Adapative memory management framework for derivative clouds

Master's Thesis Report

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

Master of Technology

by

Prashanth
Roll No: 153050095

under the guidance of

Prof. Purushottam Kulkarni

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

Abstract

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

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

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

Contents

List of Tables

List of Figures

Introduction

Mention about derivative clouds in intro itself

Memory management in clouds

Issues in native container environment

Amplification of issues in derivative cloud environment

Caching in the cloud

Drawbacks of caching in native (VM) cloud setups

Hypervisor managed caching

Issues of caching frameworks in derivative clouds

Lack of framework support in derivative clouds

Dual layers of isolated control

Derivative provider has no control over cache partioning Cache-level sentivity

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

Application cache sensitivity is unaccounted

Problem description

Phase-1

Phase-2

Background

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.
3. Mapped pages: File page with VA mappings

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

Two lists

Containers

Control groups (Cgroups)

Memory Cgroup

Memory reclamation with Cgroups

Caching

Hypervisor managed caching

T-MEM cache

Multilevel caches

Application specific cache partitoning

MRC construction

Double decker: Second chance cache for derivataive clouds

Differentiated memory management controller for containers

Overview here

Drawbacks of existing memory management for containers

Issues in native environment

Reclamation above soft limits

Reclamation below soft limits

Amplification of issue in derivative clouds

Requirements for a new memory management controller

Proposed memory management controller

Controller architecture

Policies supported by the controller

Modifications made to Linux memory Cgroup

Per container configurable weights

Deterministic reclamation

Flexible reclamation size

Empirical evaluation of our controller

Effectiveness of our controller

Differential QOS containers

Impact of reclamation chunk size

Double decker: A memory management framework for derivative clouds

Application cache sensitivity

Provisioning of caches at different levels based on application sensitivity

Inability of cache partitioning framework to support anonymous memory applications

Rethink of existing design

Decentralized memory management framework

Native provider cache partitioning framework

Derivative provider memory management framework

Hybrid cache

Multilevel configurable caches

Movement of cache objects

Implementation details

Existing implementation status

Hybrid cache

Pools to accommodate both memory and SSD objects

Asynchronous kernel threads for movement of objects

Multilevel stats

Correctness of implementation

Experimental setup

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

Experimental configurations

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

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

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

Metrics of interest

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

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

The following metrics are collected both for memory and SSD cache

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

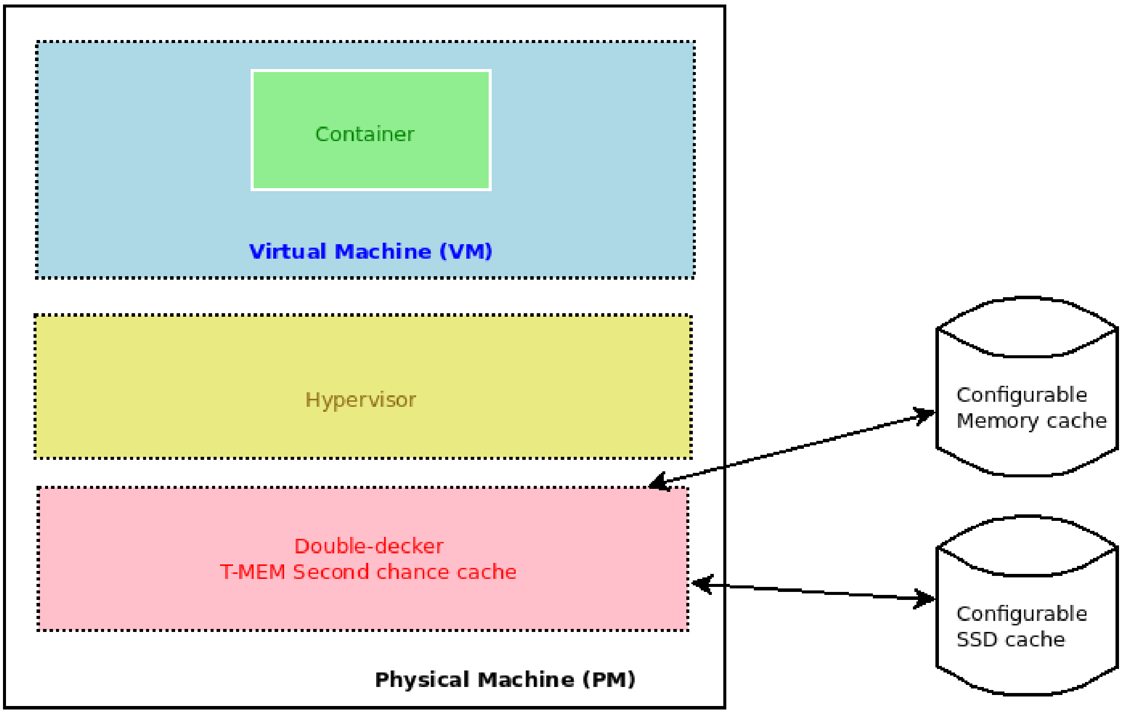

Figure 4.1: Experimental testbed for checking correctness

Workload

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

Testbed

Our testbed consists of a single VM, single container running on top of our hybrid implementation of Double decker as shown in Fig 4.1. The hypervisor used is KVM, and the container manager used is LXC.

The physical machine configuration used is as described below,

1. Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
2. 4 CPU cores (with multi-threading)
3. 8 GB of physical RAM
4. 120 GB SSD disk

Arithematic validation of stats

Movement of objects between both levels of cache

Memory to SSD cache

Question

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

Procedure

H

SSD to memory cache

Evaluation of Double Decker

Experimental setup

Experimental configurations

Metrics of interest

Workload

Testbed

Provisioning for anonymous and file backed workloads

Hybrid cache provisioning

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.

Future Extensions

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

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

Bibliography

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

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

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