

# 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

# Abstract

Cloud computing has emerged as one of the hot topics in the computing community today. Most servers these days are either already running on cloud, or are in the virtue of shifting base to cloud. Cloud providers traditionally multiplex a set of compute resources, to group of isolated clients using hardware level virtualization techniques that make use of Virtual Machines (VM) to deploy isolated Virtual Environments (VE).

Although VMs provide a very effective methodology in provisioning compute over the cloud, they incur heavy overheads there by degrading efficiency while provisioning. Lately, there has been a new direction in the flow of research in virtualization, i.e OS-level virtualization in which compute resources in a system are virtualized at an OS-level to provision light weight isolated VEs called Containers. Containers provide similar features to that as VMs but incur much lesser overheads [1] [2] . A recent work [3] has also tried to take it a step ahead, by provisioning compute resources to clients using an nested approach in which repackages and resells resources purchased from native Infrastructure-as-a-service (IaaS) cloud provider. This approach is coined as derivative cloud.

In this work, we have made an initial attempt to understand memory management between containers. We started off with purposing hypotheses based on theoretical evidences. We performed analysis to verify the correctness of our purposed hypotheses and understand parts of memory management for which hypotheses couldn't be drawn. We then tried to extrapolate its implications on real world applications running inside a derivative cloud environment running VMs on the host machine and containers in the guest machine. These implications strongly suggested that existing memory management techniques may impact higher provisioned containers negatively. We conclude by purposing the requirements of a new desired policy that provides this notion of a differentiated reclamation to enforce deterministic allocation when the system is under memory pressure. The end goal of our work is to provide an adaptive deterministic resource provisioning framework for container based services.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Memory management in clouds . . . . .	1
1.1.1	Issues in native container environment . . . . .	1
1.1.2	Amplification of issues in derivative cloud environment . . . . .	1
1.2	Caching in the cloud . . . . .	1
1.2.1	Drawbacks of caching in native (VM) cloud setups . . . . .	1
1.2.2	Hypervisor managed caching . . . . .	1
1.2.3	Issues of caching frameworks in derivative clouds . . . . .	1
1.2.4	Application cache sensitivity is unaccounted . . . . .	2
1.3	Problem description . . . . .	2
1.3.1	Phase-1 . . . . .	2
1.3.2	Phase-2 . . . . .	2
<b>2</b>	<b>Background</b>	<b>3</b>
2.1	Memory management between processes in Linux . . . . .	3
2.1.1	Memory pages used by a process . . . . .	3
2.1.2	Memory allocation . . . . .	3
2.1.3	Memory reclamation without container support . . . . .	4
2.2	Containers . . . . .	4
2.2.1	Control groups . . . . .	5
2.3	Double decker: Second chance cache for derivataive clouds . . . . .	9
<b>3</b>	<b>Related work</b>	<b>10</b>
3.1	Memory management . . . . .	10
3.2	Caching . . . . .	11
<b>4</b>	<b>Differentiated memory management controller for containers</b>	<b>12</b>
4.1	Drawbacks of existing memory management for containers . . . . .	12
4.1.1	Experimental setup . . . . .	12
4.1.2	Issues in native environment . . . . .	14
4.1.3	Amplification of issue in derivative clouds . . . . .	19
4.1.4	Key sights and drawbacks of existing system . . . . .	22
4.2	Requirements for a new memory management controller . . . . .	22

4.3	Proposed memory management controller . . . . .	23
4.3.1	Controller logic . . . . .	23
4.3.2	Policies supported by our controller . . . . .	24
4.4	Modifications made to Linux memory Cgroup . . . . .	24
4.4.1	Per container configurable weights . . . . .	24
4.4.2	Deterministic reclamation . . . . .	25
4.4.3	Flexible reclamation size . . . . .	25
4.5	Empirical evaluation of our controller . . . . .	25
4.5.1	Effectiveness of our controller . . . . .	25
4.5.2	Differential QOS containers . . . . .	25
4.5.3	Impact of reclamation chunk size . . . . .	25
<b>5</b>	<b>Double decker - Memory management framework for derivative clouds</b>	<b>26</b>
5.1	Application cache sensitivity . . . . .	27
5.1.1	Provisioning of caches at different levels based on application sensitivity . . . . .	27
5.1.2	Inability of cache partitioning framework to support anonymous memory applications	27
5.2	Rethink of existing design . . . . .	27
5.2.1	Decentralized memory management framework . . . . .	27
5.2.2	Hybrid cache . . . . .	27
5.3	Implementation details . . . . .	27
5.3.1	Existing implementation status . . . . .	27
5.3.2	Hybrid cache . . . . .	27
5.4	Correctness of implementation . . . . .	28
5.4.1	Experimental setup . . . . .	28
5.4.2	Arithmetic validation of stats . . . . .	29
5.4.3	Movement of objects between both levels of cache . . . . .	30
5.5	Evaluation of Double Decker . . . . .	32
5.5.1	Experimental setup . . . . .	32
5.5.2	Performance comparion with old implementation . . . . .	32
5.5.3	Provisioning for anonymous and file backed workloads . . . . .	32
5.5.4	Hybrid cache provisioning . . . . .	32
<b>6</b>	<b>Conclusions</b>	<b>33</b>
<b>7</b>	<b>Future Extensions</b>	<b>34</b>

# List of Tables

4.1	Base configuration for native container experimentation . . . . .	15
4.2	Base configuration for derived cloud experimentation . . . . .	20
5.1	Comparison between expected and actual values . . . . .	30

# List of Figures

2.1	Difference between a VM and Container . . . . .	4
2.2	Control groups illustration using 3 controllers, Source:[4] . . . . .	5
2.3	Mapping of pages to LRU lists . . . . .	7
2.4	Kernel Function Call Trace for System-Wide Reclamation . . . . .	8
2.5	Existing policy for Memory Reclamation . . . . .	9
4.1	Native container testbed . . . . .	14
4.2	Plots for analysis of reclamation when both containers are exceeding by same value . .	16
4.3	Plots for when both containers are having same usage but no exceeds . . . . .	17
4.4	Plots for analyzing effect of workloads characteristics on reclamation . . . . .	18
4.5	Derivative cloud testbed . . . . .	19
4.6	Plots for analysis of complete (above & below SL) reclamation in derivative setup . . .	21
5.1	Experimental testbed for checking correctness . . . . .	29

# Chapter 1

## Introduction

Mention about derivative clouds in intro itself

### 1.1 Memory management in clouds

#### 1.1.1 Issues in native container environment

#### 1.1.2 Amplification of issues in derivative cloud environment

### 1.2 Caching in the cloud

#### 1.2.1 Drawbacks of caching in native (VM) cloud setups

#### 1.2.2 Hypervisor managed caching

#### 1.2.3 Issues of caching frameworks in derivative clouds

Lack of framework support in derivative clouds

Dual layers of isolated control

Derivative provider has no control over cache partitioning    Cache-level sensitivity

Native provider has no control over application memory allocations    Anonymous memory sensitivity

#### **1.2.4 Application cache sensitivity is unaccounted**

### **1.3 Problem description**

#### **1.3.1 Phase-1**

In our work we wish to,

1. Understand the existing memory management policies used by Linux to manage containers.
2. Purpose hypotheses on existing memory management in containers and verify them using experimental results.
3. Extrapolate this understanding to analyze, how it impacts derivative cloud environment.
4. Identify issues with existing policies and purpose requirements for a new policy

#### **1.3.2 Phase-2**

sadsa



# Chapter 2

## Background

### 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.

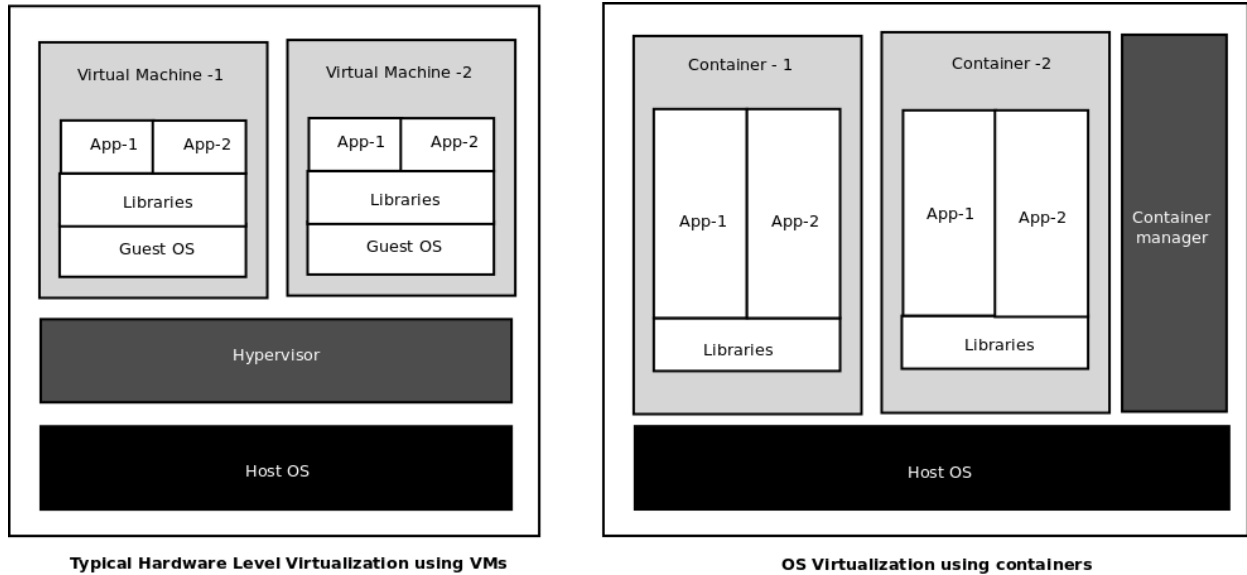


Figure 2.1: Difference between a VM and Container

### 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 [4] 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.

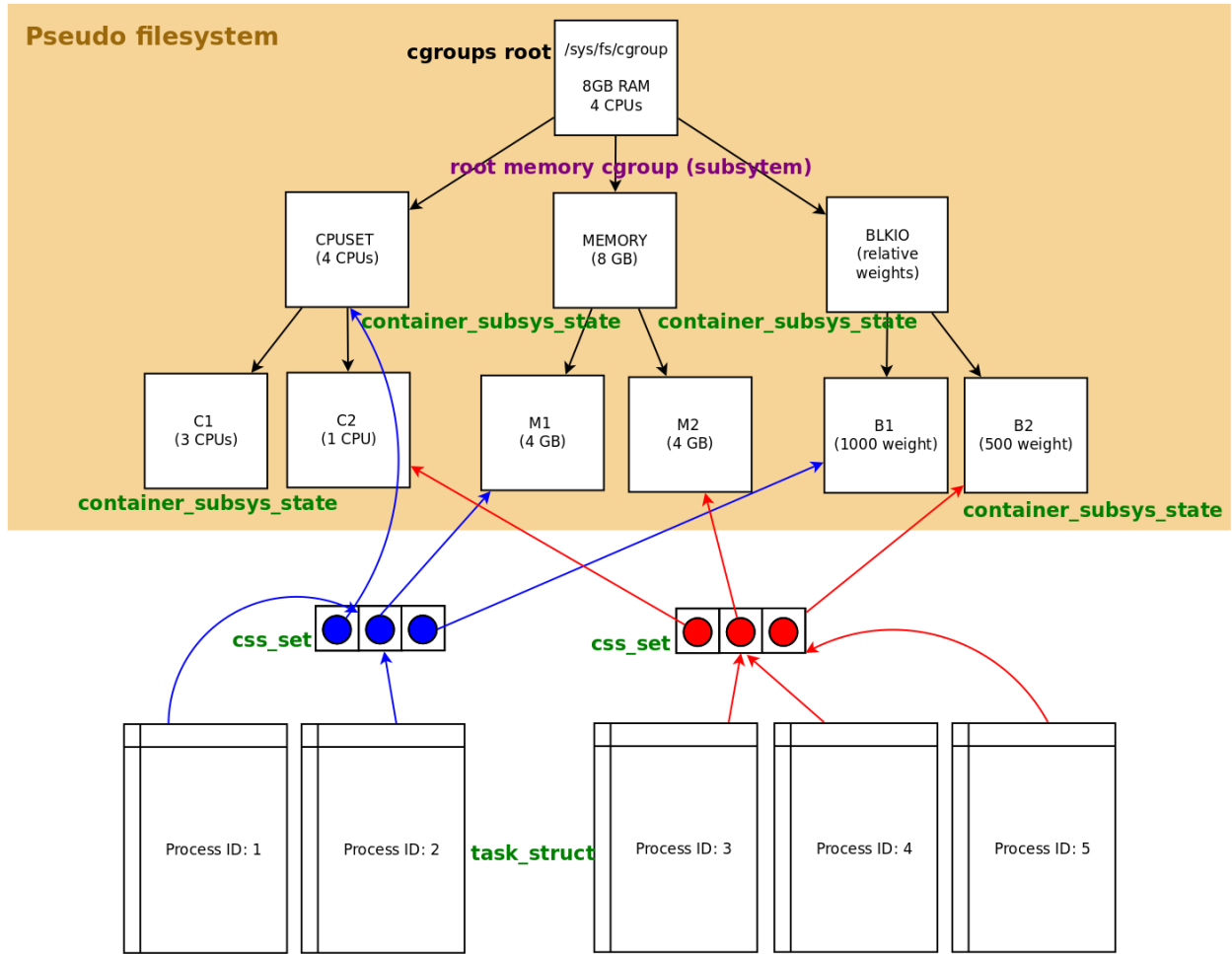


Figure 2.2: Control groups illustration using 3 controllers, Source:[4]

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 [5], LXD [6], OpenVZ [7] 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 pupose of this discussion, we would focus on control Cgroups (Cgroups) as this provides the mechanism to control resources which includes performing memory management.

### 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 [8] 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 cpuset 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.

**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.

**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

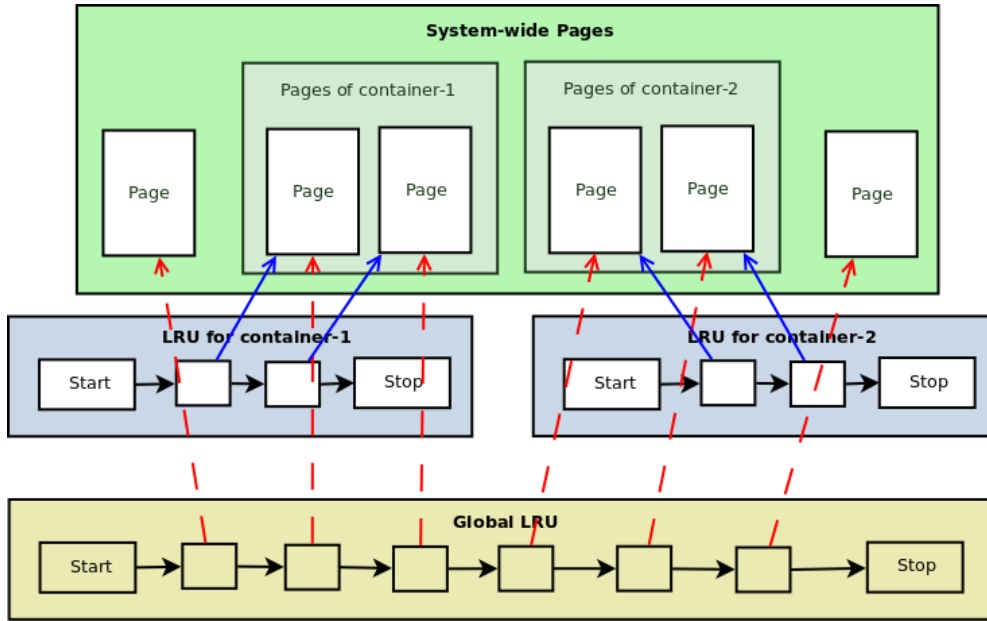


Figure 2.3: Mapping of pages to LRU lists

- 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.

An container (Memory cgroup) when has its soft limit set, has a value called excess computed for



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

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,

$$excess = usage - softlimit$$

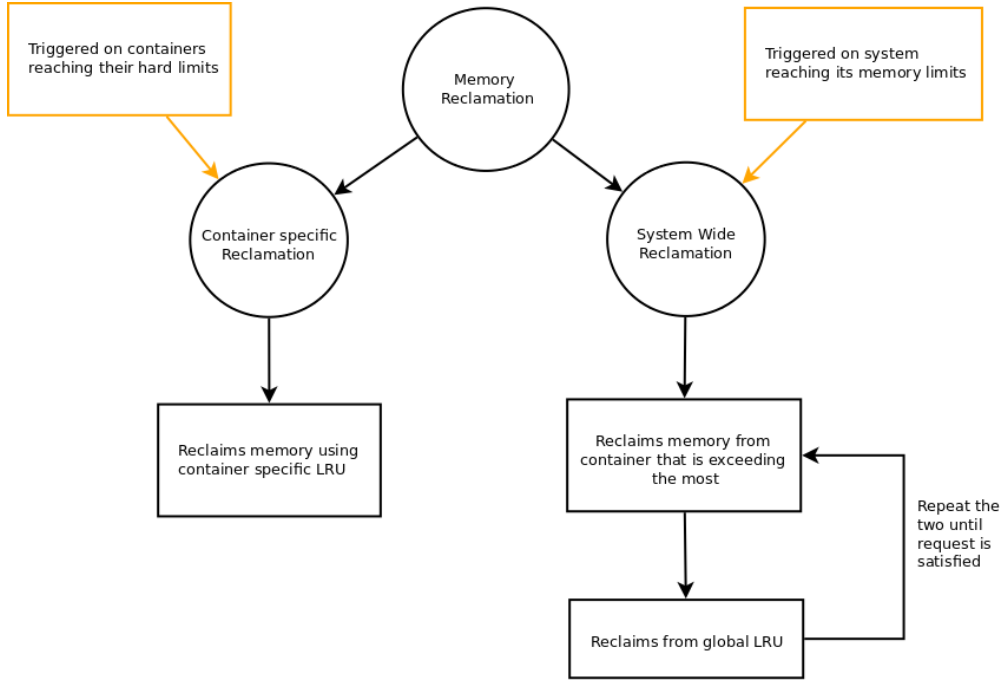


Figure 2.5: Existing policy for Memory Reclamation

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 Double decker: Second chance cache for derivataive clouds

TBD

# Chapter 3

## Related work

### 3.1 Memory management

There have been several attempts to provide efficient memory management for virtual machines. The most prominent used approach is of that Ballooning [9]. Ballooning is a 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 [10] 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 [11] 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 [12] 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.

Looking at researches that have looked at resource provisioning, CloudScale [13] 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 Dornemann[14] 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 [15], 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) [16] is a virtual resource unit for delivering better resource efficiency and more scalable cloud applications.

The idea behind derivative cloud for our discussions have emerged from SpotCheck [3] provides the illusion of an IaaS platform that offers always-available VMs on demand for a cost near that of spot servers using a nest setup of virtual execution environments.

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.



## 3.2 Caching

## Chapter 4

# Differentiated memory management controller for containers

Overview here. TBD

### 4.1 Drawbacks of existing memory management for containers

TBD.

#### 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 [17] 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.

### MongoDB

MongoDB [18] 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

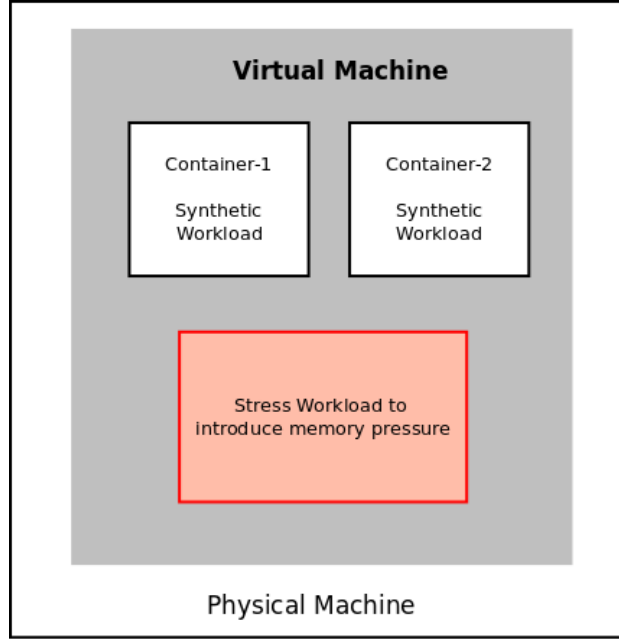


Figure 4.1: Native container testbed

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 [19] 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 [20] (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.

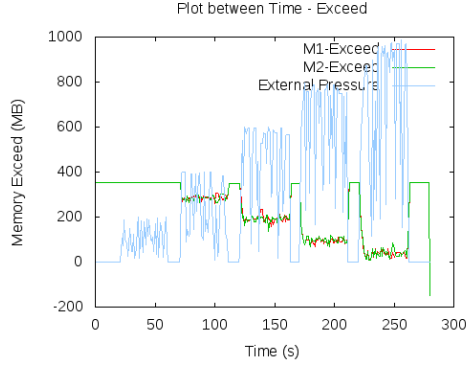
Most experiments involved setting up of 2 containers. Workloads were used to introduce system memory pressure from containers. At this point there was no memory pressure in the system (free memory was still available). Now the external pressure using Stress was introduced after about 20s which created 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

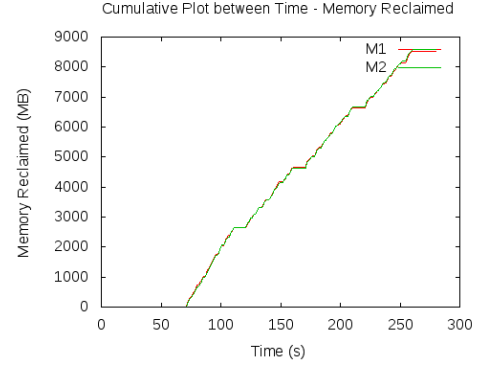
## Hypothesis

Hypothesis to be verified,

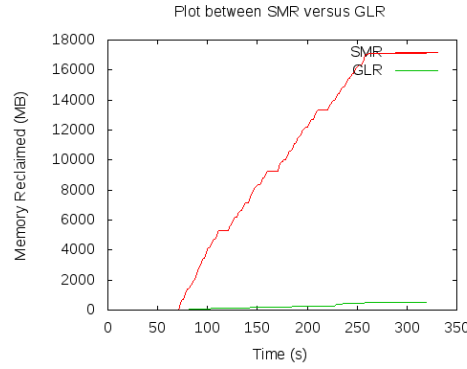
1. Majority of reclamation when containers exceed occurs using SMR (Soft Memory Reclamation)



(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

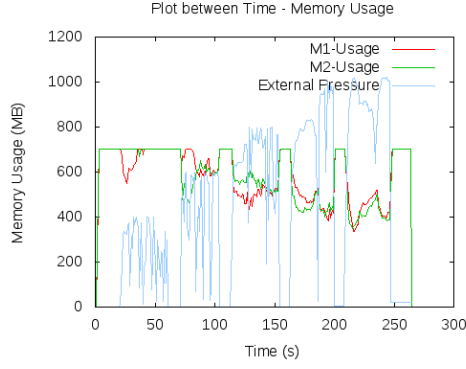
The following are the observations,

- 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.

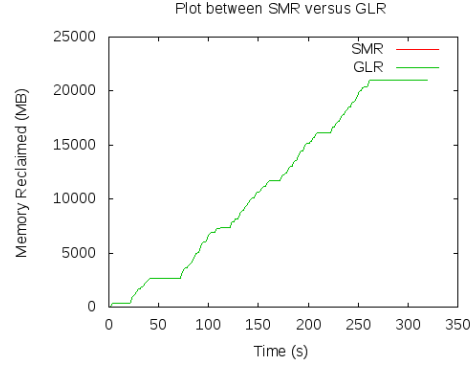
### Inference

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).



(a) Memory Usage Plot



(b) SMR versus GLR Plot

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

- 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

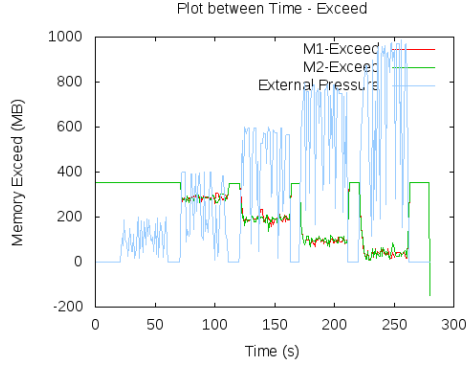
The following are the observations.

- 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

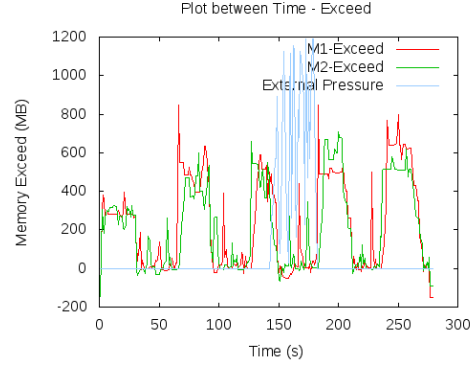
### Inferences

The following are the inferences.

- 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.



(a) Exceed plot for Experiment-4a



(b) Exceed plot for Experiment-4b

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

## 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 chunks gave us the reclamation chunks to be

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

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

### 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.



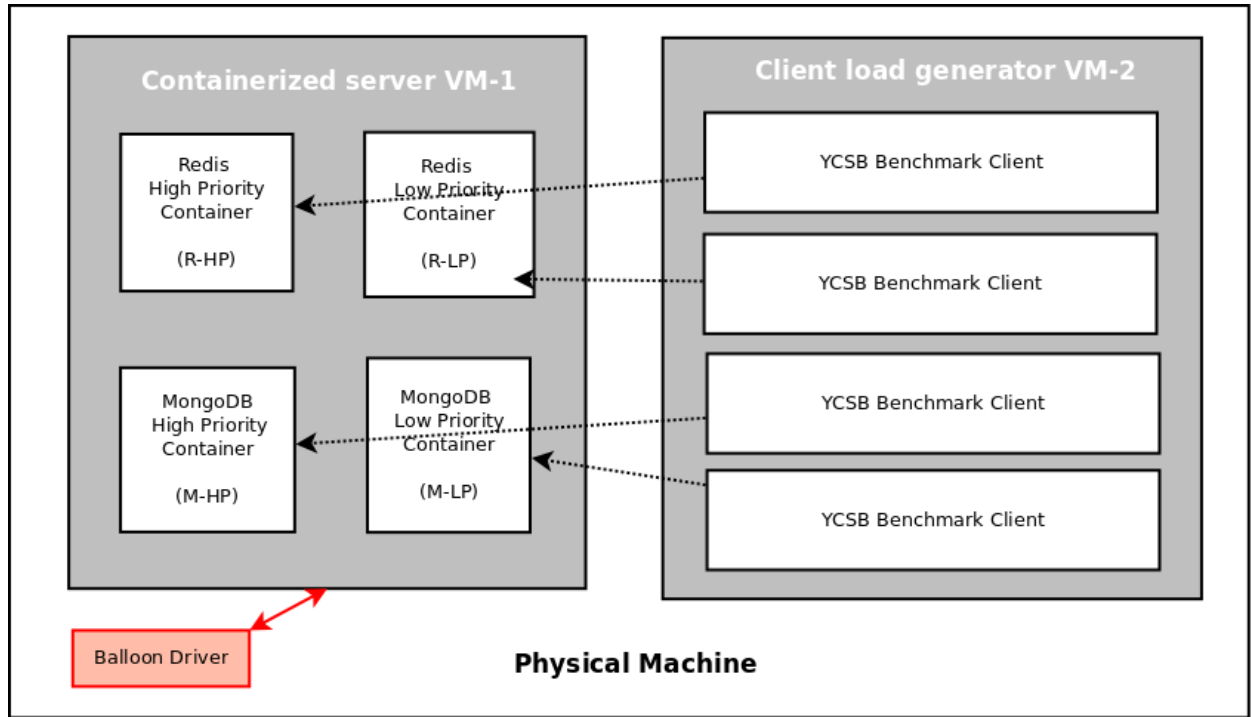


Figure 4.5: Derivative cloud testbed

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.

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

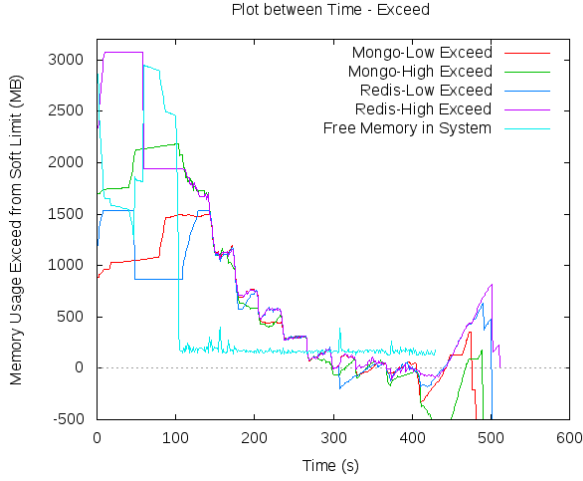
In the experimental setup, 4 containers configured as discussed above. Containers contained server workloads (Redis/MongoDB) to which clients had to connect. There were two types of containers for each workload i.e Redis and MongoDB. One of each with low priority and other of each with higher priority.

The relative avg. usage of the low : higher priority containers are in the ratio of 1:2 and so are their provisioning. Such provisioning makes us **expect as 1:1 throughput** in terms of application performance between the low and high priority workloads in each case in an ideal scenario. The default configurations for the four containers are given on Table 4.2.

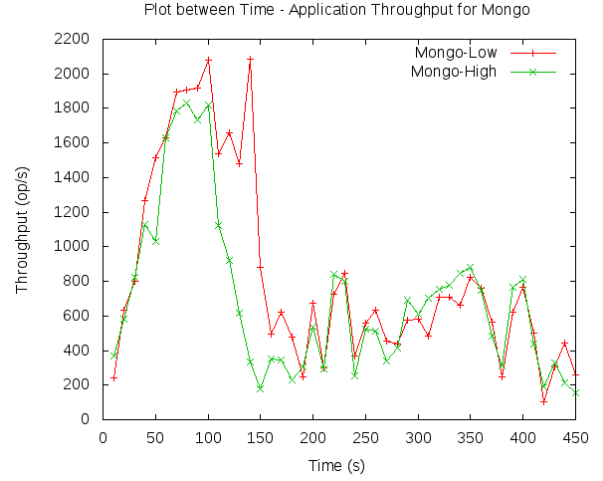
Container	HL (GB)	SL (GB)	Workload Size (records)	Avg. Usage (GB)
Redis-Low	2	0.5	500K	1.3
Mongo-Low	2	0.5	500K	1.3
Redis-High	4	1	1000K	2.6
Mongo-High	4	1	1000K	2.6

Table 4.2: Base configuration for derived cloud experimentation

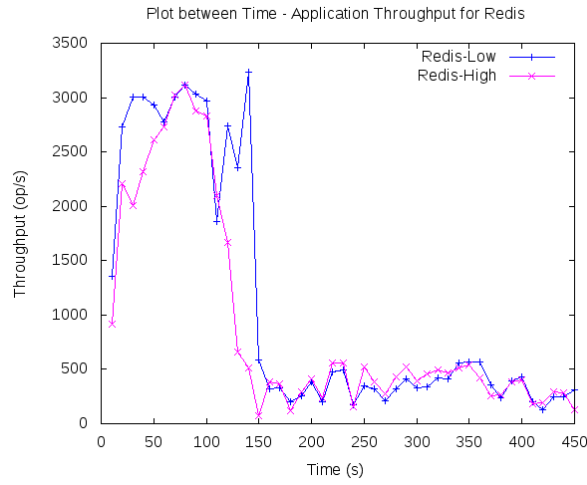
Containers were created and datasets were loaded off-line to generated the initial memory pressure



(a) Exceed plot



(b) Throughput of MongoDB containers



(c) Throughput of Redis containers

Figure 4.6: Plots for analysis of complete (above & below SL) reclamation in derivative setup

required. Each container had only two clients attached to each of them to avoid CPU bounded contention. The clients were setup on VM-2. All containers were over provisioned for all resources other than memory.

Initially 100s was given to allow the containers to setup and consume desired memory without any external pressure. Once they settled, pressure was constantly generated at a rate of  $x$  GB every 30s from the host using a balloon driver to constantly reduce the memory available to VM-1. Experiments were carried out by varying soft limits, workload size (usage) and external pressure (generated by balloon driver).

### Impact in derivative environment

We demonstrate the impact of existing memory reclamation on containers in a derivative setup and how it affects its application performance.

**Question:**

1. How does the entire process of reclamation after throughput of containers moving from all containers exceeding their soft limits to none of them exceeding in a derivative environment ?

**Procedure:**

We begin with containers configured as described in Table 4.2. The balloon driver is inflated inside the guest to change memory inside VM in steps of 16-14-12-10-9-8-7-6-5-4-3 GB every 30s.

**Observations:**

1. Fig 4.6b, Fig 4.6c shows how application throughput of MongoDB and Redis is impacted negatively using the existing knobs when provisioning for deterministic allocation as observed in between  $t=100$  and  $t=200$ .
2. When reclamation goes agnostic of existing knobs (after  $t=200$ ), it appears to have more desired throughputs but this maybe accounted for the nature of the current workloads that actively consume memory.
3. In case of a workload that uses memory after certain intervals, this situation may achieve undesired throughputs even below soft limits. This needs further experimentation.
4. As observed here container soft limits are violated when the system is under immense pressure or the container soft limits are over provisioned. However, the system tries its best efforts to maintain soft limits.

**Inferences:**

1. The container aware reclamation above soft limits impact negatively in when trying to achieve deterministic provisioning with QOS guarantees.
2. There is a lot of non determinism in reclamations below soft limits in the current policy.
3. Soft limits are not guarantees, but are mere best effort approach.

**4.1.4 Key sights and drawbacks of existing system**

1. Reclamation above SL is based on exceed values of each container, may impact negatively while we try to provision containers based on QOS guarantees.
2. Reclamation below SL falls back to host LRU based reclamation without taking into container provisioning and this leads to a lot of non-determinism.
3. Soft Limit is not a definite guarantee, it is mere best effort approach.

**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 resizable 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.

$$T\_U = \sum_{i=1}^n U_i \quad (4.1)$$

$$T\_W = \sum_{i=1}^n W_i \quad (4.2)$$

$$PA_i = T\_U \times \left( \frac{W_i}{T\_W} \right) \quad (4.3)$$

$$EX_i = U_i - PA_i \quad (4.4)$$

$U_i$ : 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 policies 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.

#### **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**

#### **4.5.1 Effectiveness of our controller**

#### **4.5.2 Differential QOS containers**

#### **4.5.3 Impact of reclamation chunk size**





## Chapter 5

# Double decker - Memory management framework for derivative clouds

### 5.1 Application cache sensitivity

#### 5.1.1 Provisioning of caches at different levels based on application sensitivity

#### 5.1.2 Inability of cache partitioning framework to support anonymous memory applications

### 5.2 Rethink of existing design

#### 5.2.1 Decentralized memory management framework

Native provider cache partitioning framework

Derivative provider memory management framework

#### 5.2.2 Hybrid cache

Multilevel configurable caches

Movement of cache objects

### 5.3 Implementation details

#### 5.3.1 Existing implementation status

#### 5.3.2 Hybrid cache

Pools to accommodate both memory and SSD objects

Asynchronous kernel threads for movement<sup>27</sup> of objects

Multilevel state

## 5.4 Correctness of implementation

### 5.4.1 Experimental setup

The following section describes the experimental setup used to verify the correctness of our implementation.

#### 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.

#### Workload

For establishing the correctness of our workload, we have considered a self generated workload generated using `cat` command that outputs the content of a file onto `/dev/null`.

#### 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.

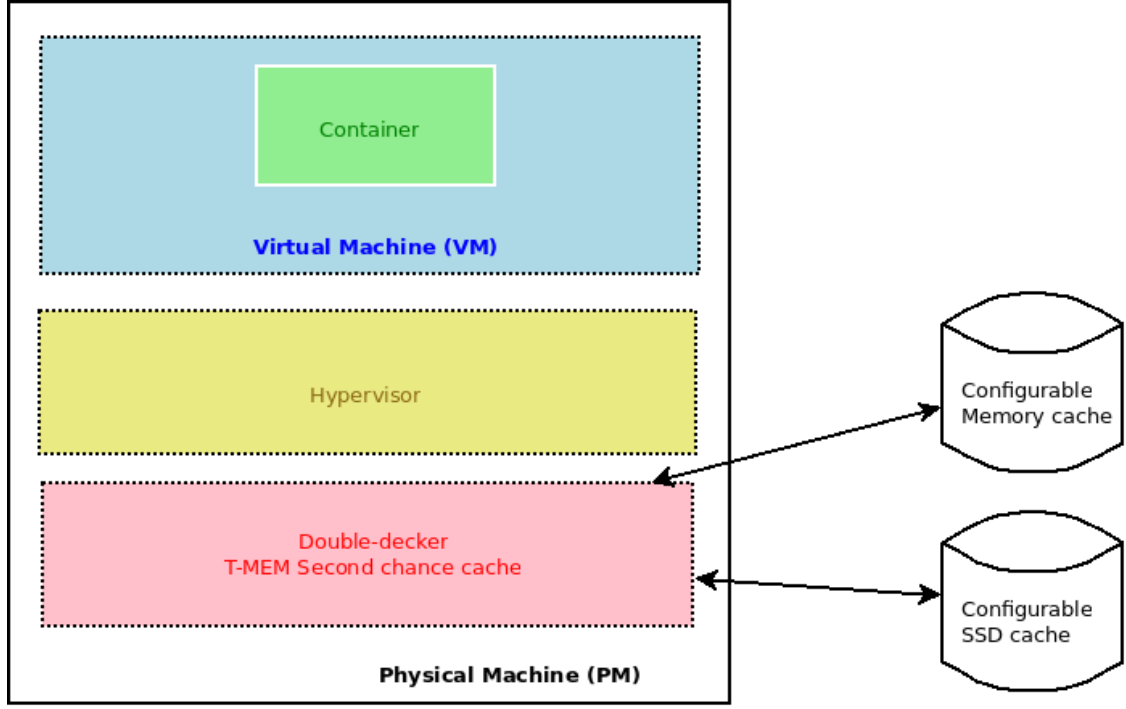


Figure 5.1: Experimental testbed for checking correctness

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

#### 5.4.2 Arithmetic 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 our stat counter) matched in most cases. However, 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,

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

and the matching of *EstimatedUsed* and *ActualUsed* would only mean that all the stats collected are right.

### 5.4.3 Movement of objects between both levels of cache

To verify the correctness of our implementation empirically, we have taken our synthetic workload described in section 5.4.1 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. Hence movement of objects from both memory to SSD and vice-versa have been verified empirically.

## 5.5 Evaluation of Double Decker

### 5.5.1 Experimental setup

Experimental configurations

Metrics of interest

Workload

Testbed

### 5.5.2 Performance comparion with old implementation

### 5.5.3 Provisioning for anonymous and file backed workloads

### 5.5.4 Hybrid cache provisioning

## Chapter 6

# Conclusions

We have made an initial attempt to understand memory management in Linux containers. We started off with purposing hypotheses based on theoretical evidences. We performed empirical analysis to verify the correctness of our purposed hypotheses. We also performed a few more empirical analysis to establish parts of memory management for which hypotheses couldn't be drawn. We then tried to extrapolate its implications in the real world applications running inside a derivative cloud environment. These implications strongly suggested that existing memory management techniques may impact higher provisioned containers negatively, when the system is under memory pressure. We conclude by purposing the requirements of a new desired policy that provides this notion of a differentiated reclamation to enforce deterministic allocation when the system is under memory pressure.

## Chapter 7

# Future Extensions

The following are the list of works that are to be taken up in the near future,

1. Design and implement a new memory management policy for containers.
2. Analyze memory hierarchy in cgroups, and see how this affects containers.
3. Explore other resource controller in the container framework, identify issues and provide appropriate fixes.
4. The end goal is to provide an adaptive resource provisioning framework for containers.



# Bibliography

- [1] 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.
- [2] 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.
- [3] 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.
- [4] “Getting your hands dirty with containers.” <https://www.cse.iitb.ac.in/~prashanth/containers/seminar/manual.pdf>.
- [5] D. Inc., “Docker official documentation,” 2016. <https://docs.docker.com/>.
- [6] “Linux container hypervisor,” 2016. <https://linuxcontainers.org/lxd/>.
- [7] K. Kolyshkin, “Virtualization in linux,” *White paper, OpenVZ*, vol. 3, p. 39, 2006.
- [8] P. B. Menage, “Adding generic process containers to the linux kernel,” in *Proceedings of the Linux Symposium*, vol. 2, pp. 45–57, Citeseer, 2007.
- [9] C. A. Waldspurger, “Memory resource management in vmware esx server,” *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 181–194, 2002.
- [10] 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.
- [11] 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.
- [12] 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.
- [13] 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.
- [14] T. Dörnemann, E. Juhnke, and B. Freisleben, “On-demand resource provisioning for bpm 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.

- [15] 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.
- [16] 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.
- [17] “Stress workload generator.” <http://people.seas.harvard.edu/~apw/stress/>.
- [18] “Mongodb.” <https://docs.mongodb.com/v3.2/>.
- [19] “Redis in-memory key-value store.” <http://redis.io/>.
- [20] 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.