Improving IO latency of Containers by Adaptive Provisioning of Non volatile memory

Shyamli Rao
Roll Number : 153050009

Under Guidance of Prof. Purushottam KulKarni

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

October 14, 2016

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],[14]. 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.

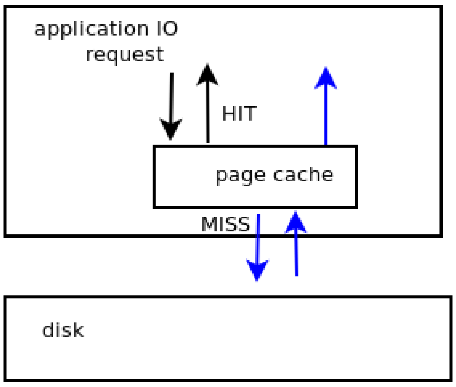


Figure 1.1: IO request flow

Using non volatile as a second chance cache will improve the IO latency. But when we expose NVM 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 both the layers of cache. 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 or write around. The selection of write policy depends on several factors such as workload (read intensive or write intensive), cache configuration, disk latency.

Latency is due to slower devices like hard disk. Hence use SSD a faster device as cache. Already much explored area in virtualised environments. But is not done explored for containers. This type of framework is not yet designed for containers. What is different from VM?

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 are 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 which 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 NVM on per container basis statically. Extend the same to partition NVM dynamically such that a user can change the NVM 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. In above setup, the IO request are coming after the page cache layer and the second chance cache size prediction is done based on these requests. Since our module is in host OS we can get the actual IO requests by by-passing the page cache. The cache size prediction using these request may be more accurate and also we can design eviction policy according to replacement policy of page cache.

5. Containers are assigned resource limits via cgroups. We can provision NVM as a resource to container via cgroup interface.

2

Background

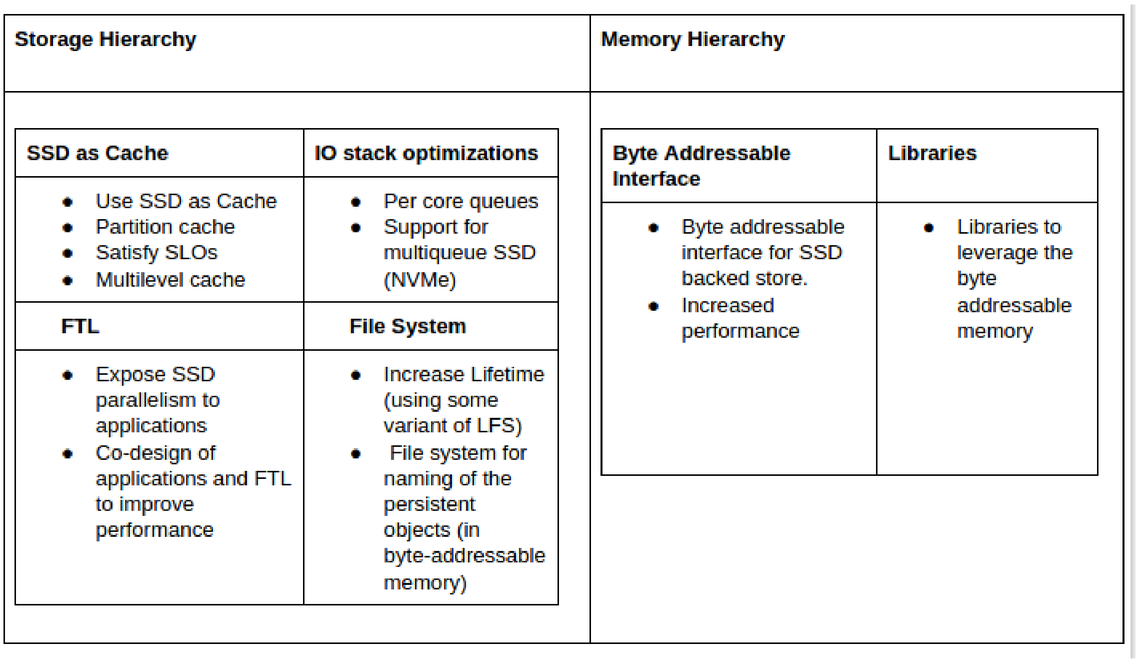
Non volatile memory is a type of computer memory which can stored information even after switching off power. There are many types of non volatile memory like flash memory, read only memory, magnetic tapes, hard disks, optical disks etc...[15]. We are working on flash storage i.e. SSD. Below Table 2.1,[3] 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

	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

2.1 Non Volatile Memory Usecases

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:



Storage Hierarchy		Memory Hierarchy	
SSD as Cache	**IO stack optimizations**	**Byte Addressable Interface**	**Libraries**
• Use SSD as Cache • Partition cache • Satisfy SLOs • Multilevel cache	• Per core queues • Support for multiqueue SSD (NVMe)	• Byte addressable interface for SSD backed store. • Increased performance	• Libraries to leverage the byte addressable memory
FTL	**File System**		
• Expose SSD parallelism to applications • Co-design of applications and FTL to improve performance	• Increase Lifetime (using some variant of LFS) • File system for naming of the persistent objects (in byte-addressable memory)		

Figure 2.1: 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 nad application-level performance [10].

- 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 [9]. 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 [11]. 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 [13],[12].

2.2 Caching

Caching 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

2.3 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:

2.3.1 Unified cache

The cache is shared by all containers as shown in Figure 2.2. 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 2.2: Unified Cache

2.3.2 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 won't 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.

2.3.3 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.

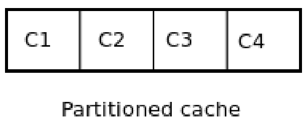


Figure 2.3: Partitioned Cache

2.3.4 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 The partitioned cache can be static or dynamic.

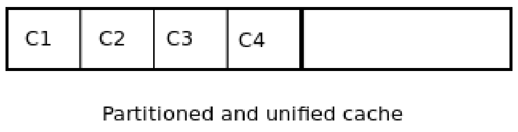


Figure 2.4: Unified and Partitioned Cache

2.4 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 **??** 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.] Generation of miss rate curve from the IO traces is memory as well as time consuming process. The IO traces tells the access pattern of 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 mattson's stack algorithm or by using cache miss equation [5].
The mattson's 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. Recently SHARDS algorithm [7] (a variant of mattson's 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.

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

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 as shown in Figure .

2.5 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 2.5 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 the 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 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 utilization is given by soft limits.
In virtualised systems when we were 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 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.

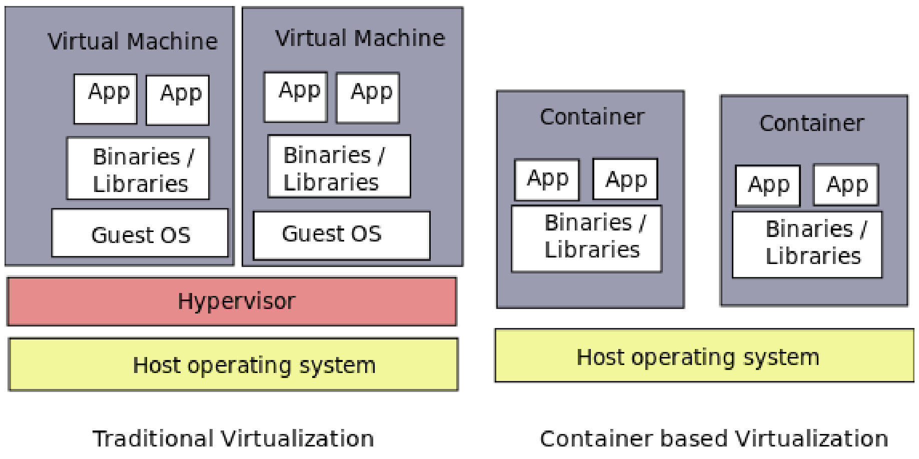


Figure 2.5: Traditional virtualization and Container based virtualization

Figure 4.2 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 their 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.

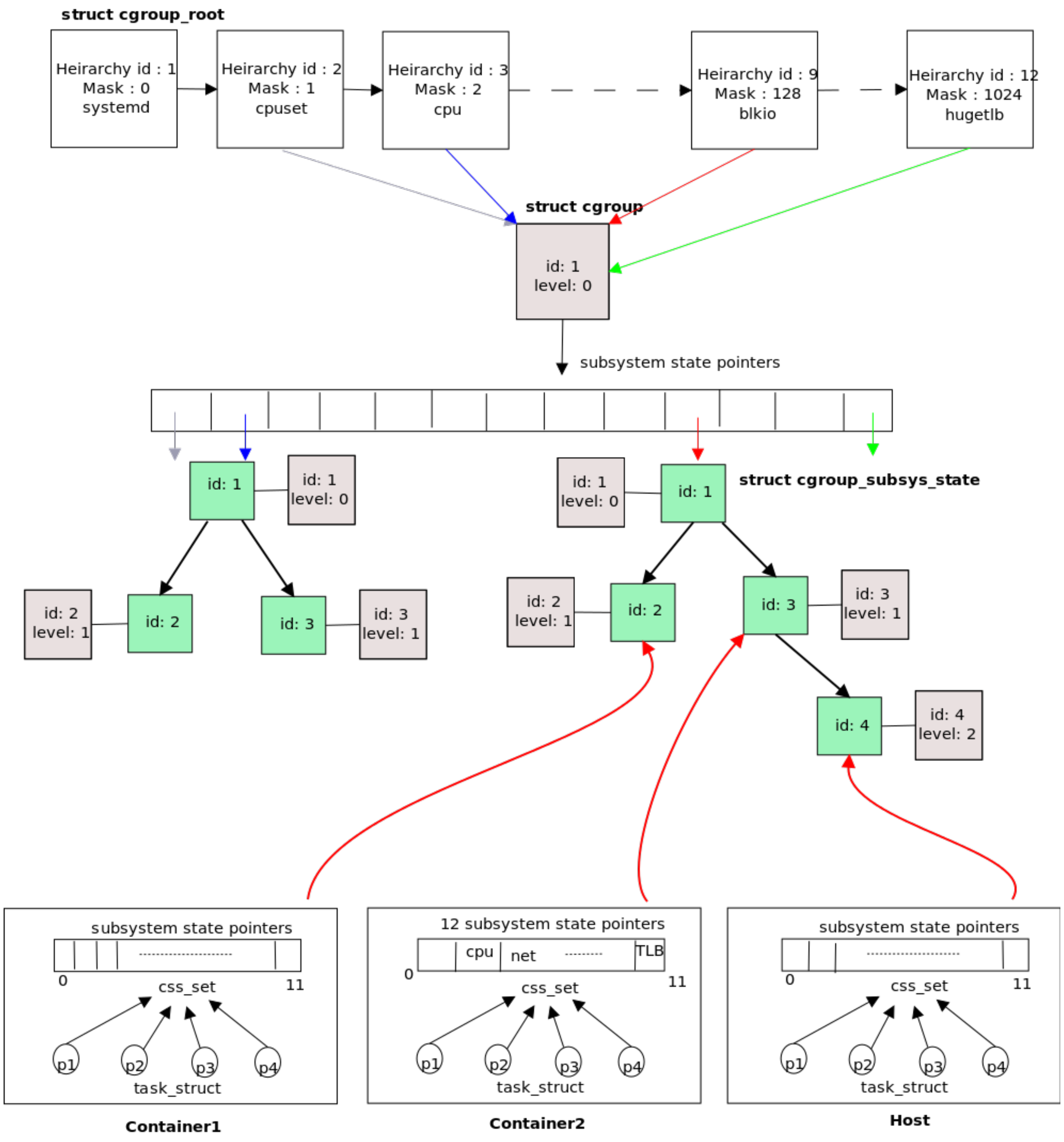


Figure 2.6: Internals of cgroup

3

Design

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 4.2 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

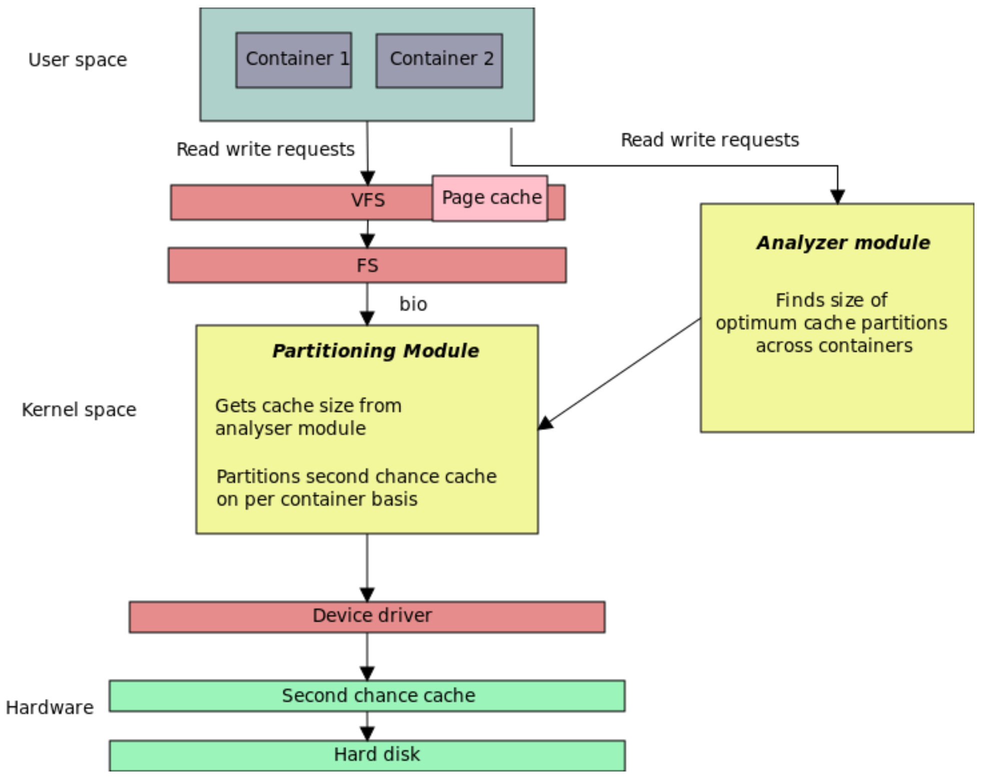


Figure 3.1: Architecture

has mapping of disk block number to cache block number. Also it maintains a global list of free cache blocks 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.

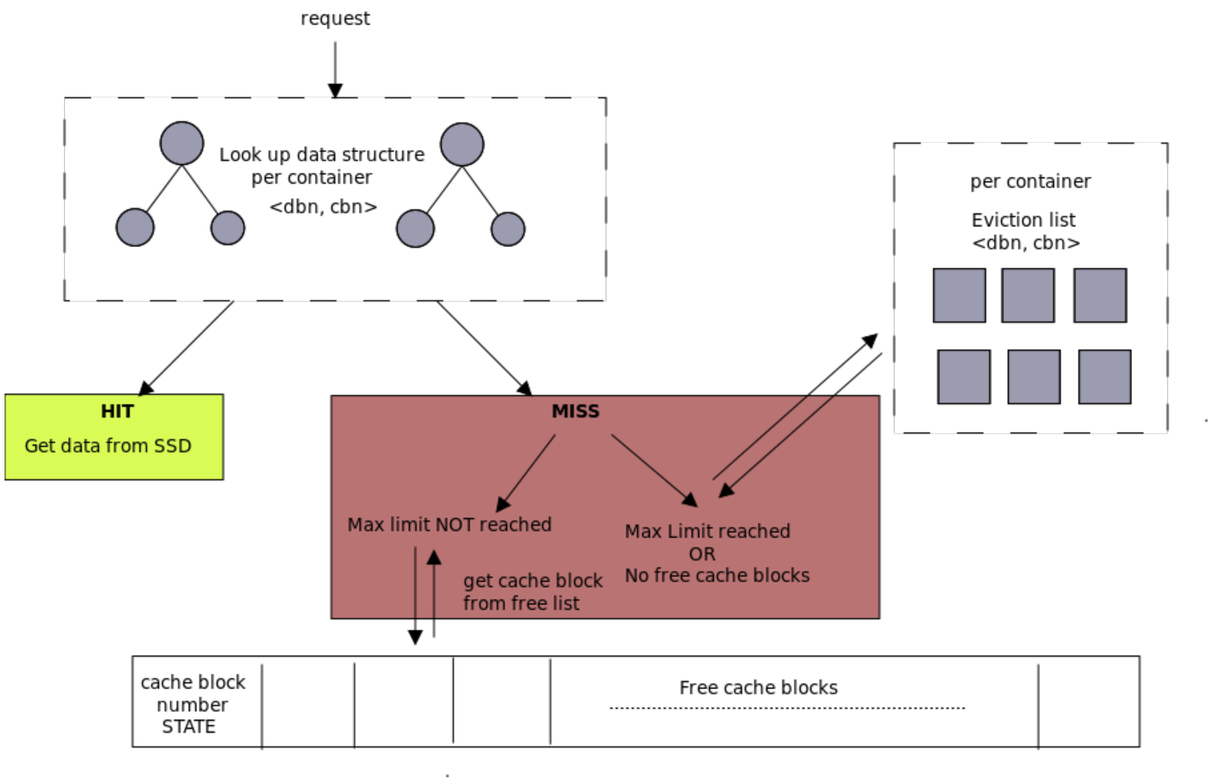


Figure 3.2: Proposed design

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 : READ or WRITE request, time stamp, container ID, is it present in page cache. 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.

4

Implementation

We implement the above architecture in Linux operating system using LXC containers. Before getting into actual implementation, we have briefly described about the components needed to implement the partitioning module.

4.1 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 4.1, 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.

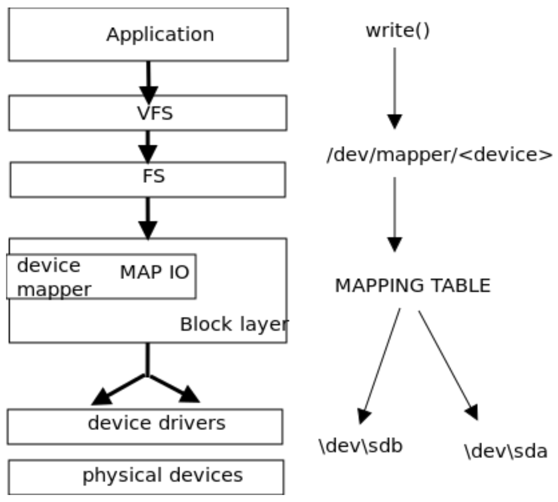


Figure 4.1: Device mapper

- bcache : This is also present in kernel source tree. It does not use device mapper framework and works at block layer. Problem with this is, before using SSD as a cache to HDD, the disk had 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.

4.2 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 goto 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 compiled)
- It works in block caching layer below file systems.
- IO request → SSD cache → HDD.

- Build using device mapper(dm)

Flashcache features:

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

4.2.1 Architecture

The cache structure is 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 4.2, 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 = (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 requests goto disk.

4.2.2 Flaws

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

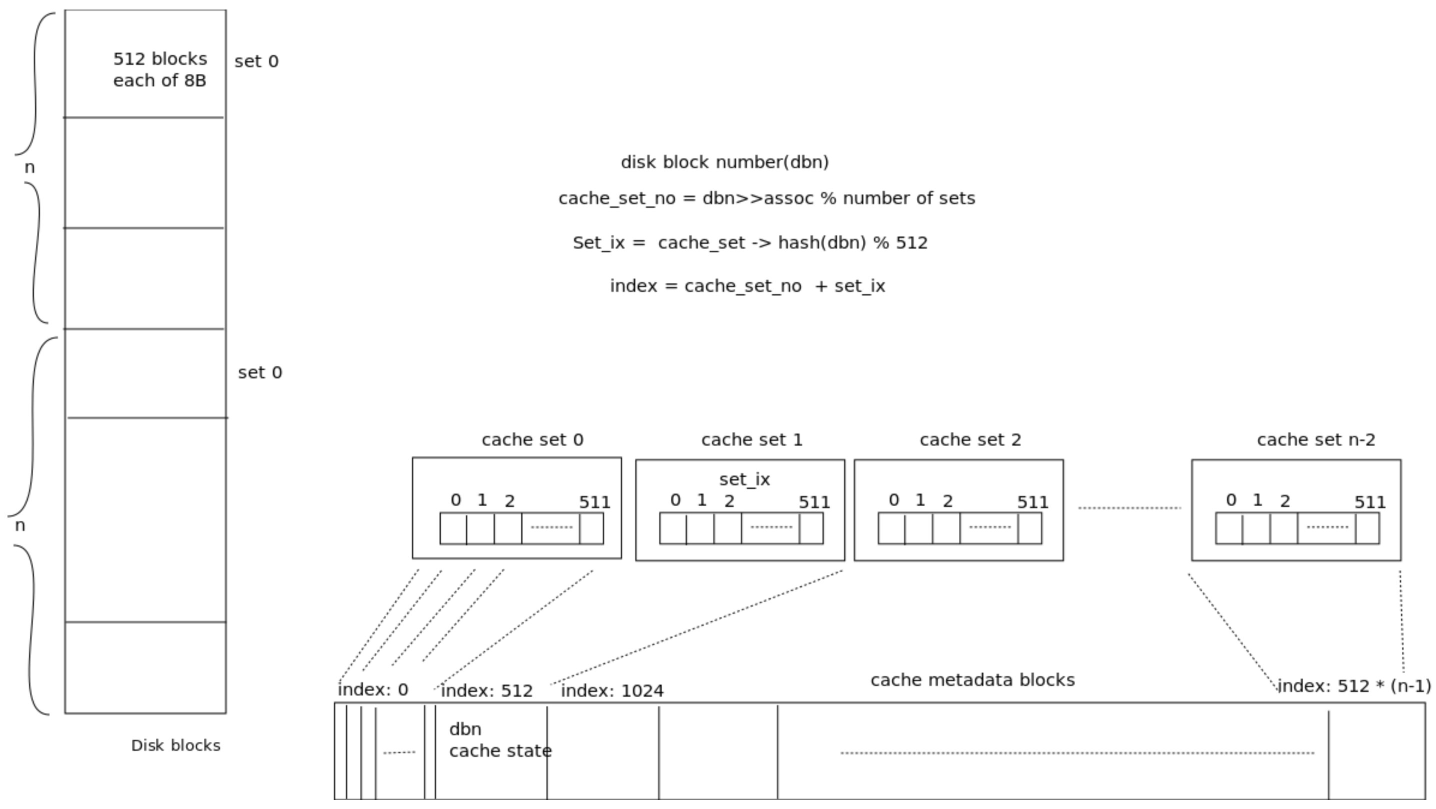


Figure 4.2: Flashcache cache structure

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.

4.3 Current implementation

We will look into two aspects, one is lookup and other is handling of read write requests.

4.3.1 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 FIFO but we will make it a pluggable interface so that we can call any eviction policy according to container's workload.

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
```

4.3.2 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

4.1. 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 is write bypass, where all the writes are bypassed to disk. If there is a write hit, then data present in SSD is flushed back to disk abd removed from SSD. Later this may change to write through or write back according to our design.

Table 4.1: 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

5

Experiments

All the experiments were done on Linux operating system (kernel -4.6) with 8GB RAM. Container used was LXC with the default parameters set.

5.1 Correctness experiment

This experiment was performed to check if SSD is caching the blocks correctly and there was no limits on SSD cache size.
Setup : Initially a file is read, then same file is read again without dropping page cache contents and also by dropping page cache contents. The block size is 4KB. This experiment was done for various file sizes as shown in Table 5.1. The misses and hits shown in table are for SSD.

Observation : Initially a file is read, then the blocks of file will not be present in SSD hence all the blocks will MISS the cache. When these blocks are read from disk, both page cache and SSD is populated. Again when the same file is read, HITs on SSD will be zero since all the blocks will be present in page cache. Hence we have dropped the page cache contents. Now when we read the same file again all the blocks will HIT SSD.

Conclusion : It can be verified that the module is storing 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.

Table 5.1: Correctness verification

	File size	No. of Blocks	1st run	2nd run with page cache	3rd run without page cache
Experiment 1	8KB	2	MISS : 2 HIT : 0	MISS: 0 HIT : 0	MISS: 0 HIT : 2
Experiment 2	1MB	256	MISS : 256 HIT : 0	MISS: 0 HIT : 0	MISS: 0 HIT : 256
Experiment 3	8MB	2048	MISS : 2048 HIT : 0	MISS: 0 HIT : 0	MISS: 0 HIT : 2048

5.2 Adhere to cache limit

This experiment is to check if SSD cache limit of a container is set, then the container size must not exceed this limit.
Setup : A 8MB file (2048 blocks) is read sequentially in experiment 1 and in a loop in experiment 2 . The SSD cache limit is set to 1000 blocks. The misses and hits on SSD cache are recorded. The eviction policy used is FIFO. The results are shown in Table 5.2.

Table 5.2: Eviction verification

	File size	No. of Blocks	1st run	2nd run with page cache	3rd run without page cache
Experiment 1 Sequential	8MB	2048	MISS : 2048 HIT : 0	MISS: 0 HIT : 0	MISS: 2048 HIT : 0
Experiment 2 In loop of 60 blocks	8MB	2048	MISS : 60 HIT : 0	MISS: 0 HIT : 0	MISS: 0 HIT : 60

Observation : In experiment 1, a file of 2048 blocks is read sequentially. Initially 1000 blocks will be miss and will be written into SSD. Next all blocks will replace these 1000 blocks and occupy SSD cache. Since cache limit is 1000. Hence in 3rd run when this file is read again all be misses on SSD. In experiment 2, 60 blocks of a file were read in a loop twice. So first time it will be miss for all blocks. Since cache limit has not been reached, no blocks will be replaced. Hence in second read without page cache, all will be hits.

Conclusion : Eviction algorithm FIFO works well.

5.3 IO latency and throughput

This experiment is to see how provisioning SSD as cache can decrease IO latency improve throughput. The throughput of SSD without using it as cache is 250 MB/s
Setup : Files of different sizes 8MB, 256MB, 512MB and 1GB were read from disk, then from page cache, then by dropping page cache, data was read from SSD.

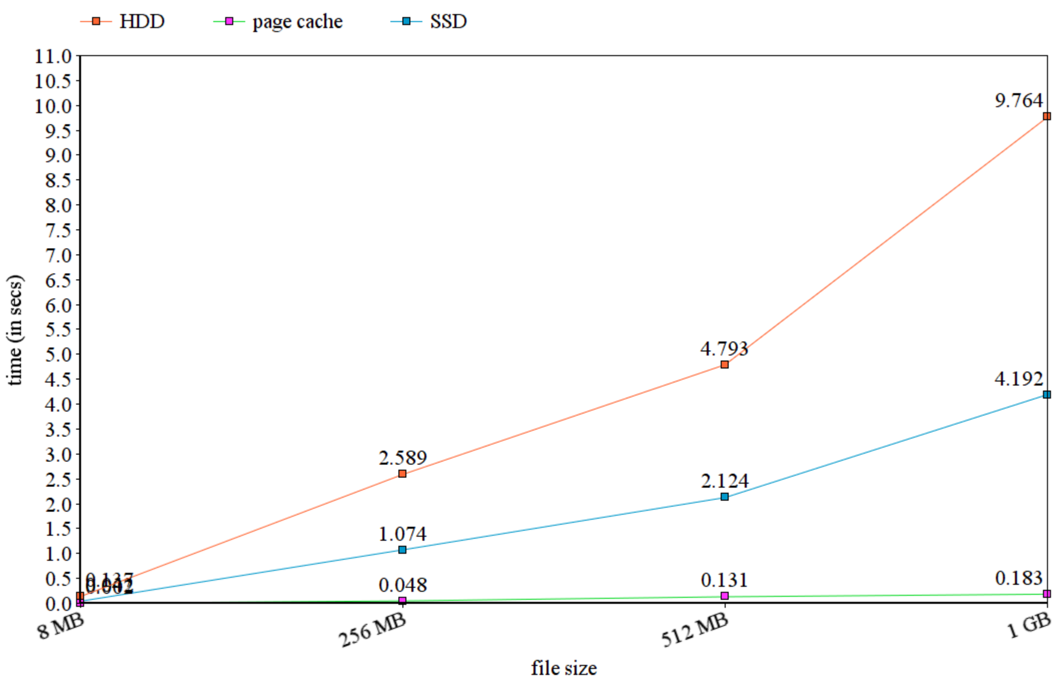


Figure 5.1: Access time

Observation : Reading from disk takes the longest time. SSD takes much less time than disk.

Table 5.3: Throughput comparison

File size	Disk	SSD	Page cache
256 MB	98.304 MB/s	235.52 MB/s	5.2 GB/s
512 MB	106.82 MB/s	235.52 MB/s	3.8 GB/s
1 GB	104.87 MB/s	235.52 MB/s	5.46 GB/s

And page cache takes least time. As file size increases the time difference is also increasing. Throughput of the RAM is much grater than SSD and HDD.
Conclusion : Using SSD as second chance cache can improve IO latency.

6

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 explore 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 some 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. In future following things are need to be done :

- We have to extend this for dynamic partitioning.
- The analyzer module is to be designed and implemented which will monitor each containers working set size and predict the optimum partition sizes.
- Design an eviction policy which makes decisions according to the eviction policy of page cache.
- Extending implementation of cgroups for setting up SSD storage limits via our module.

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 16)*, 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 16)*, 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 966–973, 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 Nonvolatile Memory for Virtual Machines. *Proceedings of the 2014 IEEE Intl Conf on High Performance Computing and Communications*, pages 51–60, 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 95–110, 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 133–144, 2014.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] Wikipedia. Non Volatile memory.