Adapative memory management frameworks for derivative clouds

Master's Thesis Report

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

Master of Technology

by

Prashanth
Roll No: 153050095

under the guidance of

Prof. Purushottam Kulkarni

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

Abstract

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

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

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

Contents

List of Tables

List of Figures

Chapter 1

Memory management framework for derivative clouds

Our objective here is to provide a memory management framework for derivative clouds. In order to achieve the same we start off with understanding existing memory management and cache partitioning frameworks. We analyzed them to come up their drawbacks while provisioning for a derivative cloud setup. We made updates to an previous caching partitioning framework to make it a more full pledged memory management framework than just a cache partitioning framework. My initial work involved understanding the existing infrastructure and come up with drawbacks and make updates to the design to accommodate the downfalls. We came up with a revised design and implemented the same. We tested it for correctness and empirically evaluated it as well.

1.1 Drawbacks of existing framework

The following section tries to bring out the drawbacks of existing hypervisor cache partitioning frameworks and how fail to satisfy application SLA requirements. We demonstrate how an two-level exclusive (either level-1 or level-2) cache partitioning framework could fail to satisfy requirements and an intermediate partitioning framework would be desirable.

1.1.1 Experimental setup

The following section describes the experimental setup used to establish issues, verify correctness and evaluate our solution. Any changes made to the below setup, shall be mentioned beforehand.

Testbed
Our testbed consists of a single VM, single container running on top of our hybrid implementation of Double decker as shown in Fig 1.1. 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)

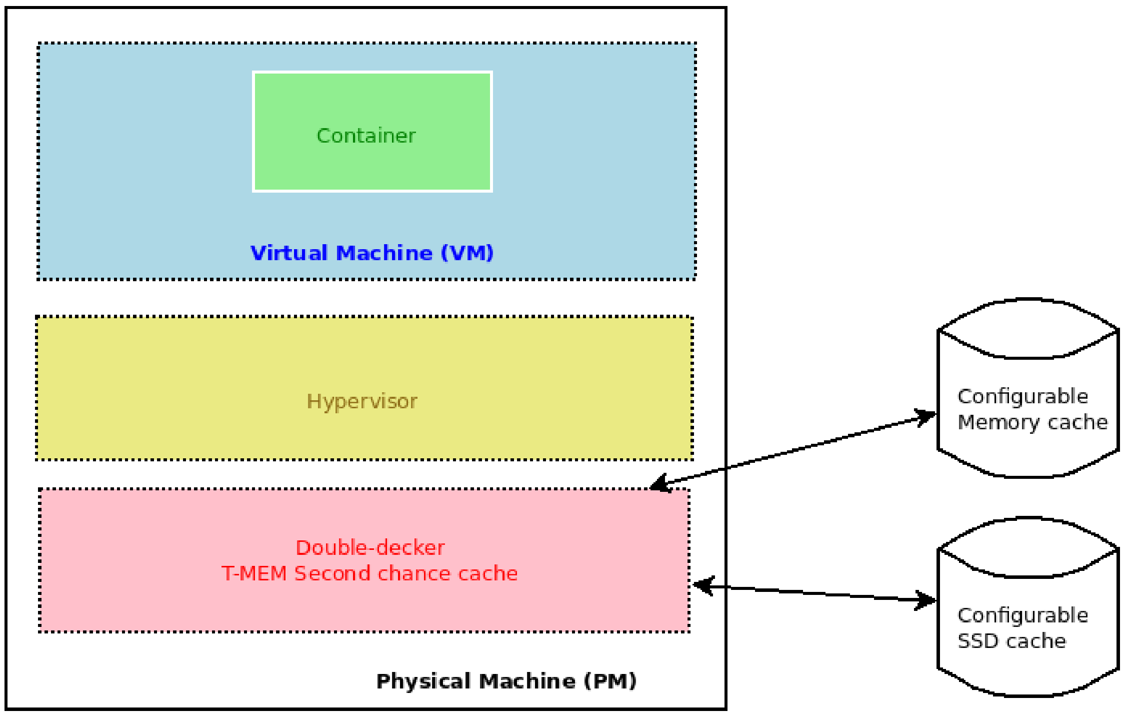

Figure 1.1: Experimental testbed for checking correctness

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

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.

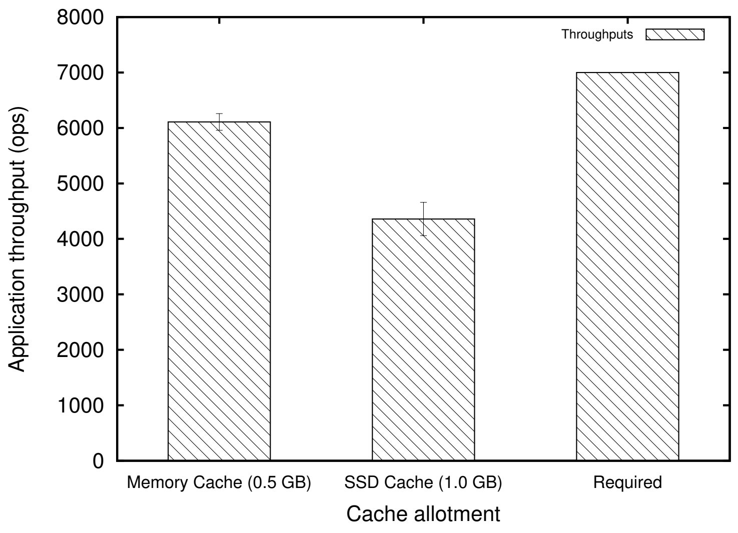

Figure 1.2: Inadequate exclusive two level cache provisioning framework

- **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.

1.1.2 Provisioning of caches at different levels based on application requirements

Previous results using Double-decker[4] have shown that certain application requirements (along with their configurations) could be satisfied better by provisioning on a cache of certain performance guarantees. In the previous work[4] it was shown how an application like Mail-server could be better provisioned on the level-2 (SSD) cache, as its requirement involved having a large WSS (working set size) with slow access rates. On contrary it also showed by cache sensitive applications like the Web-server workload had to be provisioned onto the level-1 (memory) cache. Now let's see if this setup of an exclusive two level cache provisioning schema would satisfy application specific need in all cases.

Inadequate exclusive two level cache provisioning

Consider the following scenario to demonstrate the short comings of the existing exclusive two level cache provisioning framework. Consider the case of a container with a WSS to provisioned onto the hypervisor cache with no-free memory to be further allocated at the guest VM and the hypervisor cache available is 0.5 GB at level-1 (Memory) and 1 GB at level-2 (SSD). Consider a Web-server workload where the desired application throughput is 7000 op/s.

Using our existing double decker cache, we can at max provision at 6000 op/s using memory cache of 0.5 GB as shown in Fig 1.2. But say our requirement was beyond that at 7000 op/s. Is there a workaround to satisfy this requirement ? Could we come up with a better design to satisfy this configuration ?

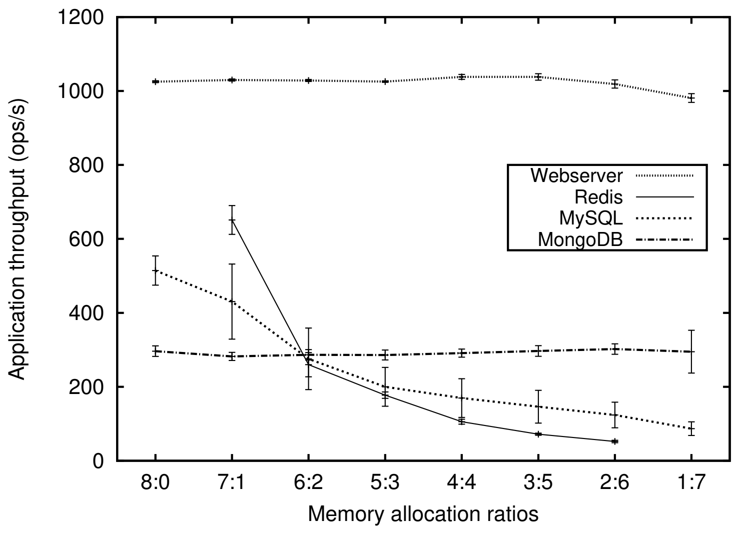

Figure 1.3: Split of memory allocation ratios (In-VM:Cache) to affect application performance

1.2 Inability of cache partitioning framework to support anonymous memory applications

Application memory requirement is vastly classified into two types - *anonymous and disk backed.* Application that require disk backed memory pages can be satisfied be either provisioning them at the VM(guest) memory or at the hypervisor cache. This effect is seen in Fig 1.3 where the memory requirement is split in ratios of in-VM:cache allocations. The plot shows how applications like Web-server and MongoDB which require disk backed pages can be satisfied either in-VM or at the hypervisor cache. Kindly note that the application performance is nearly constant in all cases of both these applications as the in-VM memory and the hypervisor cache operate at similar speeds in this case, and had there been difference in operational speeds, we would have observed degradation in application performance when moving from left to right in the plot.

Now, if you look at applications like Redis and MySQL, their performance is highly affected while provisioning them on cache, as they heavily depend on anonymous memory allocated to them (especially Redis). Hence such applications need to be provisioned at the guest, however existing cache partitioning frameworks [5, 6] can provision for disk backed workloads fail to address the needs of anonymous memory hungry workloads.

1.3 Rethink of existing design

1.3.1 Decentralized memory management framework

Native provider cache partitioning framework

Derivative provider memory management framework

1.3.2 Hybrid cache

Multilevel configurable caches

Movement of cache objects

1.4 Implementation details

1.4.1 Existing implementation status

1.4.2 Hybrid cache

Pools to accommodate both memory and SSD objects

Asynchronous kernel threads for movement of objects

Multilevel stats

1.5 Correctness of implementation

The experimental setup is just as described in Section 1.1.1. For establishing the correctness of our workload, we have considered a self generated synthetic workload generated using `cat` command that outputs the content of a file onto `/dev/null`. This workloads helps us to validate the correctness of our implementation by predicting deterministic outputs.

1.5.1 Arithematic validation of stats

Question
To verify the correctness in accounting of stats while accessing cache at both levels.

Procedure
We ran several experiments and computed the actual cache usage (memory and SSD) in our implementation. To an calculated an estimated cache usage we used the formula given below. We used the same formula for both memory and SSD cache.

$$EstimatedUsed = Puts + ObjectsMovedIn - (Gets + Flushes + ObjectsMovedOut) \qquad (1.1)$$

Observations
The values for *EstimatedUsed* and *ActualUsed* (present in out stat counter) matched in most cases. However, over long periods of run, with quite a large number of cache operations there was a marginal difference between the two (<1%).

Inference
Since the correctness of the actual value of cache used depend on the other stats that we have collected, and the matching of *EstimatedUsed* and *ActualUsed* would only mean that all the stats collected are right.

1.5.2 Movement of objects between both levels of cache

To verify the correctness of our implementation empirically, we have taken our synthetic workload described in above and ran a couple of simple experiments to demonstrate the expected behavior of our cache to support cache operations like puts, gets, promotions and demotions.

Memory to SSD cache

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

Procedure
We start of the experiment with powering on the VM, followed by the container. We assigned complete memory and SSD cache at the double-decker back-end to support the container. The container had a

Metric	Approx. estimated Value	Observed Value
Puts (MB)	2078	2080
Gets (MB)	0	1
Container memory usage (MB)	512	509
Memory cache usage (MB)	504	503
SSD cache usage (MB)	1008	1000
Evicts (MB)	54	64
Flushes (MB)	0	0
Cumulative usage (MB)	2078	2076
Demotions (MB)	1062	1064

Table 1.1: Comparison between expected and actual values

memory requirement of 2078 MB, 2048 MB workload requirement and 30 MB container requirement (container requirement was obtained by running the same experiment while having a nearly 0 MB workload. The container was allocated with 2048 MB (512 MB of container memory + 512 MB of memory cache + 1024 MB of SSD cache). Now, the workload performed a sequential read of its workload (i.e 2048 MB) once. Table 1.1 shows the list of approx. estimated values (which are based on our implementation) and observed values for the metrics at the cache at the end of the experiment.

Observations

The following are the observations,

1. There is a small number of gets, probably occurring due to pages used by other container applications.
2. Puts in the cache, is marginally (<2 MB) greater than expected value, and this deviation is due to the small number of Gets which are occurring.
3. The memory cache usage, is exactly as expected. However, SSD cache usage is slightly lesser than the expected value, but however the deviation seems to be an acceptable value.
4. The demotions (movement from memory to SSD) and cumulative usage values are nearly the same with a subtle deviation (<2 MB) which is an acceptable value.

Inference

The accounting stats, are nearly as expected. This verifies the correctness of most of the stats of our implementation (except promotions).

SSD to memory cache

Question

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

Procedure

We start of the experiment with powering on the VM, followed by the container. We assigned complete memory and SSD cache at the double-decker back-end to support the container. The container had a **memory requirement of 2078 MB**, 2048 MB workload requirement and 30 MB container requirement (container requirement was obtained by running the same experiment while having a nearly 0 MB workload. The container was allocated with 2560 MB (512 MB of container memory + 2048 MB of SSD cache). The workload performed a sequential read of its workload (i.e 2048 MB) once, this lead to using up of nearly 1566 MB of SSD cache.

Now, we changed the memory cache size to 256 MB while performing basic operations at the container which triggered the promotion (movement of objects from SSD to memory) of the objects to the memory cache. The promotion triggers all objects until the memory cache reaches a threshold usage - 192 MB in our case, as this threshold is calculated as,

$$MemoryCacheLowerThreshold(192MB) = MemoryLimit(256MB) - LimitSize(64MB) \tag{1.2}$$

Hence we would expect 192 MB worth of objects be promoted from SSD to memory cache in an ideal case.

Observation
Using our stats, it was observed that the estimated promotion and the actual promotion of objects were of an exact match with the number being 192 MB.

Inference
This verifies the correctness in the accounting stats in movement of objects from SSD to memory cache. Hence movement of objects from both memory to SSD and vice-versa have been verified empirically.

1.6 Evaluation of Double Decker

1.6.1 Experimental setup

Experimental configurations

Metrics of interest

Workload

Testbed

1.6.2 Provisioning for anonymous and file backed workloads

1.6.3 Hybrid cache provisioning

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] D. Mishra and P. Kulkarni, "Doubledecker: Differentiated hypervisor caching for derivative clouds," 2016.

[5] R. Koller, A. J. Mashtizadeh, and R. Rangaswami, "Centaur: Host-side ssd caching for storage performance control," in *Autonomic Computing (ICAC), 2015 IEEE International Conference on*, pp. 51–60, IEEE, 2015.

[6] J. H. Schopp, K. Fraser, and M. J. Silbermann, "Resizing memory with balloons and hotplug," in *Proceedings of the Linux Symposium*, vol. 2, pp. 313–319, 2006.