ADAPTIVE PROVISIONING OF HYPERVISOR CACHE MEMORY WITH NESTED CONTAINERS

M.Tech Project Report

Submitted in partial fulfillment of the requirements
for the degree of

Master of Technology

by

Kanika Pant

Roll No: 153050042

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 past decade has witnessed the emergence of Cloud Computing, a new paradigm for offering services which has dramatically changed and simplified the way consumers and businesses operate. Formally, cloud computing is based on the concept of offering computational resources as a service. This service can be in the form of software (software as a service or SaaS), a product (platform as a service or PaaS), or an infrastructure (infrastructure as a service or IaaS), or recently a new type of servicecalled Containers as a Service (CaaS)[ref].

In addtion to this general cloud platform, there exists an another version called *derivative cloud platform.*Derivative cloud platform repackages and resells resources purchased from native IaaS platform[ref].CaaS is something closely related to this platform.

Cloud service providers use server consolidation where one server is packed with many virtual machines or containers.This is to improve the utilization and power consumption of their servers.This server consolidation is possible with resource overcommittment.Of all the resources, memory is different as it cannot be scheduled like processor.Large memory chunks can be allocated and de-allocated over relatively larger time intervals than the processor.Managing memory efficiently in shared hosting platform is very important for over-commitment based provisioning.There are two types of memory usage—*anonymous* memory, applications request memory regions with system call interface and *page-cache memory*, improving the file access latency by caching the disk blocks.Efficiently managing the page cache pages.This can be done with additional caching layer for caching page cache pages.This single level caching also require efficient management.

1.1 Use-Cases of Derivative Clouds

It shouldn't be a question Derivative cloud is like a broker which leases the service from the native IaaS clod provider and resells them to customer.There folllowing diagram depicts the basic setup of derivative cloud.

There are good reasons for its usage:

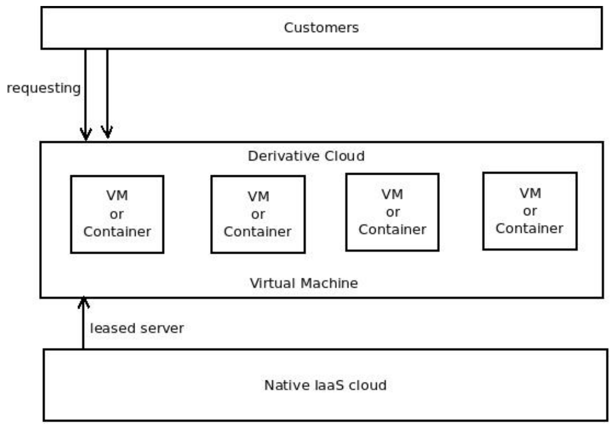


Figure 1.1: Virtual machines and conatainers in derivative clouds.

- One aspect is business.Buying at some price and then partitioning the machine in some more machines.Selling them to earn profit.There are companies which are doing it.PiCloud offers a batch processing service that enables customers to submit compute tasks. PiCloud[9] charges customers for their compute time, and is able to offer lower prices than on-demand servers by using cheaper spot servers to execute compute tasks. Similarly, Heroku[9] offers a Platform-as-a-Service by repackaging and reselling IaaS resources as containers.
- Cost of VMs does not scale with size [11].

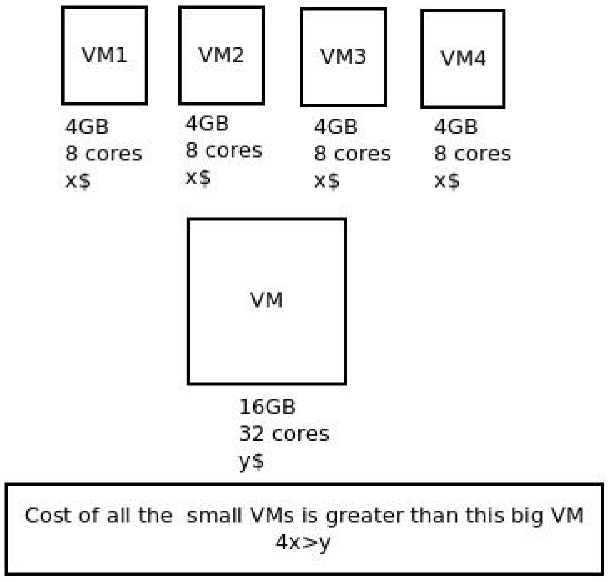


Figure 1.2: Cost difference between big VM and small VMs.

Therefore it is beneficial to buy one big VM than small VMs comprising of the same size as big VM.

- Another advantage which was recently seen[ref:spot-check].It says the derivative cloud design proves quite useful for balancing the tradeoff between cost and availability of different types of servers [9].
  The cloud business works with basically two types of servers namely:
  - On-Demand Servers:
    They work with a simplest type of contract in which a customer may request at any time and incurs a fixed cost per unit time of use. On-demand servers are non-revocable: customers may use these servers until they explicitly decide to relinquish them.
  - On-Spot Servers:
    They work with contrastic type of contract where they incur a variable cost per unit time of use as the cost fluctuates continuously based on the spot markets instantaneous supply and demand.On-spot servers are revocable: typically, a customer specifies an upper limit on the price they are willing to pay for a server, and the plat-

form reclaims the server whenever the servers spot price rises above the specified limit.

Therefore derivative cloud platform 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 contracts. The motivation for derivative clouds stems from the need to better support specialized use-cases that are not directly supported (or are complex for end-users to implement) by the server types and contracts that native platforms offer. Derivative clouds rent servers from native platforms, and then repackage and resell them under contract terms tailored to a specific class of user.

1.2 Single level Caching

Single level caching helps in minimizing disk I/O latency.This cache can be a in-memory cache or a SSD cache which will come in between the disk and the guest page cache.

1.3 Problem Description

The servers which are a part of derivative cloud come with different SLA(Service level agreement) from the native IaaS.An example scenario for this can be, suppose a customer buys a big virtual machine from a IaaS provider and that customer partition the virtual machines with several containers and resells those containers.The native IaaS provider has promised the VM customer with some resources.Further the VM owner also does some resource provisioning among the container owners. Resource management is an issue and this should be done dynamically.

Memory is an important resource and is usually overcommited in virtualized scenario.The scenario just discussed above can be also explained with respect to memory.Suppose native cloud provider sells a VM with 2GB memory.Memory is used for page cache and if facilitated can also be used as hypervisor in-memory cache in the single level caching configuration.The VM owner has his distribution of priorities and memory share percentages among the containers based on the service level agreements(SLA).These priorities and memory shares percentages can also change dynamically.

Therefore the problem is present unified hypervisor in-memory cache setup which cannot address the following issue:

- Service level agreements based on the priorities promised to the container customers.

1.4 Contribution

- Association of second chance cache with the containers
- Dynamically partitioning the hypervisor managed in-memory cache among the containers according to priorities

2. Related Work

There has been a substantial quantity of research directed towards understanding and managing cache efficiently.Cache can be used in different ways: unified , statically partitioned or dynamically partitioned.

These different use-cases of cache forms the basis for the efficient cache management for different scenarios.For our purpose, discussion would be about single level caching and multilevel caching.

2.1 Dynamic Partitioning of SSD based Cache

Literature for dynamic partitioning of SSD based cache already exists.Since dynamic partitioning can be done in several ways therefore there are different papers each with different way of partitioning.Partitioning can be done for fair allocation, for minimizing overall IO latency, for improving per VM performance, for increasing IOPS, or to satisfy service level agreement.The following papers [2], [6], [4], [8], [10] partition cache for different metrics.

2.2 Dynamic Partitioning with multilevel cache

Partitioning can be done at each level of cache.[10]

3. Background

3.1 Container Basics

3.1.1 Containers

Containers act as an alternative to Virtual machines.Virtual machine comparison is given to the containers because they provide OS based virtualization.This means from a single OS kernel, resources can be shared among multiple separate execution environment.Apart from resources, binaries and libraries may also be shared but if required each individual container can have them exclusively for their use.Because of the shared kernel and other resources the size of the software stack gets reduced.That is why we call containers as light-weight virtual machines. Thus these light-weight virtual machines require less processing, memory and storage resources and thousands of containers can be hosted on the single physical system.Another advantage would be rapid start-up and shut-down capabilities of containers. Containers can be better suited than virtual machines in various scenarios of:

- Low latency and lossless networking usage.
- Applications with fewer security requirements
- Implementations that involve only limited groups of users

3.1.2 Linux Containers(lxc)

Linux container or simply lxc is an open source project which work with Linux kernel to provide multiple containers on the same physical host.There are two features included in linux kernel which provide support for workload isolation in linux containers.Since these features are included in linux kernel therefore we call these mechanisms as core software based.Those two features are:

- Control Groups
  Cgroups[1] are collection of processes such that kernel gets a software based facility to control the subsystems such as memory, processor usage and disk I/O.Thus cgroupsgives the facility of resource control.this property can be exploited by the LXC.

- Namespace
  Linux Namespaces provides process groups their own isolated system view. Namespaces exist for different types of resources as in the case of cgroups.This allows individual containers to have their own IP addresses and interfaces.It also enables limitation to each control groups view for the physical or virtual resources such as details about file systems, processes, network interfaces, inter-process communications, host names, and user information.

3.2 Hypervisor Cache

Second chance cache, as the name suggests is giving second chance to page cache victim pages.The victim pages here are clean pages.This second chance cache can also be called as cleancache.The victim pages which get evicted from the page cache can cause refaults.
Refault occurs for disk request of a page which was earlier in the page cache and got evicted due memory pressure.
Second chance cache can potentially increase page cache effectiveness for many

workloads in many environments by reducing the number of refaults.Especially for virtual machines it can be quite useful.

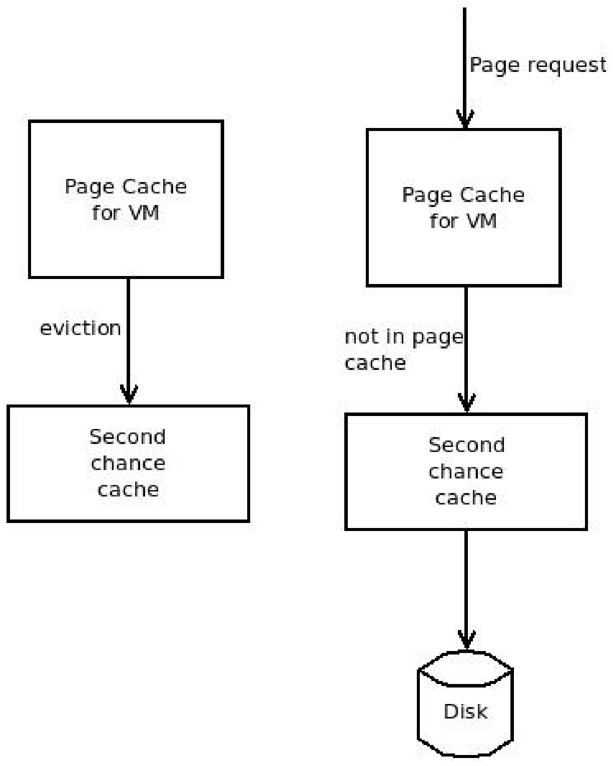


Figure 3.1: Architecture

3.3 Transcendent Memory

Transcendent memory (tmem) is used to utilize the memory so that it can improve the performance of applcations.Transcendent memory stores all free pages in one or more pool, these pages are managed by tmem host. The pages which are removed from the RAM (like disk pages and swap pages) are stored in Tmem in compressed form assuming they can be used in future after some time. Transcendent memory provided an API to indirect access to the memory used by tmem. Tmem client uses this API to indirect access to pages stored in the tmem pool. Tmem client can use tmem as extra memory which can be used to store disk and swap pages to save an I/O.

Now we have two types of memory which can be used

- Memory which can be traced by kernel, has fixed size and can be used by both system and user. In this kind of memory, each and every bytes can be read by used or kernel.
- Transcendent memory which has unknowable and dynamically varying size, which can not be directly accessed 'even from the kernel'. To access pages, user or kernel need to use API provided by tmem to access page.

3.3.1 Pools in Transcedent Memory

Transcendent memory stores pages in the pool for a tmem client. Tmem client can request to create a new pool and delete the pool using API's provided by tmem. The pages which are kep inside the pool are actually stored in idle memory which is not being used by kernel or user. Client have to create atleast one pool to save their pages in tmem. Pool can be accessed by its unique pool id.There are two important parameter passed by tmem client while creating a pool which defines the type of pool. Those parameters are as follows:

1. Shared vs Private: Shared pools are pools which may have pages shared between more than one clients. Currently shared pool is not being used because of security implications.
2. Ephemeral vs Persistent: Both private and Shared pools can be either ephemeral or persistent. A page successfully stored in ephemeral pool may lead to failed get. But in persistent pool a get after a successful put is always successful.

3.3.2 Transcedent Memory Frontend

Tmem API's can be used for storing any page and getting it back so there may be many possible frontends but there are exiting tmem frontends, 'frontswap'and ‘cleancache’. Two primary types of memory which is sensitive to memory pressure in systems are swap dirty pages and disk cache pages. Both these memory are covered by ‘frontswap’and ‘cleancache’. When the disk page is evected from memory in memory pressure than cleancache tries‘put’operation to store page in tmem to save one I/O. When page is swapped in memory pressure ‘frontswap’calls ‘put’operation to store page in tmem.

Cleancache

During a file intensive workload, the kernel loads pages from disk and, if RAM is free, kernel stores copies of such paged in memory called a disk cache because these pages may be used once in future and anyway the memory was free so in a way it utilizes the memory, it also saves one I/O if saved page is accessed again by workload. In disk cache, any clean pages are kept till lifetime but a dirty pages are flushed after a period of time. After a point of time, memory becomes full of clean “page-cache”pages which may or may not be accessed in future, now if workload want more pages,kernel reclaims pages by dropping data of some of pages. Now consider a case where kernel just drops a page, now it is again asked by workload then page must be again fetched from disk, this is called “refault”. Because kernel can not know which page is going to needed next, it may store some pages in disk cache which may never be used in future, and it may also result in often “refault”[5].

Cleancache is the approach which uses tmem to store clean disk cache pages, which leads to fewer refaults. When kernel reclaim the page, instead of dropping a data, kernel called a tmem put operation to store the page in tmem in “Ephemeral”pool. “Ephemeral”pool are used in cleancache which means pages can be discarded whenever tmem wants to. Later, if kernel want page stored in tmem, it will call tmem ‘get’operation with page id, page is searched in tmem pool by tmem, if found it is returned to kernel, if not found kernel fetches page from disk as usual (refault). “cleancache ”also ensures coherency between disk cache, disk and tmem data, cleancache 'put', 'get', 'invalidate'or other tmem functions are called while reclaim and refault is occurs.

Frontswap

In systems, total sum of physical RAM size and configures swap size(from disk) is known to be "virtual memory". The working set of the workloads may exceed physical RAM size, as soon as the physical RAM is full, one request of a new page, any older page from memory is swapped to swap area. Accessing pages from swap takes longer time because disk transfer is very less than memory to memory transfer. As the counts of swap operation increases for a workload, the performance is degraded. One solution is to increase the RAM so that kernel need never swap the pages on disk. Instead of spending cost on increasing RAM size, if we can make swapping faster than the performance will definitely increase.

Frontswap allows tmem to behave as swap. Linux swap subsystem instead of sending data to disk, first try to store data in 'persistent'tmem pool using 'put'operation. If it gets successful 'put'than it will save two I/O if that page is further required to be in memory by workload. If tmem reject a 'put'operation than data is stored in disk as usual. Once page is stored is guaranteed to retrieved from tmem using 'get'because of 'persistent'pool. The data of a process which has exited is also flushed from the tmem[5].

Unlike cleancache, in frontswap tmem can reject a 'put'operation if no more memory is available which can be given to that page or tmem exceeds it quota of using memory or may be because of its back-end constraints (like in zcache, in default configuration, if compressed size is greater than half of page size than it is rejected to store in tmem)

3.3.3 Transcedent Memory Backend

Currently multiple frontends can work with different backends but tmem allows only one backend to be configured. There are multiple backends introduces in recent years, all backends share similar properties. The operations should be synchronized, for instance incomplete "put"may not result to successful "get". All backends implementation should be tmem compatible with very less core kernel change. Some of the useful backends are as follows:

Zcache

Zcache is tmem backend which stores pages in compressed format in tmem, compressed format allows to reduce the memory requirement for data coming from frontends, which results in increase amount of disk cache pages ad swap pages to be stored in RAM, resulting decrease in significant disk I/O. Zcache is suitable to work with both frontend and cleancache or any other frontends in future. From the tmem interface,it supports both persistent and ephemeral pools. Pages for persistent store are obtained from xvmalloc, xvmalloc is a memory allocation for storing compressed pages in zram driver. Pages for ephemeral store are obtained from kernel get free page(), a standard function used in kernel to return a free page. To store data in that free page zcache compresses pair of pages using lzo1x kernel routines and matches with "compression buddies"algorithm.

Zcache stores pages which have compression ratio of two or more, if the size of compressed page is greater than half of size of uncompressed page than it is rejected to store in tmem. A zcache page may have exactly zero, one or two pages at a time[5].

RAMster

RAMster is “peer-to-peer”implementation of tmem. RAMster is useful for clustered systems connected with high-speed communication layer. RAMster allows collective RAM of all systems to be used as tmem.Here if one tmem client want to store a page in tmem, it may store its page in any system’s RAM, so for new ‘put’operation from tmem client, RAMster will search for a system where it can store that page and then stores it. RAMster improves the performance because at any point of time, each system in cluster may not be in memory stress, so the system which have free memory can be used by tmem resulting saving an disk I/O for disk cache page or swap page[5].

Xen Shim Layer

Xen shim layer allows virtual machines of XEN hypervisor to behave like tmem clients for host tmem. It allows to store swap pages and clean page cache pages to be stored in the hypervisor RAM instead of storing them in disks. This improves significant performance for virtual machine and also improves utilization of fellow memory. Shim layer allows to convert tmem front-ends call to Xen hypercall.

KVM Shim Layer

Like Xen shim layer, KVM shim layer also allows to store swap pages and clean disk cache pages in host tmem. Currently shim layer has been implemented to work with clean disk cache of virtual machine but for swap pages shim layer is not present in KVM. Experiments shows significant performance gain in disk I/O throughput in virtual machine while using KVM-tmem shim layer instead of disk cache in virtual machine[5].

3.4 Different cache configuration

We can have different cache configurations: unified, statically partitioned or dynamically partitioned.

3.4.1 Unified vs Static Partitioning

We can do the comparison with the help of an example.The example setup is such that there are 2 VMs with different type of workloads.
VM1 - workload which has less temporal locality but high IO request rates (100 blocks per sec).
VM2 - workload which has high temporal locality but IO request rate is moderate (50 blocks per sec).
Assume scheduling time is 1 sec for each VM and cache size is 100 blocks.
Case 1 : **Unified Cache** - When VM1 is scheduled it will cache 100 blocks occupying full cache.Next VM2 is scheduled which will replace 50 blocks of data of VM1 with its own data which is fine since its data blocks have better temporal locality than that of VM1. Next when VM1 is scheduled it will replace all the blocks in cache with its own data and the blocks with lesser locality replace blocks with higher locality. Next time when VM2 is scheduled it will get miss instead of hit for those replaced blocks in cache layer. This leads to higher IO latency since it decreases hit rate.
Case 2 : **Partitioned cache** - If cache is partitioned, for example 50 blocks to VM1 and 50 blocks to VM2 then performance will be better since VM1s blocks wont replace VM2s blocks.So we can say that cache partitioning is required for better hit rate. Also partitioning the cache gives hypervisors control to prioritize the VMs. For example if VM is of high priority then hypervisor can statically allocate more cache to it.

3.4.2 Static Partitioning vs Dynamic Partitioning

In the previous example we took into account only one aspect of the workload.But workload is associated with working set size.If a VM get a static cache allocation more than its working set size then there wont be any significant performance benefit as compared to cache size equal to working set size and there will be inefficient cache utilization.So if we partition the cache dynamically with changing working set size then it would give efficient cache utilization.[6]
Let us consider similar example as above.
Case 1 : static partitioning with VM1 - 50 blocks and VM2 - 50 blocks
Case 2 : dynamic partitioning which is based on temporal locality so VM1 - 5 blocks and VM2 - 95 blocks.
In case 1, VM1 wont be utilizing the whole cache since its working set size is only 5 blocks so no point in giving 50 blocks to it. So there is cache wastage of 45 blocks.
In case 2, VM1 gets sufficient amount of cache while VM2 can utilize the cache better since it has better locality of reference.

4. Design

In the previous chapters, we discussed why it is important to have an efficient memory management techniques or more specifically cache memory management techniques.Here, we will propose a possible solution design for the efficient cache memory management in nested containers setup.

4.1 Components

The following figure depicts the basic components of the design.The policy here is the priority based memory share of the containers[7].

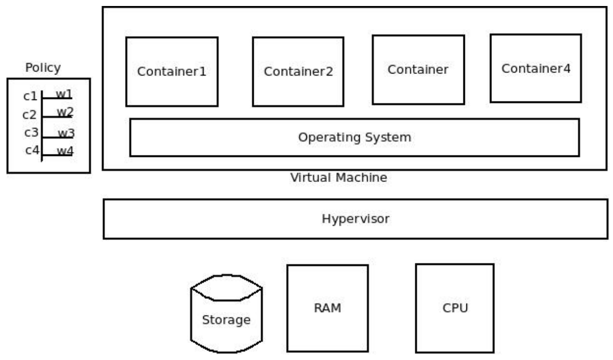


Figure 4.1: Policy for containers inside virtual machines.

4.2 Hypothesis

Propose a cache memory management architecture to dynamically partition the hypervisor cache among the containers inside the virtual machines.Following sections would give the design objectives.

4.3 Cache Partitioning

Cache partitioning is the first objective of the design.Cache should get partitioned among the containers.Each container should have its own partition.

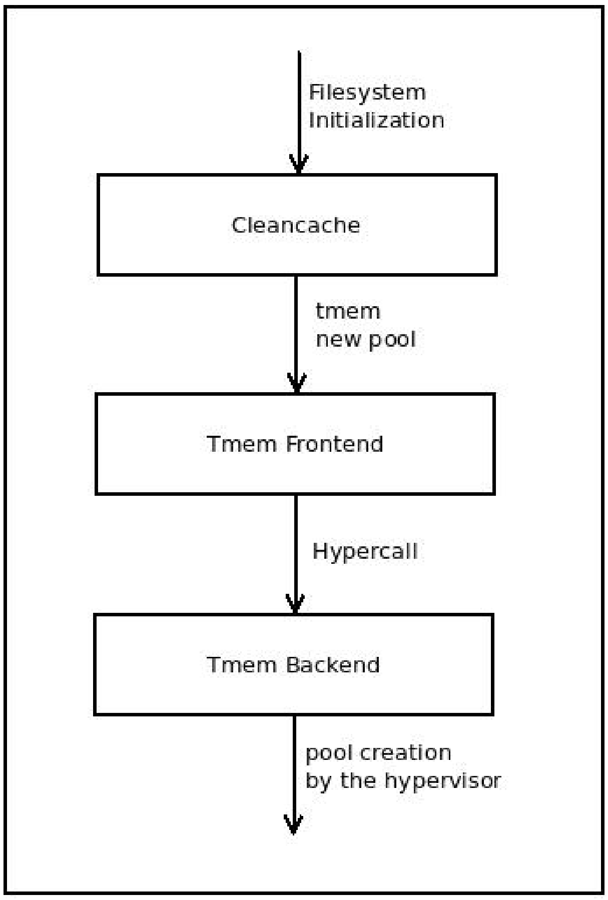


Figure 4.2: New Partition Creation for containers.

4.4 Basic Cache Functions

Another objective of design should be to handle the basic cache functions.Since we call this cache as second chance cache so the basic functions involve:

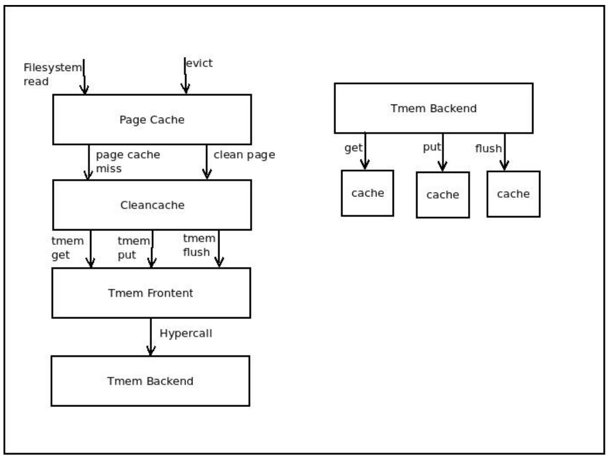


Figure 4.3: Basic cache functions for improving performance

- Whenever there is a page eviction from page cache.It should be copied to this hypervisor cache partition of that respective container.
- Whenever there is a read miss for page cache then it should be forwarded to the hypervisor cache partition and should be searched there.
- Coherency should be maintained between the backend,the page cache, and the filesystem of the partitions.

4.5 Partition Size Adjustment dynamically

Now when we have partitioned our cache and provided it with basic functionalities, there remains the need for last objective.This last objective is to dynamically change the cache partition size.

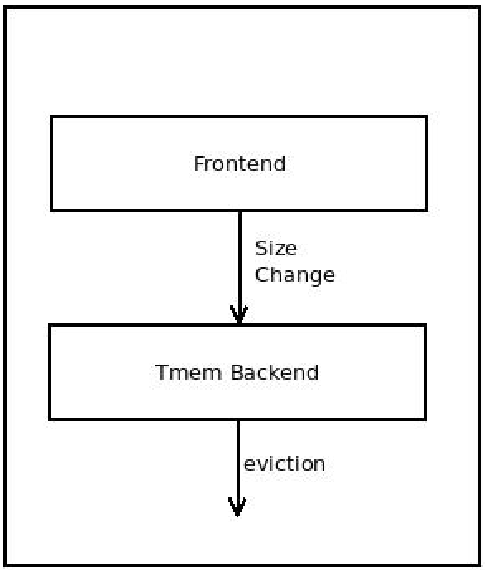


Figure 4.4: Cache Partition resizing according to policy.

5. Implementation

In order to implement the design goals of previous chapter we implement a tmem backend.But before getting into actual implementation, we should briefly go through the components needed to implement the backend module.

5.1 Cleancache Interface

Linux API for cleancache consists of the following functions[3]:
—cleancache_init_filesystem
—cleancache_init_shared_filesystem
—cleancache_put
—cleancache_get
—cleancache_invalidate
—cleancache_invalidate_inode
—cleancache_invalidate_filesystem

5.2 KVM tmem Hypercall API

This is hypercall API which is responsible for making the host manage the hypervisor cache.

—kvm_tmem_new_pool

—kvm_tmem_destroy_pool
—kvm_tmem_put
—kvm_tmem_get
—kvm_tmem_flush_page
—kvm_tmem_flush_object

The above APIs makes the messages from the guest to reach host so that the backend works properly.The backend functions in the following way:

- Different partitions are created for the containers
- These partitions are called pools
- Each partition is associated with pool_id
- The partition is uniquely identified with this pool_id
- Any operation performed on cache should have handle to access the cache.
- This handle comprises of pool_id , object_id and index
- The cache contents are managed through efficient data structures.

5.2.1 Data Structures

Data structures for storing the content from the page cache are:

- RB-Tree for storing the file objects
- Radix tree per file object

5.3 Current Implementation

We will look into four aspects, one is partitioning the cache, second is cache lookup for each page cache miss, third is page cache eviction handling and last is sizing the cache pool/partition dynamically.

5.3.1 Partition Creation

New pool is created when the filesystem for the container is mounted.This partition creation returns a pool_id which is passed back to the guest and the pool_id is registered to the superblock of the container's filesystem.

5.3.2 Search Function

The above mentioned handle(pool_id,object_id,index) is used to search the cache for the requests which got missed at the page cache level.The search involves searching of the rb-tree for the object and then the radix tree for that object.

5.3.3 Put function

The same handle is used for putting the page which was evicted from the page cache.

5.3.4 Dynamic sizing

Whenever container limit is reached.Eviction is done for that pool.

6. Experiments

6.1 Set Up

We have performed these experiments on a machine running LINUX with kernel version 4.1.3 and 8GB RAM. On this physical machine, we have configured two virtual machines (VMs) with four containers each.Both the virtual machines have same configuration.Configuration of the virtual machine is depicted in the following table.

CPU cores	2
RAM	1GB
Number of containers	4
OS	Ubuntu
Disk	70GB

Table 6.1: Virtual machine configuration for conducting experiments

6.2 Correctness Experiment

This experiment was performed to check if different pools are created for each container and further see whether the cache operations are performed consistently performed or not. To achieve our goal following steps are performed:

1. Firstly we need to install backend module as explained in Section 5.3. Then we will start the containers in our virtual machine.

2. After their start, initially we will perform a file read operation. If this file is read for the first time then its content will not be in the page cache. Therefore, this miss of the page cache will be forwarded to the respective pool of the container in the Hypervisor Cache. This lookup fails as the pool is empty for that time and the file is read from disk.

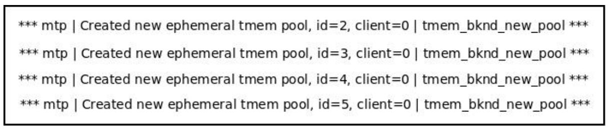
3. The purpose of this file read operation is to create memory pressure so that page cache for VM should evict the previously cached pages. This is done so that the evicted pages get cached in the respective pool/partition of container.

4. The cache operations are consistent as when a page is cached in the Hyper-ivisor cache pool then on requesting that page, it becomes a hit operation.

There is also an eviction function in the backend module which is right now tested with the help of sysfs interface. This interface will prompt the eviction function to evict the pages from a specific pool as per requirement. This eviction function can be used in future for dynamic sizing of the tmem pools/partitions.

Following images of the log would give the results of the experiments briefly :

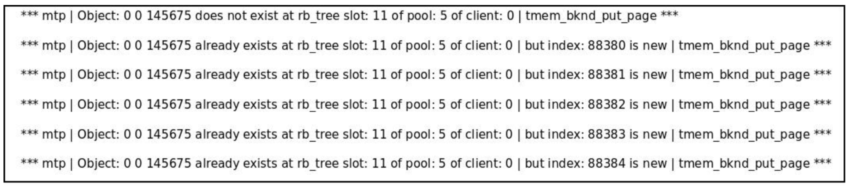
- New pool/partition creation for four containers information from the log



```
*** mtp | Created new ephemeral tmem pool, id=2, client=0 | tmem_bknd_new_pool ***
*** mtp | Created new ephemeral tmem pool, id=3, client=0 | tmem_bknd_new_pool ***
*** mtp | Created new ephemeral tmem pool, id=4, client=0 | tmem_bknd_new_pool ***
*** mtp | Created new ephemeral tmem pool, id=5, client=0 | tmem_bknd_new_pool ***
```

Figure 6.1: New pool creation log information.

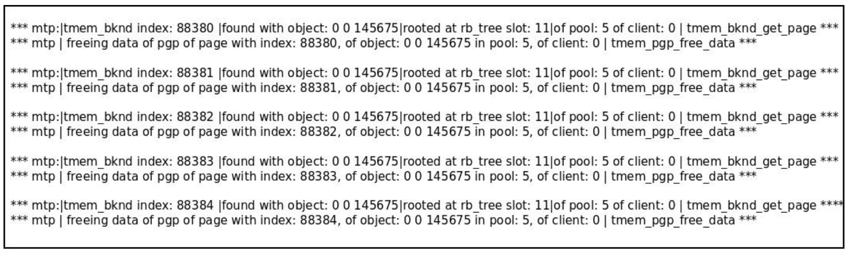
- This figure shows the five indices of the same object for container 4 being stored in the radix tree for that object.Similar happens for objects of other pools.



```
*** mtp | Object: 0 0 145675 does not exist at rb_tree slot: 11 of pool: 5 of client: 0 | tmem_bknd_put_page ***

*** mtp | Object: 0 0 145675 already exists at rb_tree slot: 11 of pool: 5 of client: 0 | but index: 88380 is new | tmem_bknd_put_page ***

*** mtp | Object: 0 0 145675 already exists at rb_tree slot: 11 of pool: 5 of client: 0 | but index: 88381 is new | tmem_bknd_put_page ***

*** mtp | Object: 0 0 145675 already exists at rb_tree slot: 11 of pool: 5 of client: 0 | but index: 88382 is new | tmem_bknd_put_page ***

*** mtp | Object: 0 0 145675 already exists at rb_tree slot: 11 of pool: 5 of client: 0 | but index: 88383 is new | tmem_bknd_put_page ***

*** mtp | Object: 0 0 145675 already exists at rb_tree slot: 11 of pool: 5 of client: 0 | but index: 88384 is new | tmem_bknd_put_page ***
```

Figure 6.2: Log of cached object indices for a particular container.

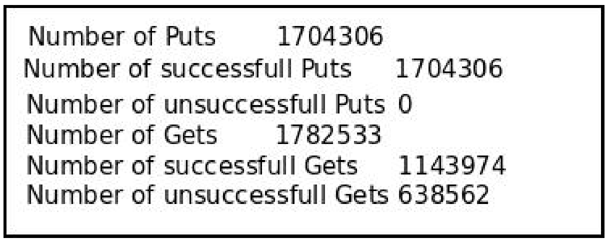
- This figure shows the five indices of the same object for container 4 retieved from the radix tree for that object and gets removed from the cache.Similar happens for objects of other pools.



```
*** mtp:|tmem_bknd index: 88380 |found with object: 0 0 145675|rooted at rb_tree slot: 11|of pool: 5 of client: 0 | tmem_bknd_get_page ***
*** mtp | freeing data of pgp of page with index: 88380, of object: 0 0 145675 in pool: 5, of client: 0 | tmem_pgp_free_data ***

*** mtp:|tmem_bknd index: 88381 |found with object: 0 0 145675|rooted at rb_tree slot: 11|of pool: 5 of client: 0 | tmem_bknd_get_page ***
*** mtp | freeing data of pgp of page with index: 88381, of object: 0 0 145675 in pool: 5, of client: 0 | tmem_pgp_free_data ***

*** mtp:|tmem_bknd index: 88382 |found with object: 0 0 145675|rooted at rb_tree slot: 11|of pool: 5 of client: 0 | tmem_bknd_get_page ***
*** mtp | freeing data of pgp of page with index: 88382, of object: 0 0 145675 in pool: 5, of client: 0 | tmem_pgp_free_data ***

*** mtp:|tmem_bknd index: 88383 |found with object: 0 0 145675|rooted at rb_tree slot: 11|of pool: 5 of client: 0 | tmem_bknd_get_page ***
*** mtp | freeing data of pgp of page with index: 88383, of object: 0 0 145675 in pool: 5, of client: 0 | tmem_pgp_free_data ***

*** mtp:|tmem_bknd index: 88384 |found with object: 0 0 145675|rooted at rb_tree slot: 11|of pool: 5 of client: 0 | tmem_bknd_get_page ****
*** mtp | freeing data of pgp of page with index: 88384, of object: 0 0 145675 in pool: 5, of client: 0 | tmem_pgp_free_data ***
```

Figure 6.3: Retrieving the cached objects from the pool.

- This figure shows the total number of each storing and retrieving operations performed on the cache.



```
Number of Puts        1704306
Number of successfull Puts      1704306
Number of unsuccessfull Puts 0
Number of Gets        1782533
Number of successfull Gets      1143974
Number of unsuccessfull Gets 638562
```

Figure 6.4: Total cached and retrieved objects from all the pools.

7. Conclusion and Future Work

In this project, we have understood the tmem functioning and implemented basic tmem backend which performs functions as creation of new pool, putting the page cache evicted page to the respective pool of second chance cache, getting the missed page cache page from our second chance cache if present, random policy page eviction.
In near future, we intend to implement dynamic sizing of the pools/partitions with the help of an eviction function.

Bibliography

[1] Neil Brown. Control groups series. 2014.

[2] Jing Xu Swaminathan Sundararaman Dulcardo Arteaga, Jorge Cabrera and Ming Zhao. Cloud-cache: On-demand flash cache management for cloud computing. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies(FAST)*, 2016.

[3] Dan Magenheimer. Chris Mason. Dave McCracken. Kurt Hackel. Transcendent memory and linux. In *Ottawa Linux Symposium (OLS)*, 2009.

[4] Ron C. Chiang Timothy Wood Jinho Hwang, Wei Zhang and H. Howie Huang. Unicache:hypervisor managed data storage in ram and flash. In *Proceedings of the 7th International Conference on Cloud Computing.*, page 216223, 2014.

[5] Dan Magenheimer. Transcendent memory in a nutshell. 2011.

[6] Ma Xiaosong Uttamchandani Sandeep Meng Fei, Zhou Li and Liu Deng. vcacheshare : Automated server flash cache space management in a virtualization environment. In *Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference*, pages 133–144, 2014.

[7] D. Mishra and P. Kulkarni. Comparative analysis of page cache provisioning in virtualized environments. In *2014 IEEE 22nd International Symposium on Modelling, Analysis Simulation of Computer and Telecommunication Systems*, pages 213–222, 2014.

[8] R. Koller A. J. Mashtizadeh R. Rangaswami. Centaur:hostside ssd caching for storage performance control. In *Proceedings of Autonomic Computing, 2015 IEEE International Conference on cloud computing,*, page 966973, 2015.

[9] Prateek Sharma, Stephen Lee, Tian Guo, David Irwin, and Prashant Shenoy. Spotcheck: Designing a derivative iaas cloud on the spot market. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, 2015.

[10] Y. C. Tay Vimalraj Venkatesan, Wei Qingsong and Yi Irvette Zhang. A 3level cache miss model for a nonvolatile extension to transcendent memory. In *Proceedings of the 6th International Conference on Cloud Computing Technology and Science*, page 218225, 2014.

[11] Dan Williams, Hani Jamjoom, and Hakim Weatherspoon. The xen-blanket: Virtualize once, run everywhere. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 113–126, 2012.