usage of a container at a given point in time, that is generated by the workload executing inside the container.

- **Workload:** The workload that is running inside each of the container. Workloads can vary based on the type of operation they perform, the ratio of anonymous memory pages they consume to that of page cache pages.
- **External memory pressure:** The memory pressure that is generated in the system in order to reduce the free memory available in the system and trigger memory reclamation. This pressure could be either generated by a process on the same system / driver that is running in the host system.
- **Size of machine:** Size of Machine refers to the maximum memory available in the system inside which all the containers are executing.

Metrics of interest

The following are the metrics of interest to us that would help us analyze the experiments.

1. **Memory assigned to each container:** Total memory assigned to a container at any given instant
2. **Soft memory reclaimed for each container:** Memory reclaimed from each container using SMR
3. **Total memory reclaimed for each container:** Total Memory reclaimed from container (SMR + GLR)
4. **Memory reclaimed using GLR:** Memory reclaimed from all containers and other processes running on system using GLR
5. **Memory reassigned for each container:** Memory reassigned to each container on freeing up of memory
6. **Application specific metrics:** Application metrics of the workload running inside containers like throughput, total time taken etc.

Workloads

This section presents the list of workloads that we have used as primary candidates to evaluate our empirical evaluations. All workloads are chosen keeping in mind the memory intensive nature of the requirement.

These are the list of Synthetic workloads we have used to establish our problem.

Stress

Stress [38] is a deliberately simple workload generator for POSIX systems. It imposes a configurable amount of CPU, memory, I/O, and disk stress on the system. It is written in C and has been developed by people at Harvard university.

Memory hogger

Memory Hogger is a simple C program that allocates an array of specified memory using a simple `malloc()` and repeatedly writes to these array locations. This only consumes anonymous memory pages.

File hogger

File Hogger is a simple python program that creates a file with specified size and repeatedly updates it line by line there by consuming both anonymous pages and file backed pages.

These are the list of real workloads we have used to show how the existing problems affect real work applications.

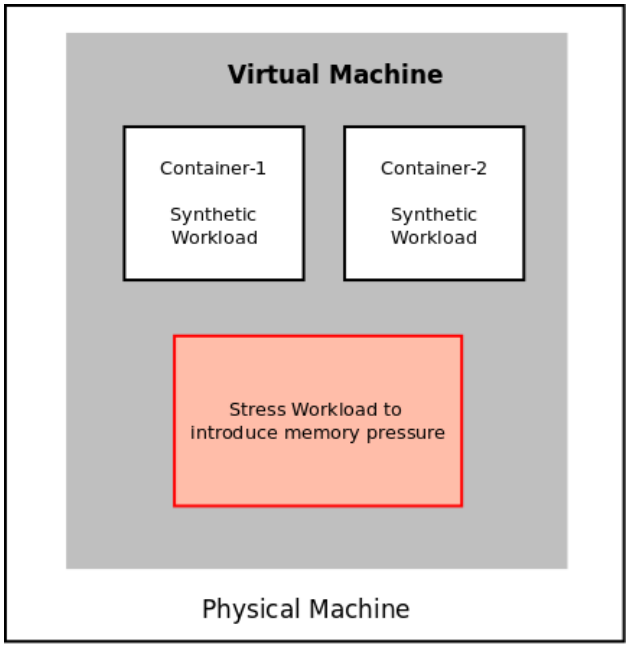


Figure 4.1: Native container testbed

MongoDB
MongoDB [39] is an open-source, document database designed for ease of development and scaling. Classified as a NoSQL database program, MongoDB avoids the traditional table-based relational database structure in favor of JSON-like documents with dynamic schema. It follows a memory hungry approach where it tries to use up most of system and it actually leaves it up to the OS's VMM to tell it to release the memory.

Redis
Redis [14] is a in-memory data structure store, used as database, cache and message broker. It is used to store a large number of in-memory key-value pairs. Its in-memory nature makes it a prime candidate to use it as a workload in our empirical evaluations.

YCSB benchmark
We use YCSB [40] (Yahoo Cloud Server Benchmark) project as the benchmark to generate the clients evaluate to the performance of our real workloads i.e MongoDB and Redis servers. The goal of YSCB is to develop performance comparisons of the new generation of cloud data serving systems. It is a framework and common set of workloads for evaluating the performance of different key-value stores.

4.1.2 Issues in native environment

We have tried to show the issues in container memory management using empirical analysis. We have taken our hypothesis/questions, and mapped them to various experiments to illustrate the issues. Inferences were drawn based on the observations in the experiments.

The native testbed consisted of running containers inside a host machine (running inside VM in our case) in complete isolation from the external environment as shown in Fig 4.5. This setup which involved a native container testbed, was used to understand the existing memory reclamations and establish the problem in a native system using **synthetic workloads**.

	Container-1 (M1)	Container-2 (M2)
Size of VM	2 GB	
Workload	Memory Hogger	Memory Hogger
Hard Limit	1000 MB	1000 MB
Soft Limit	150 MB	150 MB
Memory Usage	500 MB	500 MB
Exceed	350 MB	350 MB
External Pressure	200 - 400 - 600 - 800 - 1000 MB	

Table 4.1: Base configuration for native container experimentation

Host

1. Intel Core i5-4430 processor @ 3.00GHz
2. 4 cores of CPU (with hyper threading support)
3. 1 TB of hard disk space
4. 8 GB RAM
5. Ubuntu 14.04 LTS desktop, 64 bit
6. Kernel version 4.5
7. KVM Hypervisor

Guest

1. 3 cores of CPU (with hyper threading support)
2. 20 GB of virtual disk space
3. 2-6 GB RAM (based on experimental configuration)
4. Ubuntu 16.04 LTS desktop, 64 bit
5. Kernel version 4.7
6. Container technology: Docker

Memory Hogger and File Hogger were used to generate the memory pressure inside the containers. External pressure was generated using Stress workload running directly on the host machine.

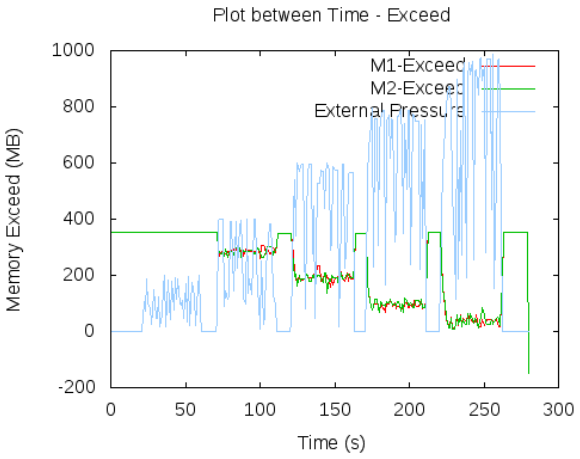
The configuration in Table 4.1 is the base configuration for all experiments in this section. Any changes the base configuration has been mentioned in the procedure of each of the experiment.

Most experiments involved setting up of 2 containers. Workloads were used to introduce system memory pressure from containers. At this point there was no memory pressure in the system (free memory was still available). Now the external pressure using Stress was introduced after about 20s which created memory pressure in the system that triggered reclamation. The external pressure kept on increasing by 200 MB in intervals of 40s. Each interval had a gap of 10s for memory to be reassigned to containers.

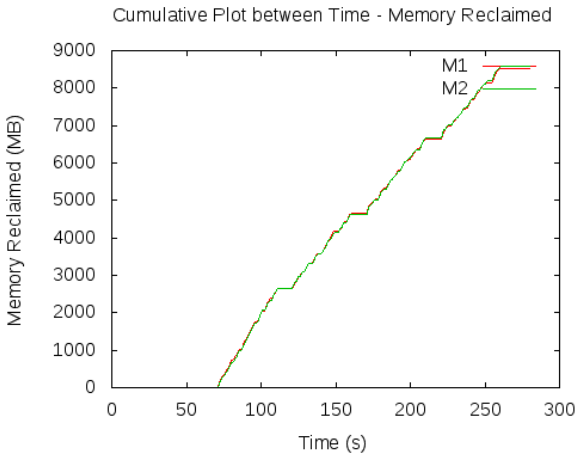
Reclamation above soft limits

Hypothesis
Hypothesis to be verified,

1. Majority of reclamation when containers exceed occurs using SMR (Soft Memory Reclamation)



(a) Memory exceed plot



(b) Cumulative soft memory reclaimed plot

(c) SMR versus GLR

Figure 4.2: Plots for analysis of reclamation when both containers are exceeding by same value

2. SMR purely based on exceed value of the container
3. Containers that exceed equally are iteratively targeted

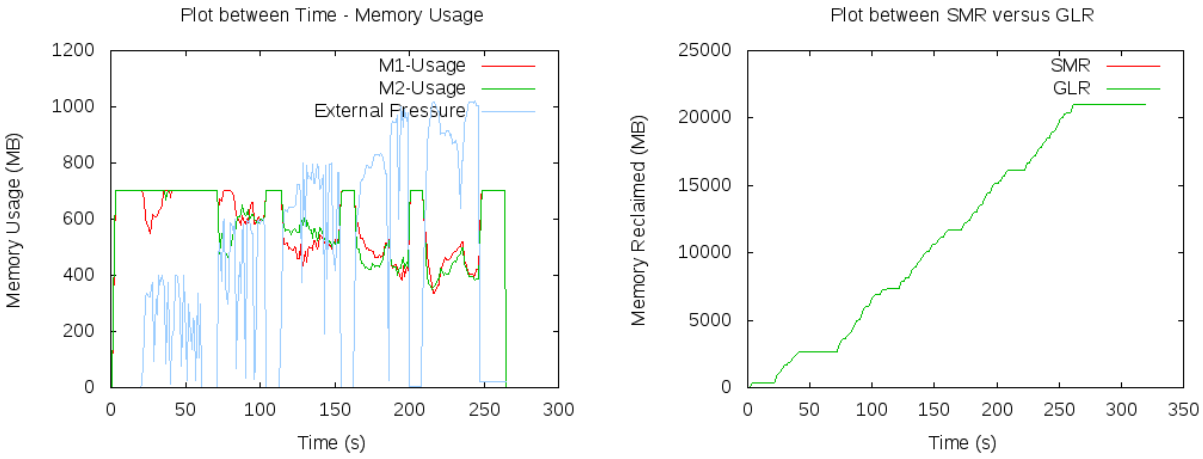
Procedure

To demonstrate the correctness of our hypothesis we the base configuration described in Table 4.1 and change the usage to 700 MB and soft limit to 350 MB there by simulating an scenario (Exp-1) where **Both containers exceeded by the same values**.

Observations

The following are the observations,

- As seen from Fig 4.2a, Fig 4.2b - memory reclaimed from containers iteratively from one after the other as their exceeds are same.
- Fig 4.2c shows how most reclamation when containers exceed occurs using SMR however it is seen that there is minimum reclamation occurring using GLR as well.



(a) Memory Usage Plot

(b) SMR versus GLR Plot

Figure 4.3: Plots for when both containers are having same usage but no exceeds

Inference
The following are the inferences,

- SMR is purely based on exceed value.
- Most reclamation when containers exceed occurs using SMR, however the GLR kicks in every reclamation request to evict any inactive page cache pages in the system (may/may not belong to container).
- Containers that exceed equally are iteratively target for reclamation one after the other.

Reclamation below soft limits

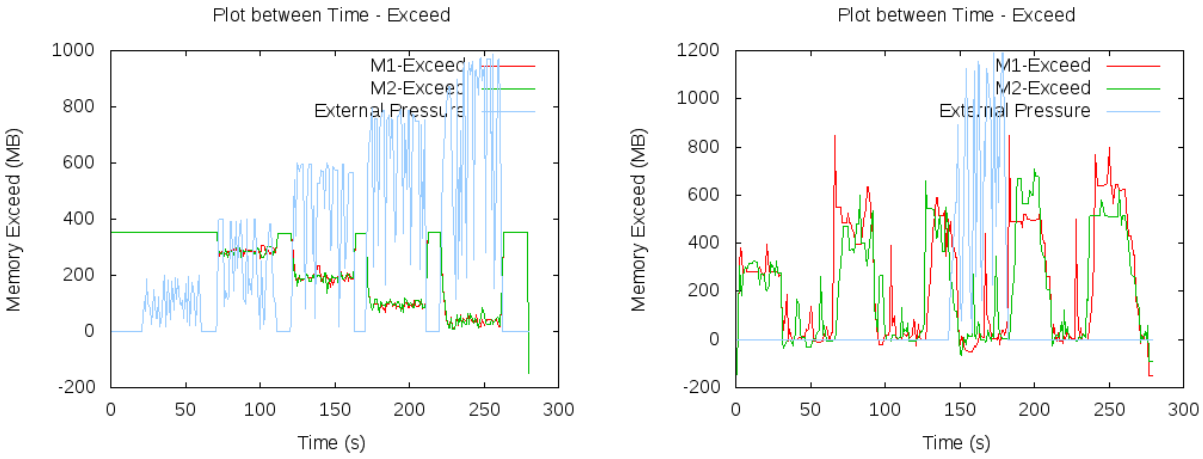
Hypothesis
Does our hypotheses of reclamation below soft limits falling back to native system reclamation hold good ?

Procedure
To test the reclamation patterns in containers below soft limits, we created containers as mentioned in Table 4.1 and changed soft limits (Exp-2) of both containers to 1000 MB there by making the current **usage of both containers below soft limits**. We used hooks in the kernel code to track requests satisfied by soft memory reclamation (SMR) and global LRU based reclamation (GLR).

Observations
The following are the observations.

- As seen from Fig 4.3a, there is no hand-in-hand reclamation that occurs to containers below their soft limits although the containers are running the same workload, unlike hand in hand reclamation that occurs in memory usage above soft limits.
- Since both containers are below SL, all reclamation is occurring using the GLR (Global LRU based reclamation) as seen by Fig 4.3b

Inferences
The following are the inferences.



(a) Exceed plot for memory hogger

(b) Exceed plot for file hogger

Figure 4.4: Plots for analyzing effect of workloads characteristics on reclamation

- Containers with memory usage below soft limits reclamation falls back to native system GLR.
- Reclamation using GLR is haphazard and there is no control over it.

Effect of workloads characteristics on reclamation

Question

Questions of our interest,

1. Effect of workload characteristics on reclamation
2. How much of memory is reclaimed from a container in a single reclamation SMR request ?

Procedure

We took our base configuration as described in Table 4.1. However we ran two workloads in this case - Memory Hogger (Exp-4a) and File Hogger (Exp-4b) workloads on it as native theory suggests that containers with page cache pages might be victimized at larger the way it occurs with GLR.

Observations

The following were the observations,

- The exceed goes hand in hand as expected but with larger deviation in Fig 4.4a and Fig 4.4b
- The larger deviation can be accounted to larger reclamation chunks in workloads that have page cache pages similar to how reclamation targets page cache pages in native system
- Further empirical analysis of the reclamation chucks gave us the reclamation chucks to be

$$\text{Reclamation chuck} = \text{Anonymous memory pages } (<25\text{MB}) + \text{Page cache pages}$$

In both cases pages from inactive zones were reclaimed before trying to reclaim from active lists.

Inference

Workloads with page cache pages are reclaimed at larger chunks per SMR request

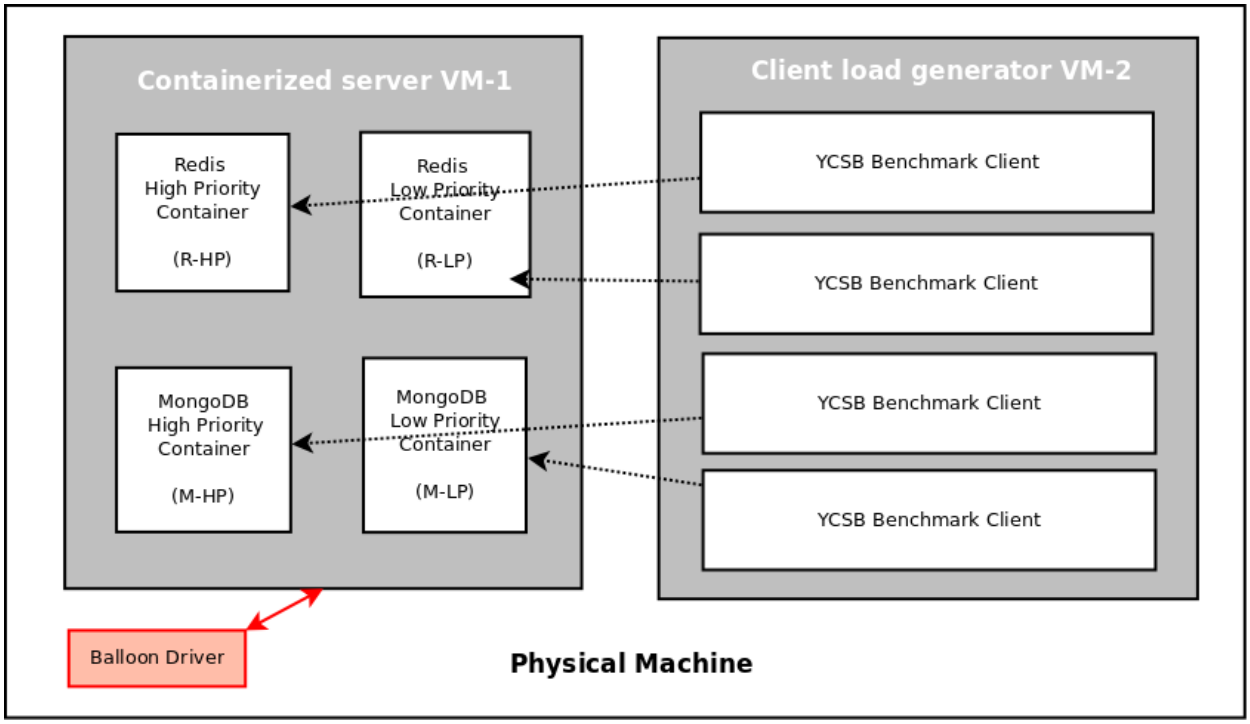


Figure 4.5: Derivative cloud testbed

Key Implications

Here are the list of key implications that were derivative from running the above experiments in an synthetic environment. We have classified it based on the scenarios as discussed earlier.

1. When containers usage are above soft limits most reclamation occurs using SMR, however the GLR kicks in every reclamation request to evict any inactive page cache pages in the system (may/may not belong to container).
2. SMR is purely based on exceed value of a container.
3. Workloads with page cache pages are reclaimed at larger chunks per SMR request
4. Containers with memory usage below soft limits reclamation falls back to native system GLR.
5. Reclamation using GLR is haphazard and there is no control over it.

4.1.3 Amplification of issue in derivative clouds

The following experiment tries to establish the implications of previously established inferences, as to how these affect applications running on a derived cloud environment.

Testbed

The derivative cloud testbed consisted of running server containers inside a virtual machine (VM-1) which was running on top of a physical host machine. Another virtual machine (VM-2) was used to generate clients who connected to servers containers running inside VM-1 as shown in Fig 4.5. This setup was used to understand the impact of existing memory reclamation patterns on real workloads running on a derivative cloud setting.

Host

1. Intel Xeon E5507 @ 2.27GHz
2. 8 cores of CPU (with hyper-threading support)
3. 125 GB of attached storage, Unlimited NFS attached storage
4. 24 GB RAM
5. Ubuntu 14.04 LTS server, 64 bit
6. Kernel version 3.13
7. KVM Hypervisor with memory ballooning enabled
8. Guest machines were connected using a software bridge

Guest
The two VMs used in this setup are described here.

VM-1: Running server containers

1. 6 cores of pinned CPUs (with hyper threading support)
2. 175 GB of virtual disk space (Storage was provisioned using NFS)
3. 16 GB RAM
4. Ubuntu 16.04 LTS desktop, 64 bit
5. Kernel version 4.7
6. Container technology: Docker
7. Containers inside guest were multiplexed using NAT forwarding

VM-2: Running clients that connect to server containers

1. 1 core of pinned CPU (with hyper threading support)
2. 20 GB of virtual disk space (Storage was provisioned using NFS)
3. 6 GB RAM
4. Ubuntu 16.04 LTS desktop, 64 bit
5. Kernel version 4.7

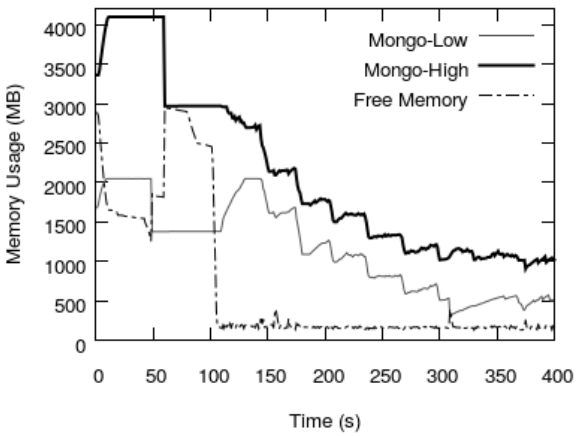
Redis and MongoDB was used to generate the memory pressure inside the containers. External pressure was generated by varying guest balloon size triggered from the host.

Experimental Flow

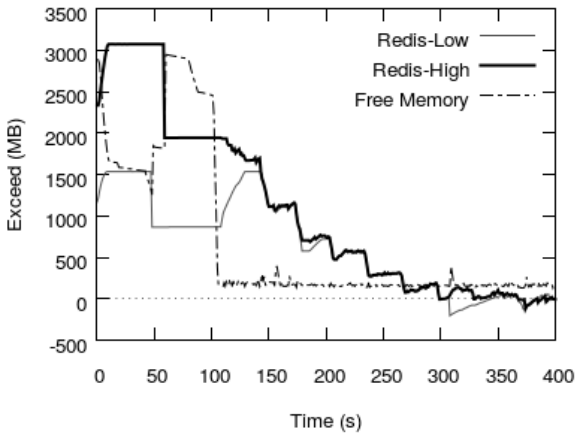
To establish the limitations of nesting-agnostic memory reclamation we performed the following experiments. Two Linux+KVM virtual machines were used, one in a nesting setup, which hosted containers, the second generated workloads for the container-hosted applications. The nested VM was provisioning with 6 vCPUs and 16 GB memory and the workload generating VM has allocated 1 vcPU and 6 GB memory. The nesting setup consisted of four Docker containers [21] which executed with Redis [14] and MongoDB [39] workloads. Default configurations of the applications executed within the contains is as shown in Table 4.2. The YSCB [40] workbench was used to generate workload datasets and as a workload generator. For the first 100 seconds the applications were executed without memory pressure to consume as much memory as required. Beyond 100 seconds, memory pressure was generated from the host and memory from the nested VM reclaimed at rate of 2 GB every 30 seconds.

Container	hard-limit (GB)	soft-limit (GB)	# of records	Usage (GB)
Redis-Low	2	0.5	500K	1.3
Redis-High	4	1	1000K	2.6
Mongo-Low	2	0.5	500K	1.3
Mongo-High	4	1	1000K	2.6

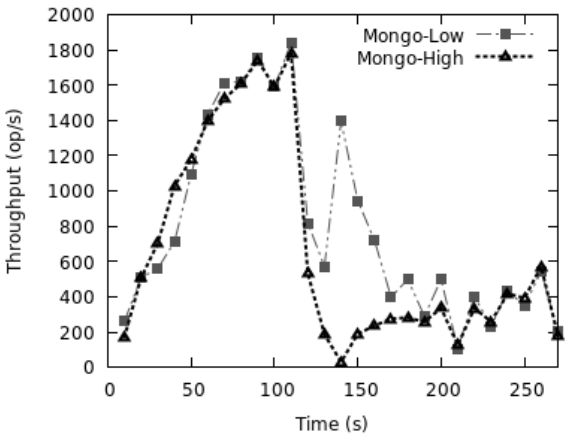
Table 4.2: Default configuration of the four application containers running in the derivative cloud setup



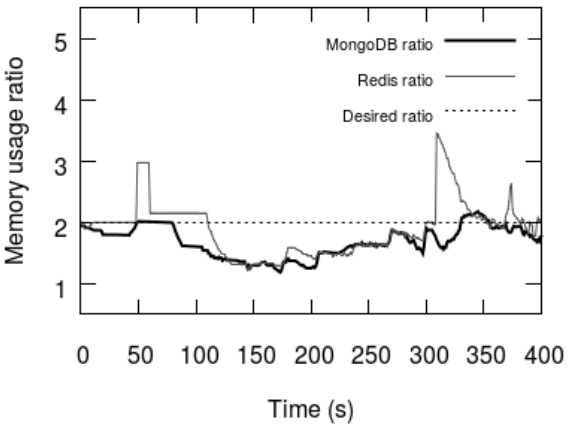
(a) Memory usage for Redis applications.



(b) Extent of memory usage exceed for Redis applications.



(c) Impact of improper memory allocations on MongoDB applications.



(d) Memory allocation ratios.

Figure 4.6: Plots illustrating limitations in maintaining memory ratios in derivative clouds

Impact in derivative environment

Figure 4.6a shows the memory consumed by the containers executing the two Redis workloads and free memory available in the virtual machine. The two containers are configured to consume memory in the ratio of 1:2 (refer to Table 4.2), via hard-limit and soft-limit specifications. At 50 seconds, both the Redis workloads, which are already at their hard-limit, try to allocate more memory, this results in container-specific memory reclamation—the free memory in the system increases and the memory usage of the two containers drops to 1.35 GB and 2.7 GB. At 100 seconds, the hypervisor-based balloon controllers starting exerting memory pressure—demanding 2 GB every 30 seconds. As shown in Figure 4.6a. just after 100 seconds, the free memory in the system drastically drops. The extent of memory *exceed* beyond the soft-limit specifications is shown in Figure 4.6b. Since, the Redis-high application exceeds by a large extent (2

GB more that soft-limit), the cg subsystem penalizes this container for reclamation. Further, due to the free memory available, and reduced memory consumption of the Redis-high workload, the usage for the Redis-low application increases for an epoch and just before 150 seconds, the extent of exceed beyond the soft-limit specifications for both the containers is similar. *Note that this is how the cg subsystem operates, it penalizes the container with the maximum extent of exceed, till all of them exceed by the same absolute value and then penalizes them equally.* This process essentially does not adhere to the memory usage ratios that were expected with the soft-limit and hard-limit specifications. Figure 4.6d shows the memory usage of the Redis and MongoDB applications, which drops below the 1:2 ratio from 100 seconds to 300 seconds. At 300 seconds, the extent of exceed decreases to zero and the system switches to the global reclamation (GLR) mode. Beyond this point, the memory reclaimed is not container-agnostic and variable ratios of usages are observed.

4.1.4 Key Implications

The key takeaways from these experiment are as follows,

- Memory reclamation uses the extent of exceeded usage above a containers soft-limit. Containers with higher exceed values are penalized to a larger extent.
- The Linux cg systems attempts to maintain similar extent of exceed for memory usage during reclamation when all container are above their soft-limit's.
- A per-container soft-limit and hard-limit specification does not guarantee proportionate memory allocation during memory pressure situations.

This Linux Cgroups reclamation policy does not accommodate different user-specified policies that are desired—proportionate memory usages at all instances, order-of-reclamation across containers etc. This is especially required in nested hosting environments where derivative service provides would be benefited by providing a rich set of prioratization features.

4.2 Requirements for a new memory management controller

We wish to build an updated memory management controller that is controller aware, and is able to enforce a differential management policy by a native or derivative cloud provider. The following are the list of requirements of policies that we would like to enforce using this controller,

1. **Prioritized memory allocation:** Currently the notion of priority doesn't exist in container specific memory allocation although the notion of priority exists in other resources. The existing knobs fail to enforce priority used to manage memory in containers.
2. **Deterministic provisioning:** The policy to be designed must eliminate existing non determinism that exists while managing memory between containers in existing system.
3. **Elastic provisioning:** Memory allocated must be re-sizable as and when required.
4. **Adaptive:** On changing resources provisioned to the system as in the case of an derivative environment, the policy enforced must still do it's best in maintaining promised QOS.
5. **Differentiated memory reclamation:** The policy could build around the notion of differentiated memory reclamation when the system falls under memory pressure.
6. **Strict enforcement of limits:** The notion of hard and soft limits that exist must be strengthened.

We would like to design a new controller keeping the above requirements in mind.

4.3 Proposed memory management controller

Enforcing memory allocation proportions across containers when no memory pressure exists is possible with the soft-limit and hard-limit configuration parameters. However, as discussed in Section 4.1, these knobs do not provide deterministic memory provisioning when the system is under memory pressure. As part of this work, we design for two new policies to provide deterministic memory provisioning in nested setups under memory pressure

4.3.1 Controller logic

When the VM in which the containers are executing comes under memory pressure, memory is reclaimed using the container specific LRU lists (as SMR) as well as the Global LRU list (as GLR) by the guest operating system. Memory reclamation depends on the extent of *exceed* in memory usage above the specified soft-limit—estimated as the difference in the two values. The container having a higher *exceed* is victimized during memory pressure first. We redefine this notion of *exceed* with one that is based on proportionality weight of each container. The proportional allocation of a container is the ratio of the container weight to the total weight across all containers in a VM multiplied by the total memory usage across all containers (Equation 4.3). The proportionate *exceed* value is the difference of the container's proportional memory allocation and its current usage (Equation 4.4). Finally, the memory reclamation policy is modified to target the container having the highest proportionate *exceed* value for each reclamation request. Similar to the default reclamation policy, the global LRU list for reclamation is used once memory usage of all containers is below soft-limit specifications.

$$T\_U = \sum_{i=1}^{n} U_i \tag{4.1}$$

$$T\_W = \sum_{i=1}^{n} W_i \tag{4.2}$$

$$PA_i = T\_U \times (\frac{W_i}{T\_W}) \tag{4.3}$$

$$EX_i = U_i - PA_i \tag{4.4}$$

U_i: Memory usage of i^{th} container
W_i: Relative weight of i^{th} container
T_U: Total memory usage by all containers
T_W: Summation of relative weights of all containers
PA_i: Proportional memory allocation of i^{th} container
EX_i: Proportionate exceed value of i^{th} container

4.3.2 Policies supported by our controller

The following are the policies enforceable by our controller.

Policy 1: Proportionate memory allocation

This policy aims to ensure that all memory allocated to container is based on relative weights specified as configuration parameters. The proportional memory allocation is enforced during situations of memory pressure and during no pressure. For example, consider two containers with an intended proportionate memory allocation in the ratio 1:2. Further, assume that their soft-limit values are set to 1 GB and 2 GB, respectively, and their current usage is 2 GB and 4 GB respectively. With a memory reclamation demand of 1 GB, most of the memory will be reclaimed from the container with the larger extent of usage, since the extent of exceed is 2 GB as compared to 1 GB of the smaller container. The resulting usages after reclamation will be 2 GB and 3 GB, respectively, violating the proportionate ratio of 1:2. The proposed proportionate allocation policy aims to maintain the 1:2 ratio in all memory pressure situations.

Policy 2: Application-specific differentiated allocation

This policy provides the flexibility of providing specific rules and or categories to application containers, e.g. gold, silver, bronze etc. Containers mapped to each of these categories have different reclamation rules or the categories themselves can imply an ordering for reclamation.

4.4 Modifications made to Linux memory Cgroup

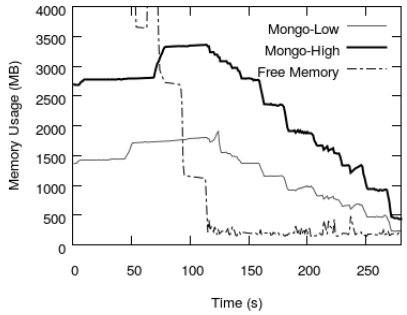
We modify memory Linux Cgroups memory management subsystem to conform to the polices specified earlier. We have made three major changes to this subsystem as described below.

4.4.1 Per container configurable weights

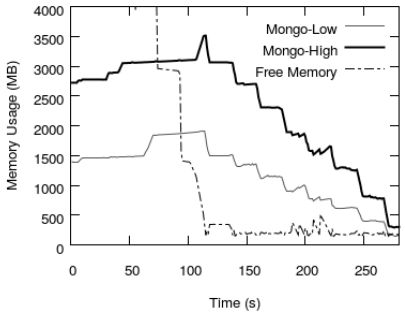
A new per-cgroup state variable is introduced to specify a *weight* parameter for each container and is also exposed through `sysfs` interface for every container. These weights enforce a notion of priority among containers; higher the weight for a container the higher its priority and proportion. The relative ratio of the weights, dictate the proportion of memory allocation. Cloud providers can modify these weights in the derivative setup (also applicable to native cloud providers using containers for provisioning) to enforce priority among containers executing within VMs.

4.4.2 Flexible reclamation size

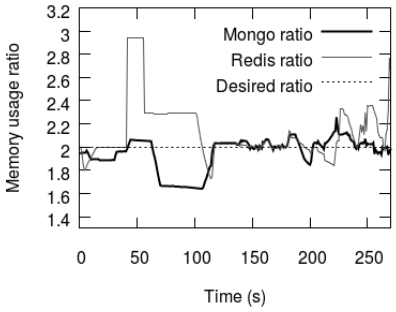
The current soft-memory reclamation policy (SMR), targets one container and reclaims a non-deterministic amount of memory from it for every request. The extent of reclamation depends on the anonymous region (from which a fixed size is reclaimed) and the disk page cache (from which a large portion is reclaimed). Since the page cache size is non-deterministic, reclamation extent can also be non-deterministic. This may lead to a container being targeted exclusively for a particular reclamation request, leading to large deviations from weighted allocations. To overcome this issue, we have capped reclamation chunk size to a maximum of 50 MB.



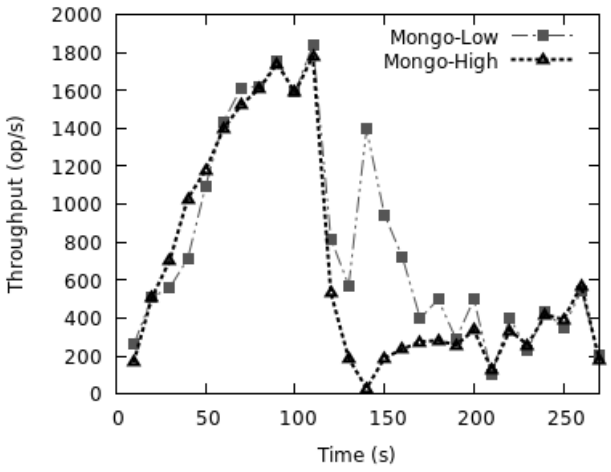
(a) Memory usage by Mongodb containers when SL=HL



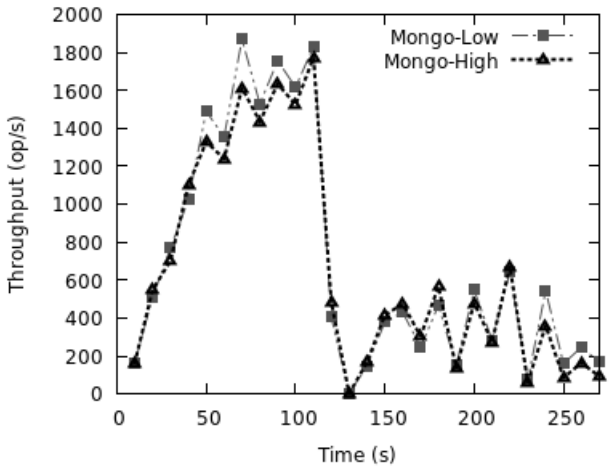
(b) Memory usage by Mongodb containers when SL!=HL



(c) Memory usage ratios between MongoDB and Redis containers



(d) Application throughputs with existing memory controller



(e) Application throughputs with modified memory controller

Figure 4.7: Impact of modified controller on low and high priority containers

4.4.3 Deterministic reclamation

As part of the Linux Cgroups memory reclamation process, soft-reclamation (SMR) reclaims memory from each container till memory usage is above the soft-limitspecification. Simultaneously, a small amount of memory is also reclaimed from the system-wide pool of pages (GLR). In our solution, to maximize deterministic reclamation we perform proportionate reclamation across containers till the usage of containers reduces to zero. Beyond this situation, the system-wide global LRU list is used for further reclamations. Since the SMR policy is container-aware, this strategy provides us a better handle over memory provisioning for containers.

4.5 Empirical evaluation of our controller

We show the effectiveness of our approach empirically in this section. Experimental set-up for the most part remains same as that mentioned in Section 4.1.3. Any changes to the set-up is mentioned in the respective subsections.

Experiments are performed with the same workload and set-up except for the size of VM-1. At the beginning of the experiment, size of VM-1 is 20 GB which is gradually reduced to 3.5 GB, at the rate of 1.5 GB every 20s (started at 50s). Relative memory weights for the MongoDB containers are in the ratio of 1:2 (1 for low and 2 for high priority containers) as are the relative weights for the Redis containers. These weights are added using the newly added *weight* parameter in the memory *Cgroup*.

4.5.1 Effectiveness of our controller

We have tested the effectiveness of our approach in both cases: (i) Soft Limit is equal to Hard Limit (SL=HL) and (ii) Soft Limit is not equal to Hard Limit (SL!=HL), because memory reclamation methods for both cases are different as mentioned in Section 4.1.2.

Figure 4.7a and Figure 4.7b depicts the memory usage patterns for MongoDB containers for cases (i) and (ii) respectively. We observe a gradual decrease in memory allocated to the low and high priority containers in both cases once the system runs out of free memory at about 120 seconds. The ratios of memory allocated to containers can be seen to be in accordance with their relative weights, unlike the earlier case when there is no modified controller. We observed similar pattern in case of Redis container. This pattern of relative memory allocations can be observed more prominently when comparing Figure 4.6d and Figure 4.7c which shows memory usage ratios between low and high priority containers (closer to 2, the better) using existing controller and our modified controller respectively.

The effect of better memory allocation is notable in the application throughputs. The existing controller produces better throughputs for the lower priority MongoDB containers in the interval 100-200s as seen in Figure 4.7d. With our modified controller, both the low priority and high priority containers produce similar throughput as seen in Figure 4.7e. The similar throughputs observed are strongly connected to the fact that low and high priority containers are provisioned with 1:2 memory allocations and so are the ratio between their usage patterns (Provisioning is done to avoid contention due to any other resource). Similar throughput impacts are also observed with Redis containers. Hence credit share not only produces better memory allocations, but also enhances application performance for containers.

4.5.2 Differential QOS containers

Consider the application specific differentiated allocation policy specified in Section 4.3.2 where we would like to prioritize a container from being victimized. Let's call this container as a gold container. We have used same set-up (SL!=HL) to test this. We have prioritized the gold container using the higher weights (Weights of the order of 10^3 relatively higher) to avoid reclamation from it unless it's necessary. There are also other containers running inside the VM with lower QOS and let's call them silver containers.

In Figure 4.8 we have plotted only the memory usage for the gold and one silver container for ease of analysis. We observe that at about 120 seconds free memory drops and the reclamation kicks in. Victimization of gold container is much lesser than other containers until the other containers usage near zero. This shows how we could use our weighted controller to enforce differential service QOS containers.

4.5.3 Impact of reclamation chunk size

The native reclamation method has no notion of limit on the amount of memory that can be reclaimed from a targeted container. This leads to larger reclamation amounts from a container thereby deviating it from the desired relative allocations.

Figure 4.9 shows the ratio of memory usage between MongoDB containers for default (native) case and reclamation chunk size of 50 MB. We consider observation after 120 seconds, as that is when the pressure kicks in and triggers reclamation. Although the 50 MB capping of reclamation chunk provides better memory ratios than the default case most of the time, it still deviates from desired behavior in the interval 180-220 seconds as seen in Figure 4.9.

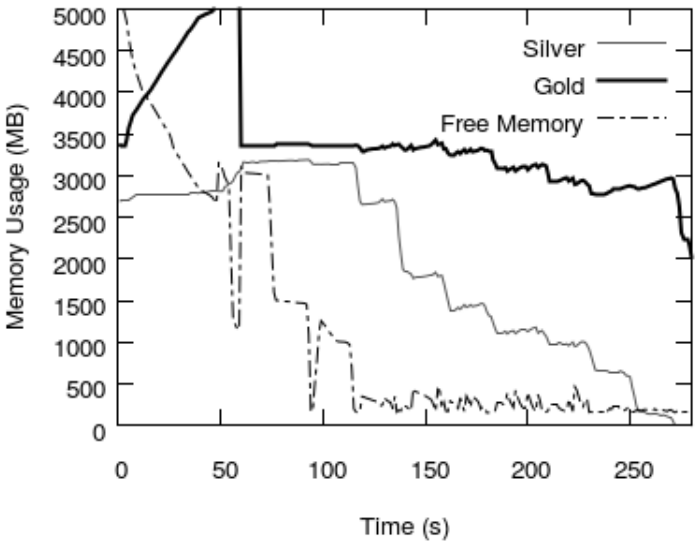


Figure 4.8: Memory usage plot showing implication of relative weights provide differential QOS guarantees

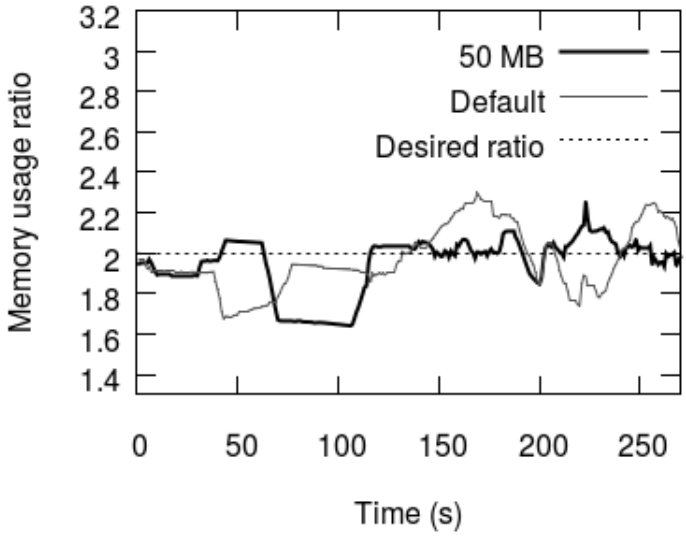


Figure 4.9: Memory usage ratio with different reclamation chunk size

With smaller maximum reclamation size, we will have finer control over the memory allocation ratios, but this comes at the cost of frequent updation. Further investigation has to be done to fix upon a reasonable size to maintain balance between performance and precision here. For now, the reclamation chunk size is parameterized in our implementation and can be easily modified using a kernel module to alter the maximum reclamation chunk size. We have used reclamation chunk size of 50 MB in our implementation and this applies to all the experiments that we have run with our modified controller.

Chapter 5

Memory management framework for derivative clouds

Our objective here is to provide a memory management framework for derivative clouds. In order to achieve the same we start off with understanding existing memory management and cache partitioning frameworks. We analyzed them to come up their drawbacks while provisioning for a derivative cloud setup. We made updates to an previous caching partitioning framework to make it a more full pledged memory management framework than just a cache partitioning framework. My initial work involved understanding the existing infrastructure and come up with drawbacks and make updates to the design to accommodate the downfalls. We came up with a revised design and implemented the same. We tested it for correctness and empirically evaluated it as well.

5.1 Drawbacks of existing framework

The following section tries to bring out the drawbacks of existing hypervisor cache partitioning frameworks and how fail to satisfy application SLA requirements. We demonstrate how an two-level exclusive (either level-1 or level-2) cache partitioning framework could fail to satisfy requirements and an intermediate partitioning framework would be desirable.

5.1.1 Experimental setup

The following section describes the experimental setup used to establish issues, verify correctness and evaluate our solution. Any changes made to the below setup, shall be mentioned beforehand.

Testbed
Our testbed consists of a single VM, single container running on top of our hybrid implementation of Double decker as shown in Fig 5.1. The hypervisor used is KVM, and the container manager used is LXC.

The physical machine configuration used is as described below,

1. Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
2. 4 CPU cores (with multi-threading)

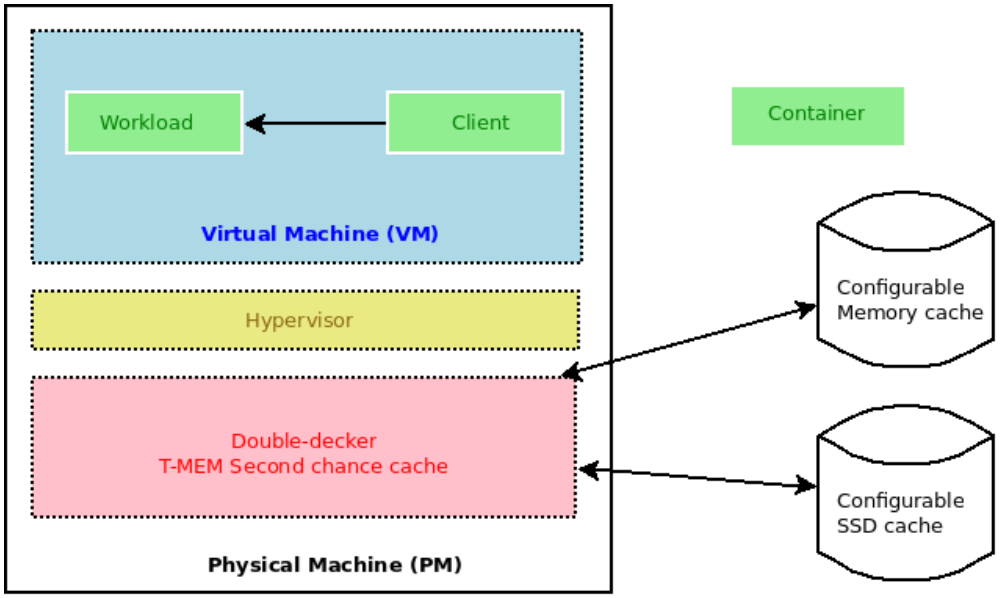


Figure 5.1: Experimental testbed for checking correctness

3. 8 GB of physical RAM
4. 120 GB SSD disk

Experimental configurations

The set of configurations used for an analysis of memory management framework for a derivative environment must be relevant, and easy to apply. The following configurations fit this criteria, and have been used for the evaluation.

- **Memory Requirement:** Memory requirement of each container, the estimated total memory used by a container.
- **Container memory limit:** Size of memory allocated to a container at the Cgroup level (soft and hard limits).
- **Memory cache limit:** Size of memory (L1) cache assigned to a container.
- **SSD cache limit:** Size of SSD (L2) cache assigned to a container.
- **Workload:** Workload application that is running inside each of the container.
- **Number of containers:** Number of containers that are currently executing in the system.
- **Number of VMs:** Number of virtual machines that are currently executing in the system.

For the sake of simplicity in the evaluations of correctness of our setup. We have only considered a single container, single VM setup which makes use of synthetic workload to stress our system.

Metrics of interest

The following are the metrics of interest that would help us establish the correctness of our implementation.

- **Container memory usage:** Guest memory usage of the container.
- **Memory cache usage:** Memory cache used by the container.
- **SSD cache usage:** SSD cache used by the container.
- **Demoted:** Objects moved from memory to SSD cache.

- **Promoted:** Objects moved from SSD to memory cache.

The following metrics are collected both for memory and SSD cache

- **Puts:** Number of objects successfully put into this container cache.
- **Gets:** Number of objects successfully got from this container cache.
- **Flushes:** Number of objects flushed from this container cache.
- **Evicts:** Number of objects evicted from this container cache.

Workloads

This section presents the list of workloads that we have used as primary candidates to evaluate our empirical evaluations. All workloads are chosen keeping in mind the memory intensive nature of the requirement.

These are the list of workloads we have used to establish issues and evaluate our framework.

Synthetic workload

A self generated synthetic workload generated using `cat` command that outputs the content of a file onto `/dev/null`

MongoDB

MongoDB [39] is an open-source, document database designed for ease of development and scaling. Classified as a NoSQL database program, MongoDB avoids the traditional table-based relational database structure in favor of JSON-like documents with dynamic schema. It follows a memory hungry approach where it tries to use up most of system and it actually leaves it up to the OS's VMM to tell it to release the memory.

Redis

Redis [14] is a in-memory data structure store, used as database, cache and message broker. It is used to store a large number of in-memory key-value pairs. Its in-memory nature makes it a prime candidate to use it as a workload in our empirical evaluations.

MySQL

MySQL [41] is a database workload that uses anonymous pages to configure its own user-space data cache.

Filebench

Filebench [42] is a synthetic workload used to generate workload patterns of different applications. In Filebench we made use of the Webserver workload to simulate a Webserver.

YCSB benchmark

We use YCSB [40] (Yahoo Cloud Server Benchmark) project as the benchmark to generate the clients evaluate to the performance of our real workloads i.e MongoDB, Redis and MySQL servers. The goal of YSCB is to develop performance comparisons of the new generation of cloud data serving systems. It is a framework and common set of workloads for evaluating the performance of different key-value stores.

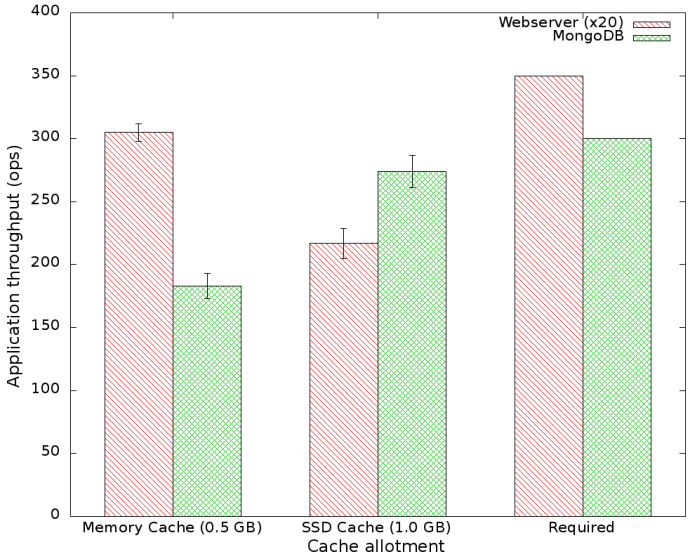


Figure 5.2: Inadequate exclusive two level cache provisioning framework

5.1.2 Provisioning of caches at different levels based on application requirements

Previous results using DoubleDecker[19] have shown that certain application requirements (along with their configurations) could be satisfied better by provisioning on a cache of certain performance guarantees. In the previous work[19] it was shown how an application like Mail-server could be better provisioned on the level-2 (SSD) cache, as its requirement involved having a large WSS (working set size) with slow access rates. On contrary it also showed by cache sensitive applications like the Web-server workload had to be provisioned onto the level-1 (memory) cache. Now let's see if this setup of an exclusive two level cache provisioning schema would satisfy application specific need in all cases.

Inadequate exclusive two level cache provisioning

Objective
To demonstrate Inadequacy in exclusive two level cache provisioning under a tight memory-cache bound scenario.

Procedure
Consider the following scenario to demonstrate the short comings of the existing exclusive two level cache provisioning framework. Consider the case of a container running web-server or MongoDB workload with a WSS to provisioned onto the hypervisor cache with no-free memory to be further allocated at the guest VM and the hypervisor cache available is 0.5 GB at level-1 (Memory) and 1 GB at level-2 (SSD). Consider a Web-server workload where the desired application throughput is 7000 op/s and a MongoDB workload where desired throughput is 300 ops/s.

Inferences
Using our existing double decker cache, we can at max provision at 6000 op/s using memory cache of 0.5 GB as shown in Fig 5.2. But say our requirement was beyond that at 7000 op/s. Similar requirements for MongoDB. Is there a workaround to satisfy this requirement ? Could we come up with a better design to satisfy this configuration ?

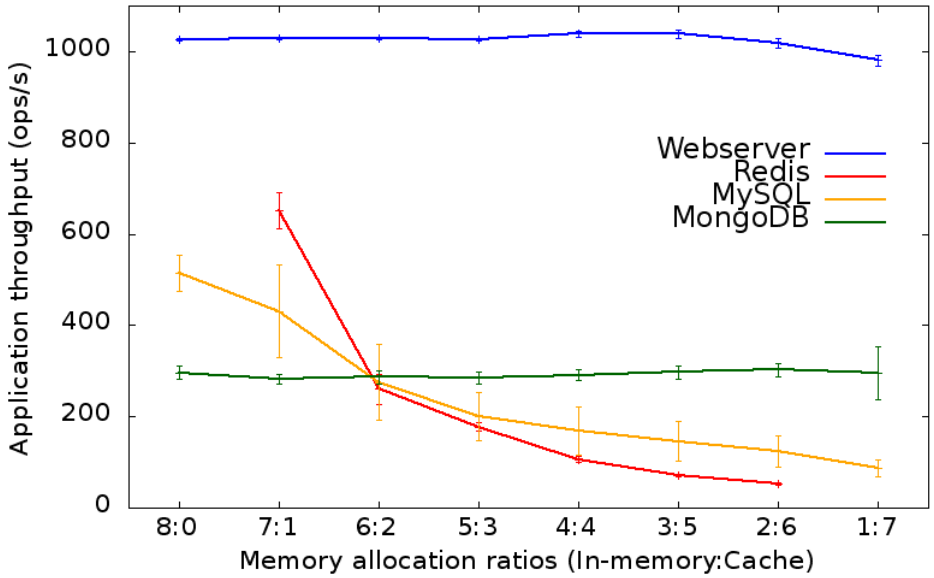


Figure 5.3: Split of memory allocation ratios (In-VM:Cache) to affect application performance

5.2 Cache partitioning framework support for anonymous memory applications

Objective

To understand impact of memory provisioning at different levels of memory (In-VM or cache) and how it impact application performance.

Procedure

Application memory requirement is vastly classified into two types - *anonymous and disk backed*. Application that require disk backed memory pages can be satisfied be either provisioning them at the VM(guest) memory or at the hypervisor cache. We ran four different applications namely, Redis, MongoDB MySQL and Webserver and analyzed their impacts on memory provisioning.

Observations

This effect is seen in Fig 5.3 where the memory requirement is split in ratios of in-VM:cache allocations. The plot shows how applications like Web-server and MongoDB which require disk backed pages can be satisfied either in-VM or at the hypervisor cache. Kindly note that the application performance is nearly constant in all cases of both these applications as the in-VM memory and the hypervisor cache operate at similar speeds in this case, and had there been difference in operational speeds, we would have observed degradation in application performance when moving from left to right in the plot.

Inference

Now, if you look at applications like Redis and MySQL, their performance is highly affected while provisioning them on cache, as they heavily depend on anonymous memory allocated to them (especially Redis). Hence such applications need to be provisioned at the guest, however existing cache partitioning frameworks [17, 33] can provision for disk backed workloads fail to address the needs of anonymous memory hungry workloads.

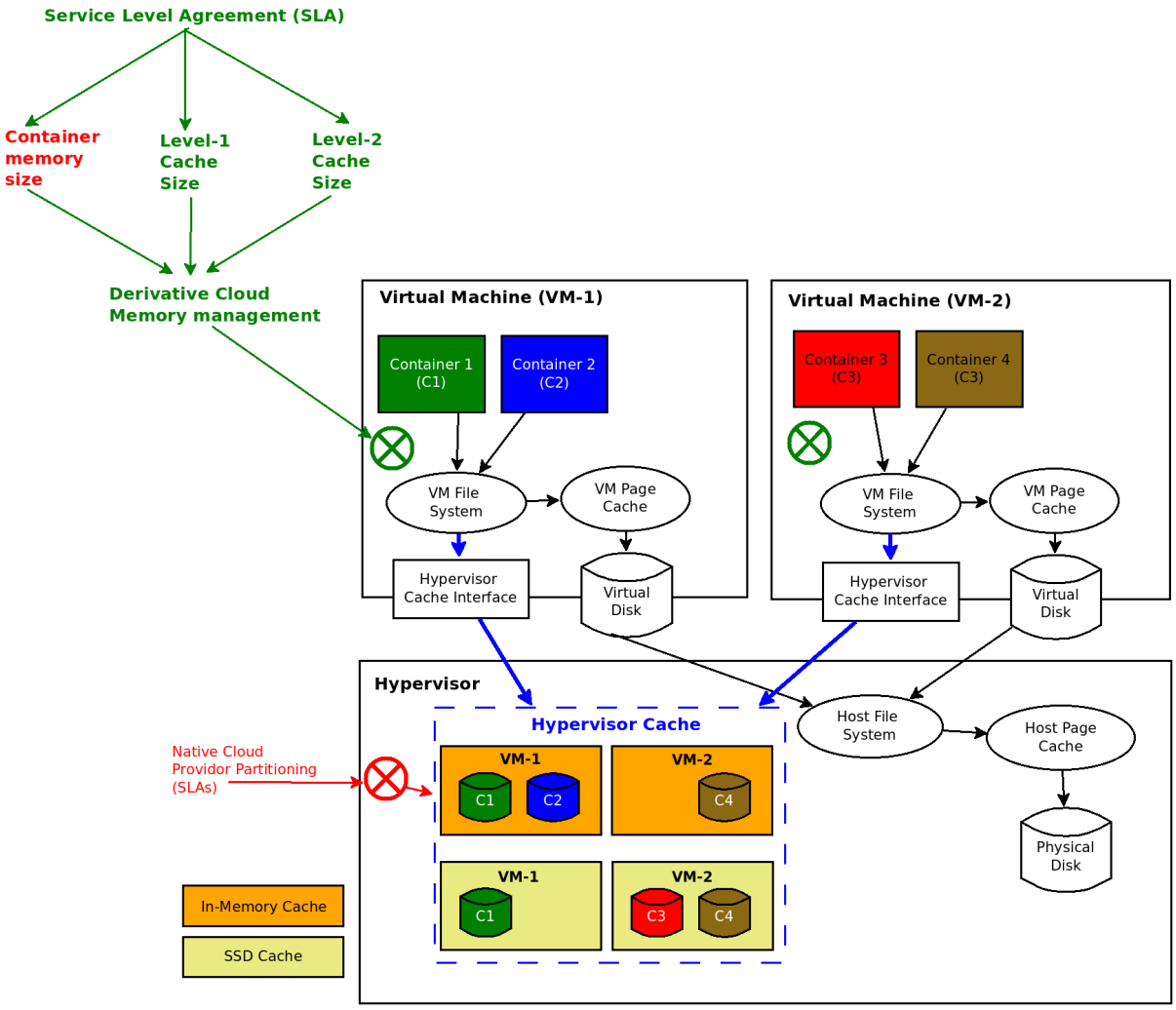


Figure 5.4: Decentralized memory management framework with a hybrid cache

5.3 Decentralized memory management framework with a hybrid cache

Keeping the above drawbacks in mind, we re-design the existing differentiated derivative cloud cache partitioning framework that DoubleDecker is and make it more of a memory management framework. There are two crucial ideas around which the design is centered around,

1. Decentralized memory management framework
2. Hybrid two-level configurable caches

These two components are discussed below.

5.4 Decentralized memory management framework

Traditional caching frameworks and the earlier design of double decker centered around the idea of a hypervisor managed cache with the hypervisor administrator (native cloud provider) as the single sole administrator of the system.

We now propose a decentralized memory management framework with two control centers who roles each have been discussed below. Most of the decision making is moved onto the derivative cloud provider sitting inside the virtual machine.

5.4.1 Native provider cache partitioning framework

The native provider is now, only responsible for partitioning the caches for each VM based on its native cloud provider SLA. The native provider need not worry about provisioning each of the container.

5.4.2 Derivative provider memory management framework

The derivative provider is now responsible for managing three levels of memory entities,

1. Container memory size
2. In-memory or level-1 cache
3. SSD or level-2 cache

The derivative provider maps each application SLA onto a combination of these three management knobs to achieve desired application level objectives. This helps even provisioning of applications that cannot be helped by cache partitioning schemes, that is that they are anonymous memory dependent.

5.5 Hybrid cache

The hybrid cache is an updated version of the previous cache where each VM/container can be configured with a in-memory and SSD cache simultaneously. But with two-levels of caches configured to each application we need an mechanism to move objects (pages) of one level to another and vice-versa.

Every PUT operation into the cache tries to first add the new object into the memory cache if configured, and if not it adds it to the SSD cache.

5.5.1 Movement of objects

There are two levels of cache movements, and have thresholds which trigger this movement. The movement of objects are described below. The thresholds and the eviction batches are all configurable. The Fig 5.5 for the same is shown.

The level-1 and level-2 caches have a higher threshold after which objects are either *demoted* to the lower level (in the case of level-1, if level-2 is configured) or is evicted from the cache. Similarly there is also an lower-threshold at level-1 and when the cache occupied at level-1 is below the lower threshold and objects are present at level-2, objects are *promoted* to level-1 is triggered.

The target execution entity (VM/container) for promotions or demotions are purely based on the entities that are exceeding their allocations the most (or) most under-utilizing their allocations. The order for selection is VM is selected first, followed by the container.

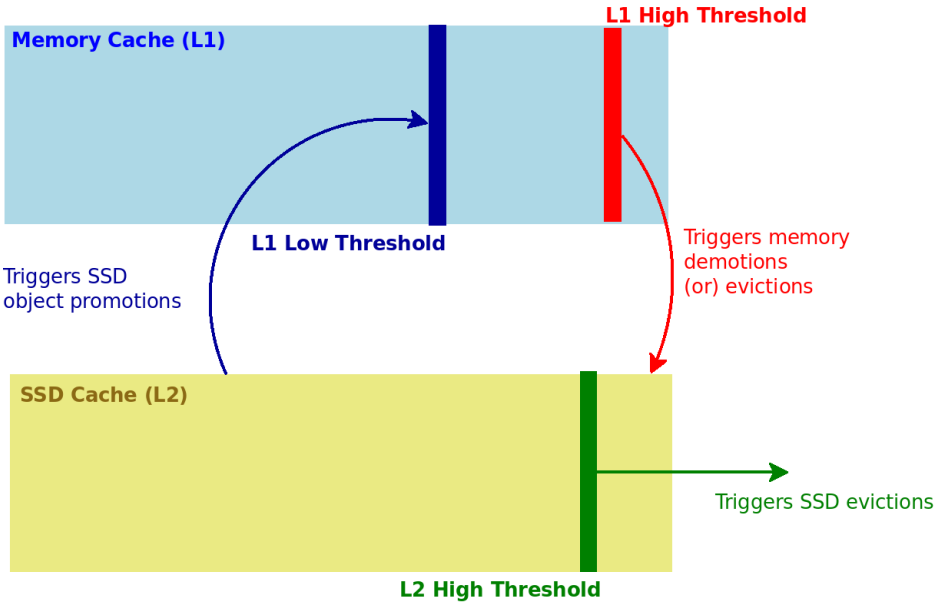


Figure 5.5: DoubleDecker movement of objects across cache levels

5.6 Implementation specifics of the hybrid cache

The hybrid cache implementation is built upon the existing implementation of the double decker cache as described in Section 2.3. Changes have been made to this implementation to accommodate the hybrid setup.

Pool to accommodate both memory and SSD objects

The notion of pools as described in Section 2.3 is carried over to the hybrid implementation. However, every pool for a particular container now contains a memory and SSD weight parameter. The VM also has a memory and SSD cache weight configuration parameter.

The pool now stores both memory and SSD objects (as shown in Fig 5.6 using the same tree structure that earlier stored a single type of object. The search mechanism is the same on the tree, but the tree node now either points to a memory page or and SSD block based on the type of the node.

Asynchronous kernel threads for movement of objects

Two kernel threads were used to promote and demote objects between the two levels of cache. These kernel threads also performed eviction if movement of objects didn't clear up enough space. Asynchronous movement of threads required proper synchronization and locking mechanisms to ensure consistency in data structure being manipulated.

Multilevel stats

The stats that were earlier based on a single level, had to be changed to incorporate a multilevel stat. These multilevel stats are developed to help provisioning of the three level knobs at the guest to satisfy application objectives.

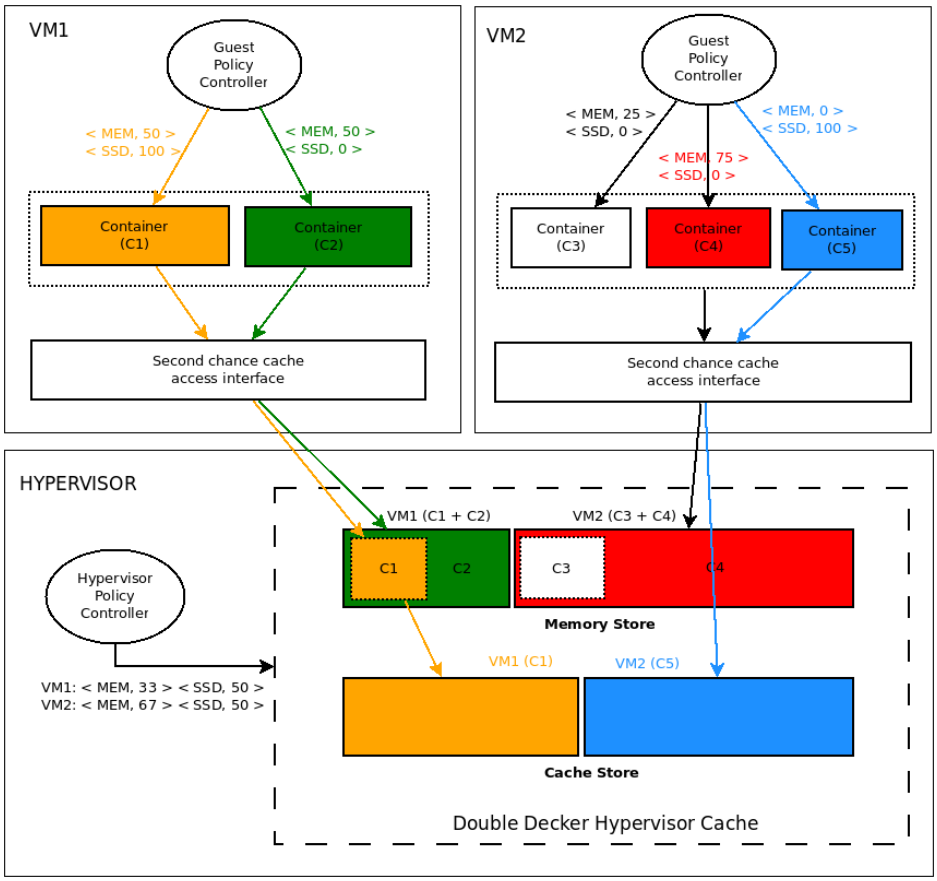


Figure 5.6: DoubleDecker internals

5.7 Correctness of implementation

The experimental setup is just as described in Section 5.1.1. For establishing the correctness of our workload, we have considered a self generated synthetic workload generated using `cat` command that outputs the content of a file onto `/dev/null`. This workloads helps us to validate the correctness of our implementation by predicting deterministic outputs.

5.7.1 Arithematic validation of stats

Question
To verify the correctness in accounting of stats while accessing cache at both levels.

Procedure
We ran several experiments and computed the actual cache usage (memory and SSD) in our implementation. To an calculated an estimated cache usage we used the formula given below. We used the same formula for both memory and SSD cache.

$$EstimatedUsed = Puts + ObjectsMovedIn - (Gets + Flushes + ObjectsMovedOut) \quad (5.1)$$

Observations
The values for *EstimatedUsed* and *ActualUsed* (present in out stat counter) matched in most cases. However,

Metric	Approx. estimated Value	Observed Value
Puts (MB)	2078	2080
Gets (MB)	0	1
Container memory usage (MB)	512	509
Memory cache usage (MB)	504	503
SSD cache usage (MB)	1008	1000
Evicts (MB)	54	64
Flushes (MB)	0	0
Cumulative usage (MB)	2078	2076
Demotions (MB)	1062	1064

Table 5.1: Comparison between expected and actual values

over long periods of run, with quite a large number of cache operations there was a marginal difference between the two (<1%).

Inference
Since the correctness of the actual value of cache used depend on the other stats that we have collected, and the matching of *EstimatedUsed* and *ActualUsed* would only mean that all the stats collected are right.

5.7.2 Movement of objects between both levels of cache

To verify the correctness of our implementation empirically, we have taken our synthetic workload described in above and ran a couple of simple experiments to demonstrate the expected behavior of our cache to support cache operations like puts, gets, promotions and demotions.

Memory to SSD cache

Question
To verify the correctness in accounting of stats while accessing cache and moving objects from memory (L1) to SSD (L2) cache.

Procedure
We start of the experiment with powering on the VM, followed by the container. We assigned complete memory and SSD cache at the double-decker back-end to support the container. The container had a **memory requirement of 2078 MB**, 2048 MB workload requirement and 30 MB container requirement (container requirement was obtained by running the same experiment while having a nearly 0 MB workload. The container was allocated with 2048 MB (512 MB of container memory + 512 MB of memory cache + 1024 MB of SSD cache). Now, the workload performed a sequential read of its workload (i.e 2048 MB) once. Table 5.1 shows the list of approx. estimated values (which are based on our implementation) and observed values for the metrics at the cache at the end of the experiment.

Observations
The following are the observations,

1. There is a small number of gets, probably occurring due to pages used by other container applications.
2. Puts in the cache, is marginally (<2 MB) greater than expected value, and this deviation is due to the small number of Gets which are occurring.
3. The memory cache usage, is exactly as expected. However, SSD cache usage is slightly lesser than the expected value, but however the deviation seems to be an acceptable value.
4. The demotions (movement from memory to SSD) and cumulative usage values are nearly the same with a subtle deviation (<2 MB) which is an acceptable value.

Inference
The accounting stats, are nearly as expected. This verifies the correctness of most of the stats of our implementation (except promotions).

SSD to memory cache

Question
To verify the correctness in accounting of stats while moving objects from SSD (L2) to memory (L1) cache.

Procedure
We start of the experiment with powering on the VM, followed by the container. We assigned complete memory and SSD cache at the double-decker back-end to support the container. The container had a **memory requirement of 2078 MB**, 2048 MB workload requirement and 30 MB container requirement (container requirement was obtained by running the same experiment while having a nearly 0 MB workload. The container was allocated with 2560 MB (512 MB of container memory + 2048 MB of SSD cache). The workload performed a sequential read of its workload (i.e 2048 MB) once, this lead to using up of nearly 1566 MB of SSD cache.

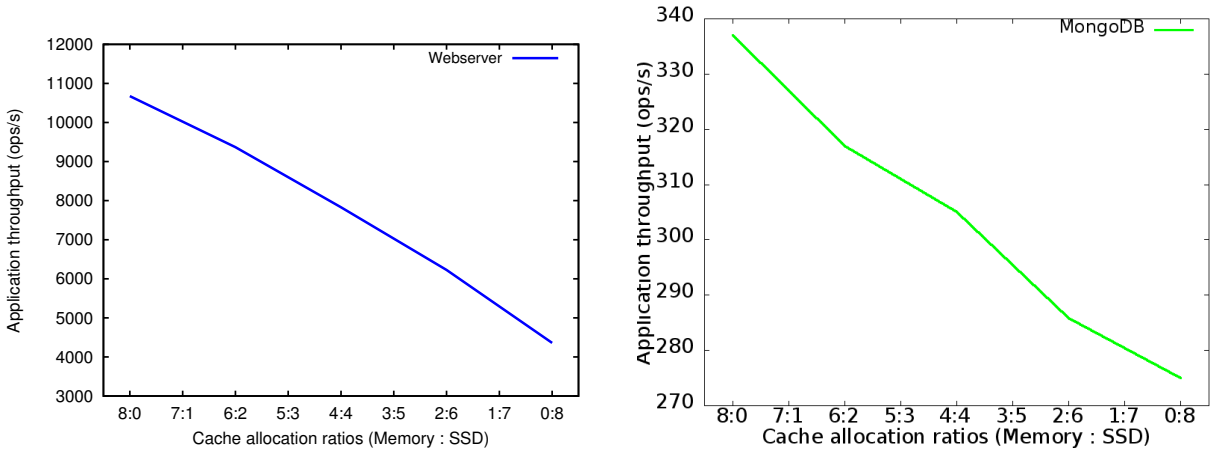
Now, we changed the memory cache size to 256 MB while performing basic operations at the container which triggered the promotion (movement of objects from SSD to memory) of the objects to the memory cache. The promotion triggers all objects until the memory cache reaches a threshold usage - 192 MB in our case, as this threshold is calculated as,

$$MemoryCacheLowerThreshold(192MB) = MemoryLimit(256MB) - LimitSize(64MB) \quad (5.2)$$

Hence we would expect 192 MB worth of objects be promoted from SSD to memory cache in an ideal case.

Observation
Using our stats, it was observed that the estimated promotion and the actual promotion of objects were of an exact match with the number being 192 MB.

Inference
This verifies the correctness in the accounting stats in movement of objects from SSD to memory cache.



(a) Webserver (b) MongoDB

Figure 5.7: Varying of cache allocation ratios while achieving application level throughputs

Hence movement of objects from both memory to SSD and vice-versa have been verified empirically.

5.8 Evaluation of Double Decker

Now, that we have established correctness of our implementation, we had put it to test its effectiveness in tackling in previously mentioned drawbacks. The experimental setup is configured as described in Section 5.1.1

5.8.1 Hybrid cache provisioning

Objective
To demonstrate the effectiveness of the hybrid cache partitioning design to achieve application objectives better than an exclusive two level cache partitioning.

Setup
Setup used is as described in Section 5.1.1.

Procedure
Configuration of the experiment is given at Section 5.1.2, the WSS of the workload is 1 GB. We fix the total cache allocation (Memory + SSD) at 1 GB. To demonstrate the effectiveness of our implementation we vary cache allocation ratios from all cache allocations at level-1 to all at level-2 and see how this affects application performance.

Observation
The application throughput increase as moving the cache partitioned from memory onto the SSD as expected. To satisfy an application objective of 7000 ops/sec as discussed in Section 5.1.2, cache would have to partitioned at about 3:5 ratio as seen from Fig 5.7a, which refers to 384 MB of memory cache and 640 MB of SSD cache.

Workload	MongoDB	MySQL	Redis	Webserver
Required throughput	15 ops/sec	100 ops/sec	5000 ops/sec	300 ops/sec

Table 5.2: Application SLA requirements

To satisfy application objective of 300 ops/sec, the case would have to be configured at 4:4 as seen from Fig 5.7b which translates to 512 MB of cache each.

Inference
An hybrid implementation is more effective at satisfying SLA objectives for cases where an two level exclusive caching setup fails.

5.8.2 Effectiveness of our decentralized memory management framework

Objective
To demonstrate the effectiveness of the cooperative memory management framework in achieving desired application objectives which traditional cache partitioning frameworks cannot.

Setup
VM hosted 4 application containers and executed the MongoDB, Redis, MySQL data stores and the Filebench Web-server workload. The data stores acted as back-ends for YCSB clients. The VM was provisioned with 6GB memory and 8 CPUs (2 CPUs pinned to each application container). The hypervisor cache was provisioned to 2GB.

Procedure
To approximate the best cache partitioning framework, we iterated through a series of possible configurations and have reported the best cache partitioning schema.

Consider the case where the application throughput requirements of the different applications running are as described in Table 5.2

The best configuration was the one met individual application SLAs and yielded the maximum application throughputs.

We have taken frameworks as described below and compare them against one another in achieving application level objectives.

1. **Best cache partitioning - No memory allocations:** Double decker configurations with best cache partitioning and no memory allocation.
2. **Best cache partitioning - Uniform memory allocations:** Double decker configurations with best cache partitioning and uniform memory allocation.
3. **Best cache partitioning - Best memory allocations:** Double decker configurations with best cache partitioning and best memory allocations given based on requirement stats obtained from memory Cgroups.

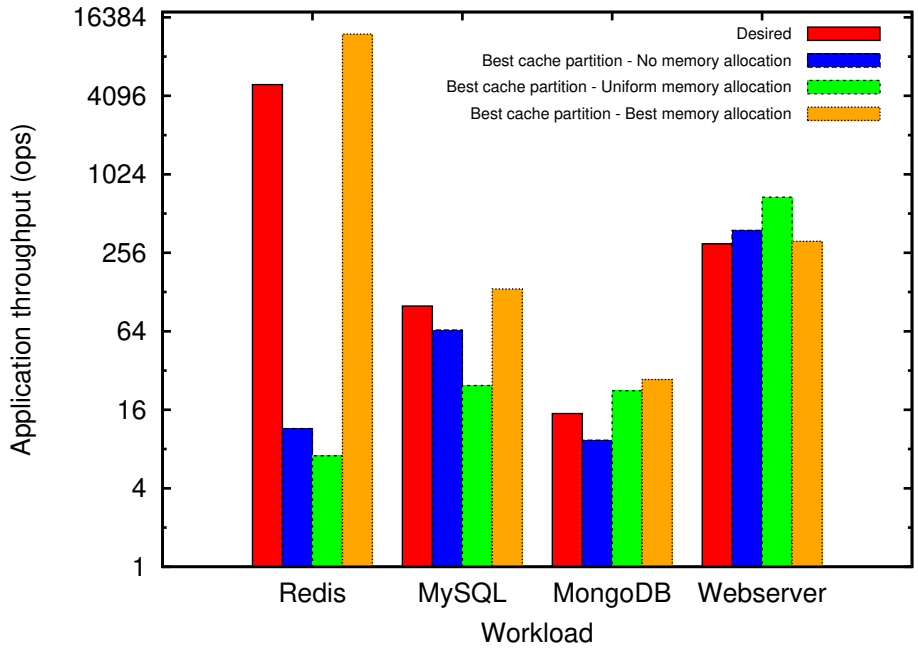


Figure 5.8: Achieved application performance using different types of frameworks

Observations

As Observed in Fig 5.8, our decentralized memory management framework depicted by *Best cache partitioning - Best memory allocations* is the only able to achieve application level objectives. This prominently true in cases of Redis and MySQL which are anonymous memory hungry workloads as provisioning in the cache isn't going to help them achieving application throughputs.

Inference

Two-level provisioning capability of DoubleDecker enables holistic memory configurations which can account for application characteristics and requirements, and yield improved application-level and system-wide performance. For workload provisioning in an adaptive manner, DoubleDecker can employ well known techniques like MRC [43, 35], WSS estimation [36], SHARDS [37] etc.. Note that, the estimation should be done from within the VM which allows the guest OS memory manager to provision memory resources at the two levels. Centralized nesting agnostic hypervisor cache management techniques cannot explore such provisioning configurations

Chapter 6

Conclusions

We have tried to proposed two different types of memory management frameworks, one is a simple update to the existing memory management controller and the other is a decentralized memory management framework for derivative clouds.

6.1 Differentiated memory management controller

We have made an initial attempt to understand the working of memory management in Linux containers. We identified issues with the existing memory management controller used to provision containers in a native and derivative cloud environment. We have empirical verified our hypotheses.

We then have proposed a differentiated memory management controller by making modifications to the existing controller. Our controller supports different types of memory management policies among containers provisioned using a Linux containers framework. We have demonstrated the working of this controller and evaluated the effectiveness of controller in overcoming the pitfalls of the existing controller. Our controller has shown its ability of differentiated memory provisioning among containers and how it positively impacts application performance.

6.2 Decentralized memory management framework

We made an attempt in understanding hypervisor caching frameworks and how they are used to meet application level objectives. We then proposed modifications to existing cache partitioning frameworks to support derivative clouds, however such a framework still failed to help anonymous memory sensitive applications and other complicated cache partitioning constraints.

We then proposed a decentralized memory management framework where the hypervisor is only responsible for partitioning the caches at the VM level and the controller inside the guest is responsible allocating the - container memory and other levels of caches at the hypervisor to meet application specific needs. Our new design also incorporates an hybrid cache design where cache partitioning at the hypervisor can incorporate a two level hybrid partition. We demonstrated the effectiveness of our new controller by provisioning for applications of all kinds and showed how our memory management framework triumphs over other traditional cache partitioning frameworks in meeting application objectives. We also demonstrated the effectiveness of our hybrid implementation by provisioning an application at two levels simultaneously.

Our former contribution on a differentiated memory management controller has been accepted at *IEEE ICDCS '17 (International Conference on Distributed Computing Systems 2017)* in the poster track, and the latter contribution on DoubleDecker—Decentralized management framework for derivative clouds is in submission at *ACM Middleware '17*.

Chapter 7

Future Extensions

The following are a list of the future possible directions of memory management controller,

- The policy enforcement using existing memory controller framework is using static weight setting, we could expand this to incorporate a dynamic weight setting based on an higher level overall application objective.
- Explore other Linux Cgroups - analyze their behavior, bring forward their drawbacks and design solutions to fix their drawbacks.

The following are a list of the future possible directions for DoubleDecker : decentralized memory management for double decker,

- To build a memory-cache allocation logic spread over the 3 levels - VM-memory, memory cache, and SSD cache.
- To come up with policies that are enforceable using our controller dynamically based on application usage patterns.
- To build a fully independent adaptive memory management framework that provisions containers across all layers satisfying individual application objectives and maximizing global allocation objectives.

Bibliography

[1] P. Sharma, S. Lee, T. Guo, D. Irwin, and P. Shenoy, "Spotcheck: Designing a derivative iaas cloud on the spot market," in *Proceedings of the Tenth European Conference on Computer Systems*, p. 16, ACM, 2015.

[2] "Getting your hands dirty with containers." `https://www.cse.iitb.ac.in/~prashanth/containers/seminar/manual.pdf`.

[3] J. McKendrick, "Forbes bussiness magzine: Is all-cloud computing inevitable? analysts suggest it is," 2016.

[4] "Amazon elastic compute cloud." `https://aws.amazon.com/ec2/`.

[5] T. Dörnemann, E. Juhnke, and B. Freisleben, "On-demand resource provisioning for bpel workflows using amazon's elastic compute cloud," in *Cluster Computing and the Grid, 2009. CCGRID'09. 9th IEEE/ACM International Symposium on*, pp. 140–147, IEEE, 2009.

[6] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "Cloudscale: elastic resource scaling for multi-tenant cloud systems," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, p. 5, ACM, 2011.

[7] R. Moreno-Vozmediano, R. S. Montero, and I. M. Llorente, "Elastic management of cluster-based services in the cloud," in *Proceedings of the 1st workshop on Automated control for datacenters and clouds*, pp. 19–24, ACM, 2009.

[8] R. N. Calheiros, C. Vecchiola, D. Karunamoorthy, and R. Buyya, "The aneka platform and qos-driven resource provisioning for elastic applications on hybrid clouds," *Future Generation Computer Systems*, vol. 28, no. 6, pp. 861–870, 2012.

[9] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, pp. 171–172, IEEE, 2015.

[10] R. Morabito, J. Kjallman, and M. Komu, "Hypervisors vs. lightweight virtualization: a performance comparison," in *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, pp. 386–393, IEEE, 2015.

[11] K. Agarwal, B. Jain, and D. E. Porter, "Containing the hype," in *Proceedings of the 6th Asia-Pacific Workshop on Systems*, p. 8, ACM, 2015.

[12] D. Beserra, E. D. Moreno, P. Takako Endo, J. Barreto, D. Sadok, and S. Fernandes, "Performance analysis of lxc for hpc environments," in *Complex, Intelligent, and Software Intensive Systems (CISIS), 2015 Ninth International Conference on*, pp. 358–363, IEEE, 2015.

[13] M. S. Rathore, M. Hidell, and P. Sjödin, "Kvm vs. lxc: comparing performance and isolation of hardware-assisted virtual routers," *American Journal of Networks and Communications*, vol. 2, no. 4, pp. 88–96, 2013.

[14] "Redis in-memory key-value store." `http://redis.io/`.

[15] "Memcached distributed in-memory key-value store." `https://memcached.org/`.

[16] D. Mishra and P. Kulkarni, "Comparative analysis of page cache provisioning in virtualized environments," in *Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2014 IEEE 22nd International Symposium on*, pp. 213–222, IEEE, 2014.

[17] R. Koller, A. J. Mashtizadeh, and R. Rangaswami, "Centaur: Host-side ssd caching for storage performance control," in *Autonomic Computing (ICAC), 2015 IEEE International Conference on*, pp. 51–60, IEEE, 2015.

[18] I. Stefanovici, E. Thereska, G. O'Shea, B. Schroeder, H. Ballani, T. Karagiannis, A. Rowstron, and T. Talpey, "Software-defined caching: Managing caches in multi-tenant data centers," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pp. 174–181, ACM, 2015.

[19] D. Mishra and P. Kulkarni, "Doubledecker: Differentiated hypervisor caching for derivative clouds," 2016.

[20] P. K. Chandra Prakash, Prashanth and U. Bellur, "Mitigating nesting-agnostic hypervisor policies in derivative clouds," in *Poster, International Conference on Distributed Computing Systems (ICDCS)*, IEEE, 2017.

[21] D. Inc., "Docker offical documentation," 2016. `https://docs.docker.com/`.

[22] "Linux container hypervisor," 2016. `https://linuxcontainers.org/lxd/`.

[23] K. Kolyshkin, "Virtualization in linux," *White paper, OpenVZ*, vol. 3, p. 39, 2006.

[24] P. B. Menage, "Adding generic process containers to the linux kernel," in *Proceedings of the Linux Symposium*, vol. 2, pp. 45–57, Citeseer, 2007.

[25] V. Venkatesan, W. Qingsong, and Y. Tay, "Ex-tmem: Extending transcendent memory with non-volatile memory for virtual machines," in *High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS), 2014 IEEE Intl Conf on*, pp. 966–973, IEEE, 2014.

[26] C. A. Waldspurger, "Memory resource management in vmware esx server," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 181–194, 2002.

[27] G. Moltó, M. Caballer, E. Romero, and C. de Alfonso, "Elastic memory management of virtualized infrastructures for applications with dynamic memory requirements," *Procedia Computer Science*, vol. 18, pp. 159–168, 2013.

[28] D. Williams, H. Jamjoom, Y.-H. Liu, and H. Weatherspoon, "Overdriver: Handling memory overload in an oversubscribed cloud," in *ACM SIGPLAN Notices*, vol. 46, pp. 205–216, ACM, 2011.

[29] S. He, L. Guo, Y. Guo, C. Wu, M. Ghanem, and R. Han, "Elastic application container: A lightweight approach for cloud resource provisioning," in *2012 IEEE 26th International Conference on Advanced Information Networking and Applications*, pp. 15–22, IEEE, 2012.

[30] D. Williams, H. Jamjoom, and H. Weatherspoon, "The xen-blanket: virtualize once, run everywhere," in *Proceedings of the 7th ACM european conference on Computer Systems*, pp. 113–126, ACM, 2012.

[31] "Google cloud plalform." `https://cloud.google.com/container-engine/`.

[32] P. Lu and K. Shen, "Virtual machine memory access tracing with hypervisor exclusive cache.," in *Usenix Annual Technical Conference*, pp. 29–43, 2007.

[33] J. H. Schopp, K. Fraser, and M. J. Silbermann, "Resizing memory with balloons and hotplug," in *Proceedings of the Linux Symposium*, vol. 2, pp. 313–319, 2006.

[34] D. Magenheimer, C. Mason, D. McCracken, and K. Hackel, "Transcendent memory and linux," in *Proceedings of the Linux Symposium*, pp. 191–200, Citeseer, 2009.

[35] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar, "Dynamic tracking of page miss ratio curve for memory management," in *ACM SIGOPS Operating Systems Review*, vol. 38, pp. 177–188, ACM, 2004.

[36] W. Zhao, X. Jin, Z. Wang, X. Wang, Y. Luo, and X. Li, "Low cost working set size tracking.," in *USENIX Annual Technical Conference*, 2011.

[37] C. A. Waldspurger, N. Park, A. T. Garthwaite, and I. Ahmad, "Efficient mrc construction with shards.," in *FAST*, pp. 95–110, 2015.

[38] "Stress workload generator." `http://people.seas.harvard.edu/~apw/stress/`.

[39] "Mongodb." `https://docs.mongodb.com/v3.2/`.

[40] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, pp. 143–154, ACM, 2010.

[41] "Mysql database server." `https://www.mysql.com/`.

[42] "Filebench." `https://github.com/filebench/filebench/wiki`.

[43] W. Zhao, Z. Wang, and Y. Luo, "Dynamic memory balancing for virtual machines," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 3, pp. 37–47, 2009.

Appendix A

DoubleDecker source code

The complete source code and documentation for DoubleDecker is available at `https://github.com/kvmprashanth/double_decker` and `https://github.com/provm/double_decker`.

DoubleDecker is implemented as an loadable kernel module. To know the complete procedure for setting up double decker on a Linux machine, kindly follow the steps provided in the README.md in the given above link.