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 [2] 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 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

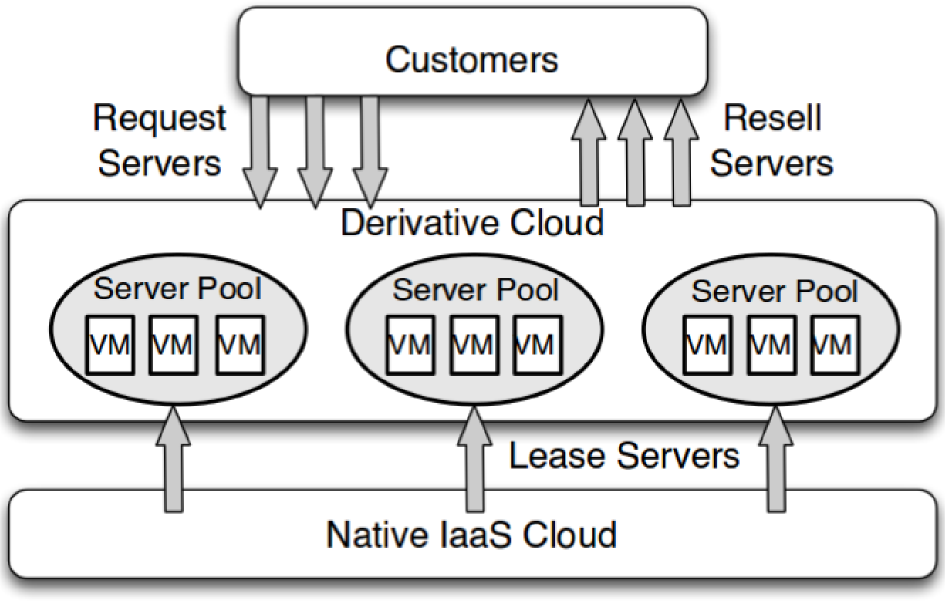


Figure 1.1: A depiction of a derivative IaaS cloud platform, Source:[1]

and servers are provisioned as per actual resource needs. A popular example for the same is Amazon's EC2 [3] 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 [4] [5] [6] [7] 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 [8] [9] [10] [11] [12] 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[13] and Heroku[14] 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 for 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

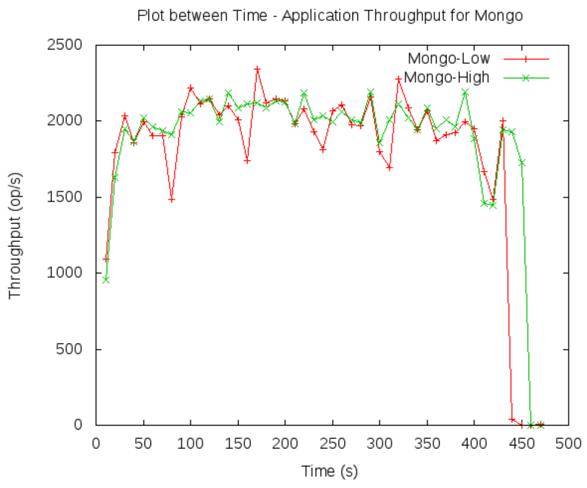
Let's see how memory overcommitment and pressure can disrupt desired functionality of the existing container framework provided by Linux containers.

Illustrate existing problem

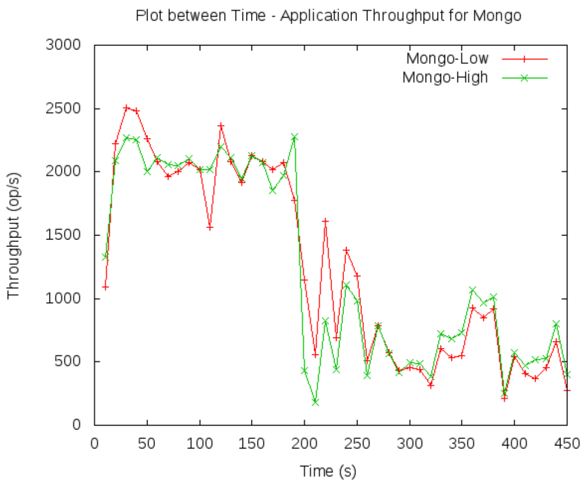
Consider two containers provisioned for running Mongo-DB containers from 2 different customers on a same host machines. Now that average memory used by the 2 containers are in the ratio of 1:2 and the customers for the 2 containers are also paying for their services in the same ratio. Let's call container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

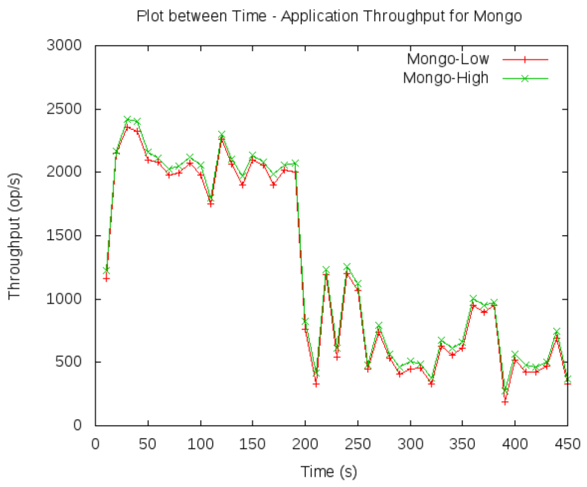
Table 1.1: Memory Provisioned for the Containers using existing memory provisioning knobs



(a) App throughput when running applications natively



(b) Observed throughputs after provisioning



(c) Desired throughputs after provisioning

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 with pressure
- Case-3: Desired throughputs with 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.

	Without 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 Considering the above setup to derivative cloud

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. Conisder 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 newly required design. Chapter 6 concludes with the future work.

2. Background

3. Design of Empherical 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 [15] 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 [16] 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 [17] 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 [18] (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.

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

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

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

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

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

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

[18] 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.