Deterministic Memory Management for Container based Services

Master's Thesis Phase-I Report

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.

Acknowledgement

I would like to thank my guide Prof. Purushottam Kulkarni for the constant support and guidance he has given me throughout, without which my research would have lacked the direction and flow necessary. I would also like to thank Prof. Umesh Bellur who has also been overlooking my work and providing me feedback from time to time.

Prashanth

Contents

List of Tables

List of Figures

1. Introduction

The present era has observed extreme levels of inflation in the number of compute systems being used to automate day to day tasks. Historically this process of automation was typically carried over a system existing locally. However, 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 [5] 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 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

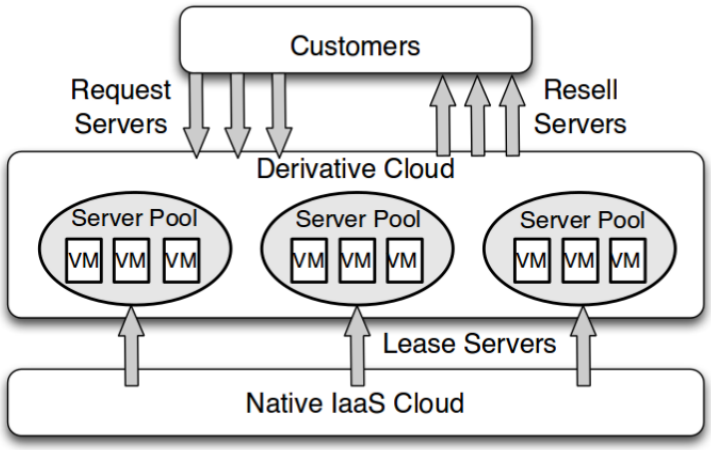


Figure 1.1: Depiction of a derivative IaaS cloud platform, Source:[3]

and servers are provisioned as per actual resource needs. A popular example for the same is Amazon's EC2 [6] 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 [7] [8] [9] [10] 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 [1] [2] [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 [3] 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 as a resource. We aspire to look into other resources as a part of our future work.

1.1.1 Provisioning memory as a resource in a native container environment

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

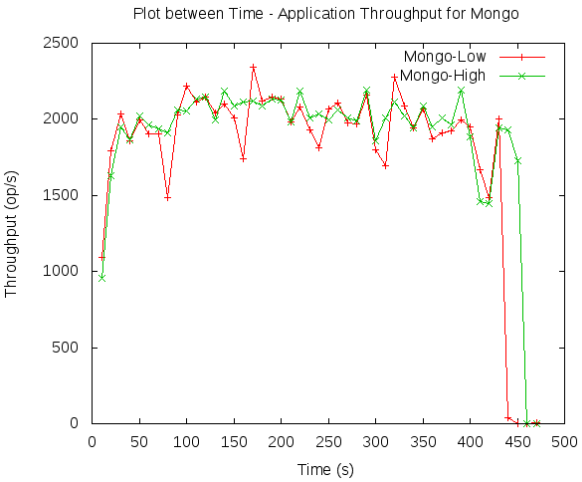
Let's see how memory overcommitment and pressure can disrupt desired functionality of the existing container framework provided by Linux containers.

1.1.2 Illustration of existing issues in native container environment

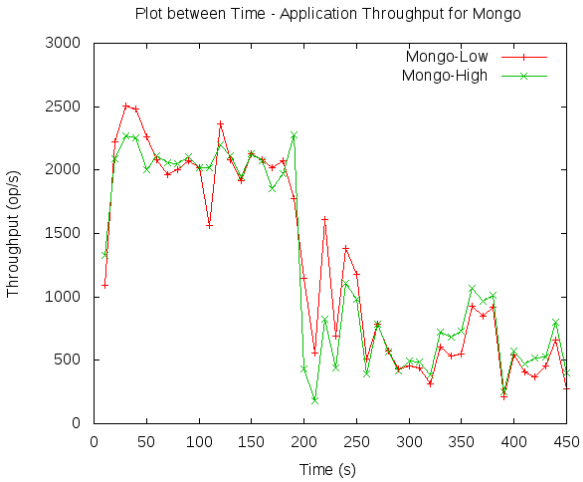
Consider two containers provisioned for running Mongo-DB containers from 2 different customers on a same host machines. Now that average memory used by the 2 containers are in the ratio of 1:2 and the customers for the 2 containers are also paying for their services in the same ratio. Let's call container 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

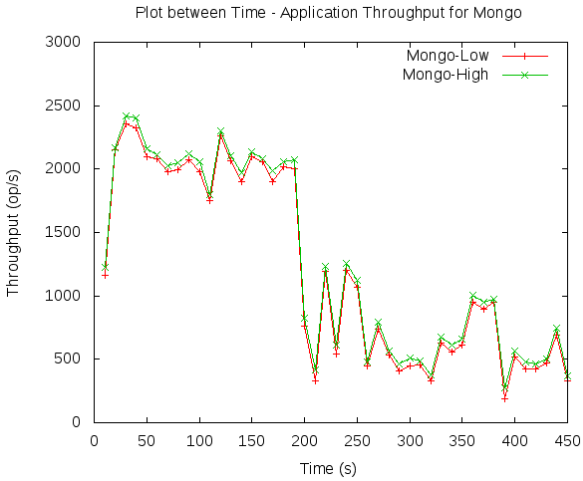
Table 1.1: Memory Provisioned for the Containers using existing memory provisioning knobs



(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.

1.1.3 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.

1.2 Problem description

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

This report is organized as follows, Chapter 2 provides the background required to read and understand the rest of the report. Chapter 3 presents related work. Chapter 4 presents the design to analyze existing system, and Chapter 5 and 6 presents preliminary results. Chapter 7 provides requirement for a desired system design. Chapter 8 concludes the key observations and future work.

2. Background

2.1 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.

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 [14], LXD [15], OpenVZ [16] 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.

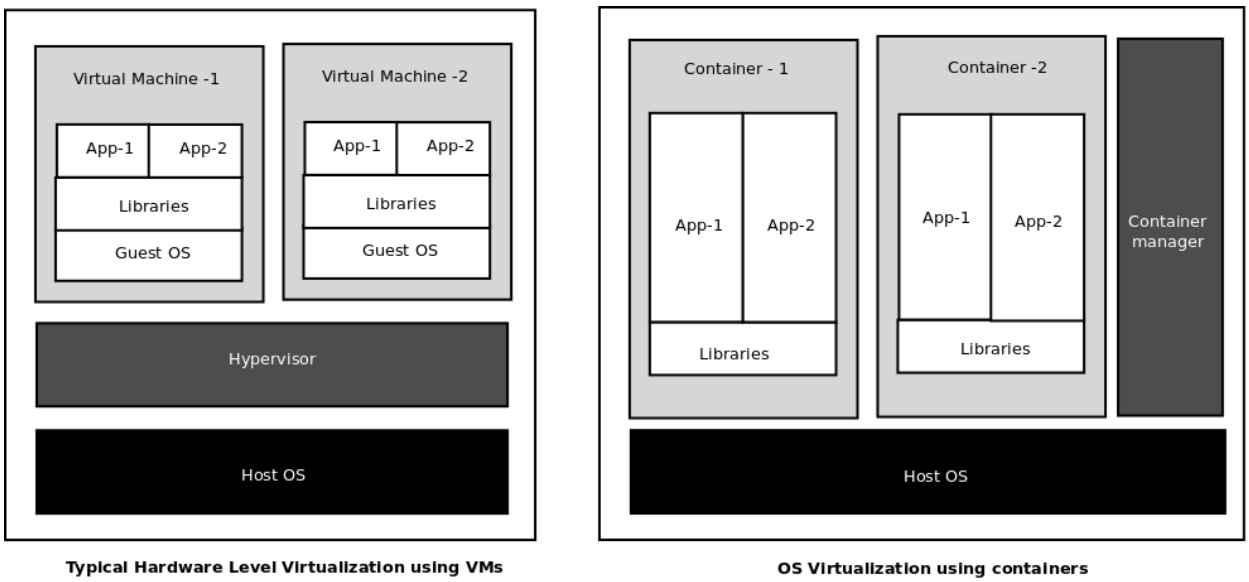


Figure 2.1: Difference between a VM and Container

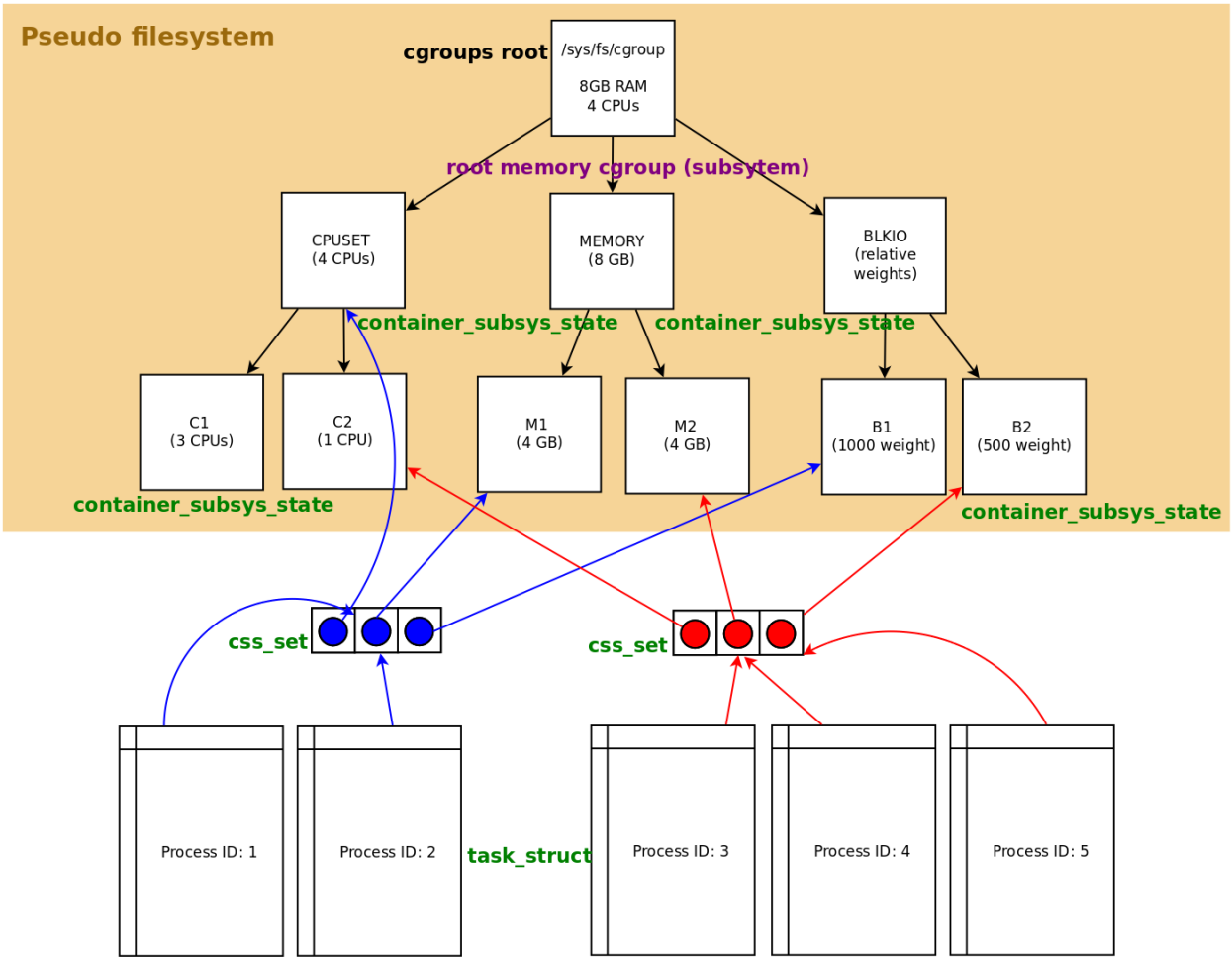


Figure 2.2: Control groups illustration using 3 controllers, Source:[4]

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.

2.1.1 Control groups

A solution to process group control and accounting was proposed by Google in 2007 which was originally called Generic Process Containers [17] 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 subsystem

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.

2.2 Memory management between processes in Linux

Memory is allocated/deallocated in terms of pages in any 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.

2.2.1 Memory pages used by a process

Memory used by processes are divided into 2 types of pages

1. Anonymous Pages: Pages those which are not associated with any files on disk. They are process memory pages.
2. Page cache pages: Are an in-memory representation of a part files on the disks.

2.2.2 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.

2.2.3 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 commonly referred as LRU/2, 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.

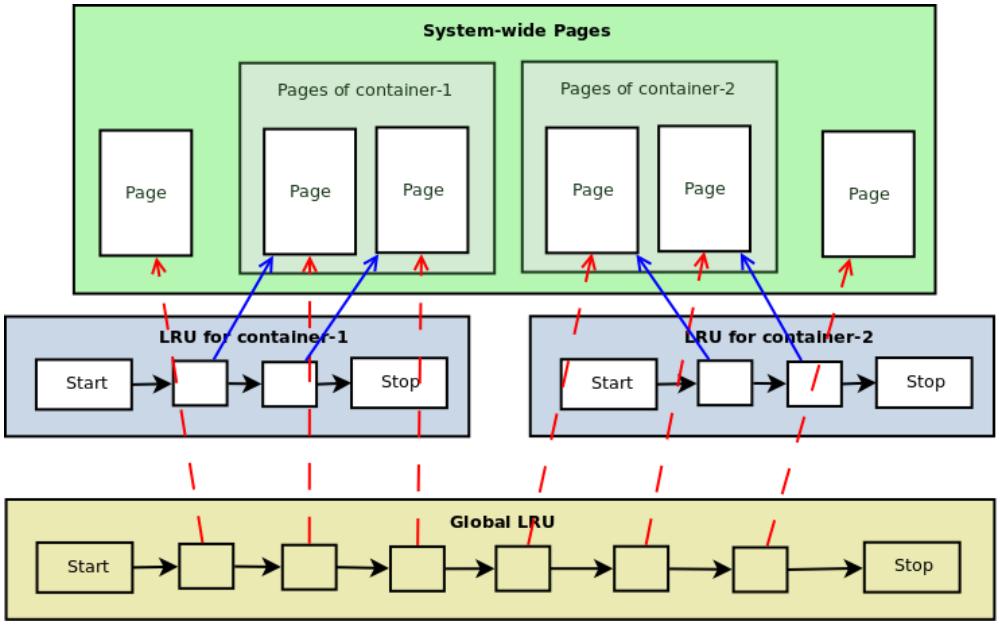


Figure 2.3: Mapping of pages to LRU lists

2.2.4 Memory reclamation with container support

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

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

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 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,

$$excess = usage - softlimit$$

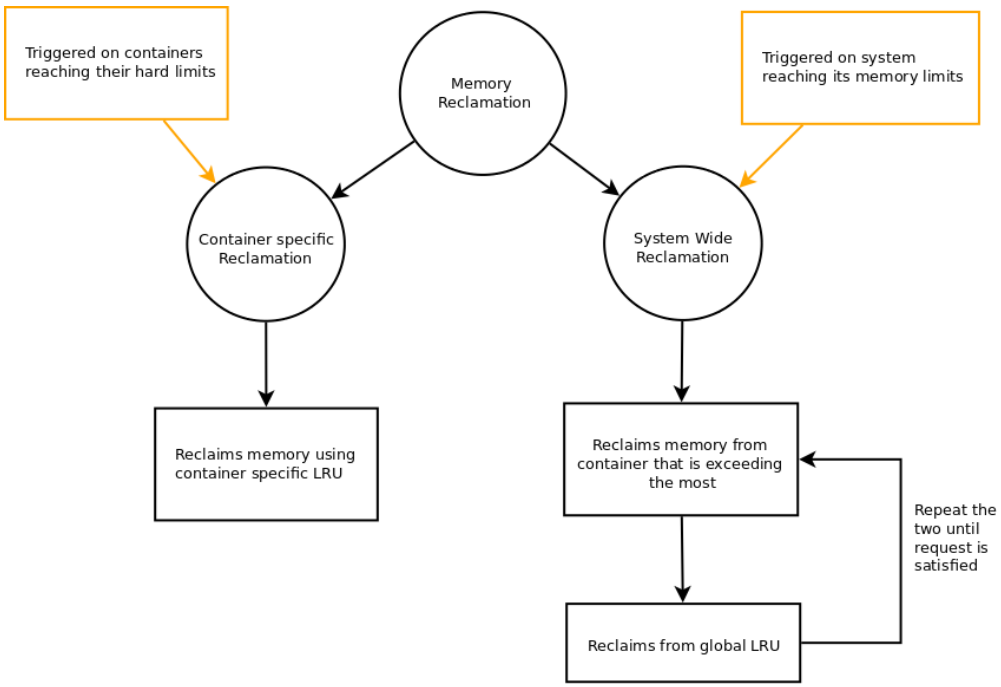


Figure 2.5: Existing policy for Memory Reclamation

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.

2.2.5 Focus on memory reclamation

Although both memory allocation and reclamation play a role in memory management, we focus on memory reclamation as reclamation is the one that penalizes containers on overcommitment. Once memory reclamation is working as desired, even negative provisioning due to improper memory allocation would still be fixed using the reclamation policy. Hence as the subject for discussion we focus on reclamation.

3. Related work

There have been several attempts to provide efficient memory management for virtual machines. The most prominent used approach is of that Ballooning [18]. 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 [19] 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 [20] 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 [21] 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 [8] 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[7] 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 [10], 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) [22] 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.

4. Design to analyze existing system

The following sections describe the set of questions that we aspire to answer and what are the experimental configurations, setup and metrics used to answer the same.

4.1 Experimental scenarios

Before we try to establish the questions that we wish to answer, we list the set of different scenarios in which a containers in a system can exist based on which we try to frame questions. The following are the possible experimental scenarios under which memory pressure maybe generated that triggers reclamation. And our main focus would be to understand how memory pressure would affect applications in these scenarios,

1. All containers exceed: All containers are above soft limits
2. None of the containers exceed: All containers are below soft limits or have no soft limits
3. Few containers exceed and the rest don't

4.2 Questions of interest

The following are the list of questions of interest in each of the above described scenarios,

4.2.1 All containers exceed

1. Is SMR purely based on exceed value of the container ?
2. Does only SMR occur when containers are exceeding ? if not, how much of memory is reclaimed using GLR ? In what ratio does SMR and GLR vary based on system memory pressure ?
3. How much of memory is reclaimed from a container in a single reclamation request ?
4. In what ratio does SMR and GLR vary based on container soft limits ?
5. Does setting containers with higher memory limits using existing memory reservations always guarantee higher priority to a one container over the other while reclamation ?
6. When containers are exceeding by the same values, in what order and how much of memory reclamation occurs from different containers ?
7. When containers are exceeding by the different values, in what order and how much of memory reclamation occurs from different containers ?

4.2.2 None of the containers exceed

1. Does our hypotheses of reclamation below soft limits falling back to native system reclamation hold good ?
2. Both containers not exceeding / containers without soft limits behave similarly ?

4.2.3 A few containers exceed, but the others do not

1. Is the soft limit that is assigned to a container a definite guarantee ? If not, what role does it play in memory reclamation ?
2. How is memory is reclaimed from the containers that exceed and the containers that dont ? Which of the two are more penalized ?

4.3 Experimental guidelines

The following are a set of guidelines that any empirical analysis of memory management techniques in a container environment must abide by. All experiments presented in this report were designed and executed around these guidelines.

1. All experiments must comprise of a set of valid configurations that could be readily applied to any container based OS-level virtualization environment. These configurations must be readily available, and easy to apply.
2. Set of workloads used must always be configured in such a way that it is memory intensive, and always throttles only on memory and no other resource.
3. All experiments must be reproducible and statistically correct.

4.4 Experimental configurations

As pointed out in section 4.3, 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.

4.5 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.

4.6 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 as mentioned in 4.3.

4.6.1 Synthetic workloads

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

Stress

Stress [23] 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.

4.6.2 Real workloads

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

MongoDB

MongoDB [24] 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 [25] 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.

4.6.3 YCSB benchmark

We use YCSB [26] (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.

4.7 Experimental setup

The following section describes the experimental setups used. There were two experimental setups used. The second setup which involved a derivative cloud setup was used to establish the problem in the derivative cloud setup using Real workloads.

4.7.1 Native container testbed

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:4.2. 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**.

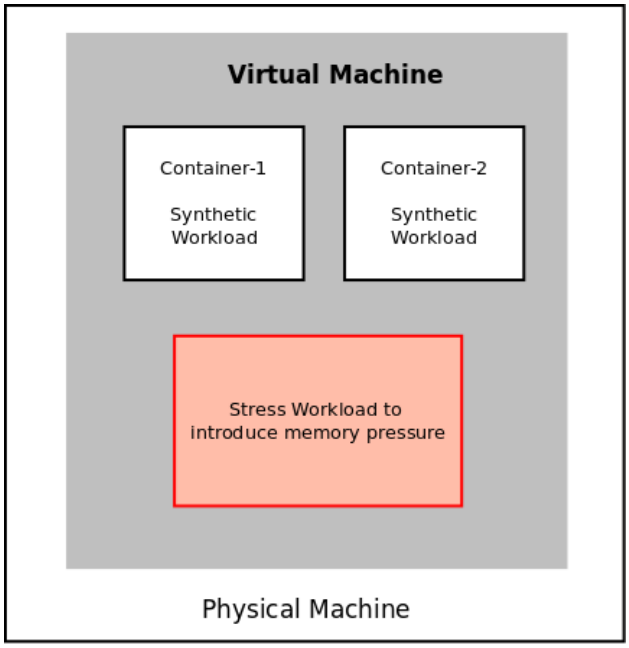


Figure 4.1: Native container testbed

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

Workloads used

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

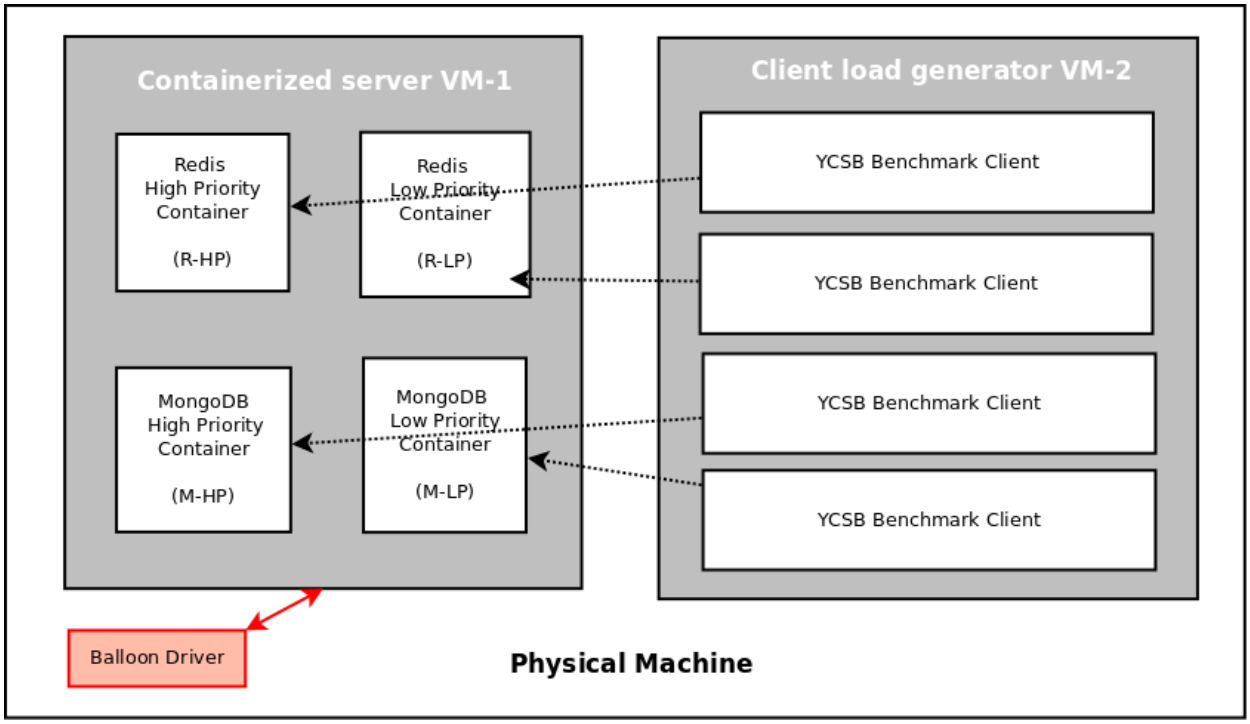


Figure 4.2: Derivative cloud testbed

Experimental flow

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.

4.7.2 Derivative cloud 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:4.2. 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

Workloads used

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

Most experiments involved setting up of 4 containers (2 Redis containers and 2 MongoDB 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 was introduced by reducing the guest VM size after 100s, which in turn trigger memory reclamation at the guest VM.

5. Analysis of memory management in native containers

The list of questions mentioned in section:4.2 are questions of interest that would help us understand the existing memory management techniques better. We have tried answer the listed questions by mapping them into appropriate experiments. Inferences were drawn based on the observations in the experiments. There were 2 different categories of questions we have tried to answer using empirical analysis in a native container environment as listed below,

1. Verify the correctness of our hypotheses
2. Understand parts of memory management for which hypothesis couldn't be drawn

The configuration in Table:5.1 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.

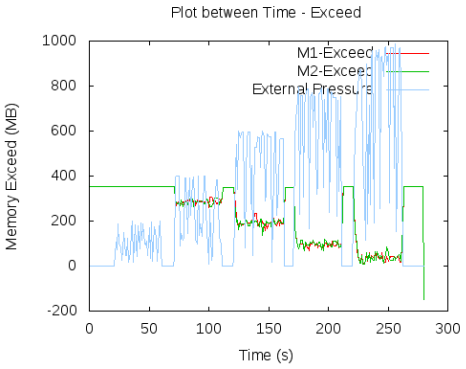
Most experiments involved setting up of 2 containers. Workloads were used to introduce system memory pressure from containers. At this point there was no memory pressure in the system (free memory was still available). Now the external pressure using Stress was introduced after about 20s which created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

5.1 Verification of hypotheses

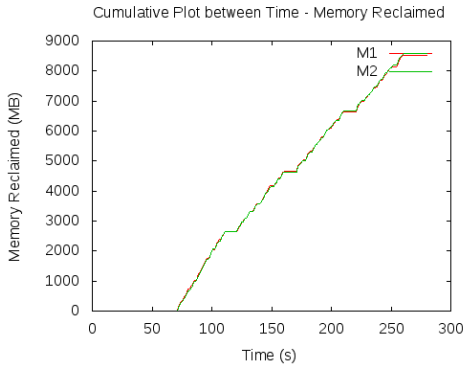
The following set of experiments were done to establish the hypotheses we purposed actually hold good

	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	

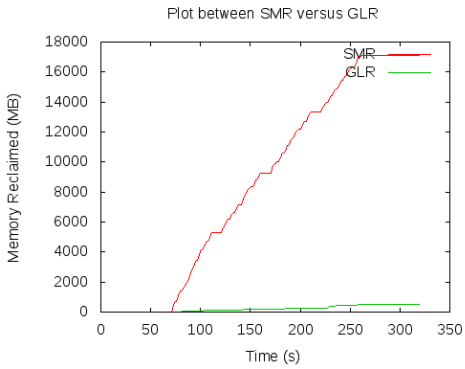
Table 5.1: Base configuration for native container experimentation



(a) Memory exceed plot



(b) Cumulative soft memory reclaimed plot



(c) SMR versus GLR

Figure 5.1: Plots for analysis of reclamation when both containers are exceeding by same value (Exp-1)

5.1.1 All containers exceed

Hypothesis:

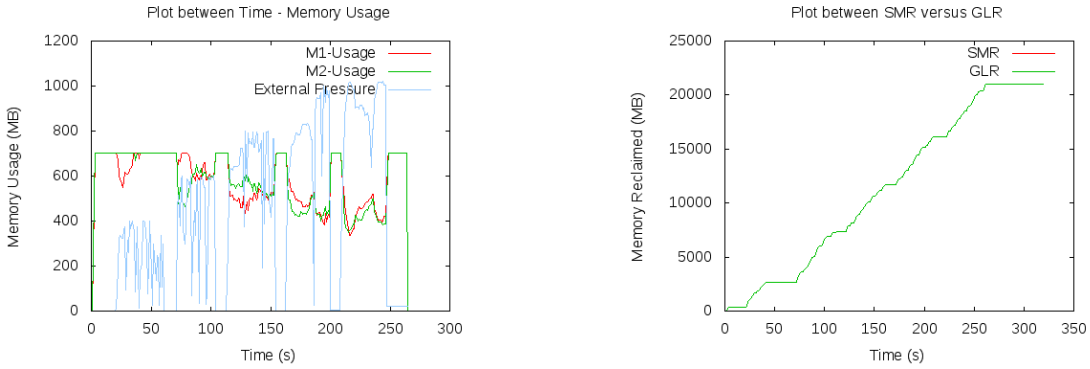
1. SMR (Soft Memory Reclamation) purely based on exceed value of the container
2. Majority of reclamation when containers exceed occurs using SMR
3. Containers that exceed equally are iteratively targeted

Procedure:

To demonstrate the correctness of our hypothesis we the base configuration described in Table:5.1 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:

- As seen from Fig:5.1a, Fig:5.1b - memory reclaimed from containers iteratively from one after the other as their exceeds are same.
- Fig:5.1c 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 5.2: Plots for when both containers are having same usage but no exceeds (Exp-2)

Inference:

- 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.

5.1.2 None of the containers exceed

Hypothesis:

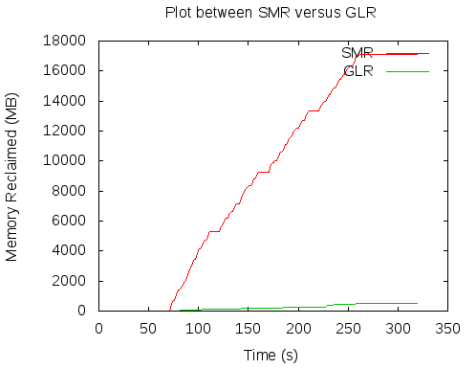
Does our hypotheses of reclamation below soft limits falling back to native system reclamation hold good ?

Procedure:

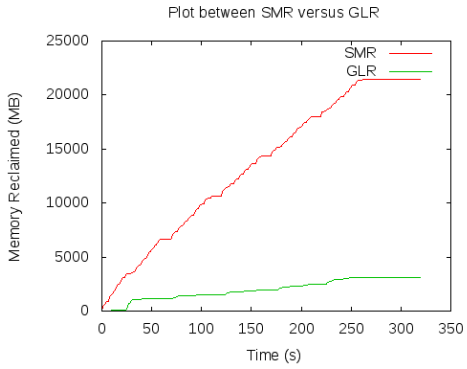
To test the reclamation patterns in containers below soft limits, we created containers as mentioned in Table:5.1 and changed soft limits (Exp-2) of both containers to 1000 MB there by making the current **usage of both containers below soft limits**. We used hooks in the kernel code to track requests satisfied by soft memory reclamation (SMR) and global LRU based reclamation (GLR).

Inference:

- As seen from Fig:5.2a, 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:5.2b



(a) SMR versus GLR for Exp-3a



(b) SMR versus GLR for Exp-3b

Figure 5.3: Plots for see effects of increasing memory pressure on reclamation

Conclusion:

- 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.

5.2 Understand memory reclamation

This section presents list of experiments to answer the list of questions for which hypotheses couldn't be drawn.

5.2.1 Effect of system memory pressure on Reclamation

Question:

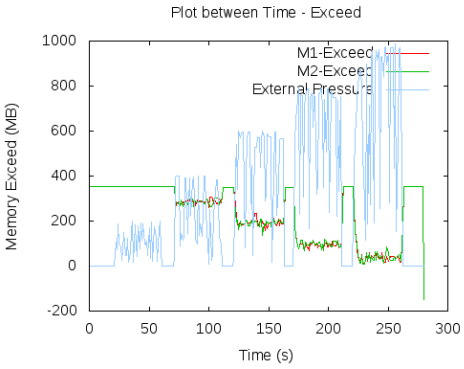
In what ratio does SMR and GLR vary based on system memory pressure ? Does only SMR occur when container exceed is more than reclamation request ?

Procedure:

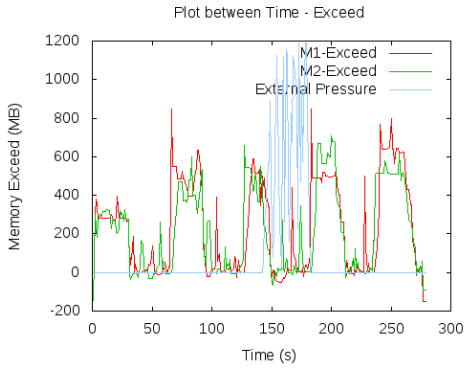
To find out the amount of memory reclaimed using SMR and GLR we took our base configuration as in Table:5.1. We run one experiment (Exp-3a) with base configuration and note its reclamation pattern. We run another experiment (Exp-3b) by increasing its external pressure from 1000-1400 MB in increments of 100MB over intervals of 50s.

Observations:

- With lesser external pressure most reclamation is satisfied by SMR as shown in Fig:5.3a with 98.1% of requests satisfied by SMR.
- As pressure increases, more and more reclamation is directed to GLR as shown in Fig:5.3b now with 87.1% (lesser) requests satisfied by SMR and the rest satified by GLR.



(a) Exceed plot for Experiment-4a



(b) Exceed plot for Experiment-4b

Figure 5.4: Plots for analyzing effect of workloads characteristics on reclamation

- Notice how GLR increase as pressure increases from left to right of both plots.

Inference:

- Both SMR and GLR occur simultaneously when containers exceed with even small reclamation requests.
- The global policy tries to satisfy most of the request through SMR, but as the reclamation demand increases it depends more on the GLR.

5.2.2 Effect of workloads characteristics on reclamation

Question:

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

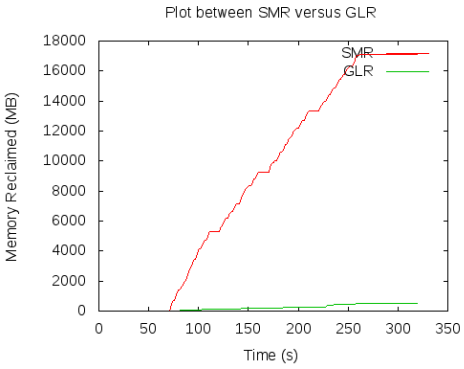
Procedure:

We took our base configuration as described in Table:5.1. 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.

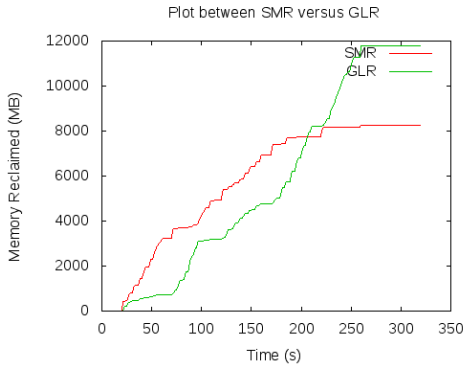
Observations:

- The exceed goes hand in hand as expected but with larger deviation in Fig:5.4a and Fig:5.4b
- 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

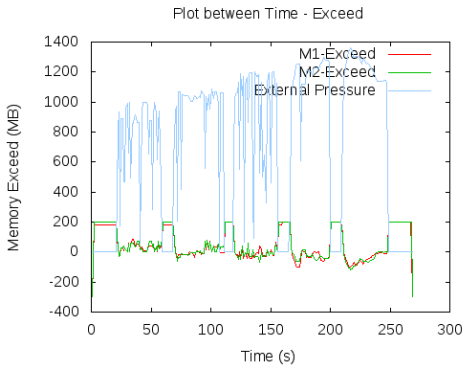
$$\text{Reclamation chuck} = \text{Anonymous memory pages } (<25\text{MB}) + \text{Page cache pages}$$



(a) SMR versus GLR for Exp-5a



(b) SMR versus GLR for Exp-5b



(c) Exceed plot for Exp-5b

Figure 5.5: Plots for understanding how soft limits effect reclamations

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

5.2.3 Effect of soft limits on reclamation

Question:

In what ratio does SMR and GLR vary based on container soft limits ?

Procedure:

We took our base configuration as described in Table:5.1. However we ran the experiments twice, once with the default soft limits that is 150 MB (Exp-5a), and then we doubled the soft limits to 300 MB (Exp-5b) and made our observations.

Observations:

- Fig:5.5a shows memory reclaimed using SMR (88.1%) in with lower soft limits is higher than the memory that is reclaimed using SMR (41.1%) with higher soft limits as shown in Fig:5.5b.
- This occurs as the higher soft limits allows lesser opportunities for SMR and more for GLR.
- Fig:5.5c shows how container exceeds can drop below 0 which indicates that the container usage is going below SL.

Inference:

- Higher soft limits invokes GLR more frequently
- Soft limit is not an absolute guarantee, it merely provides more stability to the minimum memory promised to a container

5.3 Key insights

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.

5.3.1 All containers exceed

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 that exceed equally are iteratively target for reclamation one after the other.
5. Both SMR and GLR occur simultaneously when containers exceed with even small reclamation requests.

5.3.2 None of the containers exceed

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

5.3.3 A few containers exceed, but the others dont

1. Soft limit is not an absolute guarantee, it merely provides more stability to the minimum memory promised to a container.

6. Impact of current memory management techniques in derived cloud

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

Base configuration

All experiments were configured with 4 containers as discussed in section:4.7.2. 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:6.1.

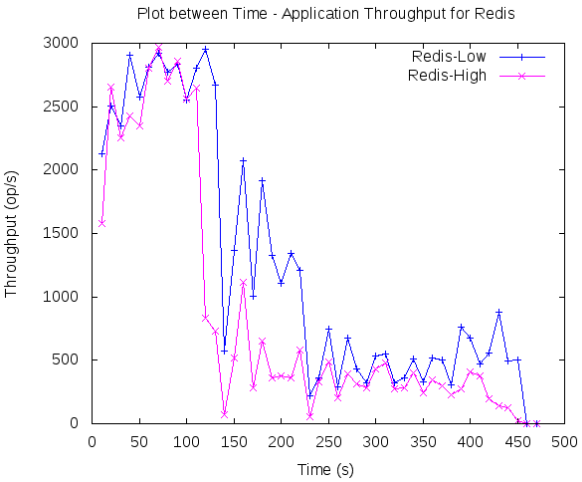
Container	HL (GB)	SL (GB)	Workload Size (records)	Avg. Usage (GB)
Redis-Low	2	0.5	500K	1.3
Mongo-Low	2	0.5	500K	1.3
Redis-High	4	1	1000K	2.6
Mongo-High	4	1	1000K	2.6

Table 6.1: Base configuration for derived cloud experimentation

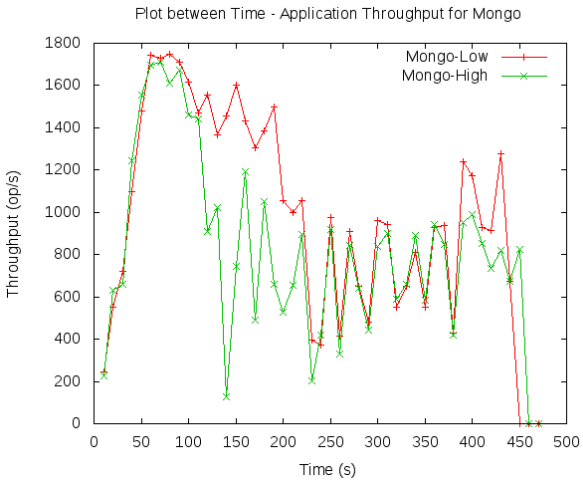
Containers were created and datasets were loaded off-line to generated the initial memory pressure required. Each container had only two clients attached to each of them to avoid CPU bounded contention. The clients were setup on VM-2. All containers were over provisioned for all resources other than memory.

Experimental Flow

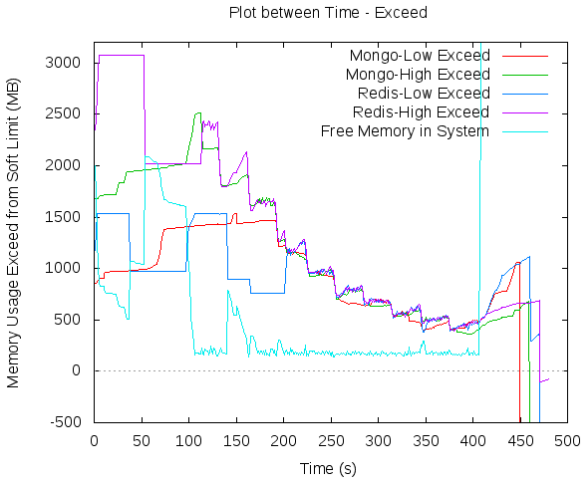
Initially 100s was given to allow the containers to setup and consume desired memory without any external pressure. Once they settled, pressure was constantly generated at a rate of x GB every 30s from the host using a balloon driver to constantly reduce the memory available to VM-1. Experiments were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



(a) Throughput of Redis containers



(b) Throughput of MongoDB containers



(c) Exceed Plot

Figure 6.1: Plots for analysis while all containers exceeding

6.1 Impact analysis

This section presents the empirical analysis in the derivative cloud environment to see how it impacts real applications.

6.1.1 Reclamation when all containers are exceeding

Question:

1. How does reclamation affect applications all containers are exceeding ?
2. When containers are exceeding by the different values, in what order and how much of memory reclamation occurs from different containers ?

Procedure:

We begin with containers configured as described in Table:6.1. The balloon driver is inflated inside the guest to change memory inside VM in steps of 16-15-14-13-12-11-10.5-10-9.5-9 GB every 30s.

Observations:

1. Fig:6.1b, Fig:6.1a shows how application throughput of both MongoDB and Redis reach desired throughputs when there is no external pressure in the initial 100s.
2. However when pressure kicks in after 100s, the higher priority containers performance degrades drastically.
3. Fig:6.1c shows how higher priority containers are penalized more while reclamation which reason for the observed decline in application throughput.
4. Fig:6.1b and Fig:6.1a shows how each of their containers that were better provisioned in memory limits are targeted negatively due to their exceeds.

Inferences:

1. Exceed based reclamation will negatively impact containers that are provisioned with more memory but are just exceeding by greater values.
2. Containers with different exceeds equalize and then reclaim alternatively.

6.1.2 Reclamation when none of the containers are exceeding

Question:

1. How does reclamation affect applications when none of the containers are exceeding ?
2. Are the application performance similar to how it occurs above exceeds ?

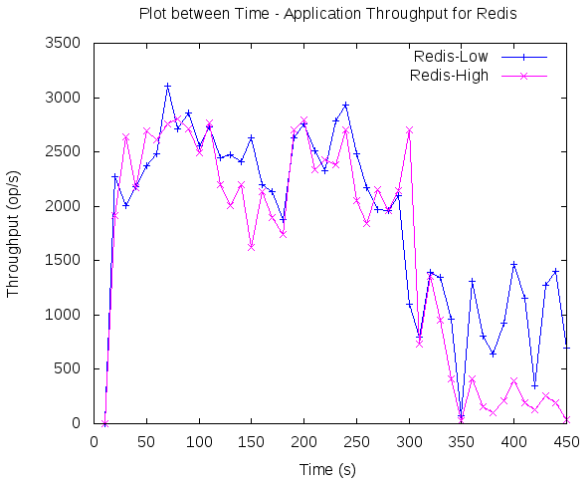
Procedure:

We begin with containers configured as described in Table:6.1. We increase the soft limits of each container upto its hard limits to simulate a situation where initial usage of the containers are below soft limits. 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.

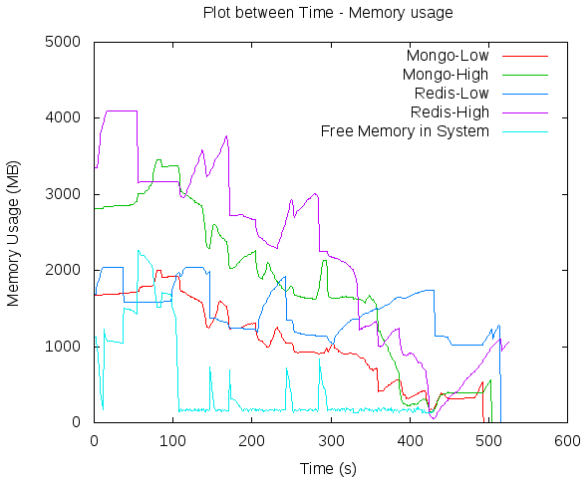
Observations:

1. Fig:6.2a, shows how application throughput of MongoDB almost reaches desired throughputs, although their throughputs never stabilize.
2. Fig:6.2b, shows how application throughput of Redis penalizes the higher priority containers due to more pages in its LRU list.

(a) Throughput of MongoDB containers



(b) Throughput of Redis containers



(c) Memory usage plot

Figure 6.2: Plots for analysis when none of the containers are exceeding

3. The unstable throughputs observed in MongoDB and negatively impacted throughputs observed in the case of Redis can be accounted to the non-determinism of the GLR which to containers below SL as shown in Fig:6.1c.

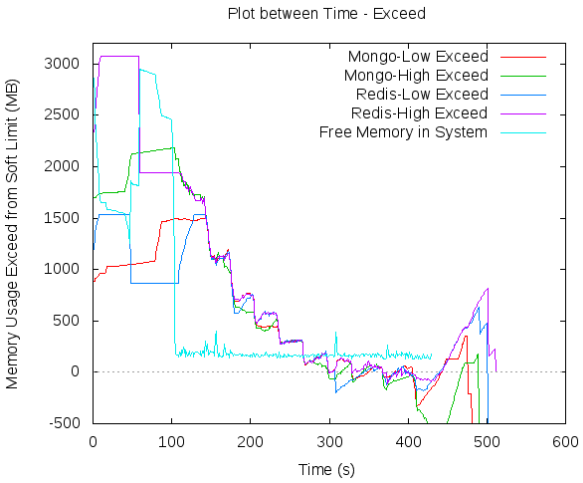
Inferences:

1. GLR based reclamation is non-deterministic which is highly undesirable quality while looking for deterministic provisioning.

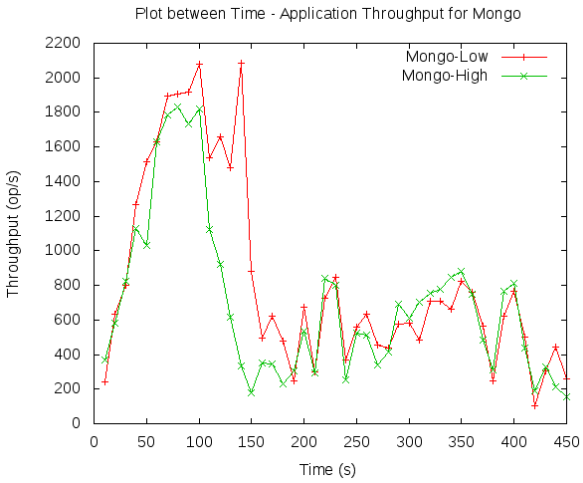
6.1.3 Complete reclamation

Question:

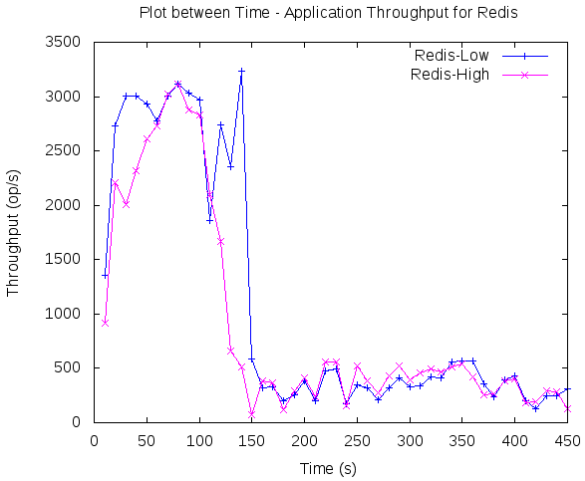
1. How does the entire process of reclamation after throughput of containers moving from all containers exceeding to none of them exceeding ?



(a) Exceed plot



(b) Throughput of MongoDB containers



(c) Throughput of Redis containers

Figure 6.3: Plots for analysis of complete reclamation starting from exceeding containers to when they aren't

Procedure:

We begin with containers configured as described in Table:6.1. 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:6.3b, Fig:6.2b 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.

6.2 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.

7. Requirements of new policy

The following are the list of requirements of new policy that we would like to enforce,

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. **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.
4. **Differentiated memory reclamation:** The policy could build around the notion of differentiated memory reclamation when the system falls under memory pressure.
5. **Strict enforcement of limits:** The notion of hard and soft limits that exist must be strengthened.

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

8. 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.

8.1 Future Work

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

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

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

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

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

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

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

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

[26] 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.