Elastic 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

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 [4] 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 Elastic Resource Provisioning for cloud services

Most commomly 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

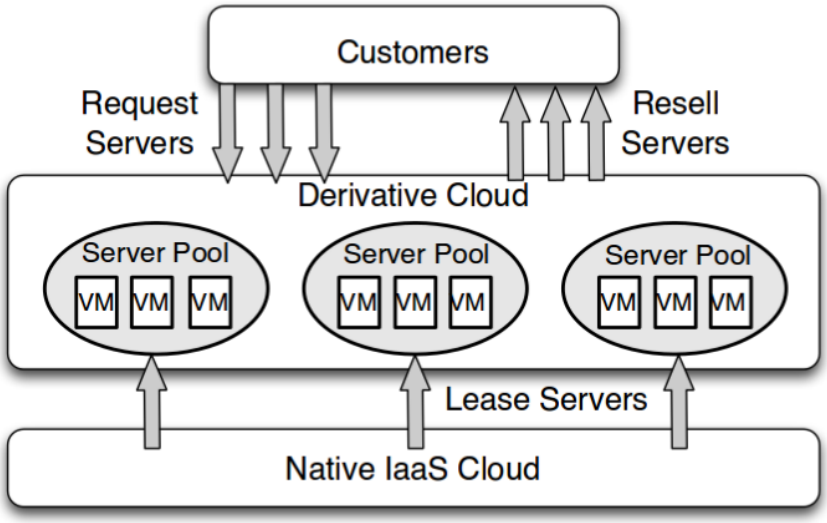


Figure 1.1: Depiction of a derivative IaaS cloud platform, Source:[1]

and servers are provisioned as per actual resource needs. A popular example for the same is Amazon's EC2 [5] 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. There are several advanced resource provisioning techniques [6] [7] [8] [9] 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 [10] [11] [12] [13] [14] 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 elastic provisioning of containers is still to be explored in depth. 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 elastic provisioning can be expanded to a *derivative cloud* environment as purposed by P.Sharma in his recent work [1] which repackages and resells resources purchased from native IaaS platforms. A derivative cloud can offer resources to customers with different pricing models and availability guarantees not provided by native platforms using a mix of resources purchased under different contract. Derivative cloud providers rent resources from native cloud providers to resell services to customers as shown in Fig:1.1. Pi-Cloud[15] and Heroku[16] are examples derivative clouds that offer batch processing service and PaaS offerings respectively by repacking native Iaas services.

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 elastic 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 using Linux Containers

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.

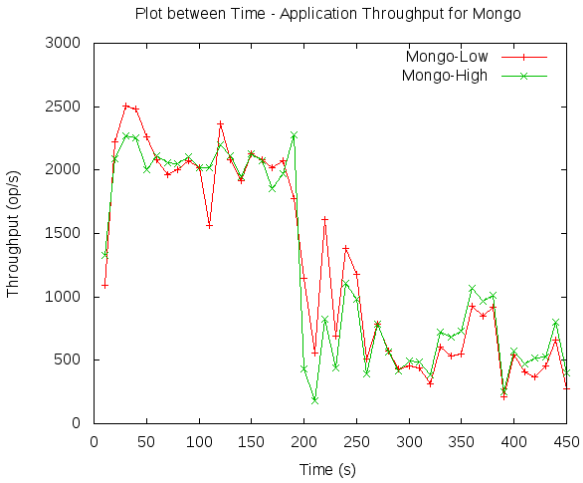
Illustrate existing problem

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) App throughput when running applications natively



(b) Observed throughputs after provisioning

(c) Desired throughputs after provisioning

Figure 1.2: Plots 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 are able to each equal throughputs initially in all 3 cases. 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. 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.

1.1.2 Amplification of problem 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,

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

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.

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 Statement

This report presents the preliminary results of an empirical evaluation of memory management patterns in the native container environment and how this is affects real application in a derivative cloud environment. It tries to understand the following questions

1. How is memory management done for Linux containers ?
2. What and when are the different types of memory management techniques invoked ?
3. How do container memory configurations impact application performance ?
4. What effect do the different container knobs have on the existing management policy ?
5. Do existing knobs work as desired ?

An comprehensive empirical evaluation which involves the above scenarios, and uses a set of workloads will help us understand the above questions. Once this has been established, we could move to purposing a solution to tackle the pitfalls of the existing approach.

1.3 Outline

This report is organized as follows, Chapter 2 provides the background required to read and understand the rest of the report, Chapter 3 presents the design of the empirical evaluation, and Chapter 4 presents preliminary results. Chapter 5 provides key insights to the desired system design. Chapter 6 concludes with the 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 [3] 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 [17], LXD [18], OpenVZ [19] 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

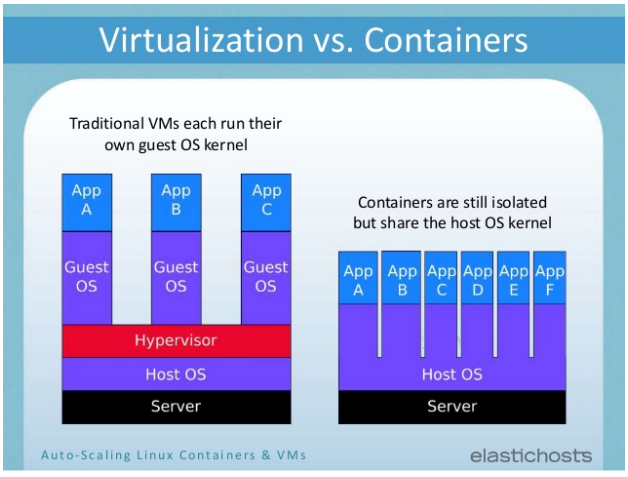


Figure 2.1: Difference between a VM and Container, Source:[2]

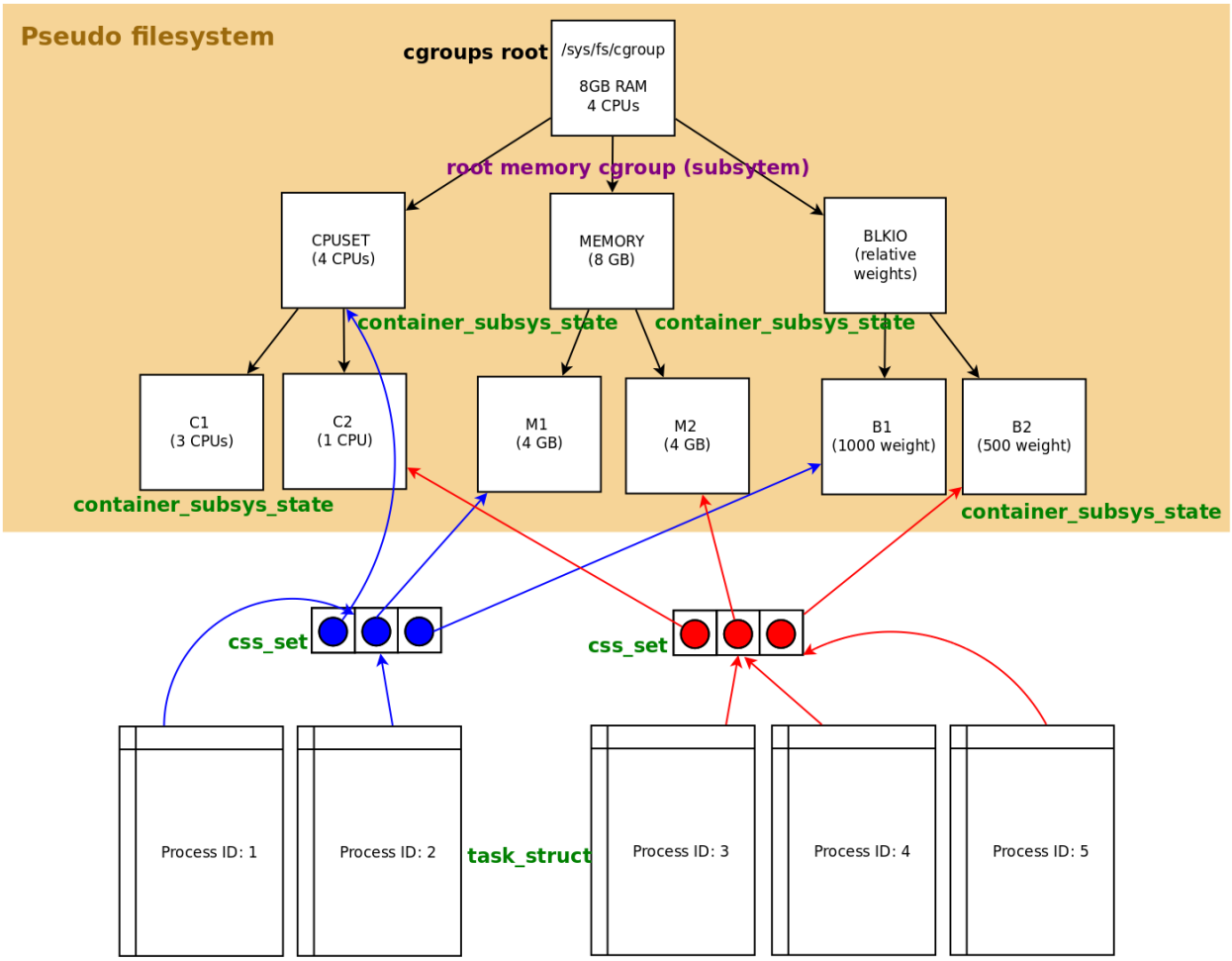


Figure 2.2: Control groups illustration using 3 controllers, Source:[3]

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 [20] 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.

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.

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.

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.

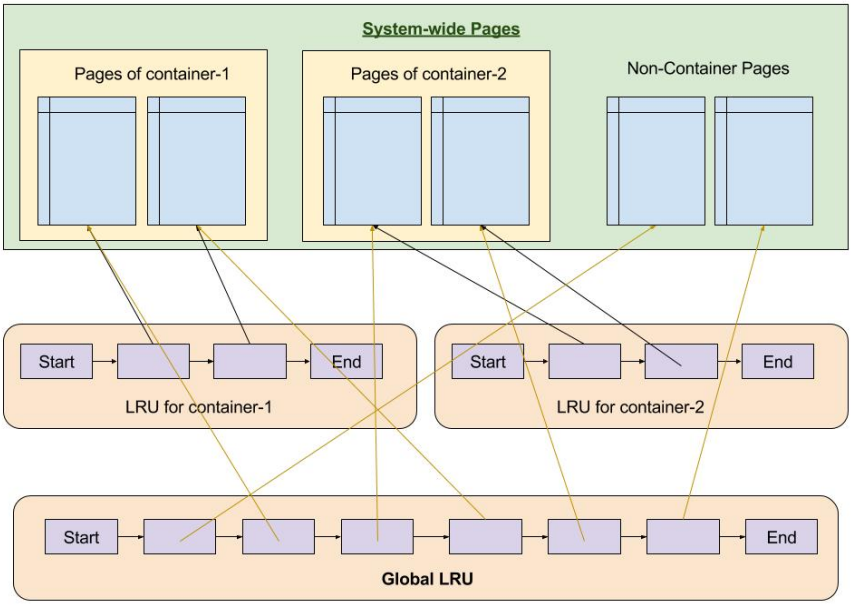


Figure 2.3: Mapping of pages to LRU lists

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.

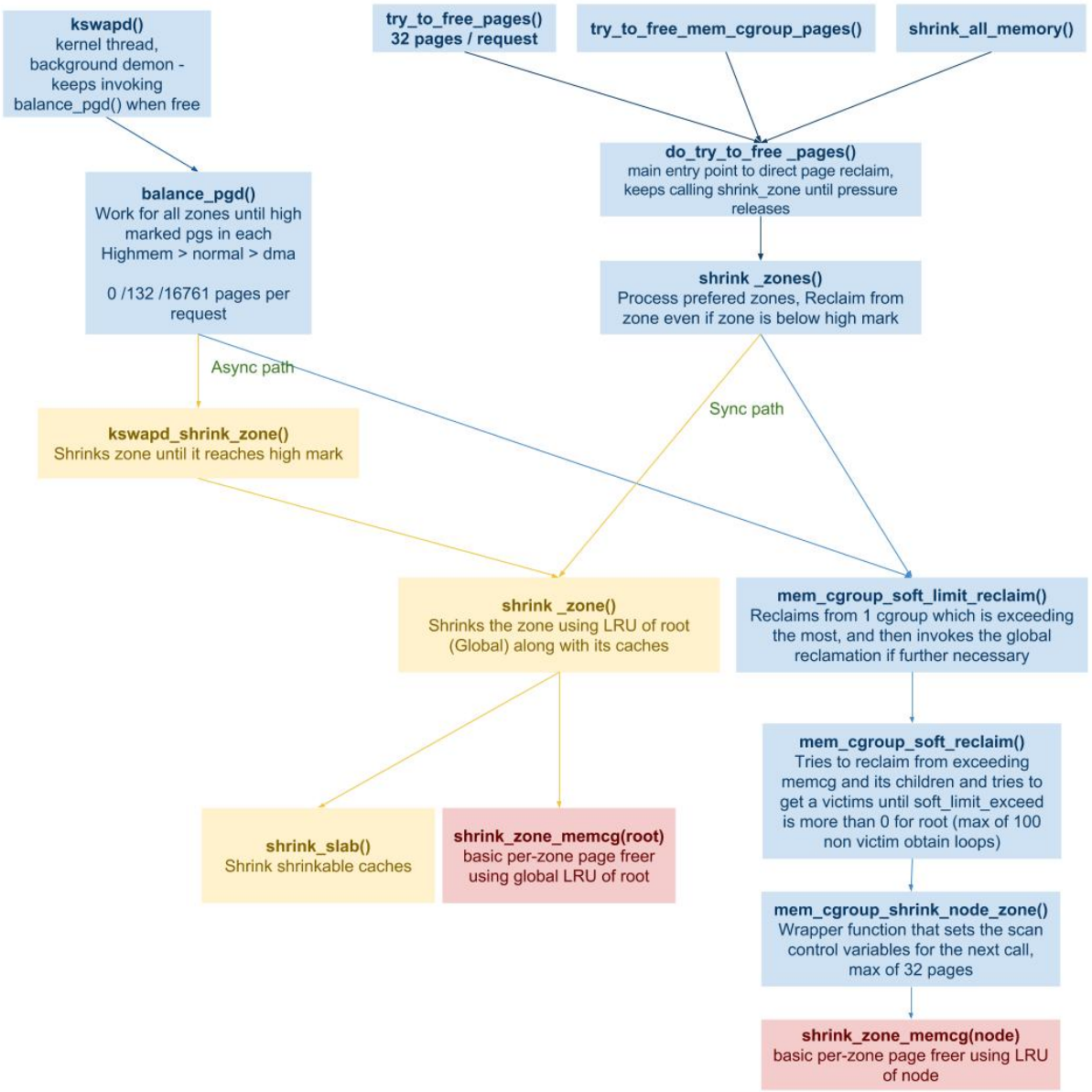


Figure 2.4: Kernel Function Call Trace for System-Wide Reclamation

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 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,

$$excess = usage - softlimit$$

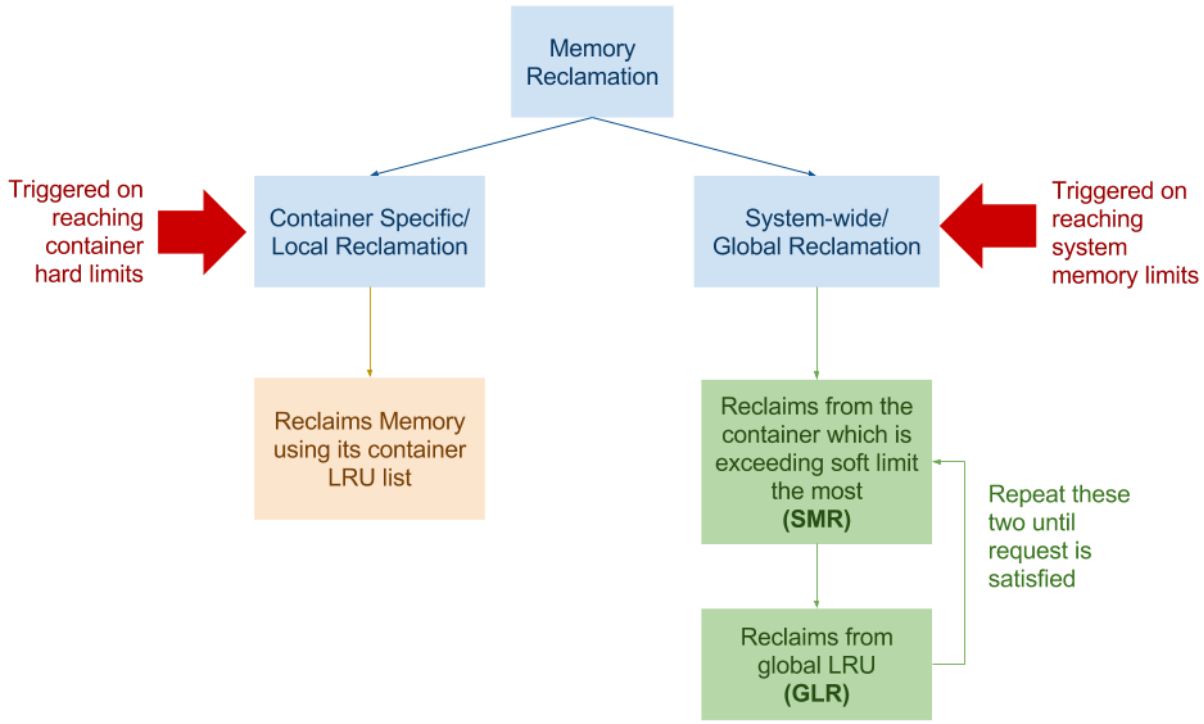


Figure 2.5: Existing policy for Memory Reclamation

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 section describes the empirical evaluations to do the same.

3. Design of Empirical Evaluation

3.1 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.

3.2 Experimental Configurations

As pointed out in section 3.1, 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.

3.3 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 3.1.

3.3.1 Synthetic Workloads

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

Stress

Stress [21] 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.

3.3.2 Real Workloads

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

MongoDB

MongoDB [22] 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 schemas. 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 [23] 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.

3.3.3 YCSB Benchmark

We use YCSB [24] (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.

3.4 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.

3.4.1 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. 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**.

Host

1. Intel Core i5-4430 processor @ 3.00GHz
2. 4 cores of cpu (with hyperthreading 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 hyperthreading 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

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.

Experimental Flow

Most experiments involved setting up of two 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 which created memory pressure in the system that triggered reclamation.

3.4.2 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. 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 hyperthreading 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 hyperthreading 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. Containers inside guest were multiplexed using NAT forwarding

VM-2: Running clients that connect to server containers

1. 1 core of pinned cpus (with hyperthreading 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 which intrun trigger memory reclamation at the guest VM.

3.5 Experimental Scenarios

The following are the possible experimental Scenarios under which memory pressure maybe generated. 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

3.6 Questions of interest

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

3.6.1 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 ?
3. When containers are exceeding by the same values, in what order and how much of memory reclamation occurs from different containers ?

4. When containers are exceeding by the different values, in what order and how much of memory reclamation occurs from different containers ?
5. How much of memory is reclaimed from a container in a single reclamation request ?
6. In what ratio does SMR and GLR vary based on container soft limits ?
7. In what ratio does SMR and GLR vary based on system memory pressure ?
8. Does setting containers with higher memory limits using existing memory reservations always guarantee higher priority to a one container over the other while reclamation ?
9. How is memory reassigned to containers when system pressure reduces ?

3.6.2 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 ?

3.6.3 A few containers exceed, but the others dont

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 ?

Experiments were designed to answer the above questions and experiments with key insights are discussed in the upcoming chapter.

4. Empirical Analysis

The list of questions mentioned in section:3.6 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 3 different categories of questions we have tried to answer using empirical analysis as listed below,

1. Verify the correctness of our hypotheses
2. Try to understand parts of memory management for which hypothesis couldn't be drawn
3. Understand the implications of existing memory management on application performance

For the sake of simplicity, questions of category 1 and 2 were answered using the native testbed using synthetic workloads as described in section:3.4.1. Questions of category 3 were answered using the derivative testbed using real workloads as described in section:3.4.2

4.1 Experiments to understand existing memory management in native environment

4.1.1 Experiment-1

4.1.2 Key Inferences

4.2 Experiments to understand memory management implications in a derivative cloud

4.2.1 Experiment-1

4.2.2 Key Inferences

4.3 Conclusions

5. Design for Proposed System

6. Conclusions

Bibliography

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

[2] R. Davies, "Linux containers: the future of iaas," 2016. http://www.slideshare.net/InternetWorld2014/19th-1400-elastic-hosts-richard-davies.

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

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

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] "Picloud." http://www.multyvac.com.

[16] "Heroku." http://www.heroku.com.

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

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

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

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

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

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

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

[24] 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.