Differentiated SSD Caching for Container-based Hosting

Master Thesis Project Report

submitted in partial fulfillment of the requirements for the degree of
Master of technology
by

Shyamli Rao
Roll Number : 153050009

Under Guidance of Prof. Purushottam KulKarni

Department of Computer Science and Engineering
I.I.T. Bombay

June 25, 2017

Abstract

Container virtualization is an emerging technology in the area of cloud computation. Containers are increasingly becoming popular amongst cloud providers since they are light weight as compared to virtual machines. Usually cloud providers have to satisfy the service level agreements of clients. Disk block caching is a resource management technique used by cloud providers to improve IO performance of the applications and hence contributing to satisfying SLAs in virtualised setup. Disk block caching is done by provisioning non volatile memory such as Solid State disks (SSDs) as second level cache to virtual machines. SSDs are preferred as caching devices since they are faster than hard disk and are cheaper than volatile memories such as RAM. The second chance cache can be utilized in different ways depending on position (caching at server side or at host side), caching policy (inclusive or exclusive), caching method (sharing or partitioning), eviction policy used.

In this report, we have extended the provisioning of SSD as second chance cache to container based virtualised environments and applied similar caching policies to second chance cache. The results show that there are certain cases where partitioning SSD cache helps improve cache hit ratio and thus improves the overall IO latency of applications running.

Contents

List of Figures

List of Tables

1 Introduction

Container virtualization is an emerging technology in the area of cloud computation. It allows multiple isolated instances to share system resources such as CPU, memory, disk IO, network etc. A container runs a set of processes in an isolated environment which is allocated a bunch of resources by host operating system. The host OS should be aware of various techniques for allocating and managing the resources such as CPU, memory, storage, network etc. across all containers. Host OS has to multiplex these resources amongst the containers in an efficient way. Here we are mainly concerned with storage management.

Modern data centers consolidate more workloads to reduce hardware and maintenance costs in virtualised environment. Storage performance of workloads under contention is poor due to high disk latency [4], [20]. Sometimes data is stored in network attached storage (NAS), therefore every disk IO request has to go through network which increases IO latency. One of the ways to deal with this problem is provisioning a non volatile memory as a second level cache between page cache and disk. For years, the increasing popularity of flash memory has been changing storage systems. Flash-based solid-state drives are widely used as a new cache tier on top of hard disk drives (HDDs) to speed up data-intensive applications [2]. Caching the blocks at host side in data centers will incur better IO performance than non caching systems.

1.1 Motivation

Caching helps in minimizing disk IO latency. As shown in Figure 1.1, application initiates an IO request. First OS checks if data is in it's page cache, if not goes to the disk. It will fetch the blocks and give it back to the application. So if there is relevant data in page cache, then every IO request must goto disk and disk access time is much higher than RAM access time.

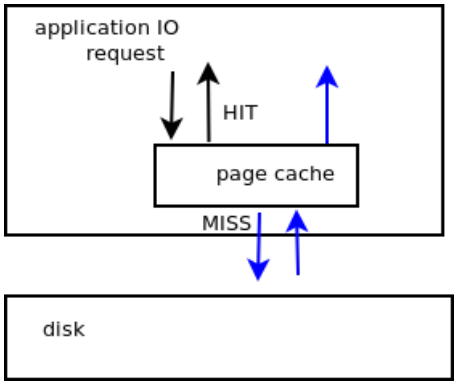


Figure 1.1: IO request flow

Using non volatile memory as a second chance cache will improve the IO latency but when we expose SSD as second chance cache we must consider following points:

1. How can NVM be provisioned for containers? Various caching techniques can be used like single level caching, multilevel caching, caching with partitioning or without partitioning.

2. Which data should be inserted or evicted from cache? We should cache the data which is frequently used by the applications. The data kept in cache will consume a memory resource and if it is going to get replaced sooner then cost of eviction will also increase. If NVM's are used as cache then wearing of cache is also a problem. So the data to be inserted or evicted from cache must be chosen wisely.

3. Which type of caching technique to use? There are two types of caching techniques : inclusive and exclusive caching. In inclusive caching, a block is copied to all the layers of cache above disk. In exclusive caching, block is present in only one of the cache layers.

4. Which write policy to use? There are different types of write policies - write back, write through and write around. The selection of write policy depends on several factors such as workload (read intensive or write intensive), cache configuration, disk latency.

IO latency is high due to slower devices like hard disk. Hence use SSD a faster device as cache to cache the disk blocks. This is already much explored area in virtualised environments but is not explored for containers. This type of framework i.e. exposing SSD as cache to disk is not yet designed for container based virtualised setups.

1.2 Problem description

We need to design a framework which would expose a NVM as a cache to hard disk. Also we have to explore which eviction policy works the best, which caching technique and write policy to use. We did some research in this area and found out that the usage of NVM especially Solid State Devices (SSDs) as caching layer has already been explored area for VMs [6], [4], [1]. But for containers such type of framework is not yet designed. In VMs, hypervisor has control over the SSD. Hypervisor used to predict each VMs requirements and eviction and allocation of blocks. But in case of containers host OS will do all this and the advantage is that host has more knowledge of access patterns than hypervisor (since host OS can look through page cache but in case VMs, hypervisor has no clue of eviction algorithms of guest VM and its page cache).

1.3 Organization of the work

The project work is planned and organized into following steps:

1. Exploring various usecases of persistent memory as byte addressable memory and as a storage device in virtualised environments. Understanding internals of containers, Linux block IO layer, IO request flow from applications to devices.

2. In the implementation thread, study existing code of block level caches such as dm-cache, flashcache and how it can be used for fulfilling our requirements.

3. Implement a layer between file system and device driver (within block IO layer) which can be used to partitioned SSD on per container basis statically. Extend the same to partition SSD dynamically such that a user can change the SSD size of container whenever needed. Implement a module for cache size prediction for each container according to the workloads running on each container such that the dynamic partitioning is automated without human intervention.

4. Doing extensive experimental evaluation of implemented partitioner and verify if the results are coming as expected.

2 Background

2.1 Containers

In traditional virtualization setup each VM runs a full operating system image, which consumes a lot of resources. The start up has high latency and also each request must go through guest OS and then the hypervisor, which increases overhead. The isolation and resource allocation to applications which is provided by virtualization can also be done using a lightweight OS-virtualization technique that is containers. Containers have a single operating system running on host machine hence the resource consumption and overhead of hypercalls is reduced. Nowadays containers have been widely used for providing virtualized environments since its lightweight as compared to VMs. So start up time is much less and the resources consumed by the containers is also less. Figure 4.4 shows difference in architectures of a virtual machine and a container.
A container runs a set of processes in an isolated environment which is allocated a bunch of resources by host operating system. Some examples of containers are lxc, docker, openVZ. Containers share operating system but have their own resources and isolation. Containers have three main components:

- Cgroups : for resource allocation like CPU, disk IO, network, memory
- Namespace : for isolation like process isolation, network isolation, mount point
- disk images : where the container has its rootfs (mount namespace)

The resource accounting is done using cgroup and isolation using namespace. Each container is being given resources : cpu, memory, blkio etc.. Each container would be running certain set of process which will have its own requirements. These resource limits are to be given by the admin via kern FS. For resources such as memory and cpu, admin can allocate the limits. For

example in case of memory, upper limit of memory utilization is given by soft limits.
In virtualised systems when we are running heavy IO workloads the performance depends on disk performance. So we introduce SSD at the block layer to improve IO latency. [4] [8]. Same problem will exist when containers will be running heavy IO workloads. Although in containers we can set IO limits dynamically but the existing implementation just throttles IO in a proportional way or using CFQ. It does not actually allocate the storage (blocks) to a container.

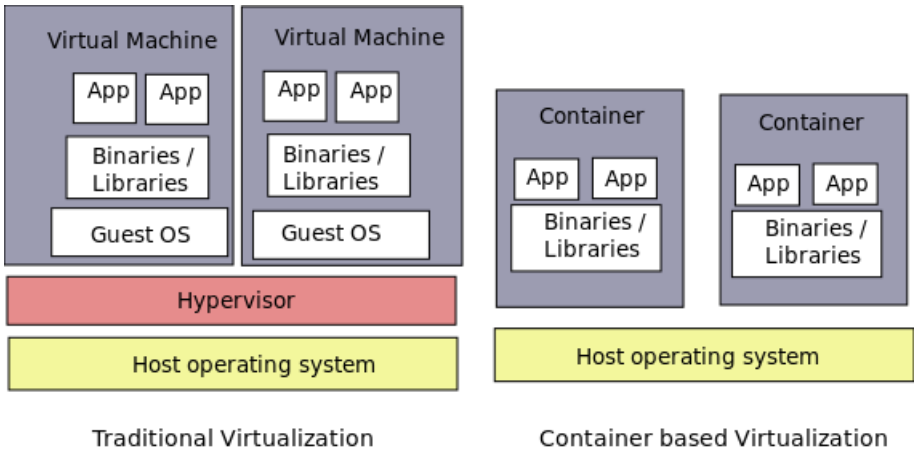


Figure 2.1: Traditional virtualization and Container based virtualization

Figure 1 in appendix shows internals of cgroups. Cgroups are hierarchical. Each resource i.e. CPU, memory, disk etc. is called a subsystem. There are 12 such subsystems, which has hierarchical id 1 to 12 as shown in figure in top. For each such subsystem there is a hierarchy of cgroups. Each cgroup (shown in grey) has an associated subsystem state (shown in green). The subsystem state has pointers to the associated resource configurations for that cgroup. The configurations are applied to whole subtree below the cgroup.
Each container has a set of processes which will have pointers to subsystem state according to their resource configurations. We will concentrate on disk IO. Cgroup subsystem blkio implements block IO controller. It is used to throttle block IO in containers. Currently two IO control policies are implemented. First is proportional weight division of bandwidth implemented in Completely Fair queuing (CFQ). The second is throttling policy which can be used to specify upper IO rate limits on devices. Both the policies are on limiting the IO rate but not on limiting the capacity of block storage.

2.1.1 Limitations

- Cgroup subsystem blkio does not allow to put a capacity limit on storage used by a container.
- In existing container setup there is no way to provision SSD as cache to hard disk and limit SSDs capacity for the container. This can be beneficial in cases where we want to control the IO performance of the application using cache.
- Weak isolation : Containers are not as secure as virtual machines are considered to be since they share the host operating system. Corruption of one container may lead to malicious attack on operating system which may in turn affect other containers.

2.2 Non Volatile Memory Usecases

Non volatile memory is a type of computer memory which can stored information even after switching off power. There are many types of non volatile memory like flash memory, read only memory, magnetic tapes, hard disks, optical disks etc...[21]. We are working on flash storage i.e. SSD. Below Table 2.1, shows comparison of hard disk, flash memory and DRAM. We can observe that NAND flash sits in between DRAM and HDD in terms of IO latency and cost.

Table 2.1: Comparison of memory technologies [3]

	DRAM	Nand Flash	HDD
Read latency	10-50 ns	25 μ s	5ms
Write latency	1GB/s	200-1500 MB/s	200MB/s
Endurance (in cycles)	$> 10^{16}$	10^4	$> 10^{16}$
Byte addressable	Yes	No	No
Volatile	Yes	No	No
Cost	\$2 per gigabyte	\$0.20 per gigabyte	\$0.03 per gigabyte

There has been lot of research going on using non volatile memory in virtualised systems and improving its performance. These can be classified into following topics:

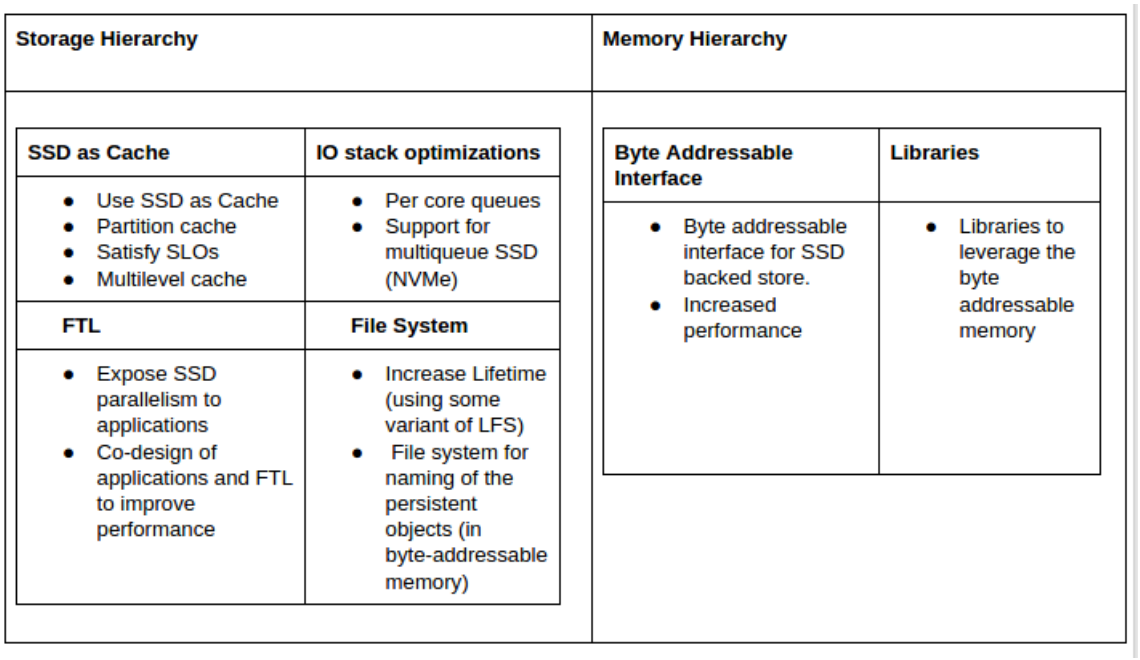


Figure 2.2: NVM classification

- NVM as cache : Using non volatile memory (specially SSDs) as cache in storage hierarchy. Various caching techniques have been designed such as single level caching, multilevel caching, static partitioning cache across VMs, dynamic partitioning of cache [4],[8],[6].

- Internals of NVM : Although flash storage is faster than HDD but there is endurance problem in it. So new cache management policies are proposed which are aware of write erasure property of SSDs. Also exposing Flash Translation Layer (FTL) to host can result in better flash endurance and application-level performance [11].

- Application has control on NVM : For memory based applications, larger memory is required. DRAM scaling is difficult and is costly too. Hence heterogeneous memory architectures are designed where NVM and DRAM both are on memory bus since NVMs are cheaper [10]. Applications can store the data in NVM as its present on data bus. Applications can be given control on NVMs in storage hierarchy as well using APIs [12]. This can result in application specific optimizations.

- NVM aware Linux IO stack : Nowadays SSDs have been adopted as storage devices and these are fast due to internal parallelism. CPUs have also improved as more cored have

been added, so the performance bottleneck exists in Linux block IO layer due to single request queue (locking). There has been work on parallelizing Linux IO queues [14],[13].

2.3 Caching disk blocks

Caching disk blocks improves IO performance since disk has higher data fetch latency than cache. In data centers storage disks are in network, so each miss in the cache must go through network and get the data from storage, this causes high latency. Hence to improve performance we must ensure that number of hits are more in the host side itself and attaching one more cache layer such as SSD can further increase the chances of getting cache hit. The objectives of various caching techniques are as follows:

1. Minimize overall IO latency
2. Minimize Per container IO latency
3. Enable maximum utilization of storage resource
4. Increase IOPS
5. Prioritize containers
6. Do Fair allocation
7. Satisfy service level objectives

2.4 Flavors of Caching

A cache can be used in different ways : unified, statically partitioned or dynamically partitioned. A brief description about these is as follows:

2.4.1 Unified cache

The cache is shared by all containers as shown in Figure 2.3. The page cache in OS is an example of unified cache since it is shared by all the processes within the system. In unified cache, blocks of one container can be replaced by the blocks of other container.

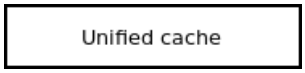


Figure 2.3: Unified Cache

2.4.2 Static partitioning

Static partitioning of cache means each container will get configured amount of cache as shown in Figure and it cannot be changed later. Following is an example which shows why static partitioned cache is better than unified cache. Let us assume two ways of caching, one in which all container use unified SSD cache and another in which SSD is partitioned on per container basis. Type of workloads running on each container is as follows:
Container 1 - workload which has less temporal locality but high IO request rates (100 blocks per sec).
Container 2 - workload which has high temporal locality but IO request rate is moderate (50 blocks per sec).
Assume scheduling time is 1 sec for each container and cache size is 100 blocks.
Case 1 : Unified Cache - When container 1 is scheduled it will cache 100 blocks occupying full cache. Next container 2 is scheduled which will replace 50 blocks of data of container 1 with its own data which is fine since its data blocks have better temporal locality than that of container 1s. Next when container 1 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 container 2 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 container 1 and 50 blocks to container 2 then performance will be better since container 1s blocks wont replace container 2s blocks.
So we can say that cache partitioning is required for better hit rate. Also partitioning the cache gives host OS control to prioritize the containers. For example if container is of high priority then user or admin can statically allocate more cache to it.

2.4.3 Dynamic partitioning

Each container runs different type of workload and each workload has its own working set size. If a container 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 we will also be wasting cache. Hence dynamic partitioning is preferred over static partitioning since it reduces cache wastage [4].
Let us consider similar example as above.
Case 1 : static partitioning with container 1 - 50 blocks and container 2 - 50 blocks
Case 2 : dynamic partitioning which is based on temporal locality so container 1 - 5 blocks and container 2 - 95 blocks.
In case 1, container 1 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, container 1 gets sufficient amount of cache while container 2 can utilize the cache better since it has better locality of reference.

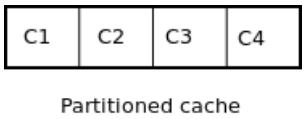


Figure 2.4: Partitioned Cache

2.4.4 Combination of unified and partitioned cache

Another variant of cache partitioning can be dividing cache into 2 parts - one is unified and other partitioned as shown in Figure 2.5 The partitioned cache can be static or dynamic.

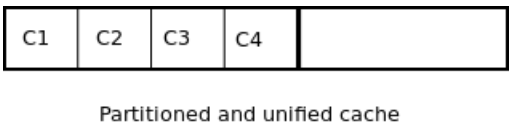


Figure 2.5: Unified and Partitioned Cache

2.5 Determining cache size according to workload

One of the most common ways to determine optimal cache size required is based on workloads (which is running on container) working set size. Working set size gives the memory required by workload beyond which cache hit ratio will not increase. From Figure 2.6 we can see that cache size beyond working set size does not further decrease miss ratio. So we try to allocate each container a cache size less than or equal to its working set size, since giving more than working set size wont add any significant amount of performance benefit. To find working set size of a workload running on container, we must construct miss rate curve of that workload.

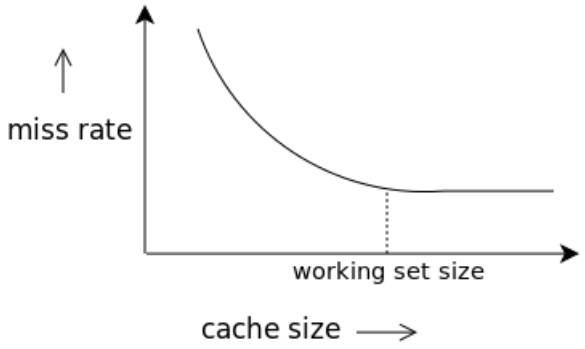


Figure 2.6: Unified and Partitioned Cache

Generation of miss rate curve from the IO traces is memory as well as time consuming process. The IO trace tells the access pattern of a block device. Each container will have its own IO trace and using these traces we must construct miss rate curve. A very naive way of doing this is by changing the cache size and calculating number of misses at each cache size. This is very time consuming since for every cache size we must traverse full trace again. There are other algorithms proposed to construct miss rate curve, such as variants of Mattsons stack algorithm or by using cache miss equation [5].

The Mattsons stack algorithm is based on reuse distances. Reuse distance of a data reference B is the number of unique data references since the previous access to B. The advantage is that the trace has to be traversed only once but the algorithm still has high space and time complexity and works only if LRU cache replacement policy is used. There has been lot of research on generating cache miss rate curve, few of them are SHARDS [7], PARDA [17], Counter Stacks [9]. Recently SHARDS algorithm [7] (a variant of Mattsons stack algorithm) has been proposed which can construct miss rate curve with constant space complexity and linear time complexity.

Cache miss equation can be directly used to find miss rate at every cache size. To formulate the equation, one needs to first find out the parameters of the equation from the IO trace using regression. The following equation 2.1 has few parameters which are dependent on the workload.

$$\begin{aligned} Missrate &= \frac{1}{2}(H + \sqrt{H^2 - 4})(P^* + P_c) - P_c \\ where : H &= 1 + \frac{M^* + M_b}{M + M_b} for M \leq M^* \end{aligned} \tag{2.1}$$

P^* : Minimum miss rate possible, i.e. miss rate at cache size = working set size
M^* : Cache size beyond which miss rate won't decrease
M_b : Memory needed by cache manager for storing metadata information
P_c : Miss rate curve's convexity

In general hypervisor managed caching can be divided into single level caching and multilevel caching. Each of this variant can be used with unified cache or partitioned cache.

2.6 Device mapper

Device mapper is a framework for constructing new block devices and for mapping them to physical block devices. This is provided by Linux kernel. As shown in Figure 2.7, device mapper lies within block IO layer. It is managed through IOCTL interface. A user space library called dmsetup is provided to create our mapping tables. When a dm target is registered in system, a new block device gets created in /dev/mapper. The write or reads requests to this device can be intercepted and manipulated according to our needs. There are various device mapper targets available such as linear (maps continuous range of blocks to a device), striped (strips data across two devices but exposes it as one), mirror (mirroring devices), cache (maps a fast device as cache to slower device) One of its usecase is to expose a faster device as cache to slower device like HDD. There are already few open source implementation which expose SSD as cache to HDD using device mapper. But all the implementations are designed to use SSD as unified cache across all containers. Few of them are :

- dm-cache : This is present in kernel source tree. Problem with this is it handles the request at granularity of group of bio requests. This makes it difficult to use for partitioning cache.
- bcache : This is also present in kernel source tree. It does not use device mapper frame-

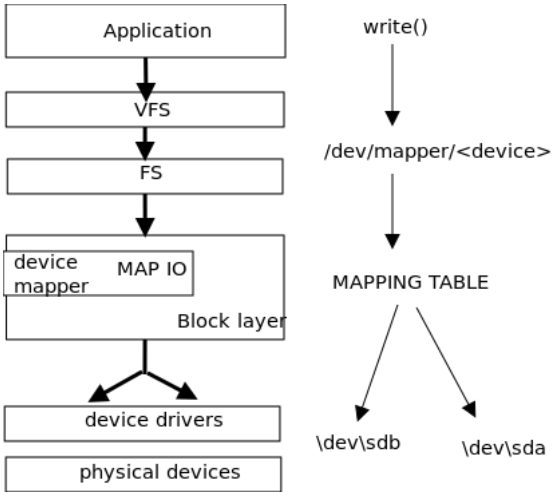


Figure 2.7: Device mapper

work and works at block layer. Problem with this is, before using SSD as a cache to HDD, the disk has to be formatted.

- flashcache : This is easier to understand and is a kernel module, so compiling is easy. Hence this is chosen for further modification according to our needs.

2.7 Flashcache

In 2010, facebook wanted a faster backend for their database. They wanted to use flash for the storage but it was costly. So one way was to use both types of storage where hot data will go to the flash device and cold would stay in HDD, but it was a complex design. Other way was to use flash as caching layer. The workload for which flashcache was primarily developed for is InnoDB - a storage engine for MySQl for facebook.
What is flashcache?

- Flashcache is loadable kernel module (advantage over bcache or dmcache since for changes in them kernel had to be recompiled)
- It works in block caching layer below file systems.
- IO request $\rightarrow$ SSD cache $\rightarrow$ HDD.
- Build using device mapper (dm)

Flashcache features:

- Caching modes - writeback, write through
- Policies used - LRU-2Q, FIFO

2.7.1 Architecture

The cache structure is a set associative cache. For each disk block number a set number is calculated and hashed into fixed sized buckets with linear probing within a set. Linear probing means if there is a clash on a set then you go to the next empty slot. Each cache block has a metadata structure which stores disk block number mapped to this cache block and the state of the cache block. As shown in Figure 2.8, cache structure has sets and cache metadata blocks. Cache metadata blocks store disk block number and the state of the cache block. Each cache set has an eviction policy and number of dirty blocks in that set. The lookup is done in following steps :

- Find the set number from disk block number, set_number = (dbn / block size / set size) mod (number of sets). The number of sets is configurable parameter.
- Within a set find which hash bucket the disk block belongs to using set_ix = hash(dbn) % 512, where 512 is set size. Set size is also configurable parameter.
- cache block number = set_number * 512 + set_ix
- If state of cache block is VALID and the disk block number matches the requested block, then it returns hit and if not present then gets an empty block from that set and returns. If there are no empty blocks in that set then returns a clean block so that write to disk can be avoided. If no clean blocks are present in the set then dirty block is returned and hence requests goto disk.

2.7.2 Flaws

1. Replacement policy is per set : Let us assume a scenario where a cache set is full and other cache sets are empty. A request gets mapped to the set which is full then the reclamation (clean block) happens from the same set. But ideally since the other cache sets are empty, a cache block should not get evicted from cache.

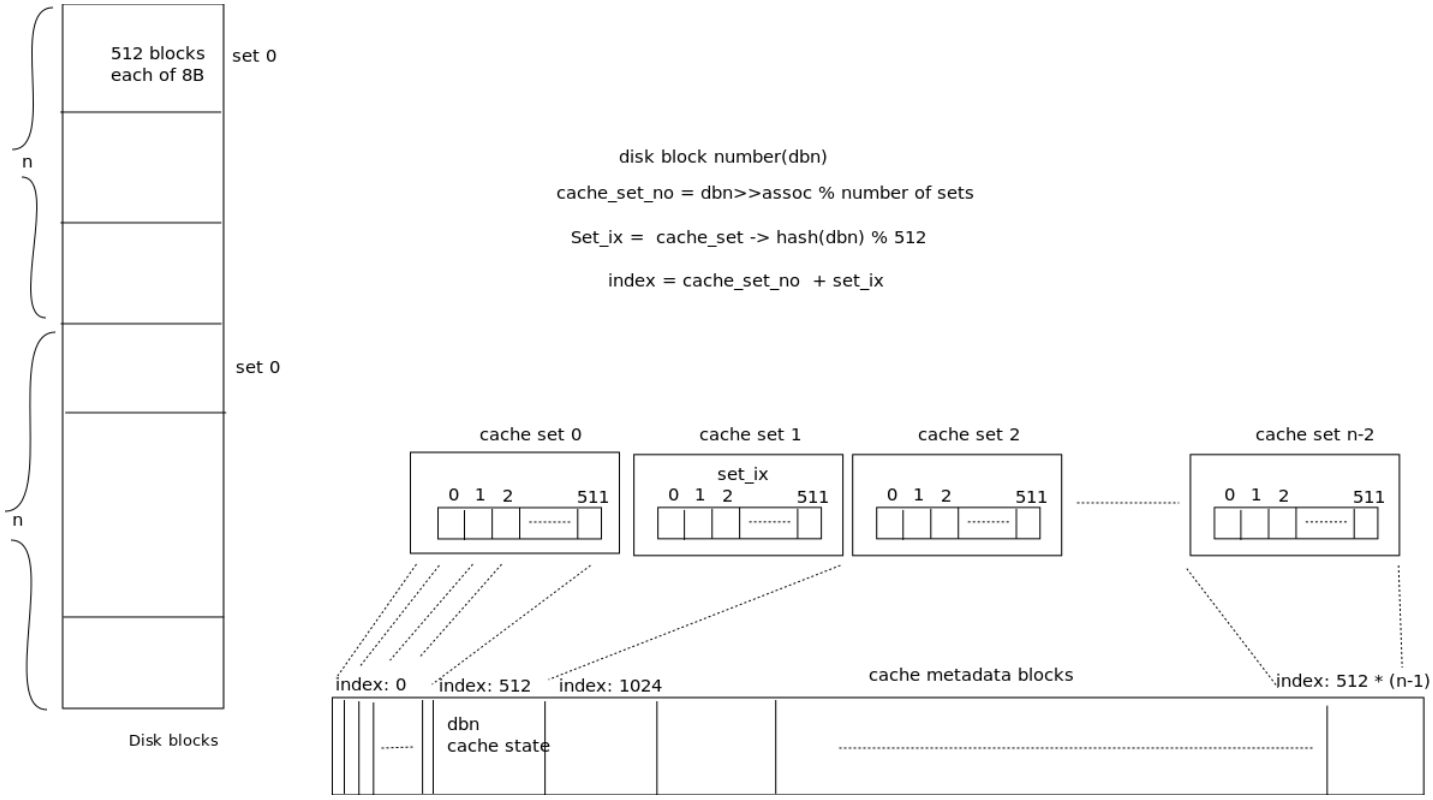


Figure 2.8: Flashcache cache structure

2. If a set is filled by one container and the requests from other container are also mapped to same set, then the blocks of one container will replace blocks of other container. But our design says that we want it to replace blocks of same container whose size has exceeded.

Hence we have removed the correspondence of a disk block number with set. We get a free cache block number by maintaining a global list of free cache blocks.

2.8 Related work

There has been lot of work already done on introducing a second chance cache in the disk IO path at different levels in virtualised setup. Table 2.2 shows various possible combinations of positioning of cache in virtualised environment. It can be cache within guest VM, memory managed by hypervisor, SSD cache managed by hypervisor or caching at multiple layers.

Table 2.2: Possible combinations of positioning cache in VM environment

	Cache in guest VM	Hypervisor managed in memory cache	Hypervisor managed SSD cache	Server side cache
References				
Default	✓			
ExTmem [6]	✓	✓		
3Level [22] Unicache [15]	✓	✓	✓	
CloudCache [1]	✓		✓	✓
VcacheShare [8] Centaur [4] SCAVE [16] GReM [23]	✓		✓	

The related work has also been done on what type of partitioning should be done, unified, static partitioned or dynamically partitioned. Table 2.3 shows which of the previous work have used which type of caching policy. As we can see most of the work has been done on dynamic partitioning. Hence we also wanted to impose this technique for container based setup.

In the space of containers following research paper was relevant to our work : Cgroup++ [24], talks about proportional IO sharing based on weight-based throttling of NVM SSDs. The usage is similar to existing cgroup that throttles IO but here its done for SSDs.

Table 2.3: Flavors of Caching

Unified cache	Statically partitioned cache	Dynamically partitioned cache
• ExTmem [6]	• None	• VcacheShare [8] • Centaur [4] • SCAVE [16] • GReM [23] • 3Level [22] • Unicache [15] • CloudCache [1]

3 Design and Implementation of Differentiated SSD Caching

The main aim of introducing a second level cache is to reduce IO latency. To achieve this, new module has to be inserted in the IO stack of current system. As shown in Figure 3.1 the internal architecture of system is modified by adding two major modules. The major components are 1) *Partitioner*, which partitions SSD on per container basis, 2) *Analyzer* Periodically processes the trace of IO requests from containers and decide the optimum partition size.

3.1 Partitioning

As shown in Figure 3.2, all IO requests are first checked in the page cache, if it is not present there then the requests are given to our module p̈artitioner p̈resent in the block layer. The input to the module is bio requests (from containers) and cache partition sizes (from analyzer module). It initially allocates default number of SSD blocks to all containers. Once the analyzer sends new cache sizes for each container, it reclaims the cache blocks from one container and gives them to other according to request. If the cache size of one container has reached its maximum limit then the reclamation must be done from the blocks of same container. The cache blocks of other containers should not be replaced.

Partitioning module does the actual allocation of cache blocks to containers. For each container it maintains an eviction list of blocks present in cache, lookup structure and the maximum cache size a container can use. A global lookup is not maintained to avoid lock contention. Separate lookups helps to parallelize lookups across containers. Lookup structure has mapping of disk block number to cache block number. Also it maintains a global list of free cache blocks

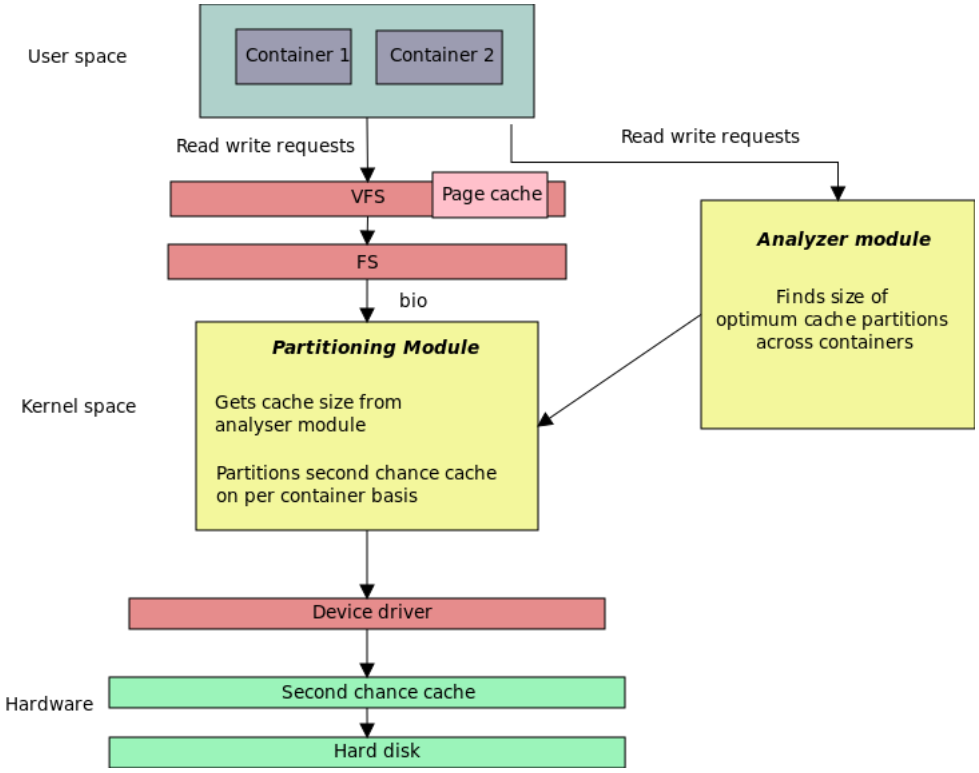


Figure 3.1: Architecture

present in SSD.

First it checks which container has requested for the IO. According to the container, check the lookup structure. If the disk block is present (cache HIT) in cache, lookup will return the corresponding cache block number. If disk block number is not present (cache MISS) in the lookup, then a new cache block has to be assigned. Each container has been given a maximum limit of second chance cache blocks. So if its a cache MISS, then there are two possibilities:

- if cache limit is not yet reached and free blocks are present in cache, then get a free block from free block list and use it to store new disk block.
- if cache limit is reached, then from the eviction list get a cache block and use it store new disk block. And the evicted block has to be written to disk if it is dirty.

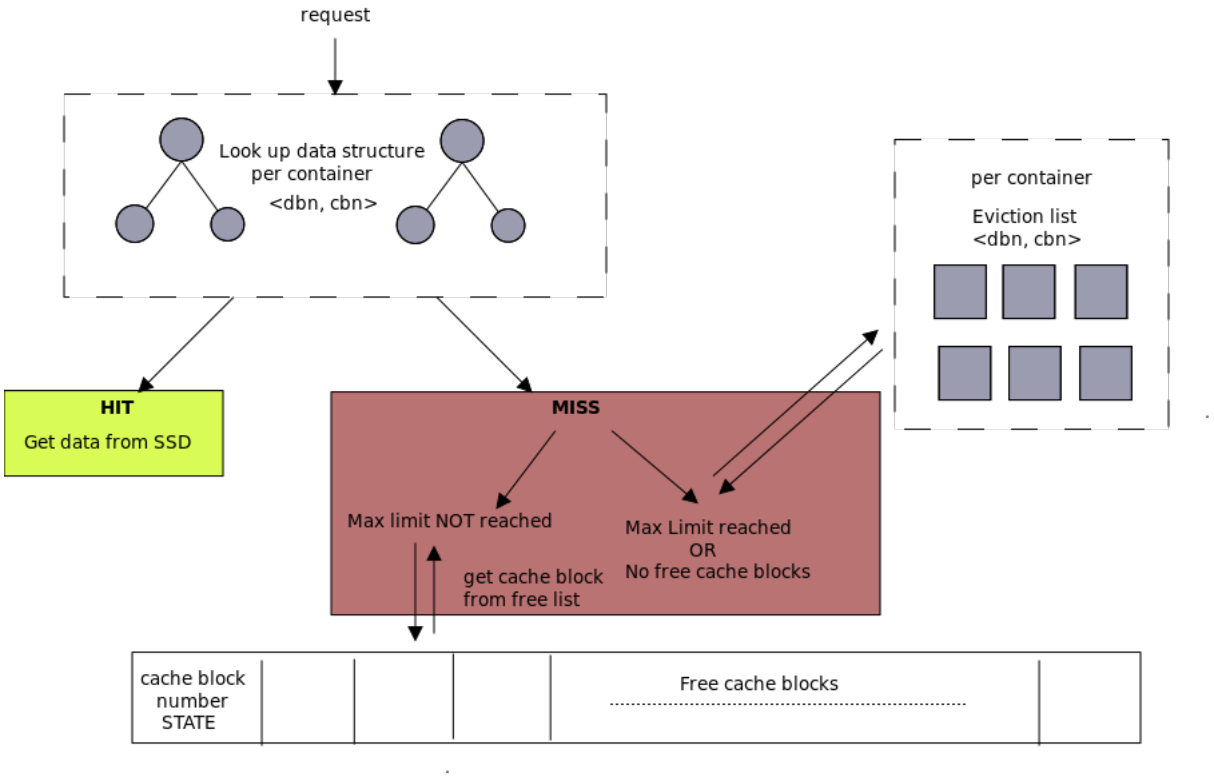


Figure 3.2: Proposed design

3.2 Analyzer

Analyzer intercepts, records and analyzes the read write requests of containers, before they hit page cache. For every container, it performs online periodic trace processing. The trace will have following parameters : disk block number and container ID. For every container, it will construct miss rate curve according to workload running in the container to find the working set size of workload. The cache size can be determined on the basis of working set size of workload running, priority of container, SLA requirement on IO latency. This is repeated after every epoch and new set of cache partitions are given as an input to partitioning module.

3.3 Current implementation

We will look into following aspects, one is lookup function, handling of read write requests, dynamically predicting the working set size.

3.3.1 Lookup function

The read write requests to the device mapper module comes in the form of struct bio (a linux struct for block IO requests). The requests come in function flashcache_map(). Then we check which container does this bio belong to. Function get_container_ID(), returns container id. The containers are distinguished by their cgroup id. We can get a cgroup id from bio by cgroup subsystem state which contains cgroup struct.

Then lookup function (flashcache_lookup(bio)) is called which checks if the disk block number is present in the cache or not. For every container, we maintain a look up tree with dbn as key and cbn as value and an eviction list. According to the container id the lookup is performed in corresponding tree. If dbn is present in the tree, it returns the cache block number and read or write is done. If its a miss then first check if container cache limit is reached or free cache blocks are not available, then get a block from eviction list. If limit is not exceeded then get the block from global free list of cache blocks. A block selected from eviction list can be dirty or clean. If its dirty then write to disk else just replace block with new one.

Currently eviction policy used is LRU but it has a pluggable interface so that we can call any eviction policy according to container's workload. We have implemented FIFO and LRU.

Algorithm 1 Cache lookup algorithm

```
1: for every bio request do
2:     container_id = get_container_ID(bio)
3:     cbn = lookup(dbn, container_id)
4:     search for dbn in rb_tree of above container_id
5:     if HIT then
6:         read from or write into cache block
7:     else MISS
8:         if cache limit for container exceeded || no free blocks present then
9:             get cache block from eviction list
10:            replace old data with new one in the returned cache block
11:        else
12:            get cache block from free list
13:            write into cache block
14:        end if
15:    end if
16: end for
```

3.3.2 Read write handling

The read write requests are handled differently in case of hit and miss. After lookup function returns a cache block number, one of the following four conditions come up as shown in Table 3.1. For every IO request a job is created with an action and dm_io_async_bvec function is called with disk or cache block number, read or write request and a callback function. After every read write request a callback function is called flashcache_io_callback. If cache is write back then write request will only write to SSD. If cache is write through then both disk and SSD are written.

In all the cases except read miss, we either read from or write into cache and in callback function check for any errors. In read miss, the data is not present in SSD, therefore first READDISK is issued where data is read from the disk. Then in the return path (in callback) get the data from disk and write that into SSD with READFILL action. Current implementation has write through and write bypass as pluggable interface. In write through all the writes are to disk and SSD as well. If there is a write hit, then data is updated on SSD as well as disk. While in write bypass all the writes are bypassed to disk. If there is a write hit, then data present in SSD is flushed back to disk and removed from SSD. Later write back policy can also be added to our design.

Table 3.1: Read Write request handling

	READ	WRITE
HIT	job action : READCACHE in callback : check for errors	job action : WRITECACHE in callback : check for errors
MISS	job action : READDISK in callback : create a job with action : READFILL	job action : WRITECACHE in callback : check for errors

3.3.3 Predicting optimum cache size

Predicting cache size can be done using reuse distance algorithm. There is an open source code called PARDA [17], which implements it. We have used and modified that code to predict the cache size after configurable time. As soon as analyzer module is inserted in kernel it will start a timer. The module will probe all the IO requests and store block number accessed in a list on per container basis. As soon as timer interrupts, it will calculate reuse distance of each block access, for each container and predict the optimal cache size and this set of cache sizes of each

container is sent to partitioner module via flashcache_change_container_size() function. The input to this function is cache size and the container id. This function will follow the given algorithm :

Algorithm 2 Cache resizing algorithm

```
Given new cache size S of a container C
if S > current size of C then
    New cache size of C = S
else
    if S > cache size occupied by C then
        New cache size of C = S
    else
        while cache size occupied by C > S do
            get a cache block from eviction list
            remove it from lookup tree
            add it to unused free list
        end while
    end if
end if
```

4 Experimental Evaluation

All the experiments were done on Linux operating system (kernel-4.6) with hard disk of size 100 GB on linux containers.

4.1 Correctness Verification Experiments

4.1.1 Experiment 1 : SSD caching without evictions

Question : Is SSD caching the blocks correctly when there is no contention on SSD resource?
Setup :

Number of containers - 1
Page cache - 256 MB
SSD - 2 GB
File of size 512 MB (131072 blocks) was read sequentially

Results:

Table 4.1: Correctness verification without evictions

	Container requests	Hits	Misses	Evictions
1st Run	131072	0	131072	0
2nd Run	131072	131072	0	0

Observation : Initially a file is read, then all the blocks of file will not be present in SSD hence all the blocks will MISS the second level cache. Hence all these blocks are read from disk and both page cache and SSD will be populated since we have chosen inclusive caching policy.

In the second run, when the same file is read again, there won't be any hits on page cache since its size is 256 MB. Since the file is read sequentially the blocks will be replaced by the later part of file. Therefore all the requests are MISS on page cache and the requests reach SSD. These are HITs on SSD since all the blocks are present in SSD.

Conclusion : It can be verified that the module is caching the blocks when they are read from disk. And when blocks are not present in page cache, same blocks are there in SSD. So SSD is acting as cache to disk.

4.1.2 Experiment 2 : SSD caching with evictions

Question : Can SSD cache the blocks correctly given a specific SSD size? Is eviction working?
Setup :

Number of containers - 1

Page cache - 256 MB

SSD - 500 MB (128000 blocks)

File of size 512 MB (131072 blocks) was read sequentially

Table 4.2: Correctness verification with evictions

	Container requests	Hits	Misses	Evictions
1st Run	131072	0	131072	3072
2nd Run	131072	0	131072	131072

Observation : Initially a file is read, then all the blocks of file will not be present in SSD hence all the blocks will be MISS on the cache. When these blocks are read from disk, both page cache and SSD will be populated due to its inclusive nature. In second run, when the same file is read again, there wont be any hits on page cache since its size is 256 MB and it will keep on replacing former blocks of file with later ones. Therefore all the IO requests are MISS on page cache. Since the size of SSD is smaller than the file size all the blocks cached would have been replaced by newer blocks. Hence all requests are miss on SSD and all blocks will be evicted in second run. The number of blocks evicted in first run will be number of IO requests minus total SSD size (131072 - 128000 = 3072 blocks) which is matching the result .

Conclusion : The module is caching and evicting the blocks correctly.

4.1.3 Experiment 3 : Effect of cache size on performance

Question : How does SSD cache size effect cache hit ratio and completion time?
Setup :

Number of Containers - 1

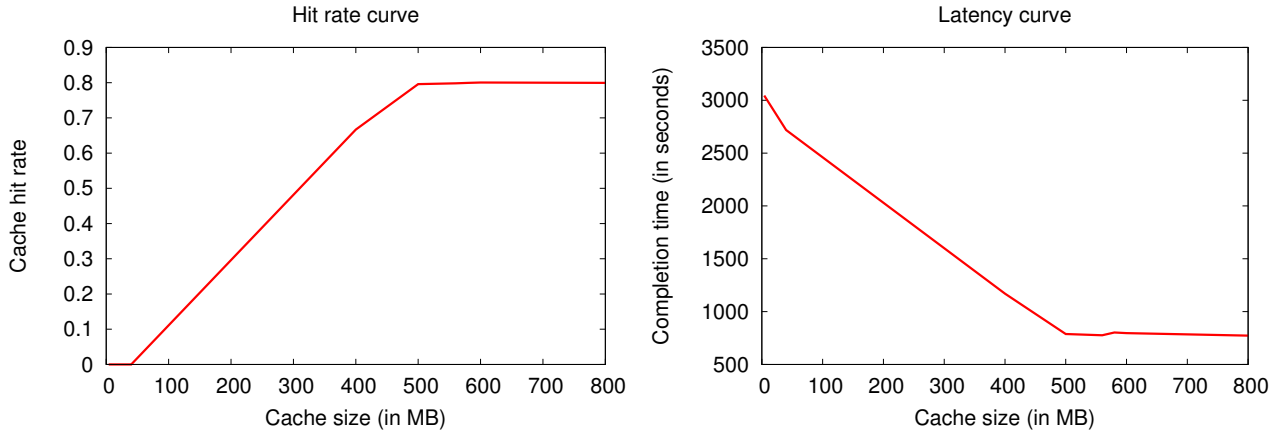
Page Cache - 256 MB

Workload - Reading file of working set size = 768 MB having a access pattern following Gaussian distribution.

Metric - Completion time, cache hit ratio

Parameter - SSD Cache size

Expected result : It is expected that as cache size increases, number of evictions decrease, cache hit ratio increases and completion time must decrease.



(a) Hit rate curve

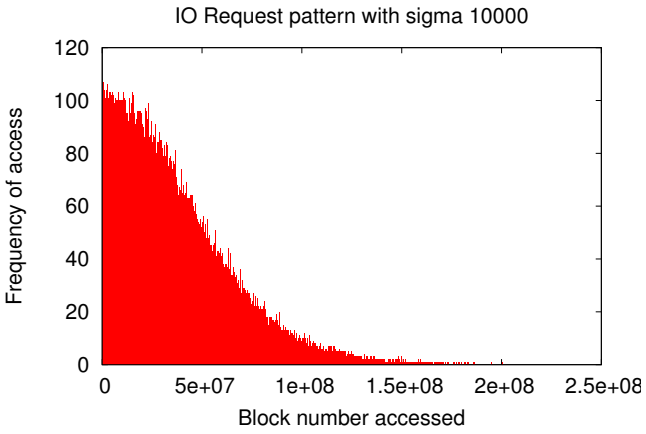
(b) Latency curve

Conclusion : Figure 4.1a shows that cache hit ratio increases as cache size increases but as soon as cache size exceeds 500 MB, the cache hit ratio becomes constant. Since the working set size of workload is 768 MB, 256 MB is in page cache and rest will be on SSD. So if SSD size is 512 MB (768 - 256) then we wont be able to see further improvements in hit rate due to compulsory misses.

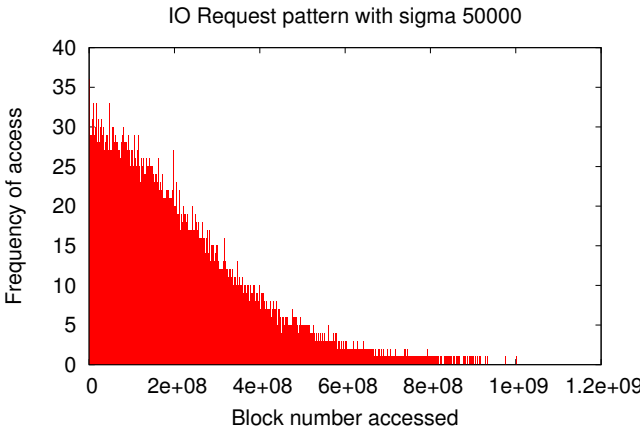
The completion time will depend on the hits on SSD since more the hits on SSD, lesser will be the time required. Figure 4.1b shows that completion time also becomes constant as cache hit ratio becomes stagnant after 500 MB.

4.2 Unified SSD cache vs Partitioned SSD cache performance

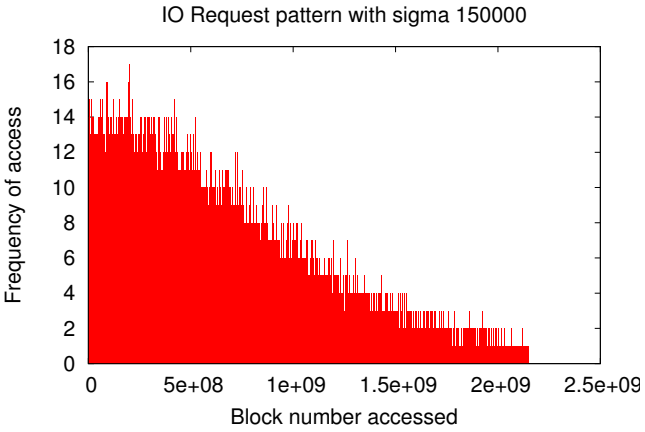
The setup consists of two lxc containers running different workloads. Both are configured with 256 MB cgroup memory limit. Each experiment was done for unified as well as for partitioned SSD. In unified case 1 GB SSD cache was shared across both the containers. In partitioned case, SSD was equally divided amongst two containers, 0.5 GB each. In these experiments a set of workloads was generated by a user program. The workload reads a 10 GB file with configurable access pattern and IO rate. The C code written uses GSL (GNU scientific library) [18] to generate IO requests with Gaussian probability distribution. The access pattern is changed by changing the sigma of distribution and the IOPS is changed by adding delay between the IO requests. Below graphs show the access patterns of 5 different workloads which were generated for experiments.



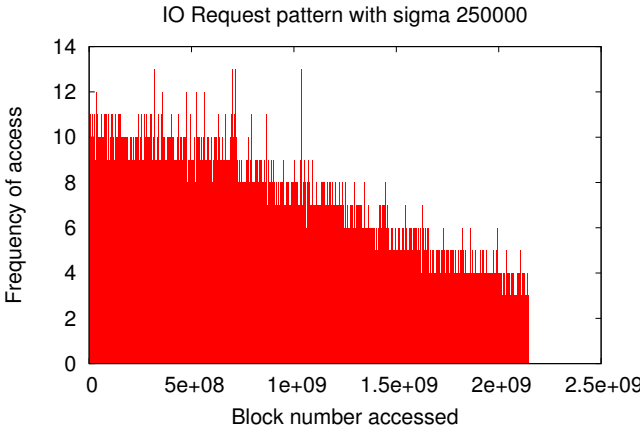
IO Request pattern with σ = 10000



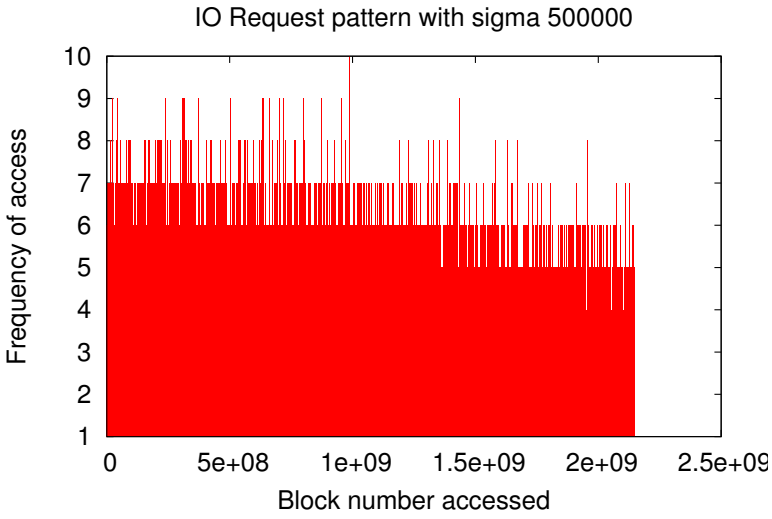
IO Request pattern with σ = 50000



IO Request pattern with σ = 150000



IO Request pattern with σ = 250000



IO Request pattern with σ = 500000

4.2.1 Experiment 1 : Effect of varying IOPS

Question : How does performance of one container gets affected by IO rate of other container which is running in parallel?
Setup : Container 1 has IO rate of 358.16 IOPS and has access pattern with σ = 50000. For Container 2 we vary the IO rate from 358.16 IOPS to 607.53 IOPS and access pattern remains same with σ = 500000 (i.e. more random accesses).
Expected result : In Unified cache setup, container 2's workload will replace the blocks of container 1 due to higher IOPS. Since container 2 has more randomness, the blocks will have lesser temporal locality and blocks with higher temporal locality are getting replaced. Therefore overall as well as container 1's cache hit ratio get affected.

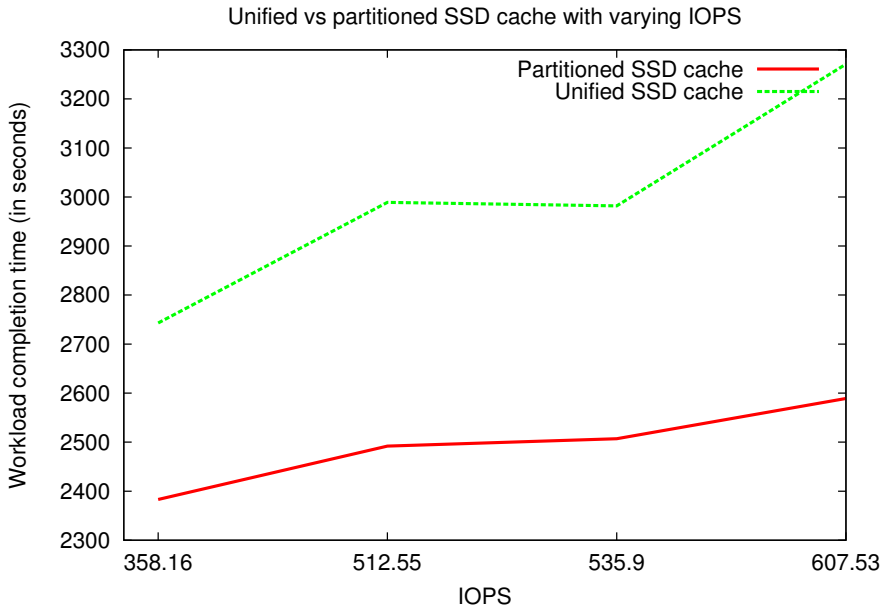


Figure 4.1: Effect on performance of Container 1 with varying IO rate of Container 2

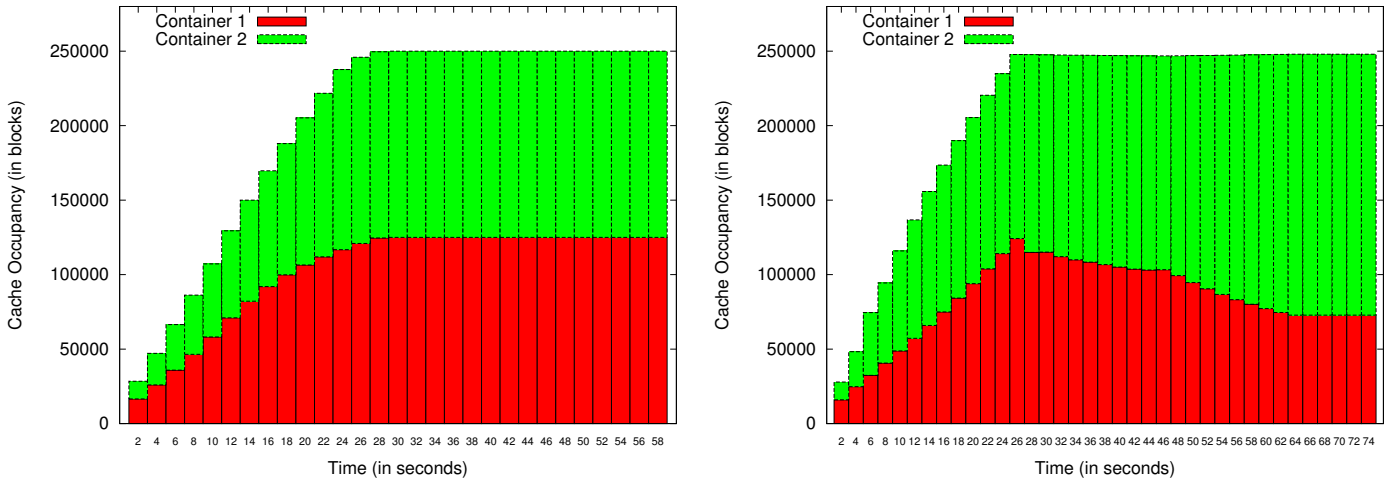
Observation : Table 4.3 shows the cache hit ratios of both the containers for unified and partitioned case. As we can see in partitioned case there is no significant change in cache hit ratio with change in IOPS because both containers have their own allocated cache and no one is interfering or replacing other containers blocks. But in case of unified cache the number of hits for container 1 are decreasing without increase in hit rate of container 2. Since container 2 is occupying more cache as shown in Figure 4.2, it was supposed to increasing the hit rate but as

Table 4.3: Cache hit ratio

	Partitioned cache hit ratio		Unified cache hit ratio		Overall Hit rate	
IOPS	C1	C2	C1	C2	Partitioned	Unified
358.16	78.26	29.48	68.19	33.96	53.87	51.075
512.55	79.58	29.46	53.25	34.41	54.52	43.83
535.90	79.54	29.47	48.06	33.89	54.50	40.97
607.53	79.58	28.89	41.29	34.15	54.23	37.72

said in setup that container 2 has workload which has more random accesses, the provision of more SSD cache has not shown much improvement cache hit ratio. Therefore the overall hit ratio of partitioned cache is much better than unified, thats why we see in Figure 4.1 that completion time for container 1 has been drastically affected due to IOPS of container 2.

Inference : The results are as expected that is as the IOPS of one container increases it keeps on replacing the blocks of other container.



(a) Cache occupancy by partitioned SSD cache

(b) Cache occupancy by Unified SSD cache

Figure 4.2: Cache occupancy

4.2.2 Experiment 2 : Effect of varying workloads

Aim : How does performance of one container gets affected by working set size and access pattern of other container running on same host?
Setup :
Container 1 : Reading a file with IO access pattern generated using σ = 50000
Container 2 : Reading a file with IO access pattern of varying σ but at a rate of 607.53 IOPS.

Expected result : Container 2's change in working set size will affect container 1's performance in unified case because as the randomness of workload increases, it will start polluting the cache and effect the overall performance of system.

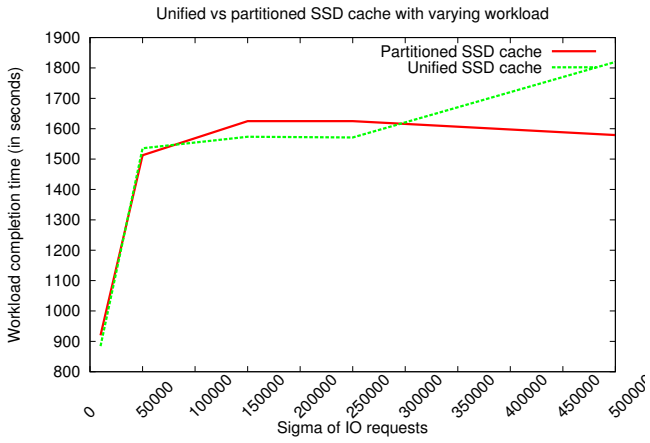


Figure 4.3: Effect on performance with varying working set size

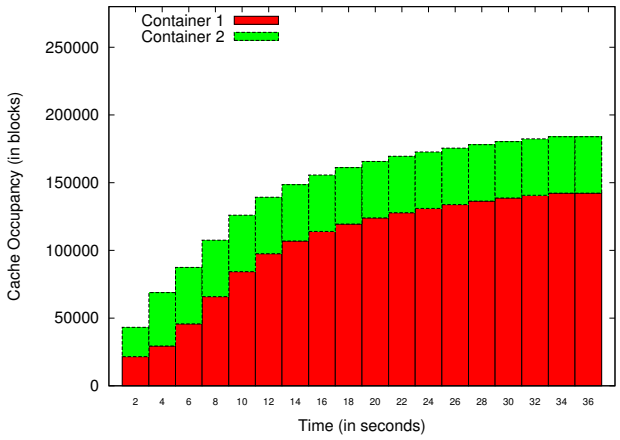
Observation : Figure 4.3, shows the effect on performance of container 1 with varying workload pattern of container 2. Initially both unified and partitioned perform similarly but as sigma goes beyond certain value the completion time for unified becomes much higher than partitioned cache. When sigma is between 10000 and 25000 the overhead of maintaining partitions may lead to increase in time. As shown in Figure 4.6c, container 2 blocks are not interfering with container 1s blocks since the working set size of container 2 is less. Thats why both partitioned and unified perform similarly. In Figure 4.4c, 4.4d, 4.4e, we can see that container 2 blocks are dominating the SSD, replacing the blocks of container 1 which are of more temporal locality with the ones with lesser temporal locality i.e. of container 2. For σ between 10000 to 250000 the workload's working set size has increased and cache hit rate increases since cache is being

utilized by both containers in a good way. Specially in unified case the But as soon as sigma exceeds 250000 workload becomes more random and hence its occupancy wont make any improvement in performance. Table 4.4 shows the cache hit ratio of containers in both partitioned and unified setup. As we can see from the table the cache hit ratio of container 1 remains same for partitioned cache while in unified case it keeps on decreasing. For container 2 as workload becomes more random the hit ratio decreases.

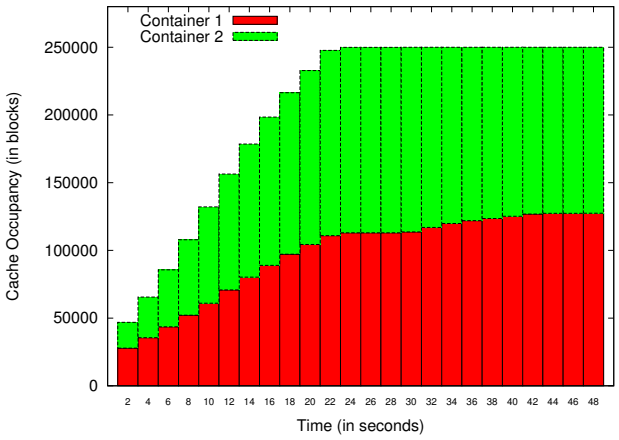
Table 4.4: Cache hit ratio of both the containers

	Partitioned cache hit ratio		Unified cache hit ratio		Overall Hit rate	
Sigma	C1	C2	C1	C2	Partitioned	Unified
10000	80.18	in page cache	79.39	in page cache	80.18	79.39
50000	79.62	79.42	78.18	78.80	79.52	78.49
150000	79.59	33.84	66.02	51.66	56.71	58.84
250000	79.44	29.91	65.87	44.95	54.67	55.41
500000	79.36	29.5	58.47	32.92	54.43	45.695

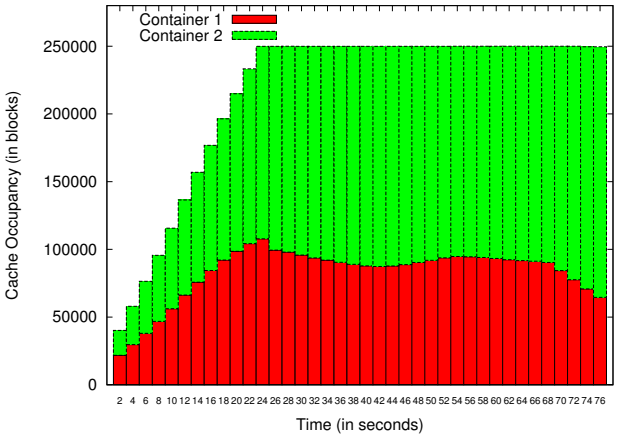
Inference: We expected that as sigma increases the unified would perform worse but from results we can say that there can be certain cases where unified may perform better if workload is having working set size very large but not completely random.



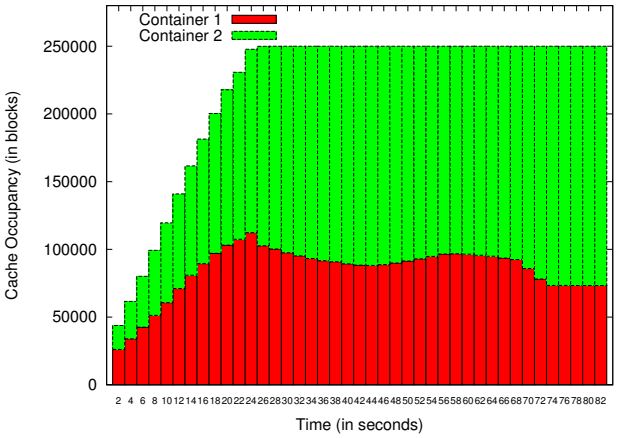
(a) IO Request pattern with σ = 10000



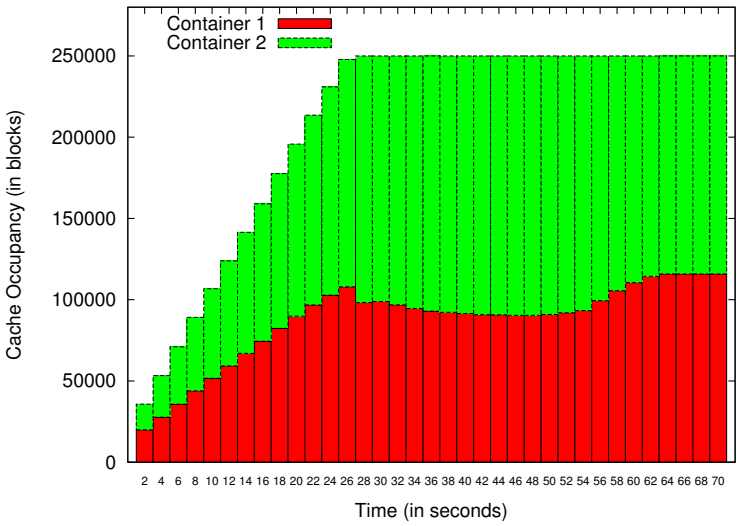
(b) IO Request pattern with σ = 50000



(c) IO Request pattern with σ = 150000



(d) IO Request pattern with σ = 250000



(e) IO Request pattern with σ = 500000

4.2.3 Real workloads

Question: How does performance get effected with statically partitioned cache and unified cache?
Setup:
Number of containers : 4
SSD - 1GB, cgroup memory limit - 256MB
Workload - 3 containers doing random reads of 2GB file using filebench benchmarking tool [19], other container running filebench's webserver workload.
Random reads has following filebench configuration : A file of size 2GB is read by 32 threads simultaneously with IO size of 1MB. Webserver has following filebench configuration : A set of files of total size 700MB is read by 10 threads simultaneously with io size of 16KB.
In unified case cache is shared across containers and in partitioned case container 1 is given cache size of 0.5 GB and rest of the containers get 0.16 GB.

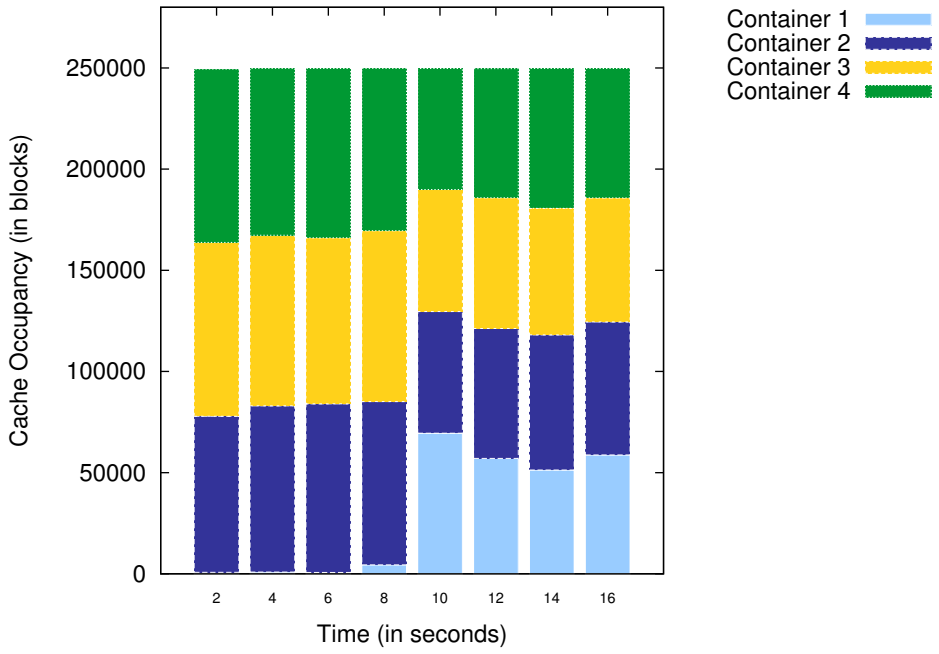


Figure 4.4: Cache occupancy of unified cache

Observation: From Figure 4.4 we can see that the workloads running in container 2, container 3 and container 4 are dominating the cache. While webserver running in container 1 has to fight for the cache. Although container 1 occupies some cache but that is not enough to improve the performance of system. As shown in Table 4.5, in partitioned setup web server has huge impact

in cache hit ratio. The performance improvement can be seen in Table 4.6. The throughput of container 1 has significant improvement in partitioned case while other containers are not effected much with the partitioning even though their hit rate reduced.

Table 4.5: Cache hit ratio of workloads

Workload	Hit ratio Unified	Hit ratio partitioned
Webserver	15.9	86.94
Random reads	12.75	10.75
Random reads	12.32	10.98
Random reads	12.97	10.60

Table 4.6: Throughput comparison

Workload	Throughput Unified (MB/s)	Throughput Partitioned (MB/s)
Webserver	12.8	28.5
Random reads	11.3	10.3
Random reads	12.6	10.1
Random reads	12.8	10.6

Inference: Thus partitioning cache helps to improve IO latency but correct partition size is apt else overall performance may drop. Also if a container is occupying more cache does not mean it will get benefited by that. As we can see in unified case even though cache was more utilized by three containers still throughput has not significantly changed with respect to partitioned case where case provisioned to these three containers was much less.

4.3 Dynamic partitioning

4.3.1 Verification experiment

Question: Is the module predicting the optimal cache size correctly?
Setup:

Number of containers : 1

Epoch time : 120 seconds

Workload : Reading a file of size 10 GB with Gaussian distribution with working set size of 142276 blocks .

Observation: Figure 4.5, shows the predicted cache size after every 120 seconds. In the end it

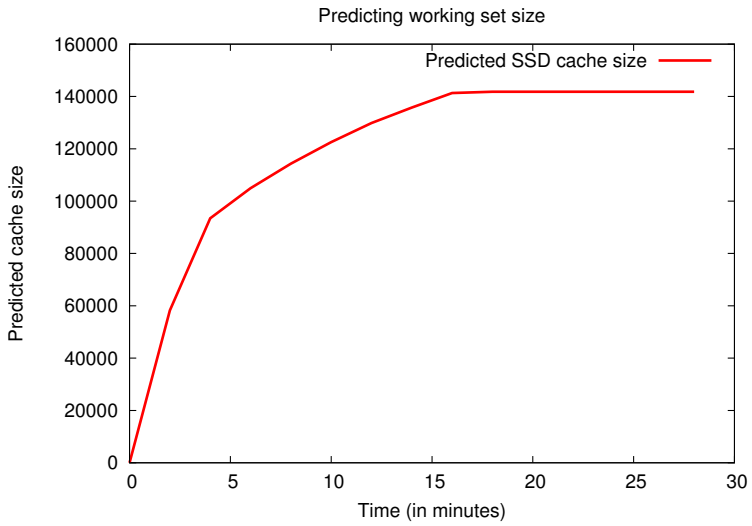


Figure 4.5: Predicting cache size dynamically

predicts working set size as 141792 blocks, which is nearly same as the expected value.

4.4 Overheads

4.4.1 Memory overhead

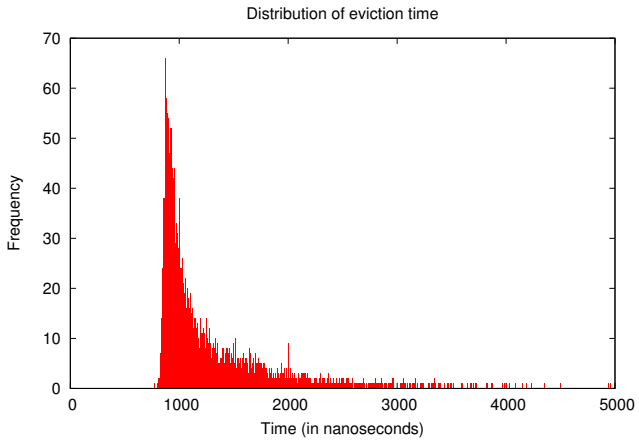
All the metadata structures are stored in memory. Data structures used are radix tree for lookup, linked list for storing free blocks, linked list for eviction. Each unique block referred will have a

structure of radix tree which uses 28 B and linked list node for eviction which uses 24 B. Each free block node uses 12 B. The per container metadata uses 40 B.

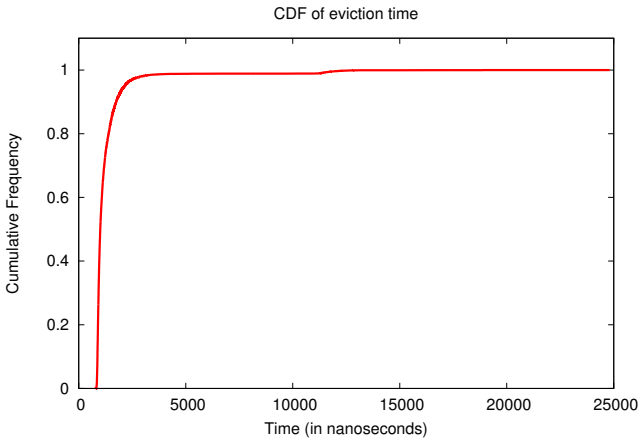
4.4.2 Time overhead

The overhead of using the module mostly lies in evicting the blocks from SSD and looking them up in search tree. Figure 4.6c shows the eviction time from SSD which is around 1000 nanoseconds. Figure 4.6d shows the lookup time in radix tree which is less than 100 nanoseconds.

Eviction Time

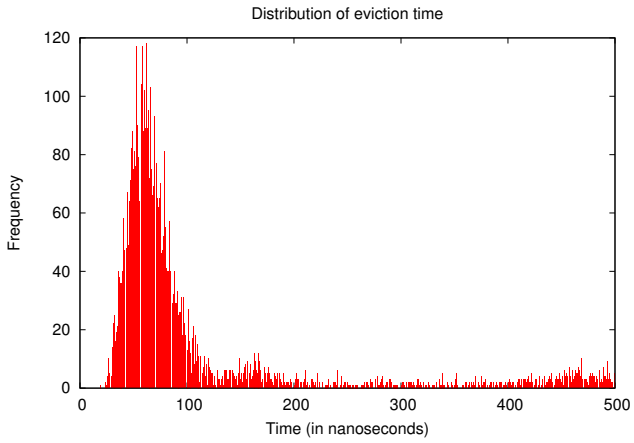


Eviction time

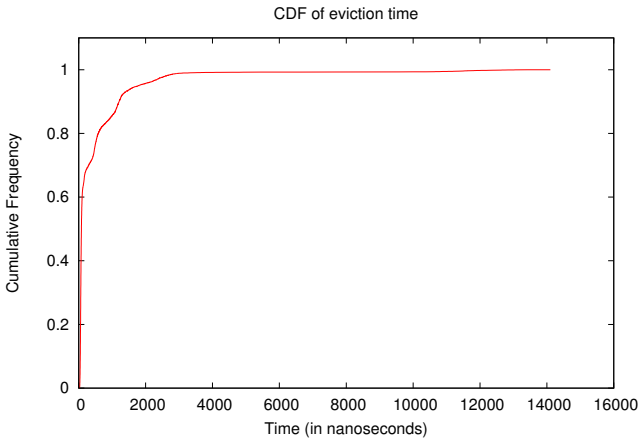


Cumulative distribution function of eviction time

Lookup Time



Lookup time



Cumulative distribution function of lookup time

5 Conclusion and Future work

The disk IO latency in virtualised systems is bottleneck in overall performance of the system. Hence introducing a new level of memory i.e. SSD as a cache to disk can help improve IO performance. We have explored a way in which SSD can be exposed as a caching layer in host operating system. Then we designed the architecture of partitioner and implemented it. We performed experiments to prove the correctness and need for extra caching layer.
As a first step we have designed and implemented a static partitioned SSD cache on per container basis and written a module which can predict the cache size online. In future following things are need to be done :

- Extensive performance based testing for dynamic partitioning.
- Design an eviction policy which makes decisions according to the eviction policy of page cache.
- Add write back caching policy as well

Appendices

.1 Cgroup Internals

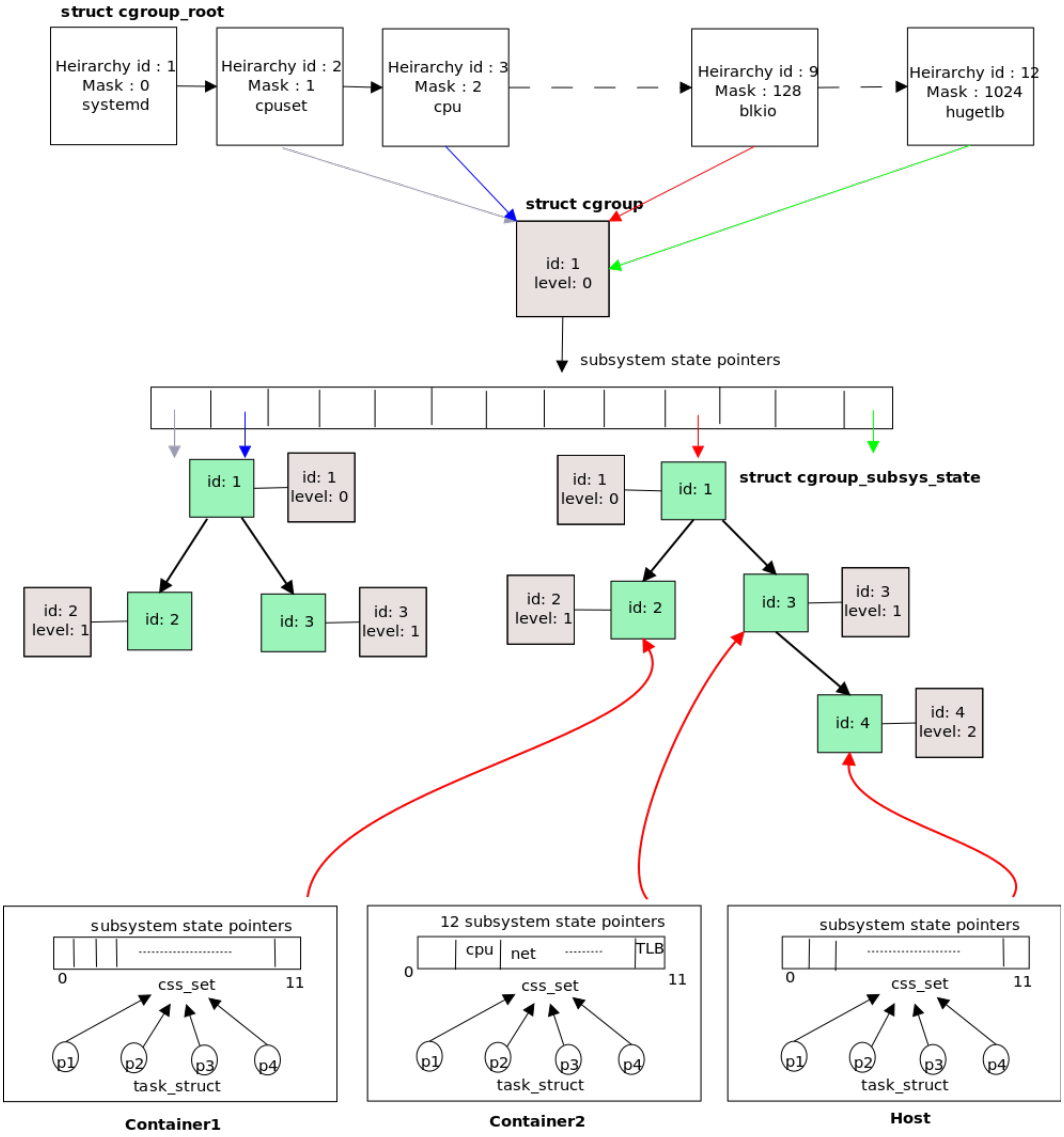


Figure 1: Internals of cgroup

Bibliography

[1] Dulcardo Arteaga, Jorge Cabrera, Jing Xu, Swaminathan Sundararaman, and Ming Zhao. CloudCache: On-demand Flash Cache Management for Cloud Computing. *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST 2016)*, 2016.

[2] Sai Huang, Qingsong Wei, Dan Feng, Jianxi Chen, Cheng Chen. Improving Flash-Based Disk Cache with Lazy Adaptive Replacement. *ACM Transactions on Storage, Volume 12 (TOS 2016)*, 2016.

[3] Yiying Zhang, Steven Swanson A study of application performance with non-volatile main memory . *Proceedings of the 31th symposium on Mass Storage Systems and Technologies (MSST), 2015*

[4] R. Koller, A. J. Mashtizadeh, R. Rangaswami. Centaur: HostSide SSD Caching for Storage Performance Control. *Proceedings of Autonomic Computing, 2015 IEEE International Conference on cloud computing*, pages 966-Âŋ973, 2015.

[5] Y. C. Tay and M.Zou. A page fault equation for modeling effect of the memory size. *Proceedings of Perform Eval*, pages 99–130, 2006.

[6] Vimalraj Venkatesan, Wei Qingsong, and Y. C. Tay. ExTmem Extending Transcendent Memory with NonÂŋvolatile Memory for Virtual Machines. *Proceedings of the 2014 IEEE Intl Conf on High Performance Computing and Communications*, pages 51Âŋ–60, 2014.

[7] Carl A. Waldspurger, Nohhyun Park, Alex T Garthwaite, and Irfan Ahmad. Efficient MRC construction with SHARDS. *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, pages 95Âŋ–110, 2015.

[8] Fei, Meng and Li, Zhou and Xiaosong, Ma and Sandeep, Uttamchandani and Deng, Liu vCacheShare : Automated Server Flash Cache Space Management in a Virtualization Environment. *Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference*, pages 133–Âŋ144, 2014.

[9] WIRES, J., INGRAM, S., DRUDI, Z., HARVEY, N. J. A.,ANDWARFIELD, A. Characterizing storage workloads with counter stacks. *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (Berkeley, CA, USA, 2014), OSDI 14* , pages 335âĂŞ349, 2014.

[10] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, Karsten Schwan Data tiering in heterogeneous memory systems. *Proceedings of the Eleventh European Conference on Computer Systems*, 2016.

[11] Dan He, Fang Wang, Hong Jiang, Dan Feng, Jing Ning Liu, Wei Tong, Zheng Zhang Improving Hybrid FTL by Fully Exploiting Internal SSD Parallelism with Virtual Blocks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2015.

[12] Hyeong-Jun Kim, Young-Sik Lee, Jin-Soo Kim. NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs. *Conference of Hot Storage*, 2016.

[13] Matias Bjorling, Jens Axboe, David Nellans, Philippe Bonnet. Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems. *Proceedings of the 6th International Systems and Storage Conference*, 2013.

[14] Junbin Kang, Benlong Zhang, Tianyu Wo, Chunmming Hu, Jinpeng Huai. MultiLanes: providing virtualized storage for OS-level virtualization on many cores. *Proceedings of the 12th USENIX conference on File and Storage Technologies*, pages 317–327 2014.

[15] Jinho Hwang, Wei Zhang, Ron C. Chiang, Timothy Wood, and H. Howie Huang. UniCache: Hypervisor Managed Data Storage in RAM and Flash. *Proceedings of the 7th International Conference on Cloud Computing*, pages 216–223, 2014.

[16] Tian Luo and Siyuan Ma. SCAVE: Extending Transcendent Memory with NonÂŋvolatile Memory for Virtual Machines. *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 103–112, 2013.

[17] Qingpeng Niu, James Dinan, Qingda Lu and P. Sadayappan PARDA: A Fast Parallel Reuse Distance Analysis Algorithm. *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 103–112, 2013.

[18] GNU scientific library. *https://www.gnu.org/software/gsl/*

[19] Filebench : File System and Storage Benchmark. *https://github.com/filebench/filebench*

[20] Gulati, A., Ahmad, I., AND Waldspurger. PESTO: Online Storage Performance Management in Virtualized Datacenters. *Proceedings of ACM SOCC*, 2011.

[21] Wikipedia. Non Volatile memory.

[22] Vimalraj Venkatesan, Wei Qingsong, Y. C. Tay, and Yi Irvette Zhang. A 3Âŋlevel Cache Miss Model for a Nonvolatile Extension to Transcendent Memory . *Proceedings of the 6th International Conference on Cloud Computing Technology and Science*, pages 218Âŋ–225, 2014.

[23] Zhengyu Yang, Jianzhe Tai, and Ningfang Mi. GREM: Dynamic SSD Resource Allocation In Virtualized Storage Systems With Heterogeneous VMs. *Under review*, 2016.

[24] Junghi Min, Sungyong Ahn, Kwanghyun La, Wooseok Chang, Jihong Kim Cgroup++: Enhancing I/O Resource Management of Linux Cgroup on NUMA Systems with NVMe SSDs. *Proceedings of the Posters and Demos Session of the 16th International Middleware Conference*, 2015.