Caching Techniques to Improve Disk IO Performance in Virtualized Systems

Shyamli Rao
Roll Number : 153050009
Department of Computer Science and Engineering
I.I.T. Bombay

May 2, 2016

Contents

List of Figures

List of Tables

1

Introduction

In recent years virtualization has become very common which has lead to higher virtual machine (VM) to physical machine consolidation since it helps to reduce hardware resource utilization and maintenance costs [6]. In virtualized environment, each virtual machine runs its own operating system on the same physical machine. The resources of the physical machine are virtualized and exposed to the virtual machines. The resources shared across virtual machines include cpu, memory, network and storage. The software entity called hypervisor, multiplexes these resources across VMs. Hypervisor should be aware of various techniques for allocating and managing these resources across virtual machines. Here we are mainly concerned with memory and storage management. Since the memory is shared, we need to ensure that it is allocated fairly across all virtual machines, the allocation is dynamic in nature, we should be able to transfer the memory allocations across VMs. Managing all these has to be efficient. There are several techniques for effective memory management like ballooning, page sharing, demand paging, compression, caching [12]. In a similar way like memory resources, we need effective management of storage resources too. The storage must be dynamically and fairly allocated across all virtual machines, proper mapping of file pages to disk blocks must be done.
In managing all these resources one more aspect to look into is performance. Both for memory and storage resources, access latency is important metric when performance is concerned. One of ways to improve IO performance is through caching.

Caching improves the IO performance of the system by reducing IO latency. It is more effective in modern data center architecture because the data is stored in network attached storage (NAS). Every disk IO request has to go through network which increases IO latency. Therefore 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 in guest VM initiates an IO request. First guest virtual machine checks if data is in guest VM's page cache, if not goes to hypervisor. Hypervisor issues a disk read and will fetch the blocks and gives back to the application. So if there is no cache, then every IO request must goto disk and disk access latency is much higher than RAM access latency. Recent trends have shown that caching can be done in different ways, for

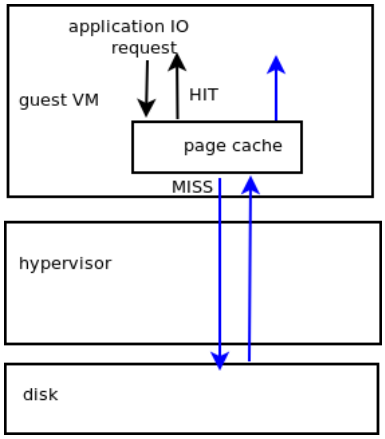


Figure 1.1: IO request flow in hypervisor

example cache can be page cache or in memory cache controlled by hypervisor [10] or SSD cache [1]. The in memory cache is volatile and smaller in size while SSD is non volatile and larger in size. There are different types of cache and various ways in which it can be used in virtualized environment. This will point to following questions:

1. Why is caching required?
    - Performance : Caching improves the performance of virtualized systems by reducing disk IO latency.
    - Hypervisor's control on storage hierarchy : If hypervisor could control guest OSs caching mechanism by controlling page cache then storage performance could be improved, but hypervisor has no control over guest's cache. Hence there is no performance control and isolation guarantee for storage. Hypervisor managed caching can be used as storage performance control and also be used in prioritizing virtual machines.
2. How cache can be used in virtualized systems?
    - Various caching techniques can be used like single level caching, multilevel caching, caching with partitioning or without partitioning.
    - Cache configuration : In virtualized systems cache can be placed anywhere, either in hypervisor or in guest VM or in server side caching or combination of these.
3. 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 SSD'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.

1.2 Scope of Seminar

The scope of the seminar is to

- study various ways in which a cache can be used within virtualized environment, understand which is better than other and why.
- study different caching configurations
- study and compare cache management techniques
- look at various experiments
- explore future work in this area

2

Background

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 virtual machine IO latency
3. Enable maximum utilization of storage resource
4. Increase IOPS
5. Prioritize virtual machines
6. Do Fair allocation
7. Satisfy service level objectives

2.1 Cache configurations in virtualized systems

From the Figure 2.1 we can see that cache can be placed at several locations in virtualized system like inside guest virtual machine (VM) or in hypervisor or in the storage. The possible cache positions are as follows:

- within guest VM : This is page cache which operating system uses for caching disk blocks in memory (RAM) which is in control of guest VM.
- hypervisor controlled in memory cache : This is in memory cache which hypervisor can use for caching any of the guest VM's data.

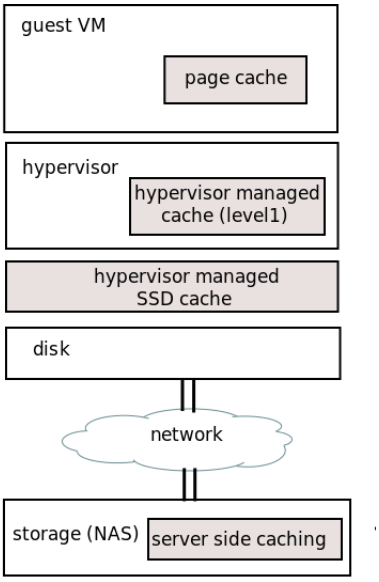


Figure 2.1: Design options for caching in virtualized environment

- hypervisor controlled SSD cache : This is flash storage which is not there in default IO request dataflow. Hence hypervisor must explicitly check if data is present in SSD or not. This is non volatile and much larger as compared to in memory cache. Also SSD IO latency is less than disk latency but higher than that of RAM.
- Caching at server side : All the above types were host side caching. We can also do server side caching where requests are checked first in server side cache then in the actual storage. This is effective when network latency of system is lesser than the disk latency at server.

There are several cache configurations possible. Each cache configuration will have combination of different possible position of cache as listed below:

1. Cache within guest VM : This is the default caching style in which page cache is in control of guest OS, while hypervisor has no control over this cache. The page cache of a virtual machine is controlled by its own guest OS.
2. Cache within guest VM and in memory cache managed by hypervisor : In this architecture, page cache is controlled by guest OS while in memory cache is controlled by hypervisor. The hypervisor controlled memory can be shared by all VMs or could be divided across VMs.
3. Cache within guest VM and SSD cache managed by hypervisor : In this architecture, the in memory cache is controlled by guest OS while SSD acting as cache, is controlled by hypervisor. The SSD could be either shared by all VMs or could be partitioned
4. Caching at multiple levels in hypervisor : Hypervisor may manage multiple levels of cache like in memory cache, SSD cache or two or more layers of SSD cache. Each layer can be used differently, for example layer 1 and layer 3 can be shared across all VMs but layer 2 could be partitioned.

Table 2.1 shows the references which have used these combinations of cache positions in virtualized environment.

Table 2.1: 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	✓			
[10]	✓	✓		
[3] [6] [5] [14]	✓		✓	
[11] [4]	✓	✓	✓	
[1]	✓		✓	✓

2.2 Cache management subcomponents

In this section we will discuss the techniques for effective management of cache in virtualized environment. Since cache size is limited we must ensure that blocks with good recency must be cached and when cache is full we must evict a block from cache. This has to be handled using proper cache eviction policy.
Cache can be allocated such that all VMs share it, or we can partition it on per VM basis. If we are partitioning cache we must find out how to partition cache across all VMs. Each VM will be running different types of workloads, hence each VM will have its own cache requirements. There are several techniques to find out cache requirement of VM which we will see.

2.2.1 Cache eviction policy

Cache in general is a limited resource which is much faster and costlier than the next level memory. Cache should be used to store the pages which have higher probability of getting accessed later. This will improve the hit rate which in turn reduces IO latency. When cache becomes full, we must evict a page to add new page which may have more future accesses than evicted page. When SSD's are used as cache, the blocks must be evicted and inserted cautiously since SSDs wear out faster. So it becomes an important concern to find which page should be inserted into cache and which page to be evicted from cache. There are different cache replacement algorithms like FIFO (First In First Out), LRU (Least recently Used), LFU (Least Frequently Used) , ARC (Adaptive Replacement Cache), mARC (multi-modal Adaptive Replacement Cache]) [7]. The usual cache replacement policies like LRU may not work well in virtualized systems. Since LRU assumes that cost of bringing back any object into cache is same and the size of object to be cached is same across all VMs [4]. Therefore we should have some custom cache eviction policy which will work effectively. In multilevel caches, each level

may use a different cache eviction policy. We can choose any replacement policy according to our workload. We can configure or dynamically determine which cache replacement policy works best for a particular workload running on VM [6]. Figure 2.2 shows how multilevel caches can use different

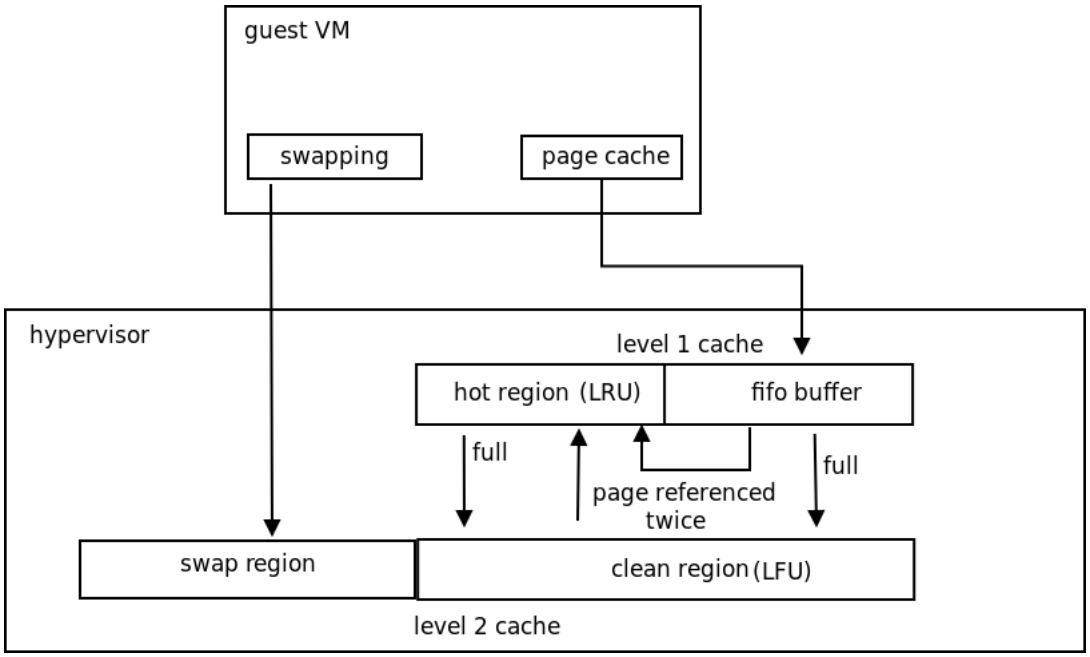


Figure 2.2: Different cache eviction policies in multilevel caching

cache replacement policies at each level [10]. As we can see, two levels of hypervisor managed cache are used. One way to evict data is by using LRU cache replacement policy at level 1. But problem is that LRU may replace a warmer page with colder page. Therefore level 1 cache is split into two regions, one is for caching hot pages which uses LRU and other is fifo buffer cache. An evicted page from guest VM is inserted into fifo buffer, if it is referenced twice then it is inserted into hot region. The level 2 cache uses LFU cache eviction policy. The first level captures captures recency and second level captures frequency of accesses.

Inclusive and Exclusive caching

Multilevel caching can be inclusive or exclusive in nature. Inclusive caching means same page can be present in all the cache levels. When there is cache miss at level 1 and cache hit at level 2 , then the page is copied from level 2 to level 1. So there will be two copies of same page at two different cache levels. If page in one of the cache levels is modified then it may lead to inconsistency. This type of caching is done in CPU L1, L2 caches or distributed network caches.
Exclusive caching means the page can be present in only one of the cache levels. When a miss occurs at level 1 and hit at level 2 then data is copied from level 2 to level 1 and then removed from level 2. The advantage of exclusive caches is that cache can store more data and there is no issue of inconsistency.

Write policies

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. Following is the brief description of each of these policies:

1. Write through : Data is read from cache if present, else read from disk. Data is modified in cache (if present) as well as in disk. Advantage is that there is no inconsistency between cache and disk, but problem is all writes must goto disk. This type of write policy is good for applications that write and re-read data or just read data (read intensive workloads).

2. Write around : Data is read from cache if present, else read from disk. Data is modified only in disk, bypassing the cache. Advantage is cache won't be flooded with write IO that will not be re-read but disadvantage is each write after read will be a cache miss.

3. Write back : Data is read form cache if present, else read from disk. Data is modified in cache if present else written to disk. This results in low latency for write intensive workloads but may lead to data inconsistency.

2.2.2 Flavors of caching

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

Unified cache

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

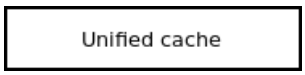


Figure 2.3: Unified cache

Static partitioning

Static partitioning of cache means each VM will get configured amount of cache as shown in Figure 2.4 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 VMs use unified SSD cache and another in which SSD is partitioned on per VM basis. Type of workloads running on each VM is as follows:
VM1 - workload which has less temporal locality but high IO request rates (100 blocks per sec).
VM2 - workload which has high temporal locality but IO request rate is moderate (50 blocks per sec).
Assume scheduling time is 1 sec for each VM and cache size is 100 blocks.
Case 1 : Unified Cache - When VM1 is scheduled it will cache 100 blocks occupying full cache.

Next VM2 is scheduled which will replace 50 blocks of data of VM1 with its own data which is fine since its data blocks have better temporal locality than that of VM1s. Next when VM1 is scheduled it will replace all the blocks in cache with its own data and the blocks with lesser locality replace blocks with higher locality. Next time when VM2 is scheduled it will get miss instead of hit for those replaced blocks in cache layer. This leads to higher IO latency since it decreases hit rate.
Case 2 : Partitioned cache - If cache is partitioned, for example 50 blocks to VM1 and 50 blocks to VM2 then performance will be better since VM1s blocks won't replace VM2s blocks.
So we can say that cache partitioning is required for better hit rate. Also partitioning the cache gives hypervisors control to prioritize the VMs. For example if VM is of high priority then hypervisor can statically allocate more cache to it.

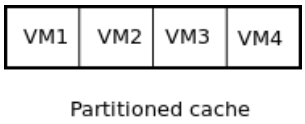


Figure 2.4: Cache partitioning on per VM basis

Dynamic partitioning

Each VM runs different type of workload and each workload has its own working set size. If a VM get a static cache allocation more than its working set size then there wont be any significant performance benefit as compared to cache size equal to working set size and we will also be wasting cache. Hence dynamic partitioning is preferred over static partitioning since it reduces cache wastage [6].
Let us consider similar example as above.
Case 1 : static partitioning with VM1 - 50 blocks and VM2 - 50 blocks
Case 2 : dynamic partitioning which is based on temporal locality so VM1 - 5 blocks and VM2 - 95 blocks.
In case 1, VM1 wont be utilizing the whole cache since its working set size is only 5 blocks so no point in giving 50 blocks to it. So there is cache wastage of 45 blocks.
In case 2, VM1 gets sufficient amount of cache while VM2 can utilize the cache better since it has better locality of reference.

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.

2.2.3 Determining per VM cache size

One of the most common ways to determine optimal cache size required is based on workloads (which is running on VM) 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

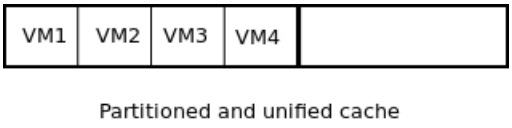


Figure 2.5: Combination of unified and partitioned cache

set size does not further decrease miss ratio. So we try to allocate each VM 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 VM, we must construct miss rate curve of that workload.

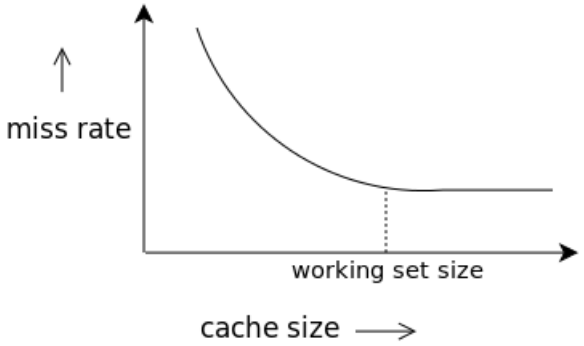


Figure 2.6: Miss rate curve

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 VM 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 [8].
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 [13] (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.

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

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 as shown in Figure 2.7.

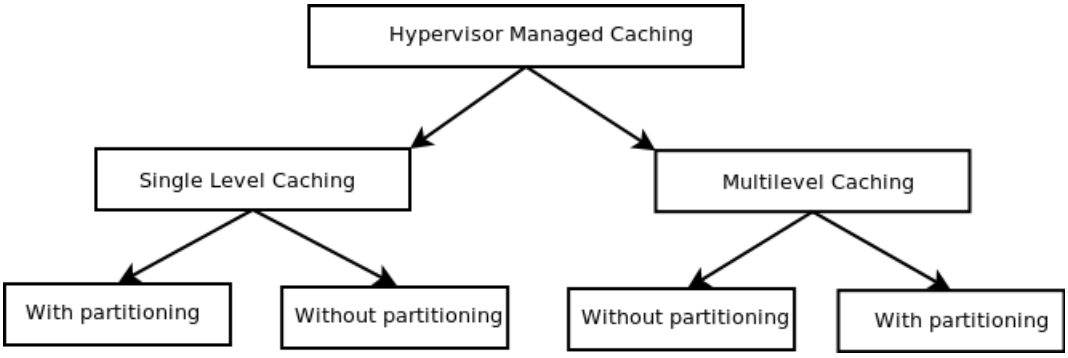


Figure 2.7: Different types of hypervisor managed caching

3

Single level caching

In this architecture only one level of hypervisor controlled cache is present. The cache can be an in memory cache or a SSD cache which will come in between the disk and the guest page cache. The cache can be either partitioned or can be used without any partitioning.

3.1 In memory cache without partitioning

3.1.1 Transcendent memory

An example of hypervisor managed in memory cache is Transcendent memory. Transcendent memory is used for improving utilization of physical memory by claiming under utilized memory in host and making it available to other VMs. This memory is in control of hypervisor and can be used for caching VM's data to improve performance. Tmem API provides two services one to create tmem pool and other is operations on the pool - put and get [9]. Transcendent memory pools can be ephemeral or persistent and shared or private. Tmem pool creation function has a flag parameter which choses the pool type. Here all VMs are sharing tmem therefore tmem pool will be shared. Tmem pool can be persistent (used as frontswap) and ephemeral (used as cleancache). A put to tmem means page is to be inserted into it. A get from tmem is to get a page from Tmem and remove it from Tmem. This makes caching exclusive in nature. Frontswap handles dirty pages while clean cache handles clean pages. Frontswap is used for swapping dirty pages, instead of throwing pages back into swap disk they are kept in frontswap. This is persistent in nature. A put in frontswap may not always succeed since if there is no space in frontswap it won't evict other pages to accommodate new pages. A get of successful put will always succeed since pages present in frontswap will only be removed if get is done on page. A successful put or get in frontswap avoids disk write or read.

Clean cache is used for caching clean pages which are removed from the page cache due to memory pressure. This is ephemeral in nature. A put in clean cache will always succeed since if there is no space then a page will be removed from the Tmem. This removal does not need disk write since it's clean page. A get (of a successful put) from cleancache may fail since we are removing the pages for insertion of new pages. A successful get from clean cache avoids disk read. By default Tmem is used as unified cache just like page cache but since it is in control of hypervisor it could be partitioned on per VM basis. Figure 3.1 shows Tmem put operation in frontswap and cleancache. When a page is

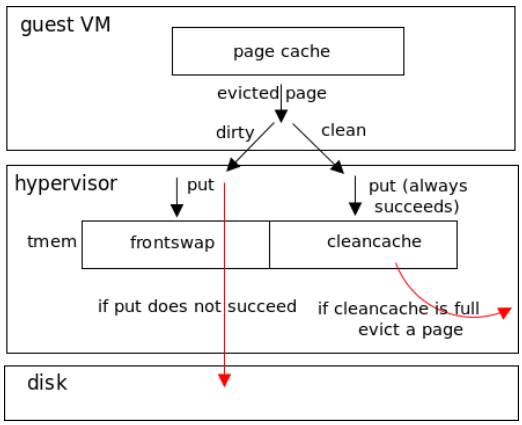


Figure 3.1: Tmem put operation

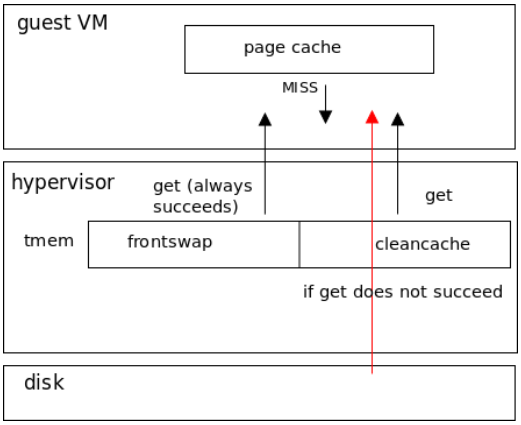


Figure 3.2: Tmem get operation

evicted from page cache of guest VM then hypervisor checks if the page is dirty or clean. If page is dirty, then tmem put is called on frontswap. If frontswap has space then page is placed in it else it is stored on disk. If page is clean, then tmem put is called on cleancache. If cleancache has free space then page is placed in it else any one of the pages is evicted and new page is inserted. In Figure 3.2 get operations on tmem are shown. When a page miss occurs in page cache and page had successful put in frontswap then get is done on frontswap. A get on cleancache may or may not fetch the page. If page is not in cleancache then issue a disk read.

3.1.2 IO request flow

Figure 3.3, depicts the application IO request flow with single level of hypervisor managed cache. The application inside guest VM initiates IO request. Guest OS first checks if page is present in page cache, if it is not then request goes to hypervisor. Hypervisor checks if page is present in hypervisor managed cache. If it is hit then page is copied to page cache and given to application, else the request goes to disk. Then data will be fetched from disk copied to guest VM's page cache and to application.

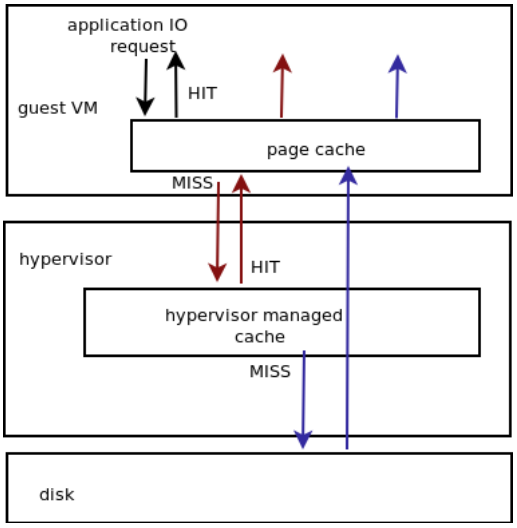


Figure 3.3: IO request flow in single level caching in hypervisor

3.2 SSD based dynamic cache partitioning

Dynamic SSD partitioning can be done in several ways depending on the objectives to be satisfied. Partitioning can be done for fair allocation, for minimizing overall IO latency, for improving per VM performance, for increasing IOPS or to satisfy service level objective. The conventional cache partitioning techniques like miss rate curve construction cannot be used for SSD partitioning since it only considers working set size of the workload but not other parameters like disk latency, recovery cost. Existing CPU cache partitioning techniques will not be able to manage SSD based caches properly since SSD caches differ in various aspects like response time, space availability, access patterns, write endurance and performance constraints.

3.2.1 Metrics used for partitioning

Following are the metrics (function of cache size) which can be used to partition cache :

1. Cache utility = Cache hit ratio × Read ratio × Reuse intensity × Device latency, where reuse intensity is the rate at which cache is accessed. This is to be maximized for every VM. [3]

2. Latency metric= Miss rate × IO latency of disk + (1 - Miss rate) × Latency of SSD. This is to be minimized for all VMs. [6]

3. Cost metric = Miss rate × recovery cost, where recovery cost depends on device latency and granularity at which SSD is accessed. [4]

4. Reuse working set size, which is the set of distinct data blocks that a workload uses more than N times in T time period. [1]

5. Cache miss equation to find cache partition size directly. [11]

Table 3.1 shows the various objectives satisfied by these metrics in the papers read. Of course all of the above metrics could be modified to satisfy all the given objectives. For example, cost metric can also be used to minimized overall IO latency, similar to cache utility.

Table 3.1: Metric considered for objectives

	Cache utility	Latency metric	Cost metric	Cache miss equation
Objectives				
For minimizing IO latency of system	✓	✓		✓
For fair allocation		✓	✓	✓
For service level objective		✓		✓
For prioritizing VMs	✓	✓	✓	

3.2.2 Algorithms

Algorithm 1 presents a high level idea on how hypervisor does cache partitioning for minimizing IO latency in virtual environment. Here t_w is window size, S_i is cache size of VM_i and P_i is the priority of VM_i. For configured time t_w, IO traces are collected for each VM. For each VM, miss rate curve is constructed using IO traces to get cache miss rate as function of cache size. Cost metric C_i is calculated for each VM. Cost metric can be any of the above stated metrics. For each VM_i, we get cost as a function of cache size since hit rate present in the metric equation depends on cache size. We have to find cache size for each VM such that the IO cost is minimized and sum of cache sizes less than total available cache. This is calculated by probabilistic search using simulated annealing tool [3] [6]. Once we get the cache size allocate it to each VM.

Algorithm 1 Cache partitioning algorithm for maximizing IO performance

1: **for** every epoch **do**
2: collect IO traces for t_w time
3: construct miss rate curve for each VM using IO traces
4: calculate cost metric C_i, using one of the above methods
5: use probabilistic search to find Si such that :
6: $\Sigma C_i(S_i) \times P_i$ = minimized or maximized
7: ΣS_i = Total cache size
8: allocate the cache size as S_i to each VM_i
9: **end for**

Algorithm 2 presents a high level idea on how hypervisor does cache partitioning in virtual environment using cache miss equation. At the start of every epoch, there is a calibration period t_w during which we vary cache partition size of each VM and measure the cache miss rate [11]. The collected data is used for regression to find parameter values (P^*, M^*, M_b, P_c) of cache miss equation

for each VM_i. We can get per VM cache size S_i directly by using below Tran el al. equation 3.1 for fair allocation.

$$S_i = \beta_i(M + \sum_{r=1}^{k} M_{br}) - M_{bi}$$
$$where : \beta_i = \frac{(\frac{P_{ci}}{P_i^*} + 1)(M_i^* + M_b i)}{\sum_{r=1}^{k}(\frac{P_{cr}}{P_r^*} + 1)(M_r^* + M_b r)} \tag{3.1}$$

Parameters are same as that of cache miss equation:
M : Total cache size
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
S_i^1 : $level_1$ cache size of VM_i
S_i^2 : $level_2$ cache size of VM_i

Algorithm 2 Cache partitioning algorithm for fairness

1: **for** every epoch **do**
2: find miss rate by changing cache size for time period t_w
3: using the above data calculate cache miss equation parameters for each VM_i using regression
4: calculate per VM cache size using Tran el al. equation of fair partitioning [11]
5: allocate the cache size as S_i for each VM_i
6: **end for**

3.2.3 Experiments

The advantage of using partitioned SSD cache on per VM basis can be seen in experimental results as shown in Figure 3.4. Here SSD cache is shared between two VMs. For VM1, both Global LRU policy (unified) cache and partitioned cache (vCS) give same hit ratio as around zero, since VM1 is running a workload of lesser temporal locality. For VM2, hit ratio for VM2 is around 0.2 using GLRU while with partitioned cache it reaches 0.7 over time span of 1200 seconds. Overall Global LRU SSD cache has 20 % less number of cache hits as compared to dynamically partitioned SSD cache.

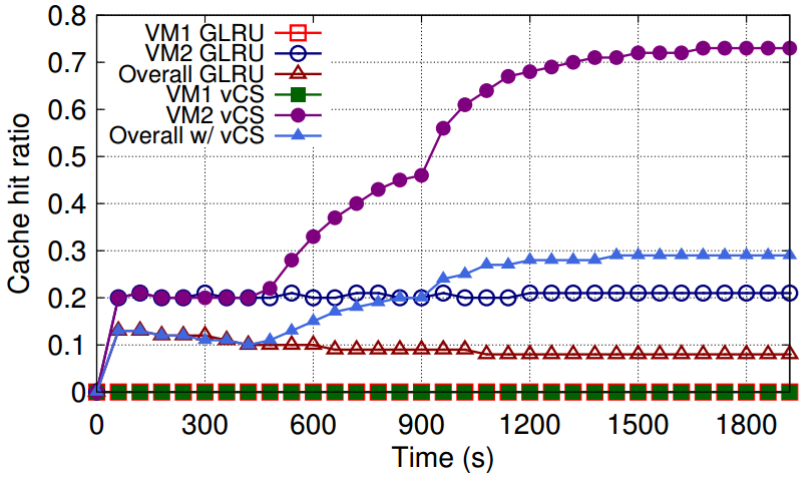


Figure 3.4: Cache partitioning effectiveness : global LRU cache vs dynamically partitioned cache [3]

Figure 3.5 shows experimental results of booting 100 VDI instance simultaneously. Flash cache is managed with dynamically partitioned cache (vCS) , static and global LRU (unified cache) allocation policies. With no cache it takes 300 seconds to boot up all machines while GLRU takes 244 seconds. The static allocation is fastest and finishes in 75 seconds while dynamic allocation takes 125 seconds. Here static allocation performs better than dynamic since all VMs have similar type of workload i.e to boot up the system, hence need static cache size. In dynamic allocation prediction of optimal cache size takes time therefore it lags behind.

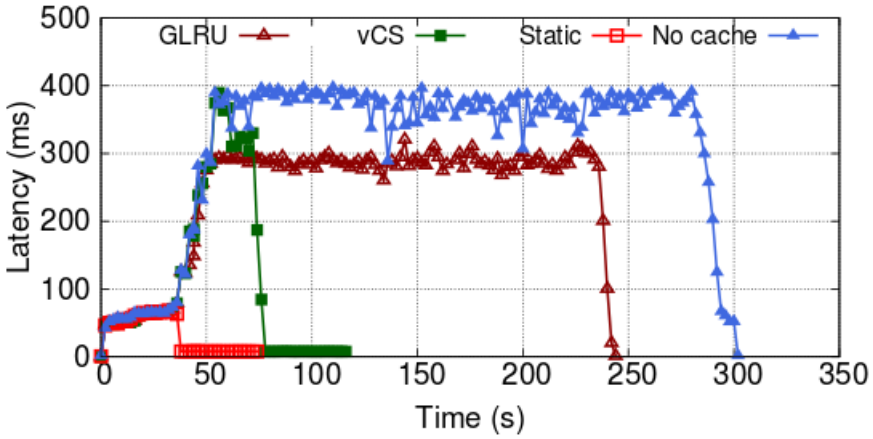


Figure 3.5: VM latency during boot storm [3]

But once the system starts and runs different types of workloads static will not perform as good as dynamic due to variable cache needs of workload. As shown in Figure 3.6, after booting i.e. 125 seconds, the latency of dynamically partitioned cache is 14 % less than statically partitioned cache.

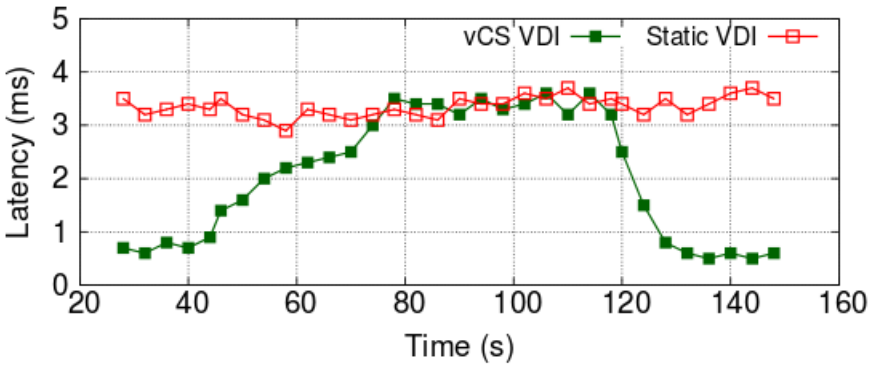


Figure 3.6: VM latency : static and dynamic cache [3]

4

Multilevel caching

In multilevel caching, two or more levels of cache are managed by hypervisor. The cache can be an in memory cache or a SSD cache which will come in between the disk and the guest page cache. The multiple levels of cache can be either partitioned or not partitioned.

4.1 Multilevel cache with no partitioning

Single level cache may not be sufficient to fulfill the cache requirements of guest VM hence multiple levels of cache can be introduced between guest VM and the disk, to increase the cache hits. But we must ensure that time required to insert, evict or search the block in all levels of cache must be less than disk latency.

4.1.1 Algorithm

The algorithm below tells how an evicted page from page cache of guest VM is inserted into hypervisor managed cache. When guest OS wants to insert a page in its page cache and it can accommodate new page then it directly adds the page. If page cache is full, then guest OS will evict a page according to some cache eviction policy (policy depends on guest OS) and new page is added. The page evicted from guest, has to be inserted into hypervisor managed cache. Hypervisor will check if level 1 cache has free space for a page, if yes then page will be inserted into $level_1$ cache. If $level_1$ is also full then hypervisor will evict a page from $level_1$ cache and insert new page. The evicted page will be similarly inserted into $level_2$ cache. The basic algorithm for managing such cache is as follows:

Algorithm 3 Page insertion in hypervisor managed two level cache

```
1: insert page X into page cache of guest VM_i
2: if VM_i page cache is full then
3:     guest VM uses its own cache replacement algorithm and evicts a page Y and inserts X
4:     if level_1 cache is full then
5:         hypervisor uses its own cache replacement algorithm and evicts page Z from level_1 and
   inserts Y
6:         if level_2 cache is full then
7:             hypervisor uses its own cache replacement policy and evicts page P from level_2 and
   inserts Z
8:         else
9:             insert Z into level_2 cache
10:        end if
11:    else
12:        insert page Y into level_1 cache
13:    end if
14: else
15:    insert X into VM_i page cache
16: end if
```

4.1.2 IO request flow

Figure 4.1, depicts the application IO request flow with two levels of hypervisor managed cache. The levels of cache managed by hypervisor are in memory cache and SSD storage. The application inside guest VM initiates IO request. Guest OS first checks if page is present in page cache. If page is present then request is directly serviced. If page is not present in page cache then request goes to hypervisor. Hypervisor checks if page is present in in-memory cache. If page is present in in-memory cache then page is copied to page cache of guest VM and given to application. If page is not present in in-memory cache then goes to SSD cache. Hypervisor checks if page is present in SSD cache. If present then page is removed from SSD and copied to guest VM's page cache. If it is not present, then data will be fetched from disk copied to guest VM's page cache and to application.

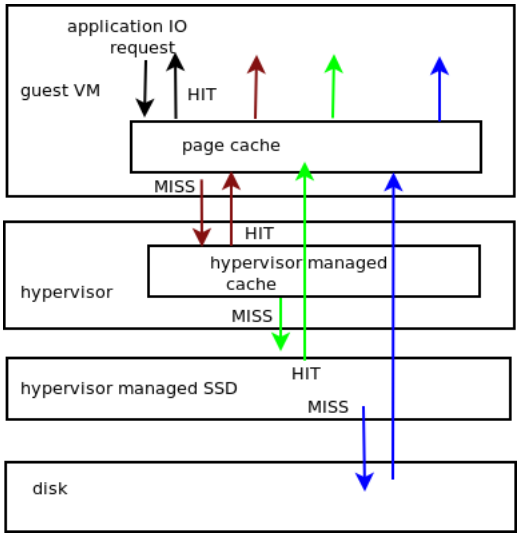


Figure 4.1: IO request flow in multiple levels of hypervisor

4.1.3 Experiments

Figure 4.2 shows the results of experiment in which different types of workloads such as Terasort, Map Reduce, TestDFSIO, Video server, web server, File server, Web proxy and OLTP were run using various cache configurations like default Tmem (simgle level cache), multilevel cache with LRU, Clock-pro and MQ as cache replacement policies. ExTmem which uses multilevel caching has more hit rate than single level Tmem and multilevel cache with other cache management techniques, for all workloads. It has 25-50 % more hit rate than default Tmem and 4-12 % more hit rate than Clock-pro and MQ. This proves that multilevel caching is better than single level caching and also the cache eviction policy can improve the performance.

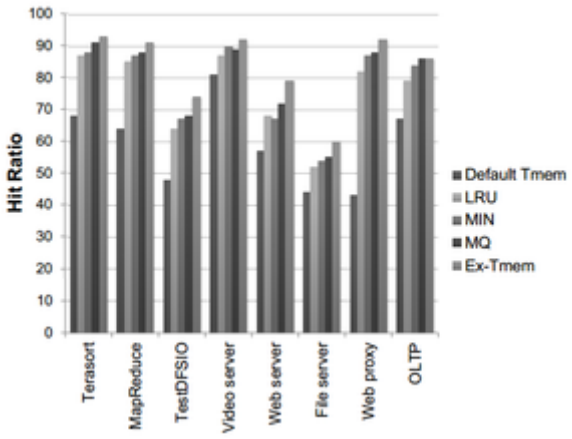


Figure 4.2: Hit ratio of various workloads

4.2 Multilevel cache with dynamic partitioning

We can partition each level of cache similar to what we have seen in previous section of single level cache partitioning. The interesting part is to find how change in cache partition size at one level affects the cache size of subsequent levels. Figure 4.3 shows multiple levels of cache - $level_1$ cache and

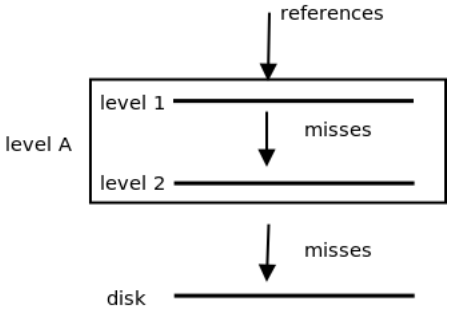
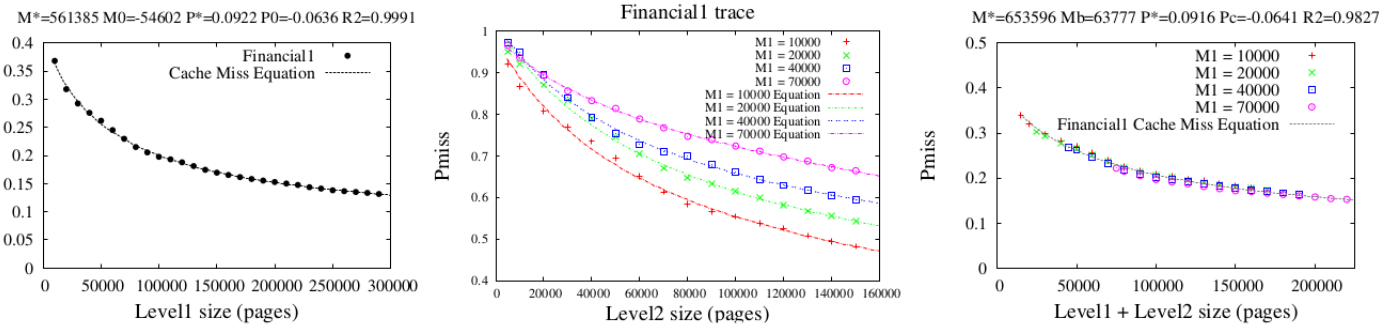


Figure 4.3: Multiple levels of cache

$level_2$ cache, managed by hypervisor. $level_1$ and $level_2$ may have different cache replacement policies. All the references from guest VM, first search $level_1$ cache, if not found checks in $level_2$. So the misses of $level_1$ become the references for $level_2$. If we change the cache partition size at $level_1$ then misses at $level_1$ will change and hence reference pattern for $level_2$ changes. Each level will therefore have its own miss rate curve which will be dependent on size of above cache levels. We can view $level_1$ and $level_2$ as an aggregate cache called $level_a$. $level_a$ gets same references as $level_1$ and suffers same misses as $level_2$. Hence $level_a$ will have its own miss rate curve, which will be independent of $level_1$ and $level_2$ cache partition size. In Figure 4.4, we can see miss rate curves of $level_1$, $level_2$ and $level_a$. As cache partition size of $level_1$ (M1) changes, the miss rate curve of $level_2$ changes. Hence for every value of M1, there is different miss rate curve at $level_2$. For $level_a$ the miss rate curve is independent of partition size of M1 and M2, rather it depends on total allocated cache on both the levels.



(a) $Level_1$ (b) $Level_2$ for different M_1 values. (c) $Level_a$: $Level_1$ plus $Level_2$.

Figure 4.4: Multiple levels of cache

4.2.1 Algorithm

The algorithm used to find per VM cache partition size is shown below. At the start of every epoch, there is a calibration period t_w during which we vary cache size of both $level_1$and $level_2$ for each VM and measure the cache miss rate [11]. The collected data is used for regression to find parameter values (P^*, M^*, M_b, P_c) of cache miss equation for each level and each VM_i. We can get per VM cache size S_i^1 for $level_1$ directly by using Tran el al. equation 4.1 for fair allocation :

$$\begin{aligned} S_i^1 &= \beta_i(M^1 + \sum_{r=1}^{k} M_{br}^1) - M_{bi}^1 \\ where : \beta_i &= \frac{(\frac{P_{ci}^1}{P_i^{1*}} + 1)(M_i^{1*} + M_{bi}^1)}{\sum_{r=1}^{k}(\frac{P_{cr}^1}{P_r^{1*}} + 1)(M_r^{1*} + M_{br}^1)} \end{aligned} \tag{4.1}$$

Parameters are same as that of cache miss equation:
S_i^1 : $level_1$ cache size of VM_i
S_i^2 : $level_2$ cache size of VM_i
M^1 : Total cache size at $level_1$
M^2 : Total cache size at $level_2$
P_i^{1*} : Minimum miss rate possible of $level_1$ for VM_i
M_i^{1*} : Cache size of $level_1$ for VM_i beyond which miss rate won't decrease
M_{bi}^1 : Memory needed by cache manager for storing metadata information of $level_1$ for VM_i
P_{ci}^1 : Miss rate curve's convexity of $level_1$ for VM_i

Now we must determine $level_2$ cache size for each VM. If some service level objective (SLO) for a VM is present then according to that, we calculate $level_2$ cache size. Let us say SLO is stated as :

$$Prob(total\ latency > L_{max}) < p_{max} \tag{4.2}$$

that is probability of total latency becoming greater than L_{max} must be less than p_{max}. L_{max} and p_{max} will be provided by SLO. From this we can find tolerable miss rate at $level_a$ for achieving given latency using 4.3 equation.

$$\begin{aligned} Prob(T > L_{max}) = Prob(x \in level_1)Prob(L_1 > L_{max}) + \\ Prob(x \notin level_1\ and\ x \in level_2)Prob(L_2 > L_{max}) + \\ Prob(x \notin level_1\ and\ x \notin level_2)Prob(L_3 > L_{max}) \\ Prob(T > L_{max}) = P_a^{miss}Prob(L_3 > L_{max}) \\ Miss\ rate\ at\ level_a = \frac{0.95 p_{max}}{Prob(disk\ latency > L_{max})} \end{aligned} \tag{4.3}$$

Here cache latencies are ignored since disk latency is much higher, therefore total latency is approximately equal to disk latency. Once we get miss rate at $level_a$, we can use cache miss equation of $level_a$ to find cache size of $level_a$ (S_i^a) for VM_i. If no SLO is present for any VM then directly divide $level_a$

equally across all VMs and get S_i^a. From the following equation 4.4 we can get $level_2$ cache partition size.

$$S_i^2 = S_i^a - S_i^1 \tag{4.4}$$

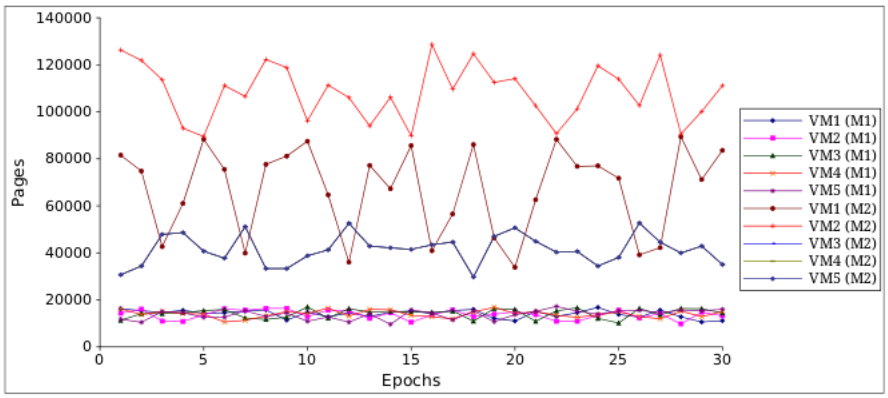
Once we get cache partition size for both the levels, allocate it to each VM. This algorithm can be applied to n levels of cache. [11]

Algorithm 4 Cache partitioning algorithm for two levels

1: **for** every epoch **do**
2: find miss ratio by keeping every possible cache size at L1 and L2 for time period t_w
3: use regression to find cache miss equation parameters for each level and for each VM
4: get the cache size as S_i^1 for each VM_i using Tran el al equation of fair partitioning [11]
5: find S_i^a according to SLO
6: calculate $S_i^2 = S_i^a$ - S_i^1
7: allocate the cache size as S_i^1 and S_i^2 for each VM_i
8: **end for**

4.2.2 Experiments

Figure 4.5 shows experimental results of running 5 VMs together using two levels of hypervisor manged cache. It shows how the memory allocation of 5 VMs changes as workload pattern changes over the time. There were no experiments done to compare how dynamic partition at both levels is better than other cache variations.



(a) Dynamic allocation of M_1 and M_2 to 5 VMs.

Figure 4.5: Multiple levels of cache

5

Additional problem statements

Here are few problem statements related to this area.

Coordinated SSD caching

The caching techniques discussed above were focused on improving performance using host side caching. We were interested in caching within single host with the static allocated cache. But in data centers there are many hosts present connected to the server. The problem which we did not look into is the capacity of cache present. The dynamic nature of workloads may lead to cache overload situations in which a guest VM will not get the desired cache capacity. The limited cache capacity is critical for flash device endurance too. This problem can be solved in two ways : one is to add more cache at host side, other is to migrate the VM along with the cache to other host machine. Adding more cache at host side is not be always possible due to cost constraints, hardware restrictions. So migrating the VM along with cache to a host with lesser cache overloading can be possible solution. This may lead to some interesting questions like which VM should be migrated on which host machine, how many cache blocks should be migrated, how to coordinate migration across different host machines? [1].

Finding best cache position in virtualized environment

Nowadays flash cache has become very popular in data centers. The problem is where to employ flash device within virtualized system for its effective performance [2]. In a virtualized system there are multiple options for SSD cache management:

1. VM-based SSD caching : Here guest VM will have control over the SSD cache.
2. Hypervisor-based SSD caching : Here SSD cache is managed by hypervisor.
3. Storage system-level SSD caching : SSD cache is managed by the storage system.

It is essential to know advantages and disadvantages of using each type of SSD caching .

6

Observations

6.1 Experimental setup

Most of the experiments were done on VMware ESX server and Xen. The host side cache added, was either SSD or NVDIMM. Since SSD is cheaper it was used more as compared to NVDIMM. SSDs used were Intel X25-M, Intel PCI-E, Intel 520 Series. The tests used VMs running windows or Ubantu Linux. Configurations of host machines used were:

Table 6.1: Configurations used in experiments

	Processor	RAM	Hard disk	Added cache
References				
[11]	Intel Xenon 2.5GHz processor 12 cores	4GB	1TB	emulated by DRAM
[10]	Intel 2.5GHz processor	8GB	1TB	16GB NVDIMM
[4]	Intel Xenon 2.67GHz processor	16GB	500GB	4 × 180GB Intel SSD 520 Series
[3]	AMD Opteron 6128 processor	16GB	1GB per VM 8GB virtual disk per VM	Intel 400GB PCI SSD 910
[6]	AMD Opteron 270 dual core processor 2 GHz	4GB	1GB per VM 40 GB virtual disk per VM	120GB Intel X25-M SSD

6.2 Questions answered by experiments

1. What is the advantage of partitioning cache?
   Metric : Cache hit rate, cache allocation, IO latency, **Parameter** : cache partitioning algo-

rithms, cache eviction policies

2. How is cache size adaptive to workload? This ensures that if workload's working set size is changing then the cache size allocated must also change.
   Metric : Cache size, **Parameter** : workload

3. How fair is cache allocation done across VMs? This ensures that all VMs have a fair share of cache
   Metric : Normalized miss ratio, i.e current miss ratio divided by minimum miss ratio possible for that workload running on VM.

4. Is the service level objective (SLOs) satisfied? The SLOs provided by customers must be satisfied since they pay more money for better response time or any other objective. Hence it is essential to check through experiments that SLO is satisfied.
   Metric : latency, responsiveness, throughput (IOPS) **Parameter** : increase load on VM (can be number of requests issued simultaneously)

6.3 Workloads used

Workloads are applications which are run on the system to test the performance. The common workloads used are:

1. The performance of the system is measured by booting several VMs simultaneously and measuring their boot time using bootchart.

2. The microbenchmarks used Flexible IO tester (fio) and Iometer for load generation [6].Flexible IO tester is used to write a job file which simulates configured IO. Iometer is an I/O subsystem measurement and characterization tool for single and clustered systems.

3. Filebench is a file system and storage benchmark that allows to generate a large variety of workloads was used for macrobenchmarking

4. Replay production storage traces were from SNIA IOTTA repository, UMASS trace repository. These are block IO traces.

5. Benchmarks in which the workload pattern running on VM can be changed dynamically : DaCapo Tradebeans Benchmark, TPC-H, TPC-VMS . DaCapo Tradebeans benchmark is java benchmark for memory management and computer architecture. TPC-H is a decision support benchmark that examines large amount of data and executes queries with high degrees of complexity. TPC-VMS measures database performance in virtualized environment.

7

Future work

- Although many papers have proposed their own way on how to manage cache so as to get good IO performance but there is no comparative analysis of all the types of these architectures. We can compare how each proposed solution and the cost metric which they have used is better than other. The comparison can be done on the basis of the common objectives like minimizing overall disk IO latency or fair partitioning or satisfying service level objective. Also we can conduct experiments to find out which caching techniques suite which types of workloads.
  As we have seen, different hardware configurations were used in all the experiments. We can study how the hardware configurations change the performance of the system like which performs better SSD or NVDIMM as a second layer cache, what is the trade off between performance and cost?

- Another aspect not covered in any of the papers is experiments to evaluate how cache hit rate varies for per-partition local cache replacement policy vs common cache replacement policy.
  Metric : Cache hit rate (per partition+overall), Latency, **Parameters** : Vary cache eviction policies (for common eviction policy).
  For this we need to come up with an algorithm which will observe the data accesses and suggest which eviction policy suits the best for a workload. As the we know workloads are dynamic in nature, their accesses will change. Hence the evict policy must also be dynamic in nature. This raise questions like does dynamic nature of eviction policy benefit the performance, how often should the eviction policy be changed?

- There are two ways to construct miss rate curve. One is using SHARDS (a variant of mattson's stack algorithm) and other is using cache miss equation. Most of the papers used mattson's stack algorithm's variants to construct miss rate curve. We can compare both the ways of constructing miss rate curve in terms of space complexity, time complexity, accuracy and limitations.

8

Conclusion

We have seen that caching can be done in many ways within virtualized environment but selecting the one which improves the workload's IO performance is important. We may can think of managing cache in multiple dimensions like multiple cache or single level cache, partitioned or unified cache, which cache eviction policy, whether to use write through or write back. In general, dynamic cache partitioning gives better results than static partitioned or unified cache. All techniques for dynamic cache partitioning have been compared to static partitioned cache or unified cache but they are not compared with each other. Conducting an experiment which could compare all these techniques could help in understanding their pros and cons with each scenario (workload).

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] Steve Byan, James Lentini, Anshul Madan, Luis Pabon, Michael Condict, Jeff Kimmel, Steve Kleiman, Christopher Small, and Mark Storer. Mercury: Host-side flash caching for the data center. *Proceedings of 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, 2012.

[3] Meng Fei, Zhou Li, Ma Xiaosong, Uttamchandani Sandeep, and Liu Deng. 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.

[4] 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.

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

[6] 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.

[7] Ricardo Santana, Steven Lyons, Ricardo Koller, Raju Rangaswami, and Jason Liu. To ARC or Not to ARC. *Proceedings of the USENIX HotStorage Workshop*, 2015.

[8] 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.

[9] Oracle Technology. Transcendent memory. `https://oss.oracle.com/projects/tmem/`.

[10] 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.

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

[12] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server . *Proceedings of the 5th symposium on Operating systems design and implementation*, pages 181–194, 2002.

[13] 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.

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