PRESTO: PREwarming STOrage Caches for Improving I/O Performance in Virtualized Infrastructure

M.Tech. Project Stage II Report

submitted in partial fulfillment of the
requirements for the degree of
Master of Technology

by

Sukrit Bhatnagar
(173059003)

under the guidance of

Prof. Purushottam Kulkarni

Department of Computer Science and Engineering
Indian Institute of Technology Bombay

Contents

List of Figures

List of Tables

Abstract

Virtualized environments nowadays employ a hypervisor cache at each node to improve performance on the storage I/O path as well as alleviate some load on the underlying storage. This further benefits the environments having networked storage where the VM disk data is not necessarily available locally.

That being said, the cache itself is local to the node as it caters to the requests coming from the VMs on that particular node. From a performance point-of-view, the cached data in this hypervisor cache is as important as the backing data. A cold hypervisor cache would not result in drastic reduction in performance as compared to performance with no hypervisor cache present. But, there will be no improvement in the overall I/O performance, and the cache will fail to fulfil its purpose.

This study aims at identifying the scenarios which will render the hypervisor cache cold and coming up with methods which can aid in effectively "warming up" the otherwise cold cache. We consider the Nutanix HCI as the base model for our experiments.

1. Introduction

In virtualized environments, where multiple VMs are running on a physical server employing a hypervisor, storage I/O often becomes the performance bottleneck. This is partly due to the nature of the underlying storage devices, which have access latency of a few milliseconds (as compared to the DRAM with latency of a few nanoseconds). A cluster of such physical servers usually have arrays of storage devices accessible through network interfaces. Two major storage setups used in such environments are: *centralized storage*, wherein a common pool of storage devices is shared among the nodes in the cluster using a network filesystem such as NFS and iSCSI, and *distributed storage*, wherein each node in the cluster consists of local storage which runs a distributed filesystem such as GlusterFS and HDFS. We will focus on virtualized environments with distributed storage for this study.

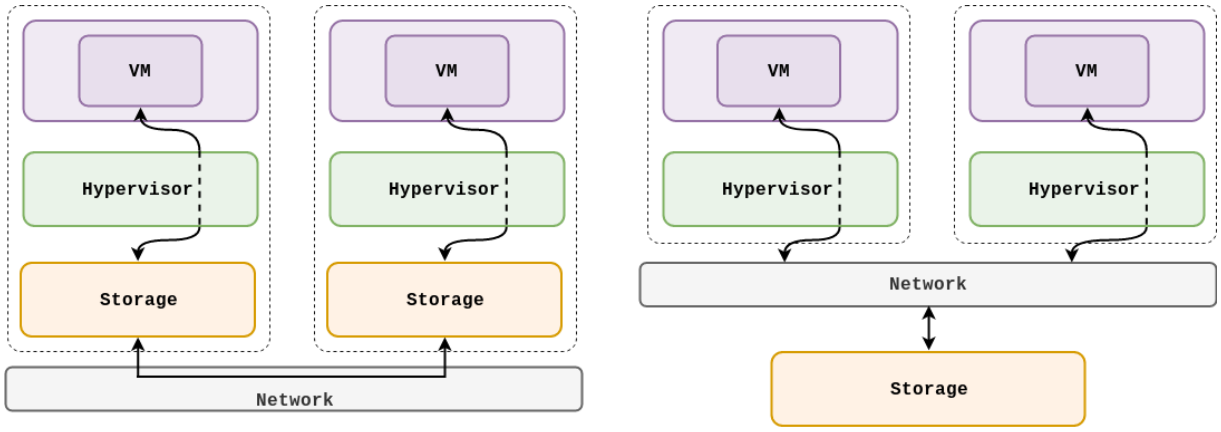

Figure 1.1: Common storage setups (distributed and centralized) in virtualized environments

Many such environments make use of a combination of HDDs and SSDs to divide the data logically into tiers, with SSDs keeping the hot data. While SSDs do provide an improvement in access latency (reduced to the order of a few microseconds), they are expensive and are limited in size as compared to HDDs. Moreover, in the networked storage setups mentioned above, the network latency can still dominate the total storage I/O latency. As a result, having such storage setup adds network access latency to the existing latency incurred by the storage devices.

When a VM performs a disk I/O operation, the hypervisor (node) intercepts the corresponding request and delegates it to the underlying storage mechanism. This process is usually called storage virtualization. Due to the distributed nature of storage, there is a chance that the data requested is not actually present in the local storage. The situation is exacerbated by the

fact that the distributed fileystems use internal metadata structures to keep track of the actual data, and we need to access the metadata first in order to get to the data. Presence of metadata information on the local storage is as important as the presence of requested data.

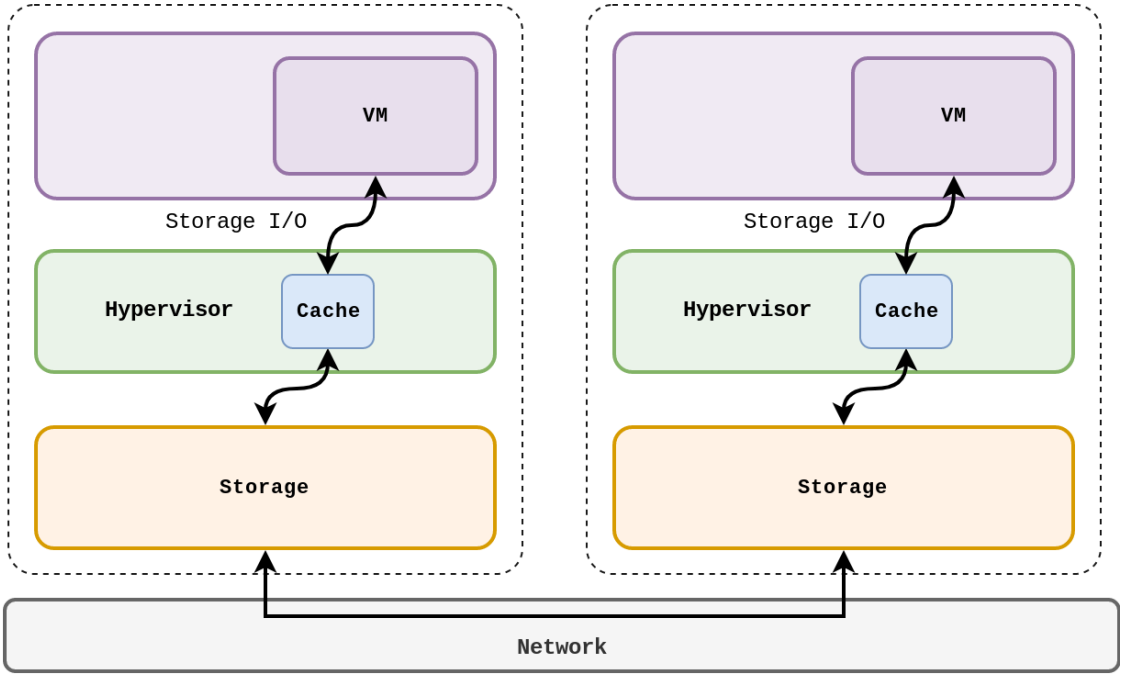

Figure 1.2: Virtualized environments with hypervisor cache

If a certain block of data or metadata required by a VM is missing in the node's local storage, it is fetched from another node's local storage over the network. An I/O operation, which would incur only one disk access if only local storage was in play, now incurs an additional network access and disk access due to the networked storage.

1.1 Hypervisor Caching

Whenever there is a need to fetch certain data from someplace, and this fetching is a costly operation, we think of caches. Hypervisor, being the manager of all system resources, is the natural choice for most virtualization-oriented I/O performance improvements and optimizations. A great deal of research is done in the past for making use of a cache inside the hypervisor layer of a node. This cache is intended to work as a client-side cache for keeping storage I/O data and metadata fetched over the network.

We define a few characteristics of our view of the cache below.

- **Hypervisor-managed**
  The hypervisor intercepts storage I/O requests from the VMs, and performs lookup in the cache. If the required data is found, it is returned. If the lookup results in a cache miss, it invokes certain primitives to fetch the data from underlying storage system. The VMs have no control over the operation of this cache, and it remains transparent to them.

- **Unified**
  Unlike a page cache employed by modern kernels which is used for caching only the data read from the disk, the cache also stores the metadata managed by the underlying distributed filesystem. We refer to individual entities in the cache (disk blocks as well as filesystem metadata) as *objects*. The cache stores heterogeneous objects, and objects of the same type may vary in size.

- **Inclusive**
  The cache is used to store disk blocks and metadata blocks read from the underlying storage system. Upon lookup for a certain block, the hypervisor will return its data to the requesting VM. The VM may have its own operating system buffer/page cache which stores this data in memory. Thus, the same data block may be cached inside the VM cache as well in the hypervisor cache.

- **Shared**
  All VMs running on a node will share this cache with equal priority, and there is no logical partitioning for each VM. This also means that there is contention among the VMs performing storage I/O operations when the cache is accessed.

- **Volatile**
  The cache is maintained in-memory, where its total size depends on the available main memory. If a node were to restart or fail, the cache contents are lost.

- **Local**
  The cache is local to a particular node, and is not accessible directly from other nodes. The contents of this cache will differ from node to node in the cluster. This is unlike the underlying storage which is shared among all the nodes in the cluster.

- **Static-sized**
  The cache is fixed in size and does not grow or shrink based on the changes in main

memory available. While caches with dynamic sizing are often used in practice, we assume its size to be fixed throughout this study.

Any cache used in compute systems (not just the hypervisor cache) is a driving force in the improved performance and smooth running of the infrastructure is often compared to an engine in a machine and analogies are set up between the their states, most notably them being cold and warm. A main memory cache is said to be cold when it is empty (or is non-empty and has irrelevant pages), and warm when it is filled with pages relevant to running processes. The transition from a cold cache to a warm cache results in improved performance of the system due to increased cache hits (and reduced penalties due to misses). We apply the same analogy to our hypervisor cache as well.

Cold cache

Cold cache refers to the state of hypervisor cache in which either the cache is empty or it contains stale objects i.e. those objects which were cached in previous runs and are not needed to serve the upcoming I/O requests. We can also have a partially cold cache which means that the objects required by a particular VM or a set of VMs are not cached. When the cache is first initialized by the hypervisor, it will essentially be empty and initial few lookup operations will result in cold misses.
Furthermore, if the cache is full with objects of a few already running VMs and a new VM comes up on that node, the cache will be cold for that VM. So, when any VM or a group of VMs start on a node, they will essentially experience a cold cache. The cache may also contain some objects of a VM from its previous run, and the VM gets reset. In this case, even when the cache contains its objects, those cached objects may not be needed now. Nevertheless, there will be a drop in the hit ratio (at least in the initial time period of storage I/O traffic) due to very high cache misses.

Warm cache

Warm cache on the other hand means that some objects are present in the cache which are relevant to one or more VMs running on the node. That is, there is a high chance that the objects will be referenced in the near future. Even if the cache is not full, but is able to accommodate all of these relevant objects, it is warm. A cache full with irrelevant objects in not warm. Having relevant objects in the caches help in increasing the performance as a result of an increase in the overall hit ratio.

Prewarming

We will reach a warm cache state (eventually, from an initial cold cache) as the incoming I/O requests will result in misses and the cache will be populated lazily. However, the VMs will suffer from an initial performance loss due to the cold state (and the associated miss penalties) as it would take a certain amount of time for the hit ratio to stabilize.

Prewarming refers to the process of proactively loading the cache with the relevant objects of one or more VMs running on the node. This is similar to cache prefetching, but we use the term prewarming instead to emphasize the cold and warm cache states.
It is a two-step process: first, we need to identify the set of objects to load, and then, we need to actually fetch these objects into the cache. Note that the objects we prewarm the cache with might be evicted from the cache eventually as the relevance of the objects to storage I/O traffic starts to fade. We are only interested in avoiding a performance loss due to a cold cache, not in minimizing the evictions from the cache.

1.2 Problem Description

As described above, the cache is local to a node as it caters to the requests coming from the VMs on that particular node. From a performance point-of-view, the cached data in this hypervisor cache is as important as the backing data on the distributed filesystem. A cold hypervisor cache would not result in drastic reduction in performance as compared to performance with no hypervisor cache present. But, there will be no improvement in the overall I/O performance, and the cache will fail to fulfil its purpose. Therefore, we can say that a cold cache will result in reduced performance.

We first identify some considerable scenarios where the cache is rendered cold, resulting in performance drop. Then we try to define the problem scope and an approach to the solution.

One cold-cache scenario is when a node boots up, and then all the VMs on it boot up. The hypervisor initializes the cache, and it is empty at the start. There are a lot of cache misses due to required objects not being present in the cache. As the VMs perform storage I/O operations starting with their bootup sequence, the cache starts getting filled with objects and hit ratio tends to increase due to possible locality in the data accessed by the VMs. A similar case is when the node is already up, but all its VMs boot up. The cache will have some objects of these VMs from their last run. But, those objects may be stale and they may not be needed at boot time, or in the initial storage I/O operations. The cache state is equivalent to being cold even when it is filled with the VMs' objects.

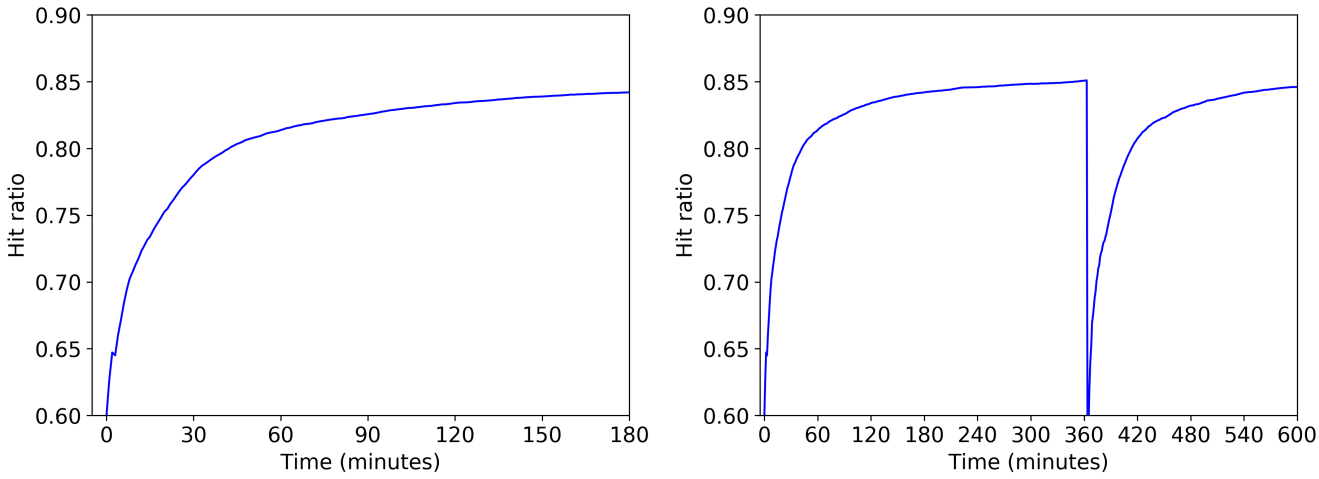

Figure 1.3: Cache hit ratio over time: when a node starts up (left) and when node failover happens (right)

Another interesting scenario is when a node experiences a failure and the cache state is lost. VMs in most virtualized setups are configure with High Availability (HA) i.e they are started on another node from the same point of execution when the failure happened. The nodes are configured to be a part of a failover cluster and guarantee a certain uptime for their guest VMs. Solutions to HA, such as lock-stepping or asynchronous replication ensure that the VM state is available at another node, but they don't account for the hypervisor cache state for the VM. When a set of VMs experience HA migration to another node, the hypervisor cache at that node is cold for them. The hot objects in cache for the VMs are not available anymore

and the cache at new node will result in misses and lookups in the underlying storage. At the time instant when VMs start executing at the new node, the overall hit ratio of the cache will take a drastic drop, and will slowly build up. It might take a considerable amount of time for the cache to reach a stable hit ratio (close to the one just before failure).

A similar case happens when a certain VM is migrated to another node (but not its hypervisor cache state), and the cache performance of the destination node is disturbed. There might be some VMs already running on the destination node and due to the addition of a new VM (for whom the cache is cold), the combined hit ratios of existing VMs, the hit ratio of migrated VM, and in general, the overall hit ratio is poor for an initial time period.

Given a storage I/O cache inside the hypervisor, how to reduce the time it takes to warm up the cache?
Simply put, how to achieve a high cache hit ratio quickly?

Ideally, we want to operate with a cache that is loaded with important objects at all times, and thus we need to find a way to keep it warm at all times. Although the cache eventually gets filled with objects actively used by the VMs as they issue storage I/O requests, we aim at keeping the cache warm even in the initial time period of the storage I/O traffic.

There are multiple questions that need to be answered as part of answering the question above.

- What are the performance implications of having a cold cache? How much degradation in performance can be expected when the VM workloads experience a series of cold cache misses?
- Will proactively prewarming the cache result in improved storage I/O performance? In other words, will the prewarming help mitigate the performance degradation due to cache being cold?
- How to determine the set of objects to be loaded into the cache so that it is effectively warm? What should be the size of this prewarm set and which objects should be a part of it?
- What is the impact of prewarming on the performance of other VMs which are actively using the cache? Is prewarming the cache for a VM worth a possible drop in hit ratio of other VMs?
- How to quantify the various overheads arising from the prewarming process and with these overheads being present, is it worthwhile to prewarm the cache?
- Can changing the behaviour and properties of cache itself help us in determining a better prewarm set (and better performance in general)? The scope includes, but is not limited to, replacement policies, logical partitioning policies and memory limits.
- For the VM workloads having specific disk block access patterns, can we recognize and leverage these patterns to make informed decisions in constructing the prewarm set for those VMs?

1.3 Solution Approach

A VM running on a hypervisor node is a black-box, i.e. we have no way of peeking inside its memory to get storage I/O-related information, such as the contents of the page cache. Moreover, we also have no information about how its disk blocks are laid out on the vdisk, and which disk blocks are the most important to consider for caching. Some examples of important disk blocks include filesystem super blocks, bitmap blocks and inode blocks. Therefore, we are restricted to using only non-intrusive approaches for mitigating the issues arising from cold hypervisor caches.

The only part visible to the hypervisor is the block I/O access request that a VM issues to its vDisk. These requests are processed by the cache and a lookup is performed. After the required data is fetched into the cache, it is served from there. We can inspect the cache to see what data is cached for a vDisk, since the cache is part of hypervisor. To find out important blocks for a vDisk, we can continuously monitor the cache and keep track of the active objects. But, as the cache is volatile, its state can be lost due to a failure and there is no way to recover the same. Losing the warm cache state implies a heavy penalty in terms of increased storage I/O completion time (experienced by the VMs) as well as increased contention for network bandwidth (due to the storage being distributed). We use a series of steps to try to mitigate these issues.

- The first step is to ensure that the cache state is persisted to a storage medium so that it can be recovered even if a failure happens. We can save the contents of cache to the disk periodically to ensure persistence. We will discuss the format for representing a persistent cache state and techniques for saving and loading the state in the next chapter.
- The second step is to enable the cache to use these saved states and determine objects that are important to a vDisk, so that we may reuse them and proactively fetch them into the cache when it goes cold (i.e. prewarm it). There are various heuristics that we can use to establish the importance of an object to a vdisk, and we will discuss some of them in the next chapter.
- The third step is to use the important objects determined in the previous step and load them into cache in a manner that gives a reasonable benefit in performance, while minimizing the interference caused due to the prewarming process on other objects in the cache. We discuss the throttling of the prewarming process in the next chapter.
- Finally, we will perform some experiments specific to a scenario where the cache goes cold and we try to prewarm it to see if the drop in hit ratio (if any) due to cache going cold can be avoided.

2. Background & Related Work

Hyperconverged Infrastructure

An infrastructure consists of various resources such as compute, storage and network with associated operations such as resource management, orchestration and automation. Converged infrastructure refers to coupling all the resources of an infrastructure into a single unit, creating a common pool of virtualized resources. Hyperconverged infrastructure, an extension to the converged one, involves performing data center operations (related to storage and networking) in the software layer, making it hardware-agnostic.

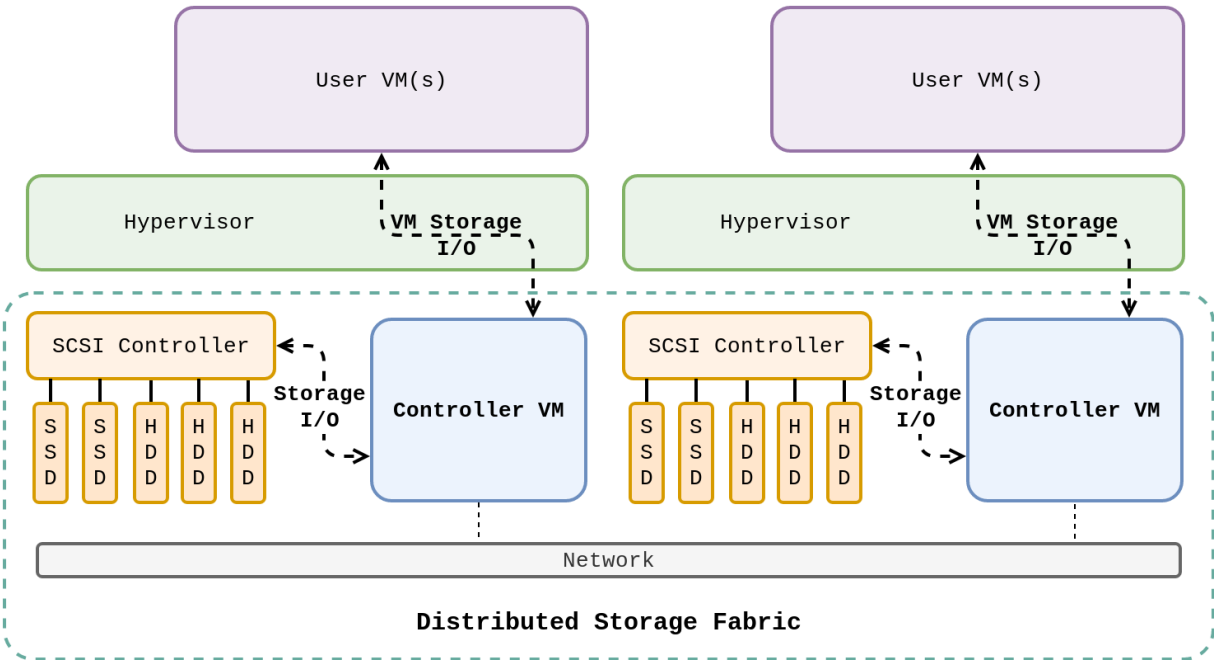

Figure 2.1: Overview of the Nutanix HCI

We use the Nutanix HCI as our base model of an infrastructure where we aim to improve the storage I/O performance. Figure 2.1 shows a simplfied view of the Nutanix HCI. It consists of clustered nodes connected through a backbone network. Each node consists of local storage, a number of client VMs, and a special VM called Controller VM (CVM). Furthermore, each node has a combination of HDDs and SSDs as part of its local storage for policy-based tiering of data. Each VM has one or more virtual disks (or vdisks) attached to it, a resource provided by the hypervisor via storage virtualization. The virtual disk data of these VMs are stored in

the node's local storage, and is optionally replicated to other nodes.

Distributed Storage Fabric

A distributed filesystem, formerly known as Nutanix Distributed Filesystem (NDFS), is used to access the underlying storage. NDFS is now integrated into Distributed Storage Fabric (DSF), which provides features such as backup, compression, deduplication and disaster recovery. DSF appears to a node as a centralized storage array, but all VM I/O operations are performed using the local storage. Due to the distributed nature of storage, DSF maintains some metadata about where the data is actually stored in the cluster. Apache Cassandra is used to store this metadata as key-value pair in a distributed fashion, where each node also acts as a node in the Cassandra ring. Thus, the actual virtual disk data, as well the associated metadata are distributed across the nodes in the cluster.

Controller VM

CVM is responsible for serving all I/O operations performed by the VMs running on that node. The local storage of a node is directly attached to the CVM using PCI passthrough mechanism. This makes CVM a privileged VM, having complete control over all storage resources and it provides storage interface (via NFS, SMB, iSCSI etc.) to all other VMs. It also realizes software-defined storage by providing features such as RAID, compression and deduplication.

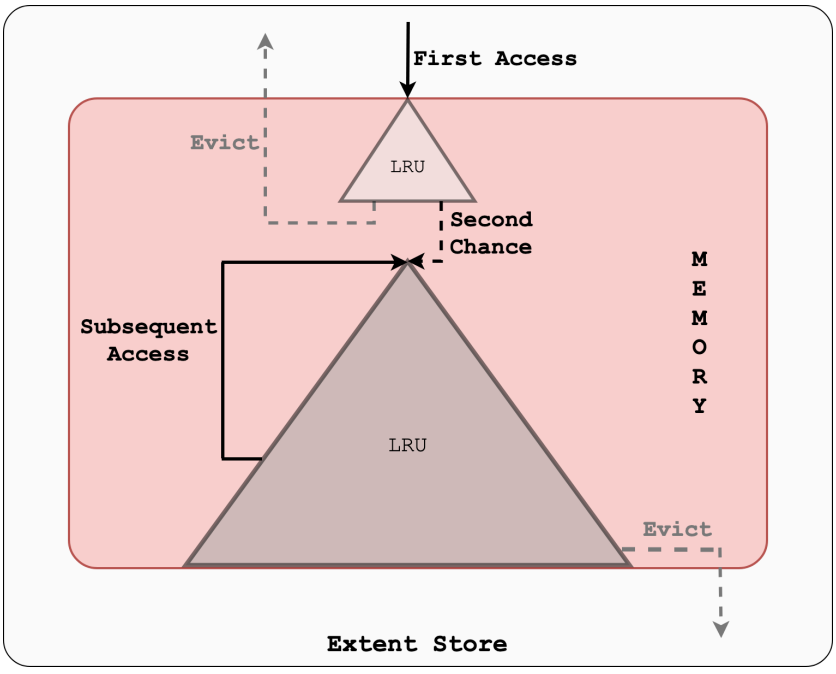

Figure 2.2: Overview of the CVM cache

The CVM on each node consists of a local cache for storing blocks of the virtual disks it is hosting. Apart from the caching the vdisk data, this cache is also utilized for storing the metadata we mentioned earlier, making the cache unified in nature. We refer to this local

cache of the CVM as the hypervisor cache.

The CVM cache consists is completely in-memory and has two pools (single-touch and multi-touch) as shown in Figure 2.2. The two triangles represent the pools, with smaller one being the single-touch pool and the larger one being the multi-touch pool. Single-touch pool is intended for storing objects that are fetched into the cache for one-time access. From this pool, an object can either get evicted according to the replacement policy, or can get promoted to the multi-touch pool on subsequent access. Multi-touch pool keeps the objects that are actively in use by the VMs. An object evicted from this pool gets evicted from the cache. An object is promoted, on subsequent access, to the top of the memory pool. As indicated in the figure, LRU policy is used for replacement in both pools.

Metadata

Nutanix DSF makes use of Apache Cassandra ring to store the distributed filesystem metadata as key-value pairs. They have layers of metadata translation such that the value returned on a key lookup at first layer is used as a key for the next layer. To get to the actual data, we have to go through a series of lookups though the metadata layers. While the actual schema of the key-value structures are complex, we use a rather simplified version. Each layer of the metadata can be visualized as a hashmap, having different data structures for the key and value parts.

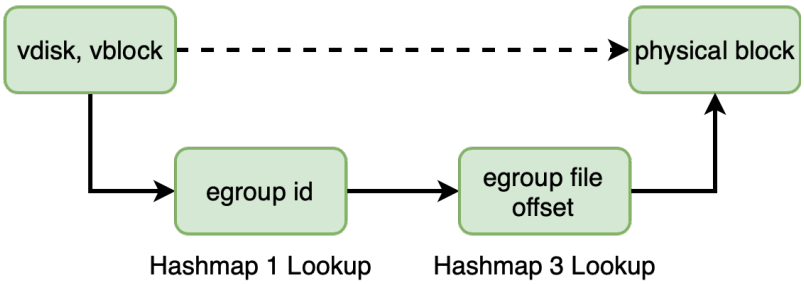

Figure 2.3: Metadata translation in Nutanix HCI

We define a few storage terms used in the Nutanix HCI:

- **vDisk** or vdisk is the storage medium from a VM's perspective. Hypervisor provides the VM with one or more vdisks, each of which has a unique identifier within the cluster.
- **Extent** or **vBlock** is the building block of a vdisk. It is 1 MB in size and a vdisk is stored as a set of extents scattered across the underlying storage. A vblock number addresses 1 MB chunk of the vdisk, and that number space is local to a particular vdisk.
- **Extent Group** or **egroup** consists of 4 extents and is 4 MB in size. Each egroup has a unique identifier within the cluster, and is stored as a file on the underlying storage media.

The first hashmap takes the cryptographic hash of a vdisk ID and a vblock number as the key and stores an egroup ID as the value. The second hashmap is similar to the first one, but is

used only if the requesting vdisk is a snapshot of another vdisk. The third hashmap takes an egroup ID as the key and produces an offset into the egroup file where the requested data is stored. The scope of this project involves prewarming the cache only with these metadata objects and not the actual data blocks. To further simplify our study, we currently use only the first and third metadata hashmaps in our metadata translation layer. **We will use the terms HM1 and HM3 to denote the first and third Hashmaps, respectively.**

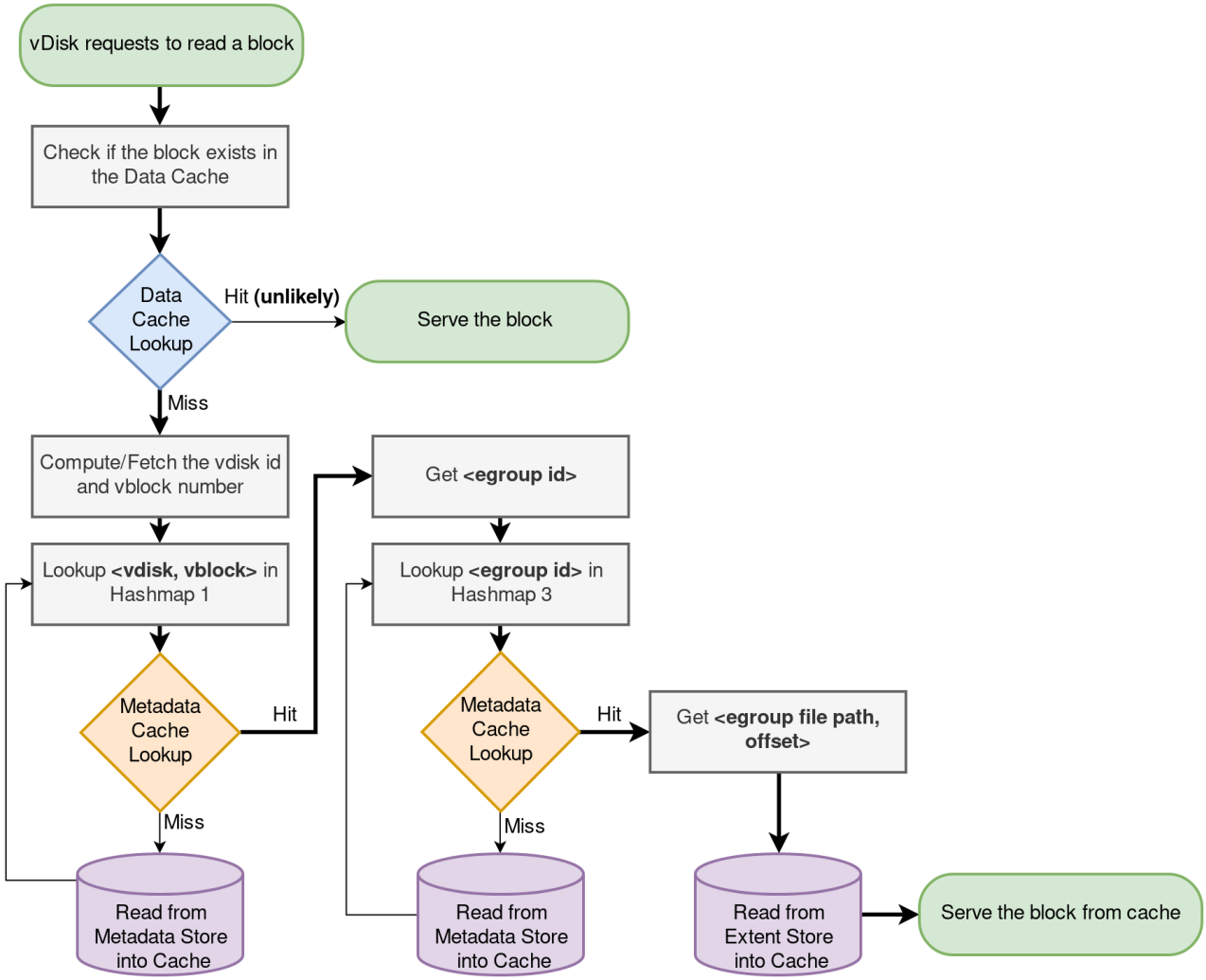

Figure 2.4: Cache Lookup Flow

Figure 2.4 shows how a block I/O request coming from a vdisk is served by the cache. The requested block is first looked up in the data portion of the unified cache, and is served directly if found. We assume that the data is most likely not present in this portion, and that the metadata cache lookups will happen. The path in bold represents the best case scenario where the required metadata objects are found in the cache.

3. Design & Implementation

Due to proprietary nature of Nutanix DSF software, we are currently not able to modify their source code to add the prewarmimg functionality. We instead aim at mimicking the relevant parts of their infrastructure in our own simulator and use it to find solutions to the problems mentioned in the previous section. The scope of our work involves the following:

- Create a simulator which offers a reasonable model of the components of the Nutanix DSF infrastructure relevant to this study
- Generate VM workloads which can be served by the hypervisor cache in the format an actual hypervisor gets storage I/O requests from its VMs
- Define a persistent cache state, as well as various primitives in the cache which will enable the prewarming, such as saving and loading the persistent state
- Come up with a set of relevant parameters and heuristics which can help us determine a good prewarm set and explore the parameter space
- Run a set of experiments which can provide us with an empirical analysis of our prewarming strategies and help us determine the effectiveness as well as feasibility of prewarming

3.1 Cache Design

3.1.1 Pools

We model our cache as having 2 pools in main memory. They are called single-touch and multi-touch pool. Intuitively, the size of these pools increases as we go from single to multi-touch. This is the same configuration as the Nutanix DSF Unified Cache. In our design of the cache, there is a 20-80 split of the total cache between the single and multi-touch pools, respectively. In the original Unified Cache this split between pools is dynamic, but we fix it statically before the simulation starts.

Single-touch pool stores the new objects that are fetched into the cache. These objects will be eventually evicted from the cache by the replacement policy unless they are accessed again. Upon a second access to an object in the single-touch pool, they are promoted to the multi-touch pool. An object will remain in the multi-touch pool until it is evicted by the replacement policy. Note that subsequent accesses to an object in the memory multi-touch pool will not change its pool, but will only result in an update in its state for replacement policy. This is the same as shown in Figure 2.2.

Note that while the cache is divided into multiple pools, there is no partitioning within the cache for the various types of objects. We do not specify the share of cache either for HM1 and HM3 objects, or for the HM1 objects of various vDisks.

For the purpose of this study, our model of the cache contains only metadata objects (not memory pages or the disk block data), and the objects are of two types (HM1 and HM3), depending on the hashmap they belong to.

The cache records several metrics during simulation, especially the following numbers:

- hits in each pool for each type of hashmap object
- overall misses for each type of hashmap object
- evictions from each pool for each type of hashmap object
- objects of each hashmap in each pool

In addition to these, it records several numbers for each vDisk that the cache serves, such as:

- HM1 objects in each pool
- hits in each pool (only for HM1 objects)
- overall misses (only for HM1 objects)

An example of these recorded metrics as they appear in the simulation output is given below:

```
_____________________________________________
| (objects)   |  HASHMAP 1 |  HASHMAP 3 |
_____________________________________________
| SINGLE POOL |      44396 |      44352 |
_____________________________________________
| MULTI POOL  |     177468 |     177427 |
_____________________________________________

_____________________________________________
| (misses)    |  HASHMAP 1 |  HASHMAP 3 |
_____________________________________________
| TOTAL       |     957363 |     915320 |
_____________________________________________

_____________________________________________
| (hits)      |  HASHMAP 1 |  HASHMAP 3 |
_____________________________________________
| SINGLE POOL |     772255 |     735580 |
_____________________________________________
| MULTI POOL  |    3922446 |    4001164 |
_____________________________________________

_____________________________________________
| (evictions) |  HASHMAP 1 |  HASHMAP 3 |
_____________________________________________
| SINGLE POOL |     140712 |     135388 |
_____________________________________________
| MULTI POOL  |     815838 |     721722 |
_____________________________________________

_____________________________________________________________________________________
| (HM1 stats) |     SINGLE |      MULTI | TOTAL HITS |     MISSES |  HIT RATIO |
_____________________________________________________________________________________
| VDISK   520 |     192315 |     466161 |     658476 |     247484 |      0.731 |
_____________________________________________________________________________________
| VDISK   521 |      42996 |     127116 |     170112 |      52856 |      0.765 |
_____________________________________________________________________________________
| VDISK   539 |      11513 |     427094 |     438607 |      11776 |      0.976 |
_____________________________________________________________________________________
| VDISK   540 |     223196 |    1318524 |    1541720 |     273130 |      0.854 |
_____________________________________________________________________________________
| VDISK   554 |      80425 |     738829 |     819254 |      84693 |      0.908 |
_____________________________________________________________________________________
| VDISK   570 |     221810 |     844722 |    1066532 |     287424 |      0.792 |
_____________________________________________________________________________________

_____________________________________________________
| (ALL objs)  |       HASHMAP 1 |       HASHMAP 3 |
_____________________________________________________
| VDISK   520 |  41082 / 100616 |  41134 / 100616 |
_____________________________________________________
| VDISK   521 |   7574 / 31629  |   7595 / 31629  |
_____________________________________________________
| VDISK   539 |  10099 / 10319  |  10099 / 10319  |
_____________________________________________________
| VDISK   540 |  78460 / 102394 |  78474 / 102394 |
_____________________________________________________
| VDISK   554 |  16641 / 25617  |  16648 / 25617  |
_____________________________________________________
| VDISK   570 |  68008 / 102354 |  68026 / 102354 |
_____________________________________________________

Total Hits  : 9431445
Total Misses: 1872683
Hit Ratio   : 0.834
```

3.1.2 Replacement Policies

LRU and LFU policies have been implemented for cache replacement. ARC policy could not be applied to this cache as we have two types of objects (with disproportional sizes) that can be stored in the cache as opposed to fixed-size pages because ARC limits the cache by number of pages, not the total available memory.

Since the cache is made up of 3 different pools with different actions for eviction and subsequent accesses, we use 3 separate instances of these policies, i.e., there is a separate LRU/LFU list for each pool of the cache.

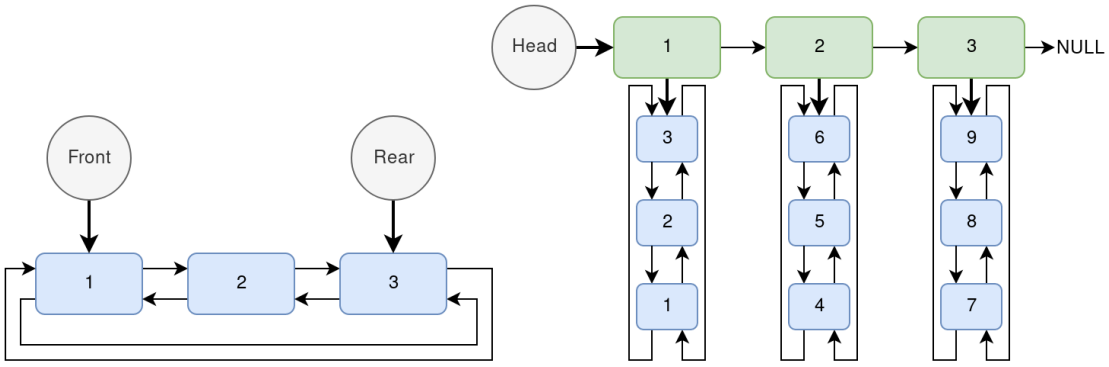

Figure 3.1: Implementation overview of LRU (left) and LFU (right)

- **LRU** was implemented using a double-linked list of key-value pairs, with the most recent entries added to the Rear and the least recent entries removed from the Front, similar to a queue. The insertion and eviction procedures take $O(1)$ time, but certain operations such as deletion take $O(n)$ where n is the total number of pairs in the cache. Binary Heaps can also be used to implement LRU, but all the operations will take $O(\log_2 n)$ time. Deletion operation is used when we move a pair from single-touch to memory multi-touch pool, or from SSD multi-touch to memory multi-touch pool. We use a different approach to finding the position of a pair in the LRU list which makes the deletion happen in $O(1)$, specified in the next section. Since the position of an object in the LRU list indicates its recency, we do not attach a timestamp counter to each object.

- **LFU** was implemented using a combination of two linked lists, first of which is single-linked and the other double-linked. The first linked list acts as the frequency list, where each node serves as the Head to a double-linked list of pairs having the same frequency. This first-level Frequency list contains nodes in ascending order of frequency. The second-level double-linked list stores the pairs in LRU manner. Using this two-dimensional linked-list setup makes insertion, eviction and deletion operations take $O(1)$ time, as opposed to using Binary Heaps which requires $O(\log_2 n)$ time.

Since the comparison of replacement policies is not the current goal of this study, and since the Nutanix Unified Cache uses LRU, we have not used LFU in the empirical analysis.

3.2 Filesystem Metadata Layer

Nutanix HCI makes use of 3 hashmaps in its metadata layer. While they are implemented using Apache Cassandra in the HCI, we make use of hashmaps (implemented using red-black trees) to simulate the behaviour. We consider only the first and third hashmap for the metadata lookups in this project.

(HM1)	Description	Type	Notes
Key	hash(vDisk ID, vblock number)	40-character long string of hexadecimals	SHA1 hash of two integers converted into character string
Value	egroup ID	32-bit integer	integer conversion of 6 hexadecimal characters taken from the hashed key

Table 3.1: Implementation details of a Hashmap 1 Key-Value pair

(HM3)	Description	Type	Notes
Key	egroup ID	32-bit integer	–
Value	list of extents and slices	–	uninitialized data; this value is ignored after lookup

Table 3.2: Implementation details of a Hashmap 3 Key-Value pair

Each hashmap object in our implementation consists of a root node and a read-write lock. A node is made up of a key-value pair and red-black tree metadata (parent, left, right and colour). We also embed certain cache metadata values within each node. The cache metadata consists of a flag indicating if the node is in the cache (and in which pool) or not, and it also contains an embedded linked list for each of the cache replacement policies, LRU and LFU.

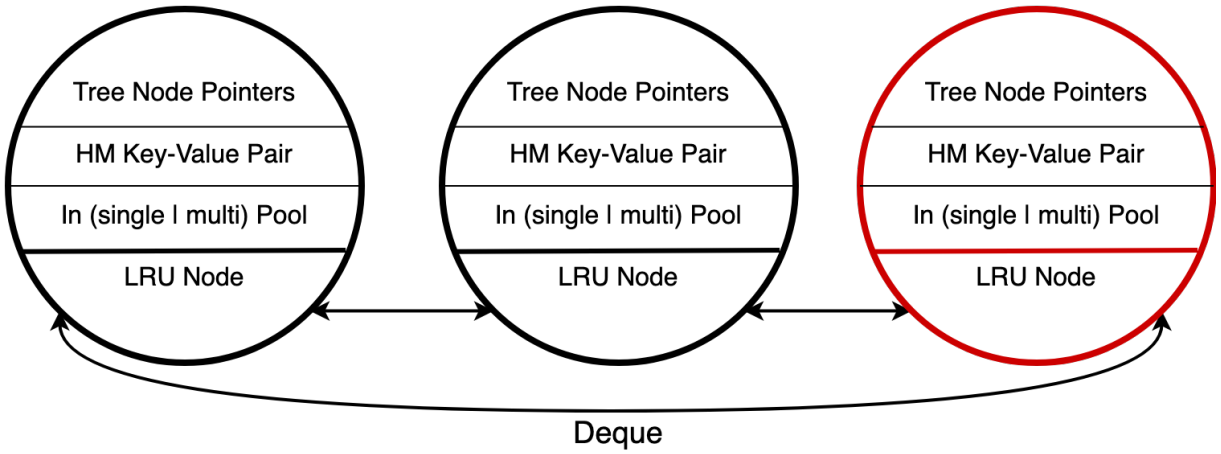

Figure 3.2: Structure of a node

The embedding of cache metadata and replacement policy metadata inside a node helps in efficiently inserting, deleting and evicting a node into/from the cache. Using this, we can simply set a few fields in each node to represent its availability in the cache, as well as its position in the LRU/LFU list. If we don't use this approach, we need to maintain another hashmap for the objects in the cache, and each insertion/deletion/eviction operation will take additional O(ln n) time. Moreover, embedding the information about LRU/LFU lists inside a node makes the list independent of the type of node which is added to the list.

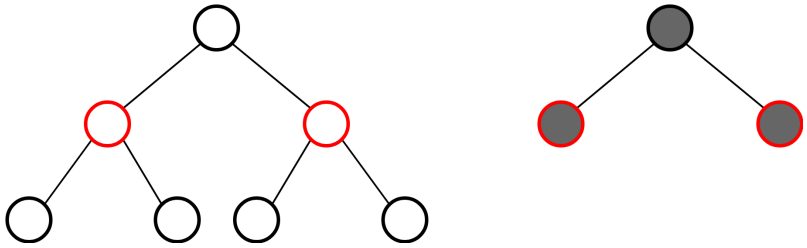

Figure 3.3: Using separate hashmaps for the metadata and the cache

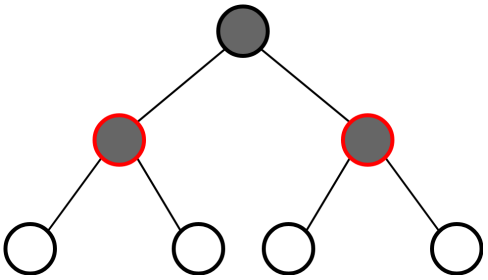

Figure 3.4: Reusing the metadata hashmap for cache lookup

Figures 3.3 and 3.4 highlight the difference made by embedding the cache metadata and replacement policy metadata inside the cache. We can reuse the same hashmaps (1 and 3) we maintain for our key-value store and

3.3 Persistent Cache State

Since the cache is prone to volatility in the face of multiple failure scenarios, we need to persist the cache state into the disk. We can save all the objects in the cache as well as its internal metadata to the disk by serializing its data structures. The problem with this approach is that we might be saving some data that is unimportant as we just need a way to know which objects were there in the cache last time the state was saved.

Another approach is to save only the data of objects currently in the cache by defining primitives to serialize/deserialize these objects. There are two problems associated with this approach. First, we will have to save a large amount of data (in the order of GBs) to the disk, resulting in storage I/O overhead. Second, the objects (especially the Hashmap 3 objects) we saved last time to the disk may have their data modified (by other nodes in the cluster) by the time we read them back.

Yet another approach is to save only the information about which objects are in the cache, and not their data, to the disk. Since the objects in the cache are key-value pairs queried from metadata hashmaps, we can only maintain the key part in our cache state as the value can always be queried again. For instance, we can simply write the following line to a file on the disk

```
37:345,5496,5876,450968
```

which will mean that objects corresponding to the vblock numbers $345, 5496, 5876, 450968$ of the vDisk 37 were in the cache at the time this state was captured. Moreover, we can have a separate file for each vDisk and one file for all the Hashmap 3 objects.

We further simplify the representation by exploiting the nature of these keys. All keys of hashmap 1 consist of a hash of the requesting vDisk and the vblock number. For this Hashmap, we maintain some state per-vDisk. Since the vblock number range for each vDisk is fixed (as its maximum size is fixed), we can keep information about all its vblocks being in the cache or not by using a **bitmap**. When a vblock belonging to a certain vDisk is added to the cache, we can turn the corresponding bit on. For example, if a vDisk is 10 GB in size, we know that it consists of 10 K extents (vblocks). Then, we need 10 K bits (≈1.2 KB) to represent Hashmap 1 cached objects for that vDisk.
For hashmap 3 objects, we need to maintain only one bitmap as its key, the egroup ID, is global to the filesystem. Moreover, as the key for Hashmap 3 is a fixed-width unsigned integer (32-bit length assumed in this study), we will need 2^{32} bits (≈512 MB) to represent the cache membership of its objects.

The bitmap representation takes some additional space in memory as part of cache-internal metadata. But, it saves a lot of space when saved to the disk, as compared to saving the complete object data. We have two choices for implementing the bitmap for persistent cache state:

- **Bit Array**
  An array of integers can be used to represent the space of all the valid bits which correspond to an object. We use an array of 64-bit unsigned integers, thereby grouping 64 objects into a single variable. To set/unset a bit, we need to first find the integer in the array which represents that bit, and then get to that bit. To get to an integer in the array, we can divide an object number by 64, and similarly to get to the specific bit, we can perform a modulo 64 operation. Bit set/unset operation itself will be a combination of bitwise shift and bitwise and/or/not operations. Setting/unsetting a particular bit is an $O(1)$ operation. Although this representation saves space by using only 1 bit for an objects, space may still be wasted if the bit array is large and only a few bits are set. This representation is costly if large contiguous ranges of bits are never accessed.

```
Bit Array for vDisk X (HM1 objects for vblocks 1 & 2 are cached)
_____________________________________________________
vblock 0  vblock 1  vblock 2  vblock 3  ...  vblock n
_____________________________________________________
   0         1         1         0      ...     0
_____________________________________________________


Bit Array for vDisk Y (HM1 objects for vblocks 0 & 1 are cached)
_____________________________________________________
vblock 0  vblock 1  vblock 2  vblock 3  ...  vblock m
_____________________________________________________
   1         1         0         0      ...     0
_____________________________________________________
```

```
Bit Array for HM3 (objects for egroup ID 1 & 3 are cached)
_____________________________________________________
egroup 0  egroup 1  egroup 2  egroup 3  ...  egroup k
_____________________________________________________
   0         1         0         1      ...     0
_____________________________________________________
```

- **Bit Set**
  A hashmap can be used to represent the bits, where the key is an integer and its value is a set of bits (which in turn is also an integer). We use 64-bit integers for the key as well the value. The key is a multiple of 64 and a key x represents all bits from x to $2 * x - 1$, i.e. 64 bits (which is its value). Moreover, the hashmap will have an entry for a particular key only if at least one bit in its 64-bit space is set. Setting/unsetting a particular bit is an $O(\log_2 n)$ operation. When the first bit is set in a 64-bit chunk, a new entry will be added to the hashmap. Similarly, when the last set bit is unset, the corresponding entry will be deleted. This representation saves space on disk as compared to the bit array representation as information is maintained only for used bit spaces.

```
Bit Set for vDisk X (HM1 objects for vblocks 192 & 3467 are cached)
3:  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000001
54: 00000000 00000000 00000000 00000000 00000000 00000000 00001000 00000000

Bit Set for vDisk Y (HM1 objects for vblocks 763245 & 763246 are cached)
11925: 00000000 00000000 01100000 00000000 00000000 00000000 00000000 00000000

Bit Set for HM3 (HM3 objects 23948 & 45678762 are cached)
374:    00000000 00000000 00000100 00000000 00000000 00000000 00010000 00000000
713730: 00000000 00000000 00000100 00000000 00000000 00000000 00000000 00000000
```

  Note that when we are saving a bit set to the disk, we don't need to write the complete 64-character binary string and can simply use its corresponding 64-bit unsigned integer value. The files the on disk having information about the bit sets mentioned in the example above can be simplified as below:

```
Bit Set for vDisk X
3,1
54,2048

Bit Set for vDisk Y
11925,105553116266496

Bit Set for HM3
374,4398046515200
713730,4398046511104
```

For empirical analysis performed in this study, we use only the Bit Array representation.

3.3.1 Cache Snapshots

Since the contents the cache will be updated very frequently, we need to capture the state of our cache at various time intervals and generate a series of snapshots.

We periodically dump the bitmaps maintained by the cache for each vDisk's hashmap 1 objects and for the hashmap 3 objects to the underlying storage in separate files. We call these files cache snapshots and in this manner, the cache is persisted. The dump file names follow the following format:

```
{Base Name}.{Node/Cache ID}.{Epoch Number}.{Bitmap Type}
```

where:

- **Base Name** denotes the absolute path to the dump file for each vDisk
- **Node/Cache ID** is an identifier for the cache generating the dumps, useful only in the scenario where we simulate two instances of the cache in parallel
- **Epoch Number** specifies the epoch at which this dump was made. Currently, we are taking the time period of 30 seconds as one epoch.
- **Bitmap Type** is either .csv (for Bit Set) or .bin (for Bit Array)

We define the term snapshot rate as the number of requests after which a snapshot of the cache is made. So, a snapshot rate of 1 K will mean that we snapshot the cache after every 1000 requests. We currently do not make the snapshots at fixed time intervals as the simulation does not take into account the time taken to serve a request. We instead record the total number of block I/O requests seen by the cache and make the snapshot if a certain number of requests were issued since last snapshot was taken.

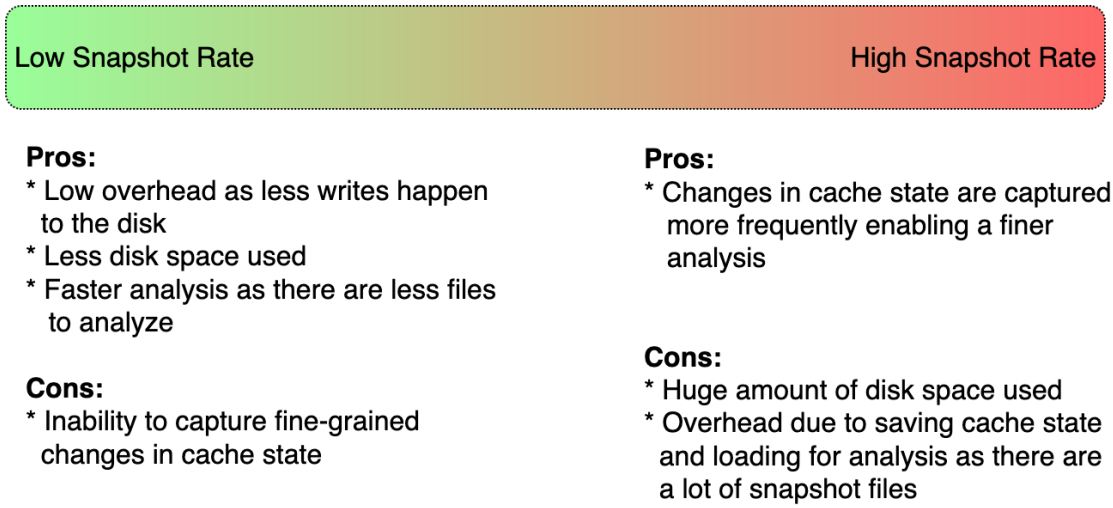

Figure 3.5: Snapshot rate tradeoffs

Figure 3.5 compares the tradeoffs associated with using very high and very low snapshot rates. We ideally want a snapshot rate which is not very extreme and avoid the respective drawbacks. Since we are saving these snapshot files to the underlying storage, and the distributed filesystem supports replication (with a certain Replication Factor), the snapshot files saved on the node that experiences a failure will still be available on the distributed filesystem.

3.4 Heuristic-based Snapshot Analysis

As part of our approach to prewarm the cache, we need to determine the set of objects to prewarm the cache with, and in order to do so, we need to analyze a series of cache snapshots taken in the past. Ideally, we want those objects in the prewarm set which are important and will be relevant after they are fetched into the cache, in the sense that loading them into the cache should result in a high hit rate and prevent cache pollution. But at the same time, we need to restrict the size of this set as loading the objects into the cache will require a series of lookups in the metadata layer (distributed key-value store), which will incur additional overhead of network transfers.

As mentioned earlier, we represent each object as a bit in a bitmap, and the collection of bitmaps of all vDisks' HM1 objects and of HM3 objects at a particular time makes up a snapshot. If a bit at a certain position is set in a snapshot, we say that the corresponding object was present in the cache at the time we captured this snapshot.

Determining the cache prewarm set can be reduced to the problem of finding most important and relevant objects optionally constrained by the total size of the prewarm set.

```
HM1 (VDISK 0)          HM1 (VDISK 1)          HM3
─────────────────────  ─────────────────────  ──────────────────────────
1  0  0  0  0  0  1    0  1  0  0  0  1  0    1  0  1  1  0  1  1  0  Snapshot 1
0  0  0  1  0  0  1    0  0  0  1  1  0  1    0  0  1  1  1  0  1  1  Snapshot 2
1  1  0  0  1  0  1    1  1  0  0  0  0  1    0  1  0  0  0  0  1  0  Snapshot 3
.  .  .  .  .  .  .    .  .  .  .  .  .  .    .  .  .  .  .  .  .  .   ...
1  0  1  0  0  1  0    0  0  1  0  0  1  0    0  0  1  0  1  0  0  1  Snapshot n
─────────────────────  ─────────────────────  ──────────────────────────
1  0  1  1  1  0  0    0  0  0  0  1  0  1    0  0  1  0  1  1  1  1  Prewarm set
─────────────────────  ─────────────────────  ──────────────────────────
```

To find a solution to this problem with the given constraint, we use a few heuristics, explained below, which can effectively help us in establishing the importance and/or relevance of an object.

1. **k-Frequent** (Frequency without memory constraints)
   Gives highest priority to the objects which appear in most snapshots. We measure the frequency of a cached object by percentage of snapshots it is present in. In our case, we can count the number of snapshots in which a particular bit was set to determine its frequency.

   ```
   1  0  0  0  0  1  1  0  0  1  Snapshot 1
   0  0  0  1  1  0  1  1  0  0  Snapshot 2
   1  1  0  0  0  0  1  0  1  1  Snapshot 3
   0  0  0  1  1  1  0  1  1  1  Snapshot 4
   1  0  1  0  1  1  0  1  1  0  Snapshot 5
   ──────────────────────────────
   3  1  1  2  3  3  3  3  3  3  Frequency count
   ──────────────────────────────
   1  0  0  1  1  1  1  1  1  1  Objects occurring in at least 50% snapshots
   ──────────────────────────────  (Prewarm set)
   ```

2. **k-Recent** (Recency without memory constraints)
   Gives highest priority to the objects which appear in the most/least recent snapshots. To

take recency into account we add all the objects in the last few (or first few) snapshots to our prewarm set. Whether to take the most recent or least recent snapshots into consideration depends on when we are prewarming the cache. We will make use of both most and least recent snapshots in our analysis for different scenarios.

```
1  0  0  0  0  1  1  0  0  1  Snapshot 1
0  0  0  1  1  0  1  1  0  0  Snapshot 2
1  1  0  0  0  0  1  0  1  1  Snapshot 3
0  0  0  1  1  1  0  1  1  1  Snapshot 4
1  0  1  0  1  1  0  1  1  0  Snapshot 5
————————————————————————————
1  0  1  1  1  1  0  1  1  1  Objects in the last 2 snapshots
————————————————————————————  (Prewarm set)
```

3. **k-Frerecent** (Frequency & Recency without memory constraints)
   Priority of an object is determined by how frequently it has occurred in the snapshots as well as how recent was its last occurrence in the snapshots. Another way to see this is that taking both of these factors into account helps us distinguish between objects with the same recency or the same frequency value. One method of applying this heuristic is to perform a weighted sum of all bits in all the snapshots. Then we can sort the bits according to their summed weight and consider all the objects having highest 'n' values or top 'x%' objects by values; we use percentage instead of an absolute number to select objects. There are many ways to define a weight for objects in a particular snapshot. For instance, a weight of 1 for all snapshots will result in the same heuristic as k-Frequent. We use two methods for determining the weight, both of which are functions of the snapshot number. One of them is the identity function: $f(x) = x$, and the other is the square function: $f(x) = x * x$. x here is the snapshot number, where a value of 1 means the first snapshot and it keeps increasing by 1 for each subsequent snapshot. In the example below, we use the square function.

   Another way to do this is to keep adding the objects into the prewarm set till the memory constraint is satisfied.

```
1  0  0  0  0  1  1  0  0  1   Snapshot 1 (weight: 1)
0  0  0  1  1  0  1  1  0  0   Snapshot 2 (weight: 4)
1  1  0  0  0  0  1  0  1  1   Snapshot 3 (weight: 9)
0  0  0  1  1  1  0  1  1  1   Snapshot 4 (weight: 16)
1  0  1  0  1  1  0  1  1  0   Snapshot 5 (weight: 25)
————————————————————————————
35 9  25 20 45 42 14 45 50 26  Weighted Sum
————————————————————————————
1  0  0  0  1  1  0  1  1  0   Objects having a score of 30 or more
————————————————————————————  (Prewarm set)
```

4. **Constrained-k-Frequent** (Frequency with memory constraints)
   This is similar to k-Frequent, but with an upper limit on the prewarm set size. We do not specify the frequency value explicitly as a parameter and keep on adding the most frequent objects until the memory limit is exceeded. In practice, we always have a constraint on the total memory available for prewarming and explicitly specifying a frequency value or percentage might result in the prewarm set being either overfilled or not being fully filled up to potential.

```
1  0  0  0  0  1  1  0  0  1  Snapshot 1
0  0  0  1  1  0  1  1  0  0  Snapshot 2
1  1  0  0  0  0  1  0  1  1  Snapshot 3
0  0  0  1  1  1  0  1  1  1  Snapshot 4
1  0  1  0  1  1  0  1  1  0  Snapshot 5
_____________________________
3  1  1  2  3  3  3  3  3  3  Frequency score
_____________________________


_____________________________
3  3  3  3  3  3  3  2  1  1  Frqequency values
0  4  5  6  7  8  9  3  1  2  Objects sorted by frequency
_____________________________
1  1  1  1  1  0  0  0  0  0  Memory constraint ~size of 5 objects
_____________________________


_____________________________
1  0  0  0  1  1  1  1  0  0  Prewarm set
_____________________________
```

5. **Constrained-k-Recent** (Recency with memory constraints)
   This is similar to k-Recent, but with an upper limit on the prewarm set size. We do not specify the recency value explicitly as a parameter and keep on adding all the objects in most recent snapshots until the memory limit is exceeded. As was the case with an explicit frequency value, explicitly specifying a recency value might result in the prewarm set being either overfilled or not being fully filled up to potential.

```
1  0  0  0  0  1  1  0  0  1  Snapshot 1
0  0  0  1  1  0  1  1  0  0  Snapshot 2
1  1  0  0  0  0  1  0  1  1  Snapshot 3
0  0  0  1  1  1  0  1  1  1  Snapshot 4
1  0  1  0  1  1  0  1  1  0  Snapshot 5
_____________________________
5  3  5  4  5  5  3  5  5  4  Recency score
_____________________________


_____________________________
5  5  5  5  5  5  4  4  3  3  Recency values
0  2  4  5  7  8  3  9  1  6  Objects sorted by recency
_____________________________
1  1  1  1  1  0  0  0  0  0  Memory constraint ~size of 5 objects
_____________________________


_____________________________
1  0  1  0  1  1  0  1  0  0  Prewarm set
_____________________________
```

6. **Constrained-k-Frerecent** (Frequency & Recency with memory constraints)
   In k-Frerecent, we defined a way to assign weights to objects in a snapshot and perform a weighted sum to establish importance, but we needed to manually specify the threshold (top x% objects). Here, we do not specify a threshold for the weighted sum of objects and

keep adding them to the prewarm set (in decreasing order of their sum values) until the upper limit on prewarm set is exceeded.

```
1  0  0  0  0  1  1  0  0  1   Snapshot 1 (weight: 1)
0  0  0  1  1  0  1  1  0  0   Snapshot 2 (weight: 4)
1  1  0  0  0  0  1  0  1  1   Snapshot 3 (weight: 9)
0  0  0  1  1  1  0  1  1  1   Snapshot 4 (weight: 16)
1  0  1  0  1  1  0  1  1  0   Snapshot 5 (weight: 25)
_____________________________
35 9  25 20 45 42 14 45 50 26  Weighted Sum
_____________________________


_____________________________
50 45 45 42 35 26 25 20 14 9   Score values
8  4  7  5  0  9  2  3  6  1   Objects sorted by score
_____________________________
1  1  1  1  1  0  0  0  0  0   Memory constraint ~size of 5 objects
_____________________________


_____________________________
1  0  0  0  1  1  0  1  1  0   Prewarm set
_____________________________
```

3.4.1 Working Set Size Estimation

In the context of this study, we define the Working Set Size (WSS) of a vDisk as the total size of its objects (HM1 & HM3) that are in present in the cache (over a period of time). For HM3 objects, we currently charge the size to all the vDisks that have their vBlocks in that HM3 object. The WSS of each vDisk gives a relative idea about the amount of cache a particular vDisk uses in its normal run. A vDisk may have a large number of objects resident in the cache, but not all of them may be actively used and will eventually be evicted. Working set essentially includes all the temporally relevant objects and enables us to determine the share of cache each vDisk needs individually.

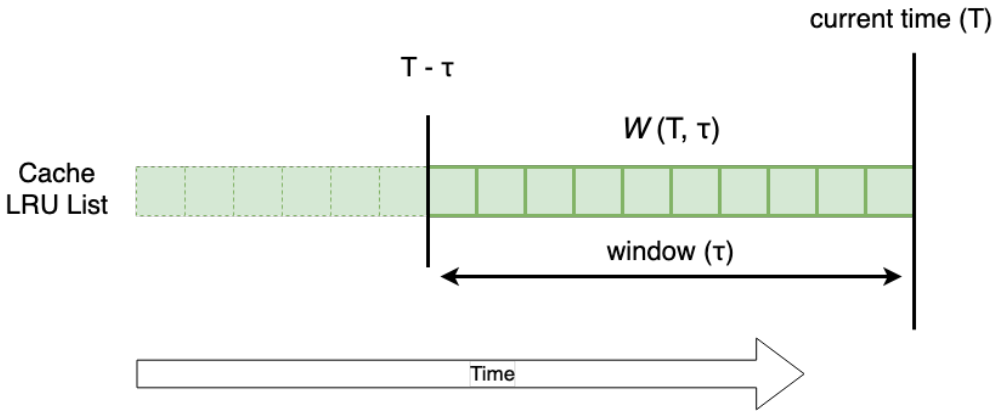

Figure 3.6: Using a moving window for estimating the WSS

In our empirical analysis, we estimate the WSS for each vDisk. This helps us in getting an estimate about each vDisk's memory needs.

3.4.2 Prewarm Set Partitioning

Since we have two types of metadata objects and a number of vDisks, we need to logically partition the set to decide the share for each type of object. This is necessary in order to ensure that one type of object (or a vDisk object) does not fill up the whole set and to accommodate a diverse set of objects.

We consider two types of partitioning for the prewarm set:

- Partition between all HM1 and HM3 objects
  We need to determine the share of the prewarm set for each type of metadata object as both HM1 and HM3 objects have different costs and benefits. There are many policies to perform this partition:
  - No partitioning: Do not reserve space in the set for HM1 and HM3 objects
  - Equal HM3: Split the prewarm set into two equal parts for HM1 and HM3 objects
  - No HM3: Consider only HM1 objects for the prewarm set, and ignore all the cached HM3 objects
  - Closed HM3: Consider only those HM3 objects which can be resolved into by another HM1 object in the cache; use the remaining space for HM1 objects.
- Partition among the HM1 objects for each vDisk
  We need to further partition the space available for HM1 objects for each vDisk to ensure that vDisks with a large number of objects in the cache do not cloud those with only a few objects. Some policies that can be used to perform this partition are:
  - No partitioning: Do not reserve space in the set for HM1 objects of vDisks
  - Proportional Share: Split the prewarm set for vDisks in ratios calculated from the number of their currently cached objects
  - Proportional Share (WSS): Split the prewarm set for vDisks in ratios calculated from their most recently determined working set sizes
  - Fair Share (vDisk): Split the prewarm set for HM1 objects equally for all vDisks
  - Fair Share (VM): Split the prewarm set for HM1 objects equally for all VMs (a VM can have more than one vDisk)

  Note that even if we partition the space for HM1 objects for each vDisk, it might be possible that some vDisk does not have sufficient cached objects to fill up its share. Hence, this partitioning will be performed iteratively, redistributing remaining space in the prewarm set among the vDisks that have more cached objects.

3.5 Cache Prewarming

After we analyze a set of snapshots taken in the past and determine the objects that will become a part of the prewarm set, we set the bits corresponding to those objects in a new (prewarm) bitmap. As before, there will one such bitmap for each vDisk's HM1 objects and one for HM3 objects. We can optionally save this prewarm bitmap to the disk to checkpoint our analysis so that for next analysis, all the snapshots taken prior to this analysis will not be needed again. With this bitmap ready, we know which objects to load into the cache as we have the key part already in the bitmaps (the position of a bit in the bitmap tells us about the vBlock/egroup ID depending on the object type). The next step is to actually fetch these objects from the metadata store and store them in the cache. We are not considering the various costs associated with the metadata lookups over the network in the current study.

After we have queried the value associated with an object in the prewarm set, we need to store the object in the cache. The cache itself can be in one of two states: cold (either empty or has stale objects) or warm (having objects of some running VMs). Moreover, since we have a cache with multiple pools, we have a choice in which pool(s) to use to load these objects. We consider using both pools for prewarming, given that there is space for the incoming prewarming object. We start loading objects in the multi-touch pool first. If it gets filled up (due to prewarming or due to normal vDisk I/O traffic), the remaining objects are loaded in the single-touch pool. When the single-touch pool gets full, we stop the prewarming and the objects remaining in the prewarm set are discarded. Some objects in the prewarm set may already be in the cache (due to normal I/O traffic). These objects are skipped and the next one in order is considered.

Note that we use multi-touch pool first to load the most important objects because unlike in single-touch pool, an object which replaces another one in the multi-touch will be equally as important. Another important point here is that the most important objects (loaded into the multi-touch) will be the ones to get evicted unless they are accessed again within a short time period.

3.6 Throttling Prewarming

Prewarming process involves the following steps:

1. Determine all the HM1/HM3 objects in the prewarm set we constructed using heuristic-based snapshot analysis, and iterate over them in the order determined by the heuristic.
2. Lookup each of these objects in the cache. If an object is already present in the cache (any of the pools), skip it.
3. If an object is not found in the cache, query the metadata store and load it into the cache. Try loading into multi-touch pool first. If it gets full, try single-touch pool.

4. Repeat these steps until either the cache gets full (both pools) or the prewarm set size limit is exceeded.

Throughout this process, we perform several operations that will affect resources in a real infrastructure. For looking up objects in the cache, we perform a main memory access. For querying and fetching objects from metadata store, we perform a query over network and this in turn will invoke more operations depending on how the metadata store is organized. Fetching the object from metadata store into the cache will involve a network transfer and another memory access. This operation of querying and fetching objects form the major part of our cache miss penalty.

In the scenario where the cache is already serving some vDisks, there might be an increased contention due to new objects being fetched to the cache. Since the prewarming process forcefully loads these new objects, objects belonging to other vDisks may start getting evicted and the associated VMs may start experiencing an increase in cache misses. Some of these misses are inevitable, but the major issue arises when a large portion of these actively used objects (of other vDisks) start getting evicted out of the cache. We have set an upper limit on the total size of objects that are in the prewarm set in our empirical analysis. But if the complete prewarm set is loaded into the cache instantly, the functioning cache state will get disturbed, affecting the existing VMs.

We try reducing these impacts due to prewarming by imposing an upper limit on the total size of objects that can be loaded into the cache withing a short time period. In our analysis, we specify a prewarm rate in terms of the total size that can be loaded within one second. This implies that if we have the prewarm set size as x *MB* and our prewarm rate as y *MBps*, at most y *MB* worth of objects can be loaded within a second, and our prewarming process will go one for at most $\lceil x/y \rceil$ seconds. We use upper bounds ("at most") here as some objects in the prewarm may already be loaded in the cache and they can be skipped over.

3.7 Simulation Environment

Our simulation environment consists of the hypervisor cache, metadata hashmaps and driver functions for running various experiments. We can view this simulation as a system which is fed a sequence of block I/O requests and under the presence of various tunable and fixed parameters produces a set of metrics that enable us to get the answer for various questions we are interested in. Figure 3.7 shows a basic model of our system.

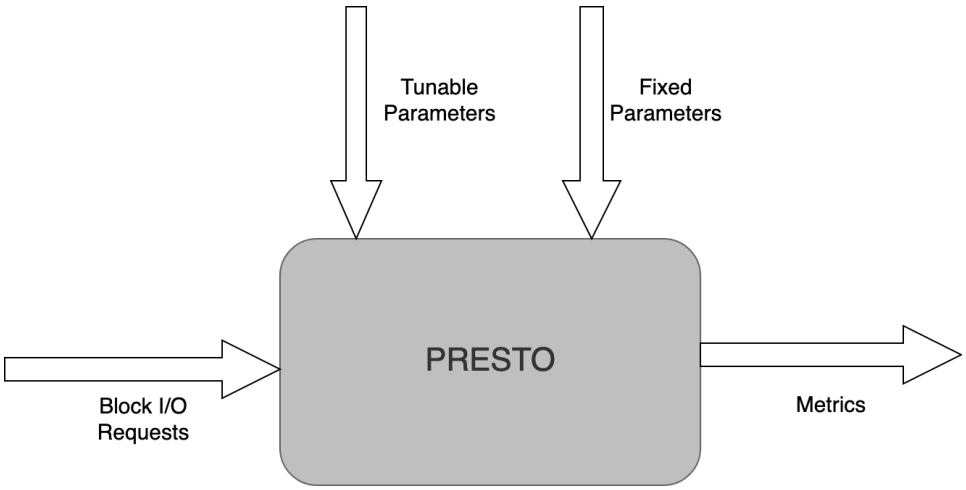

Figure 3.7: Basic overview of the simulation

The general sequence of steps taken in our experiments is as follows:

```
1. Read requests one by one from the block I/O trace file

    1a. Split the request into two if it spans over two extents
    1b. Perform lookup in the cache for HM1 key determined from the request
        – If the HM1 object is not there in the metadata map, create and insert it
        – If the HM1 object is not cached, fetch it into the cache
    1c. Perform lookup in the cache for HM3 key corresponding to the value of
        looked up HM1 object
        – If the HM1 object is not there in the metadata map, create and insert it
        – If the HM1 object is not cached, fetch it into the cache
    1d. Record metrics such as hits, misses and number of cached objects for
        lookups
    1e. If the number of requests served (since last snapshot was taken)
        exceed the snapshot rate:
            Dump the bitmaps of all vDisks and HM3 objects to the disk


2. Analyze the snapshots taken so far

    2a. Read the snapshots one–by–one starting from the last one
        – Create a bitmap for each vDisk and for HM3 representing the prewarm set
        – Perform analysis according to heuristic used
        – Mark the objects to be included in the prewarm set

3. Reset the cache and load the objects from prewarm set into the cache

    3a. Query those objects from metadata maps which are in the prewarm set
    3b. Fetch them into the cache


4. Repeat the requests from the point of reset

    4a. Perform the same lookup sequence again as before
    4b. Record metrics such as hits, misses and number of cached objects for
    lookups
```

Note that in the current simulation, we do not consider time for various events such as disk access, network transfer and even for request arrivals and completions. The simulator acts as a block I/O trace analyzer currently.

4. Experiments & Results

Our aim at performing the experiment is to mimic specific scenarios which can render the cache cold, and measure the benefits of prewarming. We particularly look at a scenario where the cache experiences a failure. Each run of the experiment involves issuing block I/O requests to the cache and measuring the hit ratio. Note that the outcome of these experiment runs depend heavily on the nature of the block I/O requests the cache serves. We perform all our experiments using 3 workload sets as described in the following section.

Each of our experiment uses the cache for a 4-hour long storage I/O traffic, and the total number of requests received in this time period depends on the nature of the workload.

4.1 VM Workloads

VM workloads are the input to our hypervisor cache, and help us simulate actual workloads running on a hypervisor. Each of these workloads is a series of pre-recorded block I/O requests, and these requests are issued to the cache, sequentially.

We create a trace file for each workload, which contains these block I/O requests. All the block I/O trace files used in the simulations have one I/O request in a line, where the lines have the following format:

```
{Type},{vDisk ID},{Sector Number},{Size},{Timestamp}
```

where:

- **Type** is either R (read) or W (write). R and W may be suffixed with F (Force Unit Access), A (readahead), S (sync) or M (metadata). We do not consider these suffixes to differentiate the request type further.
- **vDisk ID** is a global identifier for a virtual disk provisioned to a VM.
- **Sector Number** is the sector number (using 512 KB for the sector size) of the vDisk which is to be accessed. The actual offest is calculated by multiplying the sector number with this sector size.
- **Size** is the amount of data (in bytes) that will be read or written to the vDisk.

- **Timestamp** is the time (in seconds) of each request relative to the start time of our tracing utility. The precision within a second is upto a microsecond.

The traces recorded from various workloads are merged into a single CSV file. After merging, the requests are sorted on the basis of their timestamp. The trace file is read line-by-line by the simulator, and after parsing each line, a request is issued to the cache. There may be requests with I/O size over 1 MB (which is the extent size), and many requests span over more than one extent. We parse these in the same manner and issue two or more requests (which are extent-aligned) to the cache.

For instace, when the following request is read from the trace file,

```
R,2,1292882848,1310720,005286153
```

we get the following information:

```
Type: Read
vDisk ID: 2
Sector Number: 1292882848
Byte Offset: 661956018176 (Sector Number * 512)
vBlock Number: 631290 (Byte Offset / 1MB)
vBlock Offset: 475136 (Byte Offset % 1MB)
Size: 1310720 Bytes
```

Since in this case, the request size exceeds 1 MB, we split it into two requests that are extent-aligned as follows:

```
Request #1:
Type: Read
vDisk ID: 2
vBlock Number: 631290
vBlock Offset: 475136
Size: 573440 Bytes

Request #2:
Type: Read
vDisk ID: 2
vBlock Number: 631291
vBlock Offset: 0
Size: 737280 Bytes
```

Our experiment workload superset comprises of 2 types of workloads: real and synthetic.

- **Real Workloads**
  These workloads correspond to storage I/O traffic observed on a system/server used for normal operations. We have two sets of such workloads. One of them was collected from the FIU SyLab Resources [6] available on the web. The other was collected from applications running on various VMs on the IITB CSE Department Infrastructure. These traces were collected in a non-intrusive manner i.e. we have no runtime information about the resources used by these systems, but only the storage I/O information as seen from their underlying storage medium.

- **Synthetic Workloads**
  These workloads are generated with the help of an external tool and the I/O traffic does not correspond to actual real-life events. We have used two methods to generate such workloads: using storage benchmark tools inside a VM and using a custom workload generation tool which generates artificial I/O requests. One set of workloads using a storage benchmark program was collected from [4]. They have used the VMmark virtualization benchmark to generate various types of application workloads. We also have created a workload generation tool to be able to include a more diverse set of workloads into our superset. Since we are not using an actual system in a production environment here, we are able to control certain aspects of the I/O traffic that can be generated. For instance, we can set in advance what the rate of issuing I/O requests should be, or how much of the total disk space available is actually touched by the I/O requests.

The workloads comprise of trace files which are pre-recorded series of storage block I/O requests. For real workloads and for synthetic workloads running a benchmark, we need to be able to capture the disk I/O and parse it in into the format described above. There are two widely used techniques for capturing/tracing the I/O requests made by the applications/OS to the underlying storage: *NFS-based tracing* and *Block Layer I/O tracing*. We now look at Block Layer I/O tracing in detail and how we used the same to record storage I/O traffic on our servers.

Block Layer I/O tracing

The *blktrace* utility is onle of the most commonly used tool to record block I/O activity on Linux systems. We will take it as an example to explain the block layer I/O tracing. blktrace is a block-level I/O tracing utility which extracts information about various events from the kernel. The events are communicated by the kernel in debugfs files (in /sys/kernel/debug) through a series of buffers. We pass a Linux partition device file (such as /dev/sda1) as an input to it and it collects all the block I/O events associated with that device. As part of each event it traces, it captures certain attributes such as the timestamp of the event, the Logical Block Address (LBA) or the sector, the size of the request, and the type of request (Read/Write), among other attributes. Another related utility, *blkparse*, is used to provide a verbose output of the traces recorded by blktrace. We use blkparse to convert the traces into our required CSV format.

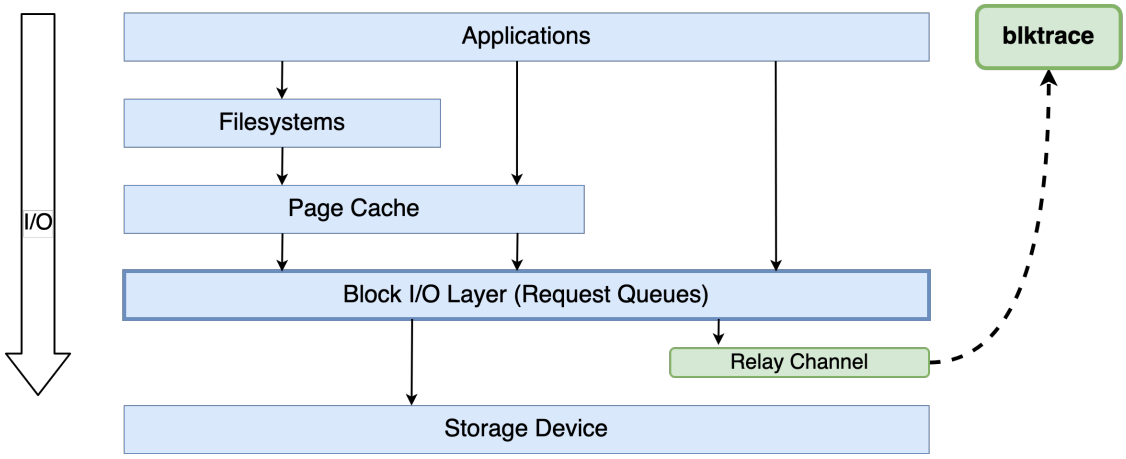

Figure 4.1: Block Layer I/O Tracing using blktrace utility

blktrace can also trace various types of block layer events such as *Inserted* (request sent to I/O scheduler and queued which will be later serviced by the driver), *Issued* (a queued request is sent to the driver) and *Complete* (an issued request is serviced). We have filtered these events and recorded only the requests that are Issued. In virtualized setups, the VM-issued storage I/O requests will eventually be serviced by the hypervisor, and in our case, the hypervisor cache. So, we are concerned only with those requests that reach the driver and can be seen by our cache.

For instance, we can use the following set of commands on VMs running a Linux kernel to record block I/O traces. The first one runs for 12 hours and captures block I/O events (filetered by Issued event). The second one parses the information captured by blktrace and produces the block I/O requests in a human-readable CSV format. This set of commands can be on each VM that we consider for workload generation.

```
blktrace –d /dev/sda1 –d /dev/sdb1 –w 43200 –a issue
blkparse sda1.blktrace.0 sdb1.blktrace.0 –q –f "%d,0,%S,%N,%T.%t\n"
```

An example of output from blkparse is given below. It was observed during the traces that most events captured on any VM (while collecting workloads for our experiment) corresponded to two types of processes, apart from the user-space applications. These two types

of processes run as background tasks, one of which is the kworker threads and the other is jbd2. An example of blkparse output is shown below.

```
  8,32   0    16132 21528.832980937   253  D  WS 1049159784 + 8 [kworker/0:1H]
  8,32   4     7556 21538.815958634   578  D  WS 1049159792 + 56 [jbd2/sdc-8]
  8,32   4     7557 21538.817070529   378  D  WS 1049159848 + 8 [kworker/4:1H]
  8,32   0    16133 21543.935871627  9005  D   W 476832560 + 8 [kworker/u16:1]
  8,32   2     1334 21545.984095780   578  D  WS 1049159856 + 144 [jbd2/sdc-8]
```

kworker is the name given to kernel worker threads which are responsible for doing all kernel-level activities such as handling hardware interrupts and performing I/O operations. Journaling Block Device (JBD) is a layer in the kernel which provides an interface to various journaling filesystems. ext4 uses a variant of JBD called jbd2.

4.1.1 Custom Workload Generation Tool

To generate workloads with desirable properties, we have created a tool in Python which generates a series of block I/O requests based on certain configurable parameters, and writes them to a trace file which we later feed as an input to the cache simulator. These requests do not originate from an actual filesystem/disk, but are randomly generated based on sampling from probability distribution(s).
Since these traces are completely synthetic, their disk block access pattern may not correspond to a real workload. We use them instead to fine-tune certain parameters, such as IOPS and total disk usage to better understand the effects of prewarming.

Configurable parameters

- **name**: name of the trace file (access log)
- **set**: name of the set to which this workload belongs
- **vDisk**:
  - id: vDisk ID (used by the cache internally)
  - size: vDisk size
- **randseed**: seed value for python RNG
- **duration**: total I/O run time of the vDisk
- **iops**: `duration` (seconds) values are be sampled from one of the probability distributions mentioned below; each sampled value specifies number of I/O operations issued at each second
  - normal(mean, stddev)
  - uniform(min, max)
  - poisson(mu)
  - beta(a, b, shift, scale)
- **sector**: (parameters for deciding sector values)

  - size: sector size (512 B is the most commonly used size)
  - cluster: it is a set of spatially close sectors; similar I/O requests are generated for a cluster of sectors
    * size: how many neighbours of this sector to access?
      A value taken from Uniform(min, max), i.e. uniformly from the closed interval `[min, max]`. It can also be called as the **locality size**.
    * stride: how far is the next sector to be accessed from the current one?
      A value taken from Uniform(min, max), i.e. uniformly from the closed interval `[min, max]`

- **hotspots**: disjoint clusters in the vDisk where I/O is performed
  The requests to a hotspot are assumed to be thread-agnostic. If a set of requests access a particular hotspot, all those requests will complete the accesses and only then another hotspot is chosen for the further requests. We use the format `[% requests, % vDisk]` to specify individual hotspots. This format indicates that a certain percentage of all I/O requests should happen to a certain portion of the vDisk.
- **readp**: percentage of I/O requests that should be reads
- **iosizes**: request size distribution, as a list of `[percentage of requests , I/O size]` pairs. All sizes are in bytes and the percentages must add up to 100.

The following steps describe the process of workload generation by the tool.

```
1. Sample IOPS values for each second from the specified distribution
    a. Total I/O requests will be the sum of these IOPS values

2. Determine hotspot ranges (start and end sectors) and their weights

3. Start with 0th second and compute timestamp for first request

4. Repeat until total I/O requests remaining becomes 0

    a. if cluster size value becomes 0
        i.   select a new hotspot using (weighted) uniform distribution
        ii.  select a new sector within the hotspot range, uniformly
        iii. select a new sector cluster size, uniformly

    b. if iops for current second is complete
        i.  move on to next second of time, and reset the iops value
        ii. calculate step value for timestamp

    c. I/O request size is chosen using (weighted) uniform distribution

    d. write an I/O request to the file as a line of CSVs

    e. select a sector stride value, uniformly and increment the sector

    f. increment timestamp value by step determined in (b)(ii)

    g. decrement cluster size and total I/O requests

    h. increment iops done for current second
```

We specify these paramaters in a YAML file which serves as the input to our tool. An example configuration with 5 hotspots is given below. Here IOPS values for each second (3600 for 1 hour duration) are sampled from a Normal distribution.

```
---
-   name: 'syn501'
    set: 'wload_6gb'
    vDisk:
        id: 501
        size: 100gb
    randseed: 1048576
    duration: 1h
    iops:
        randvar: normal
        mean: 100
        stddev: 20
    sector:
        size: 512b
        cluster:
            size:
                min: 1
                max: 10
            stride:
                min: 1
                max: 1000
    hotspots:
        - [10%, 1%]
        - [10%, 1%]
        - [30%, 10%]
        - [40%, 20%]
        - [10%, 1%]
    readp: 50%
    iosizes:
        - [80%, 4096b]
        - [10%, 8192b]
        - [5%,  16384b]
        - [5%,  131072b]
```

We have created multiple such configurations by changing iops, hotspots and sector cluster parameter values. Only a few of them are included in the synthetic workloads we will use.

Please note that the hotspots are not strictly defined in terms of sector range. A set of requests may eventually access a sector outside these hotspot ranges. But this happens with a very low chance and does not affect the total disk space used.

Since a sector is 512B is size and one HM1 object can cover 1MB of disk data, 1 HM1 represents ~2000 sectors. A sector stride of 2000 will mean that every sector access will be mapped to a different HM1 object. If we have a locality size of 10 and cluster stride of 100, all the sectors in this locality may map to the same HM1 objects. In the latter case, we will have 1 miss for the HM1 object (and the corresponding HM3 object), but 9 consecutive hits, resulting in a hit ratio of 0.9. To avoid this, we try to keep the locality size low and sector stride high.

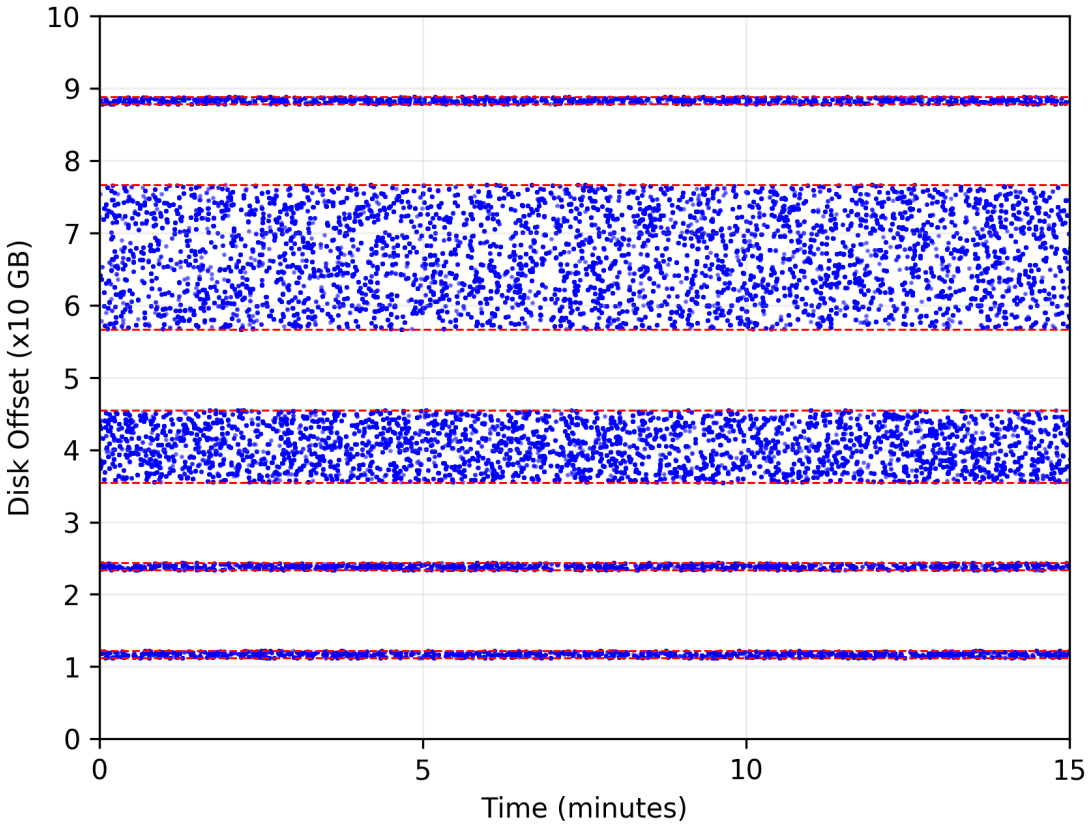

Figure 4.2: Disk block access pattern based on the configuration given in example above

The figure above shows the configuration with 5 hotspots (marked by dotted horizontal red lines). We have plotted the first 15 minutes of the disk block access pattern, but it remains the same throughout. This is one of the drawbacks of using a synthetic workload. For most real workloads, these hotspots change over time.

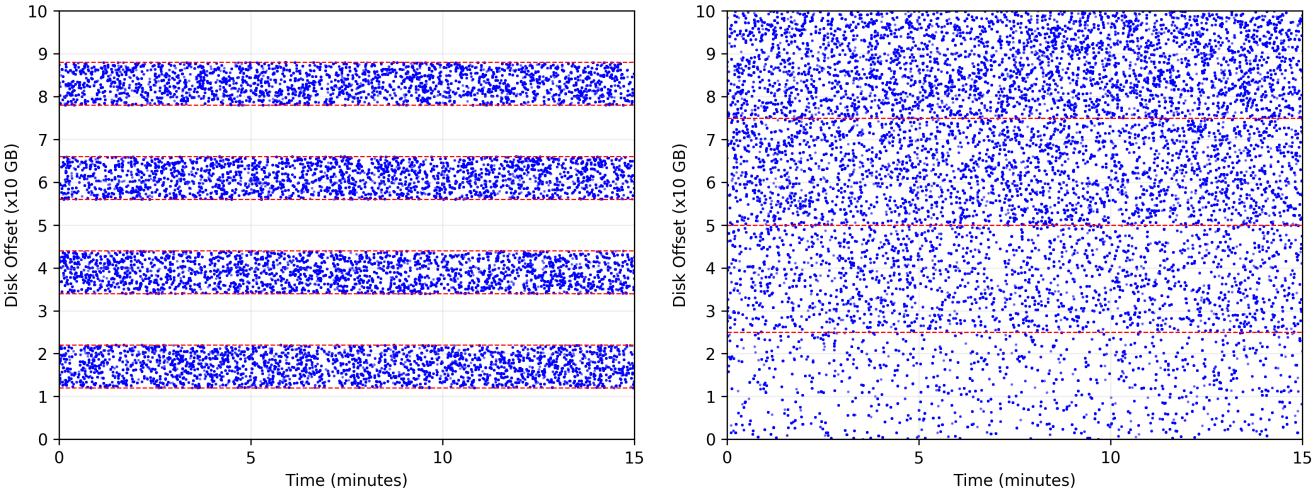

Figure 4.3: More examples of disk block access patterns of workloads generated using the tool

4.1.2 Experiment Workload Sets

For the experiment, we need at least a couple of workloads to test the effectiveness of our prewarming methods and also to test whether the same set of experiment settings can work for all workloads or not. We currently use 3 workloads and perform the experiment for each one of them, separately. These workloads comprise of various vDisks and they differ in their I/O characterisitcs. The requests from individual vDisks are seen by the cache in a time-interleaved manner. In other words, the trace file we feed as an input to the cache has the requests of various vDisks sorted by their timestamps.

When characterizing the following workloads, we used a set of metrics to evaluate the nature of each vDisk. One such metric is the IOPS value, which indicates the number of requests *issued* to the cache within a second, averaged over the total duration of experiment run. Other than their IOPS, we needed a sense of their block access pattern in terms of locality of references and the disk coverage.

- Predetermined wss80
  We need a rough approximation of the locality in access pattern of a vDisk. We estimated this by letting vDisks run individually on the cache (only one pool) and observe the average hit ratio they achieved. Initially, we set the cache to a very low size (1 MB). We gradually increase this size and stop at the one where we can get 80% average hit ratio. This roughly tells us the amount of cache a particular vDisk needs to maintain an 80% average hit ratio. We define wss90 in the same manner, but consider only wss80 in characterizing the vDisks.

- wss80-to-disk Ratio
  For the synthetic workloads we generate using the custom tool mentioned in the previous section, we know exactly how much of the disk will be accessed. This combined with the fact that 1 MB of objects in cache can represent 36 MB of disk block on disk helps us come up with another metric. *We have obtained this relation between size of cache objects and disk size they represent from empirical analysis.* We calculate how much disk is represented by the wss80 and then compare this with 80% of the total disk space used for a vDisk. This comparison made indicates if caching 80% of the disk gives us 80% average hit ratio or not. For instance, if the value of this ratio is 1, we know that wss80 represents exactly 80% of the vDisk. In other words, caching wss80 worth of a vDisk's objects will an 80% hit ratio. For values less than 1, we can infer that the vDisk can reach 80% hit ratio by caching less than 80% of its accessed vDisk blocks.
  Ideally we do not want all vDisks to have the same ratio. If all vDisks can reach that hit ratio by caching 80% of its disk blocks, we can always set a cache size which guarantees this. We intend to have a good mix of vDisks in our workload set where this ratio differs for each one of them.

We now list the workload sets and some details about them.

- **Syn6GB**
  This workload is (synthethically) generated using the tool described in the previous section.

We use the digit '6' in the name to denote the total size of cache (6 GB) this workload will run on. The vDisks in this set vary in their IOPS, going from 25 to 200. The collective IOPS of the vDisks in this set is 625 which indicates we can expect nearly 625 requests that within a second of our simulation. The collective wss80 is nearly 7350 MB. We include vDisks with various wss80-to-disk ratios, ranging from 0.31 to 1.1. We have set the size of all vDisks to 100 GB as total size does not affect our empirical analysis, but only the size of vDisk actually accessed.

vDisk	vDisk Size (GB)	Avg. IOPS	Predetermined wss80 (MB)	Data repr. by wss80 (GB)	wss80-to-disk Ratio
syn520	100	100	2000	70.31	0.88
syn521	100	25	425	14.94	0.57
syn539	100	50	250	8.79	1.1
syn540	100	200	2200	77.34	0.97
syn554	100	100	175	6.15	0.31
syn570	100	150	2300	80.86	1.01

Table 4.1: Characteristics of workload Syn6GB

- **Syn4GB**
  This workload set is similar to Syn4GB. The total cache size for this workload is set to 4 GB. The vDisk IOPS vary from 25 to 200 and the collective IOPS is 675. The collective wss80 is nearly 4425 MB. The wss80-to-disk ratios range from 0.2 to 1.26. All vDisks are 100 GB in size.

vDisk	vDisk Size (GB)	Avg. IOPS	Predetermined wss80 (MB)	Data repr. by wss80 (GB)	wss80-to-disk Ratio
syn504	100	50	650	22.85	1.14
syn506	100	25	600	21.09	1.26
syn508	100	100	325	11.43	0.95
syn540	100	150	2400	84.38	1.05
syn551	100	200	150	5.27	0.2
syn557	100	150	300	10.55	0.25

Table 4.2: Characteristics of workload Syn4GB

- **RealCSE**
  This workload is recorded using blktrace on VMs running on the IIT Bombay CSE Deparment Infrastructure. The VMs on which the traces were recorded comprise of a mail server (Postfix, Dovecot, Mailman), three web servers (httpd, Nginx), an LDAP server (OpenLDAP), and a database server (MySQL). All the traces were recorded for the same duration of 12 Hours on two consecutive weekdays. The volume of block I/O events in the traces for all these VMs differ depending on the traffic that particular VM had to serve. All these traces

were recorded on device partitions formatted with the **ext4** filesystem, but all VMs did not have the same kernel version. The IOPS range from 0.45 to 20 (collective IOPS is 51.19) and the collective wss80 is close to 105 MB. Due to the latter being very low, we have set the total cache size to 32 MB for this workload.

vDisk	vDisk Size (GB)	Avg. IOPS	Predetermined wss80 (MB)	Data repr. by wss80 (GB)
mail1	100	13.93	5	0.18
mail2	500	19.99	1	0.04
mail3	1000	8.89	90	3.16
db	19	2.54	1	0.04
ldap	19	2.12	1	0.04
web1	19	2.19	5	0.18
web2	19	1.08	1	0.04
web3	16	0.45	1	0.04

Table 4.3: Characteristics of workload RealCSE

Note that we do not include the wss80-to-disk ratio here as we do not know exactly how much of the disk in in use by the individual vDisk workloads.

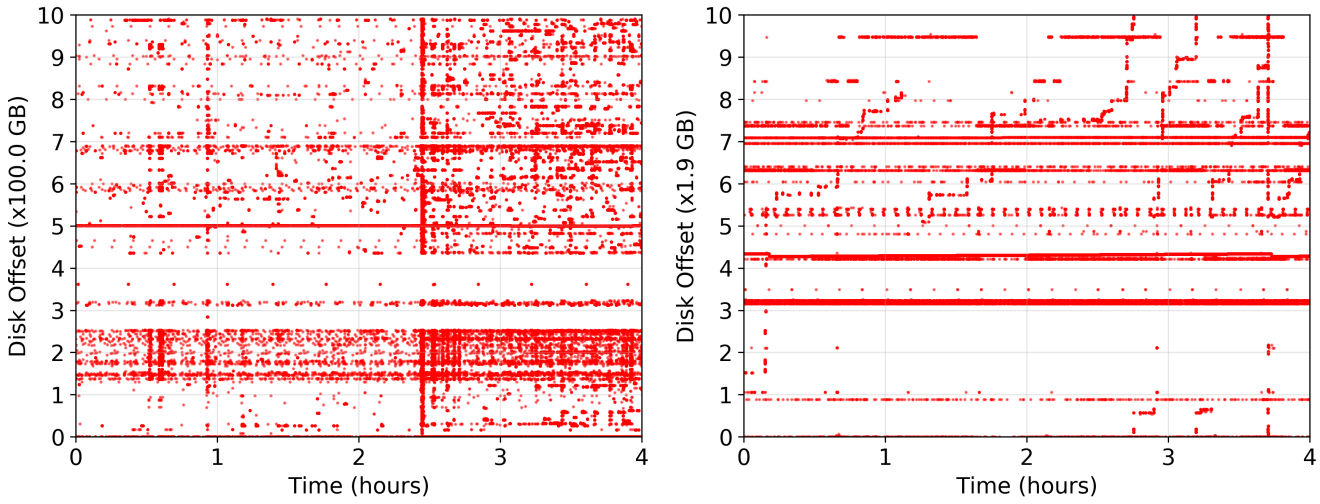

Figure 4.4: RealCSE: Disk block access patterns for mail3 (left) and web1 (right)

From the access patterns in the figures above, we can see why the wss80 values are very low for these 2 vDisks. The other vDisks have similar access patterns as well. These access exhibit very high temporal locality in addition to the visible high spatial locality. Since these workloads are running on a real filesystem, the accessed with high locality mostly correspond to system processes that perform periodic tasks such as journaling.

In the table below, we have categorized various characteristics of the 3 workloads we have described. The Average IOPS and wss80 values are relative to each other. For IOPS distribution, we know that the synthetic workloads were generated using values for sectors (blocks) sampled from uniform distribution. As for RealCSE workload, we observe from the access patterns of individual vDisks as well as their cumulative accesses that it is bursty in nature. For the synthetic workloads, spatial locality is maintained to some extent due to out choices for various sector parameters, but there is low temporal locality as the sectors accessed are sampled and they may or may not repeat within a short duration. Although, temporal locality due to consecutive accesses to same group of sectors (which can be covered by a single HM1 object) is possible. As for RealCSE workload, we have seen from both the access patterns and the static analysis that it exhibits both high spatial and high temporal localities.

Workload Set	Type	Average IOPS	IOPS Distribution	Spatial Locality	Temporal Locality	wss80
Syn6GB	Synthetic	Medium	Uniform	Medium	Low	High
Syn4GB	Synthetic	Medium	Uniform	Medium	Low	Medium
RealCSE	Real	Low	Bursty	High	High	Low

Table 4.4: High-level characteristics of workloads

4.2 Parameters & Metrics

Parameter Space

Throughout the experimentation, we make use of a large set of parameters that affect the various metrics we will measure. From this set of parameters, only some are studied in depth in the experiment, prioritized on basis of their relevance to our view of prewarming benefits. For instance, we would, as a first step, would like to observe how the choice of a heuristic for establishing an order of importance among the cache objects will affect relevance of those objects after we load them into the cache. But the total size of the hypervisor cache, or the size of the Hashmap objects is of lower relevance at this stage of our study.

We have the following parameters **fixed** throughout all the experiments:

- **HM1 object size**: 1400 B
- **HM3 object size**: 27648 B (27 KB)
- **Total cache size**: (fixed depending on the workload used)
  - Single-touch pool size: 20% of the total cache size
  - Multi-touch pool size: 80% of the total cache size

  Note: The cache size is set to the same value on the source as well as on the destination node.
- **Cache replacement policy**: LRU
- **Representation used for persistent state**: Bit Array
- **Prewarm set partition policy**:
  - Between HM1 and HM3 objects: No partitioning
  - Among the vDisks (for HM1 objects): No partitioning
- **Total duration of I/O traffic seen by cache**: 4 Hours
- **Time instant at which node failure happens**: 90th minute
- **Number of vDisks considered for failover migration**: All vDisks in the workload set
- **Number of vDisks running on target node (before failover)**: None

Note that for the current study, we do not distinguish between read and write requests during cache lookup.

We now list some fixed parameters that vary across the workloads we use for this experiment.

We start by listing some information about the workloads used for the experiments, which is fixed across the experiments. It is provided in the table below:

Workload Set	No. of vDisks	Total Requests Issued (millions)	Total Metadata Objects (thousands)	Collective Avg. IOPS of vDisks	Total Cache Size
Syn6GB	6	8.95	745.86	625	6 GiB
Syn4GB	6	9.68	504.78	675	4 GiB
RealCSE	8	0.74	39.67	51	32 MiB

Table 4.5: Workload-specific fixed parameters used in experiments

The following parameters are **tunable** and we vary at least one of these in each of our experiment runs. The values over which these parameters are varied are also mentioned below.

- **Snapshot Rate**: number of I/O requests after which we dump the cache state
  `10 K, 50 K, 100 K`
- **Prewarm Set Size Limit**: total size of metadata objects (as fraction of the total cache size) considered for prewarming the cache
  `5%, 10%, 25%, 50%, 75%, 100%`
- **Prewarm Rate**: the rate at which we load the prewarm set objects into the cache
- **Heuristic for snapshot analysis**:
  - Constrained-k-Frequent: no parameters
  - Constrained-k-Recent: no parameters
  - Constrained-k-Frerecent
    * Weight function (f(x)): weight for an object if present in the snapshot number x
      x `(Linear),` x^2 `(Quadratic)`

    We do not use the non-constrained heuristics anywhere in the experiments.

  Note: There is always an implicit upper bound on the prewarm set size. If the cache gets full while prewarming, we stop the prewarming even if there are objects remaining in the prewarm set.

Metric Space

There are several metrics that we can consider for evaluating the prewarming process. They are mainly related to either the cost or the benefit of prewarming. For instance, we can consider cache miss penalty and saving cache snapshots to disk as costs in terms of latency caused on the storage I/O path. In this study, we do not take into account these costs associated with prewarming. Instead, we focus on the hit ratio benefits as the benchmark for effectiveness of the prewarming process.

We capture the following metrics in our simulation on the destination node cache:

- **Cache Hit Ratio** (for each second)
- **per-vDisk Hit Ratio** (for each second)
- **Instantaneous Hit Ratio**: the value of cache hit ratio within 1 second after the failure
- **{5, 15, 30}-min-avg Hit Ratio**: the average cache hit ratio in a time frame of 5, 15 and 30 minutes just after the failure

We also have captured the time it takes (on destination) to reach the hit ratio before failure (on source). But in some of the experiments, we have noticed that after reaching that hit ratio at a certain point in time, the hit ratio starts to decline again. Hence, we do not consider time to reach a certain hit ratio as a metric.

4.3 Node Failover Scenario

This experiment involves simulating the scenario where a node in a failover cluster experiences failure and all its running VM are live-migrated to another node. We assume here that there are no VMs already running on the target node. Unlike the node on which the VMs were initially running, the target node will have a cold cache, i.e. the information about the objects in cache is lost. The VMs will start running from the same point at which the source node experienced failure. We expect to see a drastic drop in the cache hit ratio (as well as in the hit ratios of individual VMs) when the VMs start running on the target node, but it should stabilize as the VMs continue to run.

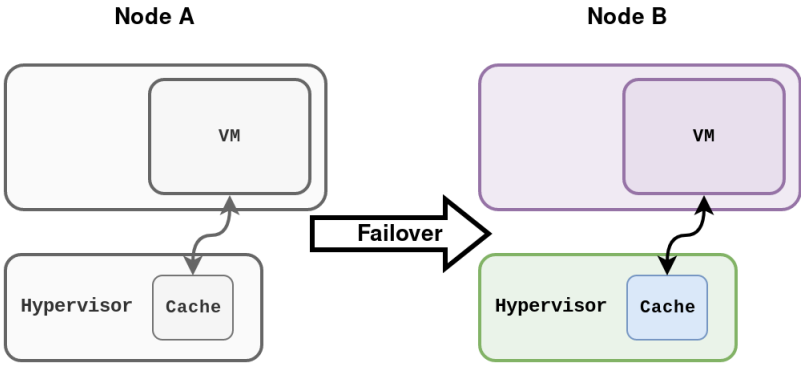

Figure 4.5: Node failover scenario

In this experiment, we answer the following question:
To what extent will prewarming the cache on the target node help in alleviating the drastic drop in the hit ratio due to the cache being cold for the incoming VMs, and in reaching a high hit ratio within a shorter time period?

We capture the hit ratio of the cache on the initial node up to the point when it fails, and then on the new node after failover. As in the previous experiment, we do this at fixed time intervals, and compare the impact on the hit ratio with and without prewarming the cache on the new node.

The procedure for performing this experiment is as follows:

```
1. Issue the block I/O requests to the cache until n seconds is served
2. Measure the hit ratio of the cache at each second
3. Dump the cache state at fixed intervals (determined by snapshot rate)
4. Analyze the snapshots using one of the heuristics to get the prewarm set
5. Reset the cache and load the objects determined in the previous step
6. Issue the block I/O request starting (n+1)th second
7. Measure the hit ratio of the cache at each second
```

Throughout this experiment, we first issue the requests for the first 90 minutes, reset the cache to simulate failover and then continue the run for another 150 minutes. At the end of this experiment, we have two sets of cache hit ratios, one where the cache on target node was cold after 5400th second (90 minutes) and the other where we prewarmed the cache with certain objects after the same point in time. Figure 4.6 shows an overview of the timeline we follow for all experiments running in this scenario.

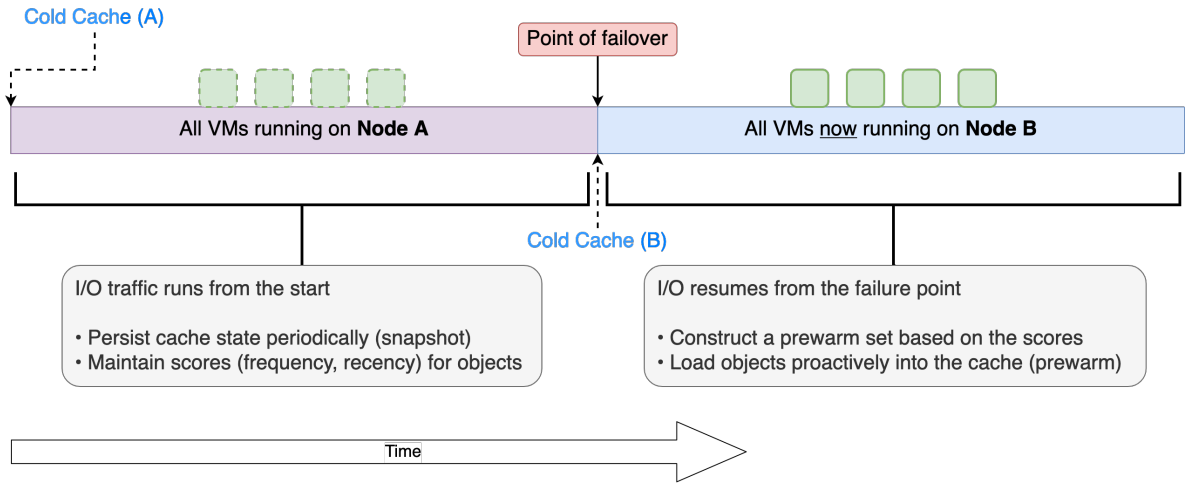

Figure 4.6: Timeline of node failover

The experiments were run for only 3 of the 6 heuristics mentioned earlier, leaving out the ones with no memory constraint (k-Frequent, k-Frerecent and k-recent) because for these heuristics, we needed to manually specify a score threshold which in some cases will result in the prewarm set not utilizing available space.

Moreover, we also skip high snapshot rates such as 1K requests and 5K requests as it involves a large size of cache state being written to the disk and will incur more overhead in terms of disk accesses and computation time on the simulation system.

In the following subsections, we have provided a subset of results for our workloads, with various combinations of parameters. These combinations may not offer the highest hit ratios, but are included for the purpose of observing the effect they have on the hit ratio. In the case of two combinations resulting in same time or hit ratio, we chose the one with lower snapshot rate (e.g. 100 K instead of 50 K).

We perform various experiments with the node failover scenario to study the effect of changing one parameter, keeping the others fixed. While we are interested in seeing the ones that give us the highest benefits, we also need to study the impact one parameter can make on our overall prewarm process. We will use the inferences gathered from these experiments to come up with a set of hypotheses which can help us determine the parameter values from the runtime state of individual vDisks.

We perform each of these experiments try to answer a set of questions, which are centred around our choice of individual (tunable) paramaters.

- What are the performance implications of choosing a prewarm set size limit for the (destination) cache? What is the cost-benefit tradeoff here, and which of those values can give us reasonable prewarm benefits?
- What are the performance implications of choosing a prewarm rate for the (destination) cache? What is the cost-benefit tradeoff here, and which of those values can give us reasonable prewarm benefits?
- What are the performance implications of choosing a certain snapshot rate for the (source) cache? What is the cost-benefit tradeoff in the rate we choose? Does one rate outperform the others in most cases?
- How does the choice of a certain heuristic affect the selection of important objects to load (and in turn affect the prewarm benefits)? Is there a heuristic we can use that gives us higher hit ratios (relative to other heuristics), regardless of our choice for other parameters?
- Given the implications of these 4 parameters, which combination should we recommended to use, given some information about the cache and individual vDisks at runtime (before failure)?

The first two questions are concerned with parameters that affect the cache on destination node, while the next two affect the cache on source.

We will look at each of these questions for all the workloads. At the end of these experiments, we will try to come up with a set of ideas for determining the values for these parameters at runtime (on source node, before failure).

4.3.1 Workload: Syn6GB

We first mention the hit ratio values we obtained on a cold cache for this workload. These values will remain constant across all experiments using this workload. The Hit Ratio before failure was obtained on the source cache, while the others were recorded on the destination cache.

HR before failure	Instantaneous HR	5-min avg. HR	15-min avg. HR	30-min avg. HR
0.8	0.36	0.43	0.52	0.6

Table 4.6: Syn6GB: Values for metrics recorded in the cold cache

Implications of prewarm set size limit on the destination

Prewarm set size limit constrains the cache memory available on the destination cache for prewarming. A large prewarm set size will be able to accommodate more objects. But at the same time, the prewarming process will need more time to actually fetch those objects from the metadata store and then load them in the memory. Moreover, if the cache is not empty (and the destination node has a few running VMs), a large prewarm set will cause a lot of evictions which may affect the VMs' performance. Ideally, we would like to keep this limit at a low value which can guarantees a reasonable improvement in the hit ratio.

Snapshot Rate (requests)	Prewarm Set Size Limit (% 6 GiB)	Prewarm Rate	Heuristic	Instant. HR	5-min avg. HR	15-min avg. HR	30-min avg. HR
100 K	5%	100 MBps	k-Recent	0.41	0.64	0.69	0.74
100 K	10%	100 MBps	k-Recent	0.41	0.65	0.7	0.75
100 K	25%	100 MBps	k-Recent	0.41	0.68	0.73	0.77
100 K	50%	100 MBps	k-Recent	0.41	0.72	0.77	0.8
100 K	75%	100 MBps	k-Recent	0.41	0.74	0.75	0.77
100 K	100%	100 MBps	k-Recent	0.41	0.72	0.73	0.76

Table 4.7: Syn6GB: Effect of various Prewarm Set Size Limits on the cache

We observe from the table above that increasing the prewarm set size limit does help the cache in terms of hit ratio, but only up to a certain extent. Even prewarming only 5% of the destination cache (~307 MB) results in an average hit ratio of 0.74 for the first 30 minutes, a significant improvement over 0.6 with cold cache. Furthermore, we observe that the hit ratio benefits start to decline after 50% limit. This happens due to the fact that loading 75% of the cache occupies most of multi-touch pool (which is 80% of the cache), and loading 100% will occupy it completely. When the VM traffic is running, it will load some objects in the cache (which were not in prewarm set), these objects will get to the multi-touch part and cause evictions. The prewarm objects that get evicted may have been needed in the near future, but their absence due to eviction causes an extra misses. Another reason for this behavior is that the heuristic used (k-Recent) will load objects that are temporally relevant (relative to the point of failure). Most of the objects in the 50% limit set are being accessed within first 30 minutes, but in 75% or 100% limit sets, they will be evicted before they are accessed.

We can say here that prewarming more than 50% of the cache does not help the cache to an extent that we would expect. In the scenario where some VMs are already running on the destination (before failover happens), prewarming with higher limits may cause evictions of their objects as well, affecting their performance. Moreover, increasing the prewarm rate helps the hit ratio for first few minutes (especially 5-min-avg), but as the traffic continues to run these differences start to fade and the hit ratios for them eventually converge to the same value.

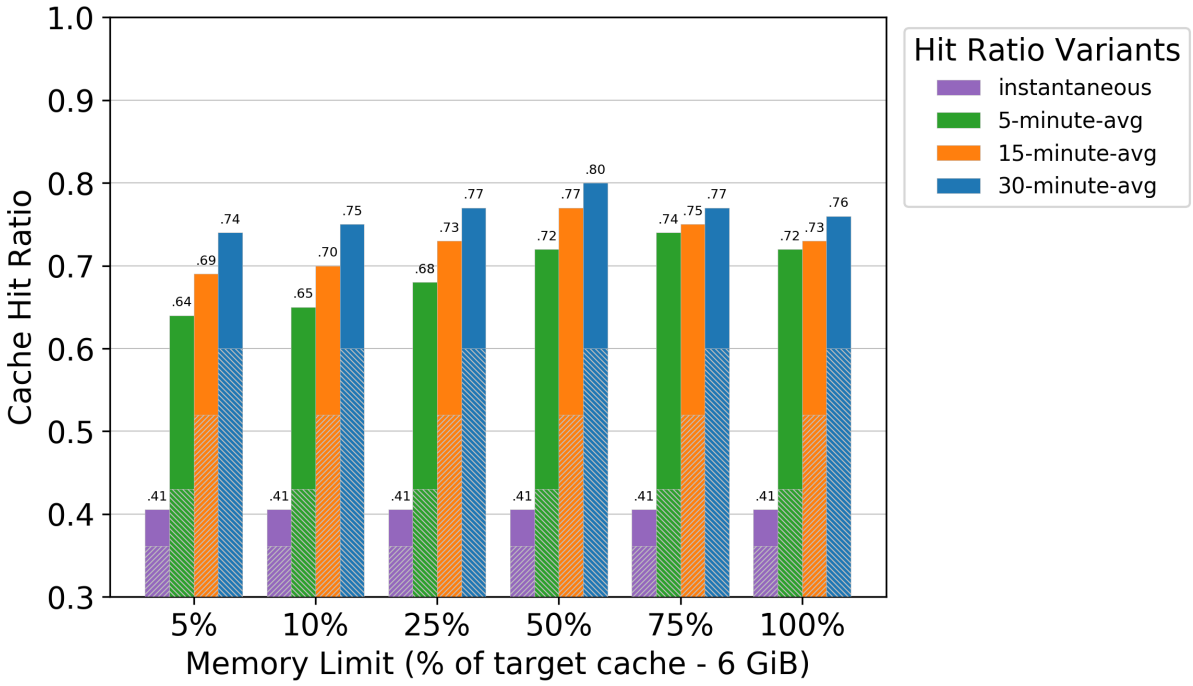

Figure 4.7: Syn6GB: Comparison of various Prewarm Set Size Limits for Hit Ratio metrics

The figure above compares the hit ratio metrics for various values of prewarm set limits, keeping snapshot rate fixed at 100K, heuristic as k-Recent and prewarm rate at 100 MBps. The lower part of each bar with hatch pattern shows the corresponding hit ratio value for cold cache.

Implications of prewarm rate on the destination

Prewarm rate affects how fast we can load the objects in the prewarm set into the destination cache. A higher rate might give a higher chance to the temporally relevant objects to be loaded into the cache. If we can load all the objects (with an infinite rate) within a second, the VMs would not experience a miss only because the required object was in the prewarm set, but could not be loaded as fast. But at the same time, a higher rate needs more bandwidth on the storage I/O path (including network resources), potentially disturbing running VM traffic as well as that of the whole cluster. Moreover, very high rates are practically infeasible due to the physical limitations of hardware resources. Ideally, we want this rate to be as low as possible, while giving us the maximum benefit in terms of hit ratio.

Snapshot Rate (requests)	Prewarm Set Size Limit (% 6 GiB)	Prewarm Rate	Heuristic	Instant. HR	5-min avg. HR	15-min avg. HR	30-min avg. HR
100 K	50%	50 MBps	k-Recent	0.38	0.7	0.76	0.79
100 K	50%	100 MBps	k-Recent	0.41	0.72	0.77	0.8
100 K	50%	500 MBps	k-Recent	0.61	0.74	0.78	0.8
100 K	50%	∞	k-Recent	0.73	0.75	0.78	0.8

Table 4.8: Syn6GB: Effect of various Prewarm Rates on the cache

We observe here that increasing the prewarm rate helps the hit ratio, but only for the first few minutes. In the table above, we can see that the 30-minute values are almost the same for all prewarm rates. Even if we load more objects into the cache, only a fixed number of those will be accessed in the first few minutes. Moreover, if the rate is low, it takes more time to prewarm the cache. For instance, to prewarm 3 GB (50%) at 50 MBps, it would take around 60 seconds. The same would take 30 seconds when prewarmed at 100 MBps.

The figure below compares the hit ratio metrics for various values of prewarm rates, keeping snapshot rate fixed at 100K, heuristic as k-Recent and prewarm set limit at 50%. The lower part of each bar with hatch pattern shows the corresponding hit ratio value for cold cache.

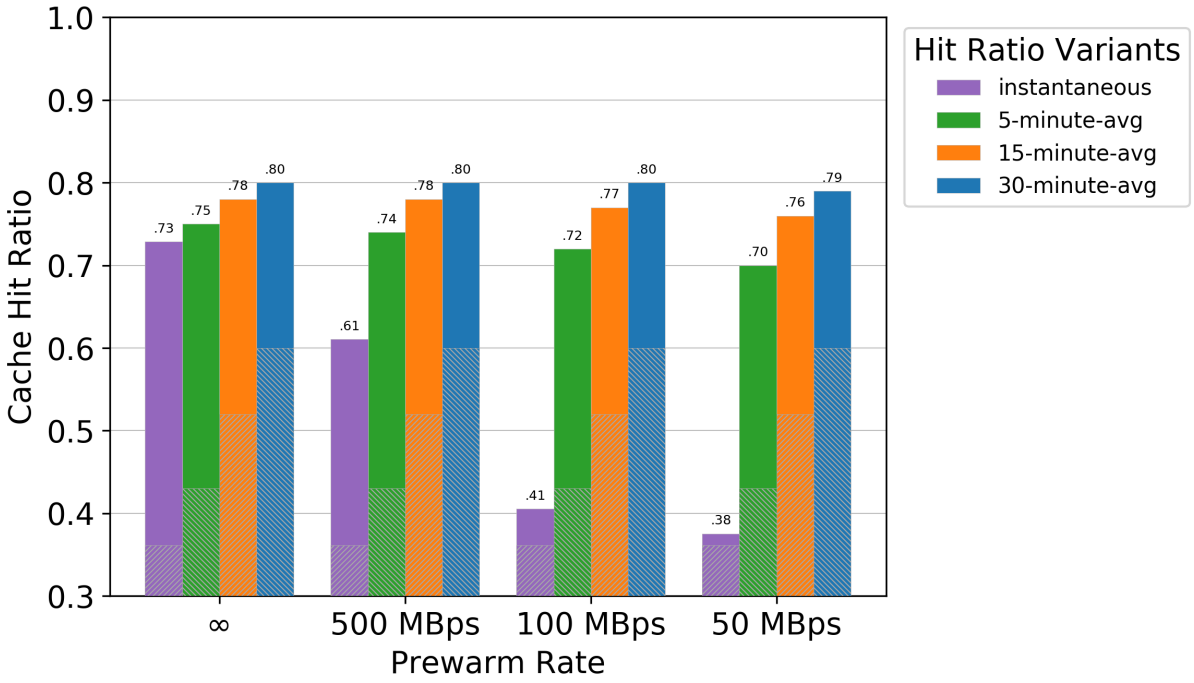

Figure 4.8: Syn6GB: Comparison of various Prewarm Rates for Hit Ratio metrics

Implications of snapshot rate used for saving the cache state

Snapshot rate affects the frequency at which we capture our in-memory cache state and save it to the disk for analysis in the near future. A high snapshot rate is able to capture the cache state at a finer granularity, i.e., if there are objects that are frequently going in and out of the cache, these will only be noticed only if the cache is snapshotted at high rates. When using snapshots for analysis, we will have a larger number of these snapshots to process, which can possibly give us a better selection of objects. But at the same time, high snapshot rate creates issues as well. High rates will result in more snapshots being written to the disk (and more frequently as well), which can affect the storage I/O path severely and result in degraded storage performance of the running VMs. Moreover, having more snapshot makes the heuristic-based analysis slower. We can perform an online analysis, where snapshots are analyzed as they are created and the prewarm set is built incrementally. But, this type of analysis will also be slower for high rates. Ideally, we want this rate to be as low as possible, while resulting in a good selection of objects for the prewarm set.

Snapshot Rate (requests)	Prewarm Set Size Limit (% 6 GiB)	Prewarm Rate	Heuristic	Instant. HR	5-min avg. HR	15-min avg. HR	30-min avg. HR
10 K	75%	100 MBps	k-Frequent	0.38	0.72	0.78	0.8
50 K	75%	100 MBps	k-Frequent	0.38	0.73	0.78	0.8
100 K	75%	100 MBps	k-Frequent	0.39	0.73	0.78	0.8

Table 4.9: Syn6GB: Effect of various Snapshot Rates on the cache

In the table above, we can observe that choice of a snapshot rate does not have a significant impact on the hit ratio. There is only a minute difference for the same with 10K rate and with 100K rate. Due to the same reason, we choose a lower rate (100K) in all our results.

Implications of heuristic used for snapshot analysis

Heuristic used for analysis affects the selection of objects for prewarm set, as well as the order of importance of objects within the set. Each heuristic has its own advantages. For instance, if we choose objects based on their recency, the storage traffic within the next few minutes may experience high hit ratio due to temporal locality of their accesses. A frequency-based selection may help in the longer run, as we are selecting mostly used objects regardless of their access time. A combination of these two (frerecency) is expected to give a benefit in the immediate time period, as well as in the an extended time period. While we want the prewarming to reduce the drastic drop in the immediate hit ratio after failure, it will be good to do it in a way which helps the cache in the longer run as well.

Snapshot Rate (requests)	Prewarm Set Size Limit (% 6 GiB)	Prewarm Rate	Heuristic	Instant. HR	5-min avg. HR	15-min avg. HR	30-min avg. HR
100 K	10%	50 MBps	k-Recent	0.38	0.64	0.7	0.75
100 K	10%	50 MBps	k-Frequent	0.38	0.51	0.59	0.65
100 K	10%	50 MBps	k-Frerecent (LNR)	0.38	0.51	0.58	0.65
100 K	10%	50 MBps	k-Frerecent (QDR)	0.38	0.51	0.58	0.65

Table 4.10: Syn6GB: Effect of various Heuristics on the cache (low prewarm set limit)

From the table above, we can infer that for the given set of other parameter values, k-Recent performs better. One of the reason for the same is that we are interested in only the gains in hit ratio for a short duration after prewarming. k-Frerecent and k-Frequent give results similar to each other as frequency heuristic is not able to select a good set of objects, as compared to the recency heuristic. Moreover, we have observed that for high prewarm size limits (75% and 100%), the k-Frequent heuristic performs better than the k-Recent one. This can observed in the table below.

Snapshot Rate (requests)	Prewarm Set Size Limit (% 6 GiB)	Prewarm Rate	Heuristic	Instant. HR	5-min avg. HR	15-min avg. HR	30-min avg. HR
100 K	75%	50 MBps	k-Recent	0.38	0.7	0.73	0.75
100 K	75%	50 MBps	k-Frequent	0.38	0.67	0.75	0.78
100 K	75%	50 MBps	k-Frerecent (LNR)	0.38	0.67	0.75	0.78
100 K	75%	50 MBps	k-Frerecent (QDR)	0.38	0.66	0.75	0.78

Table 4.11: Syn6GB: Effect of various Heuristics on the cache (high prewarm set limit)

The main reason for this is that the eviction or prewarm objects caused due to excessive prewarming here will evict the most recent or the most frequent objects first. But, eviction of most recent objects is worse than that of most frequent objects because the recent objects are meant for immediate VM traffic which will now cause many misses for them. While eviction of the most frequent objects is a problem too, from the results above, we can infer that the number of such objects evicted are fewer than those for k-Reecnt. In the complete set of results for this experiment (with same values for all other parameters), we observed that the evictions for k-Recent start to happen 20 seconds earlier than that for k-Frequent.

Per-vDisk analysis

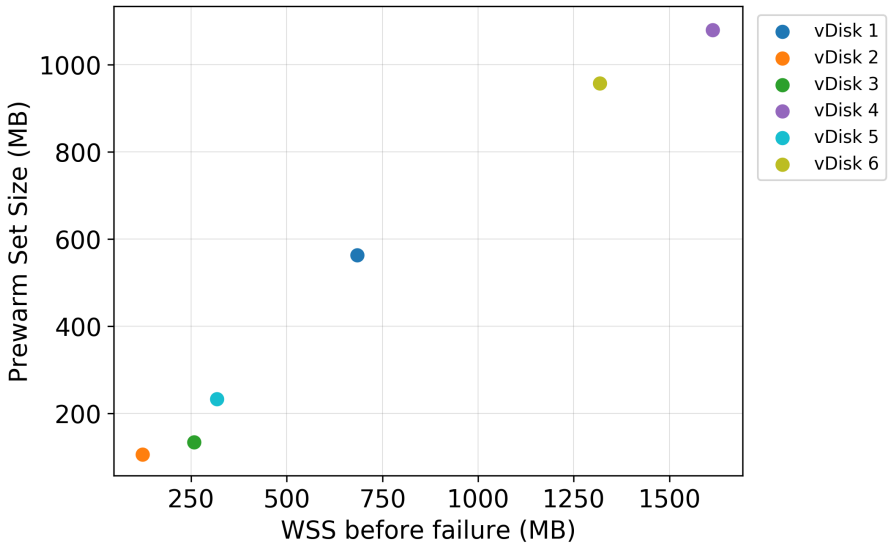

Figure 4.9: Syn6GB: Correlation between WSS and share in Prewarm Set for vDisks

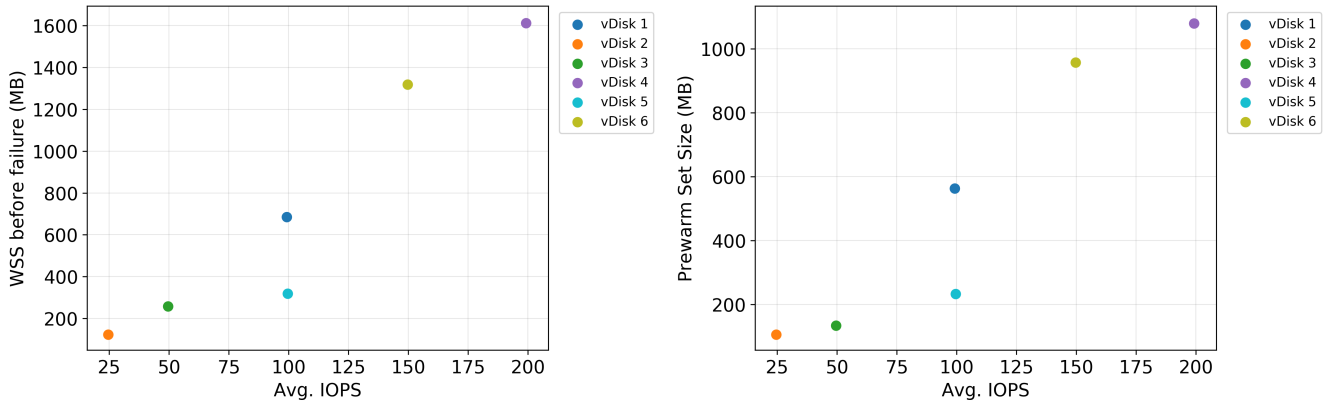

Figure 4.10: Syn6GB: Correlation between avg. IOPS and WSS (left), avg. IOPS and Prewarm Set share (right) for vDisks

In the figures above, we have plotted the correlation between the IOPS of vDisks, and their WSS and Prewarm Set Share obtained during the experiment. These numbers are obtained from the experiment run with 50% prewarm set size limit, k-Recent heuristic for snapshot analysis, and a prewarm rate of 100 MBps. The WSS was estimated using a moving 10-minute window with a stride of 5 minutes.

From the graph on top, we can infer that for all vDisks in this workload, prewarm set share is proportional to their WSS before failure. The graphs at the bottom compare how the IOPS of each vDisk affect the WSS and set share of each vDisk. It can be seen that vDisks with higher IOPS value are having higher WSS and a larger share in the prewarm set, as expected. Since these vDisks have higher IOPS, they will use the cache at a higher frequency and they will have more objects cached.

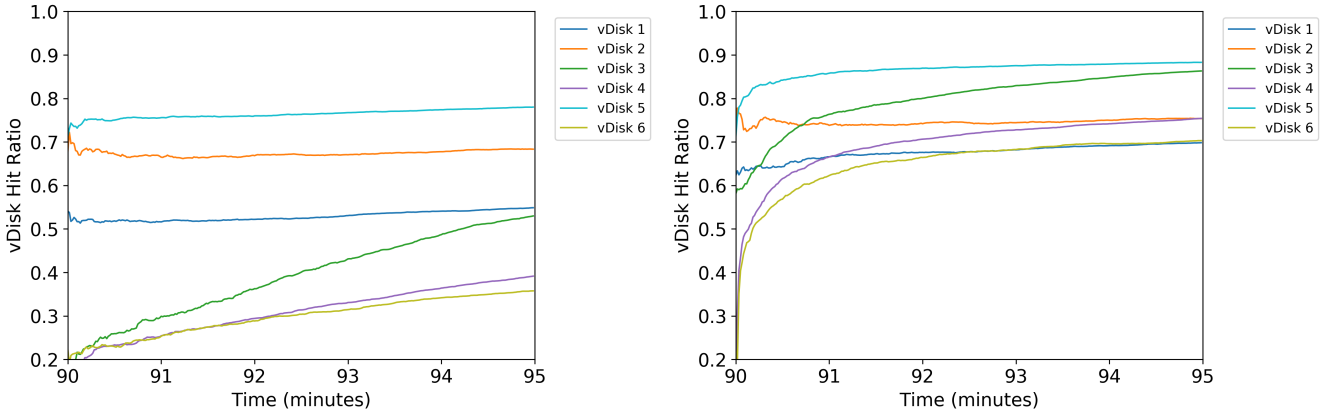

Figure 4.11: Syn6GB: Impact of prewarming on the vDisk hit ratio for 5 minutes after failure: cold cache (left) and prewarmed cache (right)

In the figures above, we have plotted the individual vDisk hit ratios, comparing the cold cache with prewarmed cache. We can infer here that prewarming does help them individually, even if it is by a small margin. There is a large difference in the hit ratio improvements for vDisks 2 (orange) and 6 (olive). This can be attributed to the differences in their IOPS as well as in their shares in the prewarm sets.

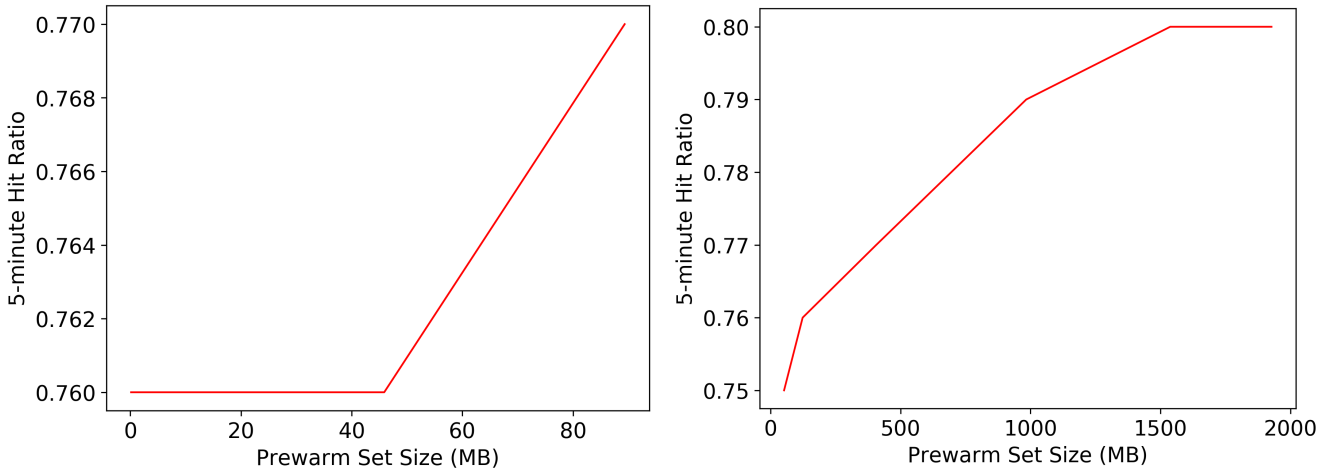

Figure 4.12: Syn6GB: 5-minute average Hit Ratios for vDisks 2 (left) and 6 (right) against their prewarm set sizes

From the 2 figures above, we can infer the extent to which 5-minute hit ratios are affected by the prewarm set sizes for those vDisks. For vDisk 2, there was an increase in the 5-min hit ratio by 0.01 when the prewarm size for it is increased by nearly 80 MB. For vDisk 6, the same amount of increment in the hit ratio can be seen with nearly 50 MB. Moreover, when the latter was able to get a good prewarm set share, we saw an increment of 0.05 in the hit ratio. We could possibly see a better increment for vDisk 2 if it were given a better share in the prewarm set. This is not possible in the current setup as there is no scheme for explicit partitioning of the set for each vDisk.

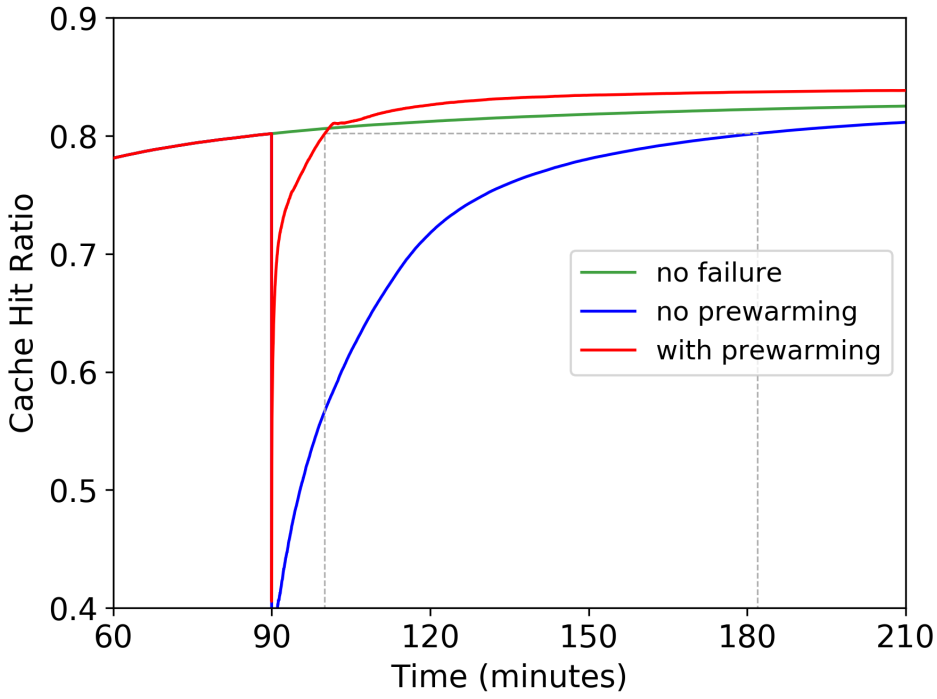

Figure 4.13: Syn6GB: Impact of prewarming on the (cumulative) cache hit ratio after failure

In the figure above, we can see the overall impact prewarming has on the cache. The parameter combinations used for prewarming here are the same as in the previous figure. One more observation made here is that the time it takes to reach the hit ratio value that was just before failure is cut by more than an hour.

4.3.2 Workload: Syn4GB

Mentioned below are the hit ratio values we obtained on a cold cache for this workload.

HR before failure	Instantaneous HR	5-min avg. HR	15-min avg. HR	30-min avg. HR
0.83	0.51	0.56	0.63	0.69

Table 4.12: Syn4GB: Values for metrics recorded in the cold cache

Implications of prewarm set size limit on the destination

Snapshot Rate (requests)	Prewarm Set Size Limit (% 4 GiB)	Prewarm Rate	Heuristic	Instant. HR	5-min avg. HR	15-min avg. HR	30-min avg. HR
100 K	5%	500 MBps	k-Recent	0.68	0.71	0.76	0.79
100 K	10%	500 MBps	k-Recent	0.69	0.72	0.77	0.8
100 K	25%	500 MBps	k-Recent	0.69	0.74	0.79	0.81
100 K	50%	500 MBps	k-Recent	0.69	0.78	0.81	0.83
100 K	75%	500 MBps	k-Recent	0.69	0.8	0.8	0.81
100 K	100%	500 MBps	k-Recent	0.69	0.76	0.78	0.8

Table 4.13: Syn4GB: Effect of various Prewarm Set Size Limits on the cache

We make the same observation here as we did in the corresponding experiment for the previous workload (Syn6GB). Increasing the prewarm set limit helps the 30-minute average hit ratio, but up to a certain extent. For 75% and 100% values, we see again that the prewarmed objects are more prone to eviction from multi-touch pool due to the running VM traffic. Moreover, loading even only 5% of the cache gives us a significant benefit in 30-minute hit ratio of 0.74 over the cold cache value of 0.6.

The figure below compares the hit ratio metrics for various values of prewarm set limits, keeping snapshot rate fixed at 100K, heuristic as k-Recent and prewarm rate at 500 MBps. The lower part of each bar with hatch pattern shows the corresponding hit ratio value for cold cache.

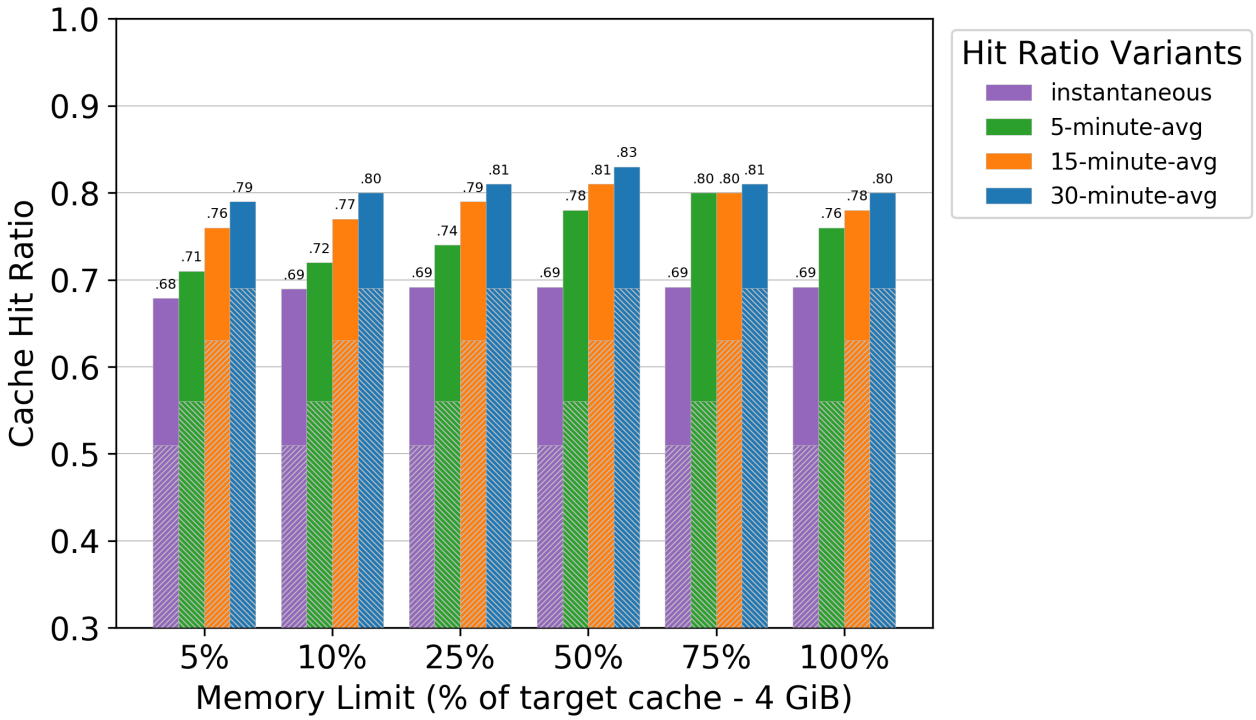

Figure 4.14: Syn4GB: Comparison of various Prewarm Set Size Limits for Hit Ratio metrics

Implications of prewarm rate on the destination

Snapshot Rate (requests)	Prewarm Set Size Limit (% 4 GiB)	Prewarm Rate	Heuristic	Instant. HR	5-min avg. HR	15-min avg. HR	30-min avg. HR
100 K	75%	50 MBps	k-Frequent	0.54	0.75	0.8	0.82
100 K	75%	100 MBps	k-Frequent	0.55	0.79	0.82	0.83
100 K	75%	500 MBps	k-Frequent	0.61	0.82	0.83	0.84
100 K	75%	∞	k-Frequent	0.82	0.83	0.84	0.84

Table 4.14: Syn4GB: Effect of various Prewarm Rates on the cache

As in the same experiment for the previous workload (Syn6GB), we see here that increasing the prewarm rate helps in the immediate hit ratios (instantaneous, 5-min-avg), but for an extended time period, the hit ratio values eventually converge. Loading the objects at only 50 MBps or 100 MBps has a great improvement in the 5-minute value which is 0.43 for cold cache.

The figure below compares the Hit Ratio metrics for various values of prewarm rates, keeping the snapshot rate fixed at 100K, heuristic as k-Recent and prewarm set limit at 75%. The lower part of each bar with hatch pattern shows the corresponding hit ratio value for cold cache.

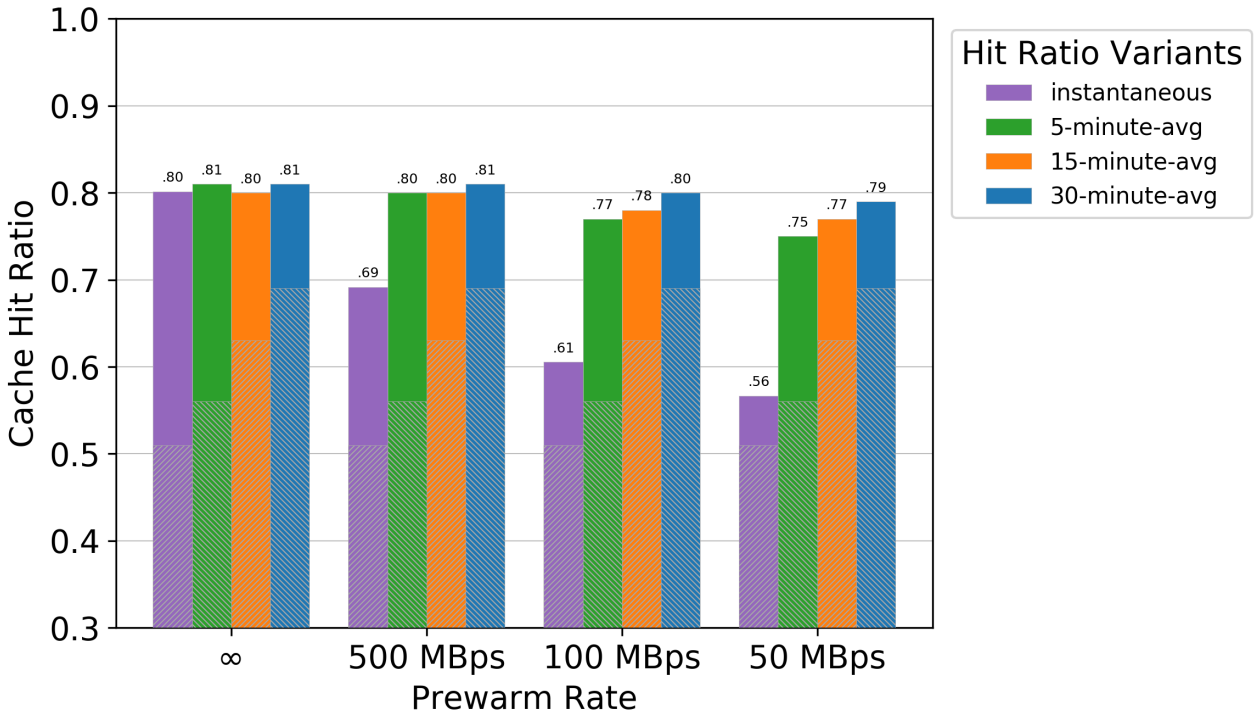

Figure 4.15: Syn4GB: Comparison of various Prewarm Rates for Hit Ratio metrics

Implications of snapshot rate used for saving the cache state

Snapshot Rate (requests)	Prewarm Set Size Limit (% 6 GiB)	Prewarm Rate	Heuristic	Instant. HR	5-min avg. HR	15-min avg. HR	30-min avg. HR
10 K	75%	100 MBps	k-Frequent	0.54	0.78	0.81	0.83
50 K	75%	100 MBps	k-Frequent	0.54	0.78	0.81	0.83
100 K	75%	100 MBps	k-Frequent	0.55	0.79	0.82	0.83

Table 4.15: Syn4GB: Effect of various Snapshot Rates on the cache

The snapshot rate does not have a significant effect for this workload as well. Since we see a very small difference in the hit ratios, we proceed with using 100K as the rate for other experiments using this workload.

Implications of heuristic used for snapshot analysis

Snapshot Rate (requests)	Prewarm Set Size Limit (% 6 GiB)	Prewarm Rate	Heuristic	Instant. HR	5-min avg. HR	15-min avg. HR	30-min avg. HR
100 K	75%	100 MBps	k-Recent	0.61	0.77	0.78	0.8
100 K	75%	100 MBps	k-Frequent	0.55	0.79	0.82	0.83
100 K	75%	100 MBps	k-Frerecent (LNR)	0.55	0.78	0.81	0.83
100 K	75%	100 MBps	k-Frerecent (QDR)	0.55	0.78	0.81	0.82

Table 4.16: Syn4GB: Effect of various Heuristics on the cache

The effect of the choice of heuristic on the hit ratios is also similar to the one we saw for the previous workload (Syn6GB). With higher prewarm size limits, k-Frequent performs better, while k-Recent performs better with lower ones.

Per-vDisk analysis

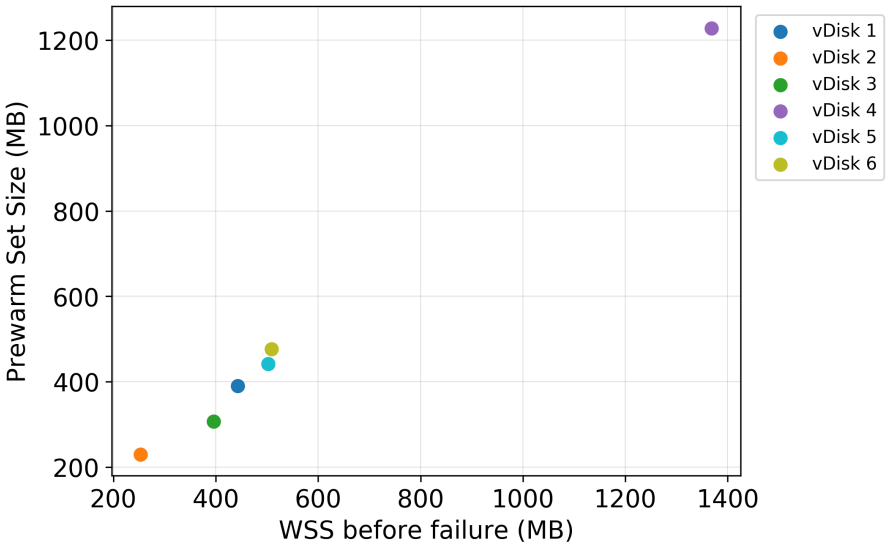

Figure 4.16: Syn4GB: Correlation between WSS and share in Prewarm Set for vDisks

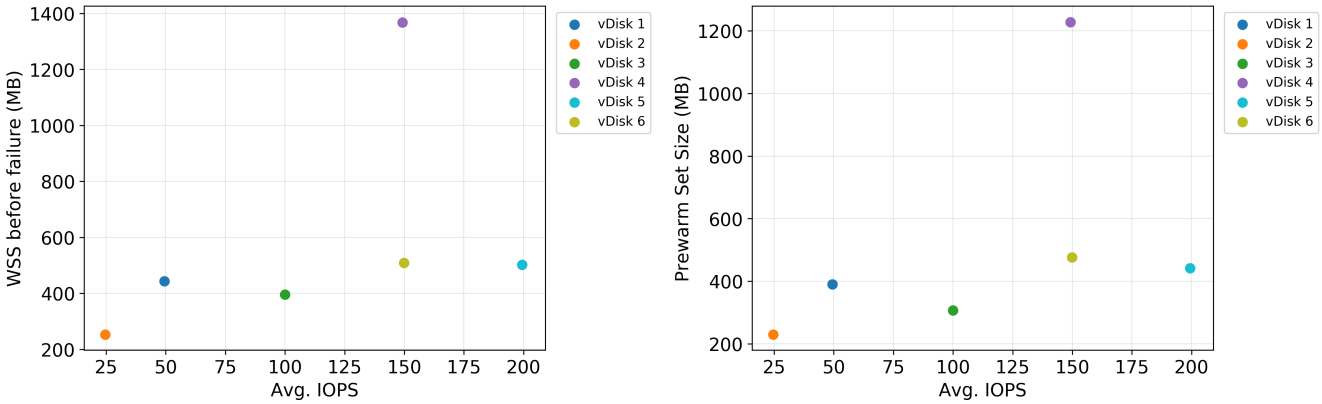

Figure 4.17: Syn4GB: Correlation between avg. IOPS and WSS (left), avg. IOPS and Prewarm Set share (right) for vDisks

We have plotted the correlation between the IOPS of vDisks, and the WSS and Prewarm Set Share obtained during the experiment. These numbers are obtained from the experiment run with 75% prewarm set size limit and k-Recent heuristic for snapshot analysis. The WSS was estimated using a moving 10-minute window with a stride of 5 minutes, same as before.

From the graph on top, we can infer that for all vDisks, prewarm set share is proportional to their WSS before failure. The graphs at the bottom compare how the IOPS of each vDisk affect the WSS and set share of each vDisk. Here, we see that all vDisks except one are having similar WSS values and Prewarm Set shares. The one vDisk with extreme values for both (vDisk 4) is the one which had the highest (predetermined) wss80 and a high IOPS value, as described in the table 4.2.

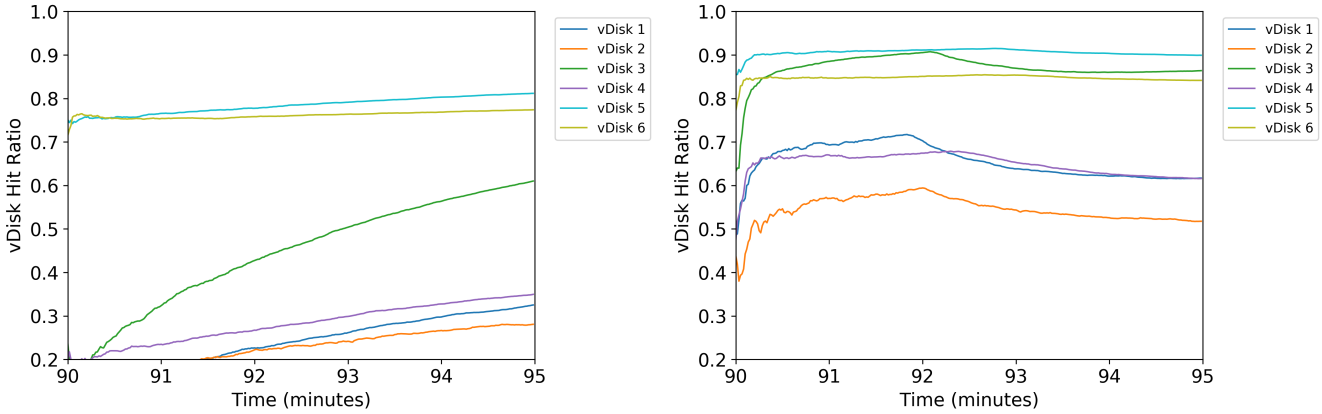

Figure 4.18: Syn4GB: Impact of prewarming on the vDisk hit ratio for 5 minutes after failure: cold cache (left) and prewarmed cache (right)

In the figures above, we can see the hit ratios of individual vDisks. Prewarming does help the vDisks in achieving high hit ratios in the first 5 minutes. For vDisks 5 and 6, there is little improvement in the same. For the others, the improvement is significant (especially for vDisks 1, 2 and 4).

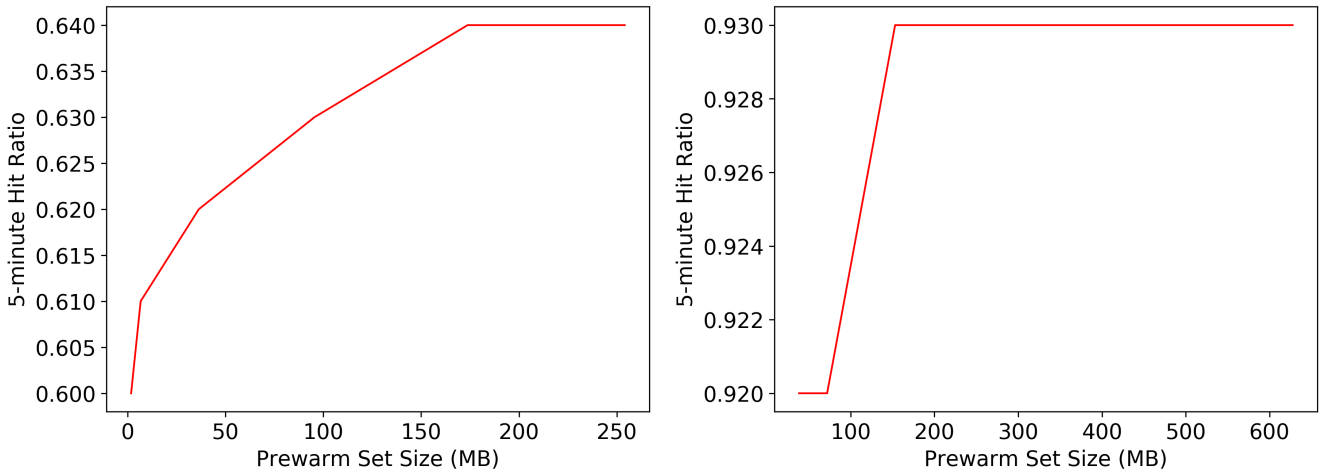

Figure 4.19: Syn6GB: 5-minute average Hit Ratios for vDisks 2 (left) and 5 (right) against their prewarm set sizes

From the 2 figures above, we can infer the extent to which 5-minute hit ratios are affected by the prewarm set sizes for those vDisks. For vDisk 2, there was an increase in the 5-min hit ratio by 0.02 when the prewarm size for it is increased by nearly 50 MB. For vDisk 5, there is not improvement observed with 50 MB. Moreover, when the latter was able to get prewarm set share of nearly 150 MB, we saw an increment of 0.01 in the hit ratio; no increment with further increase in prewarm set share. With 150 MB of prewarm set share, vDisk 2 could get an increment of nearly 0.03. With a partioning in place for each vDisk, we should ideally limit the share for vDisk at around 150 MB.

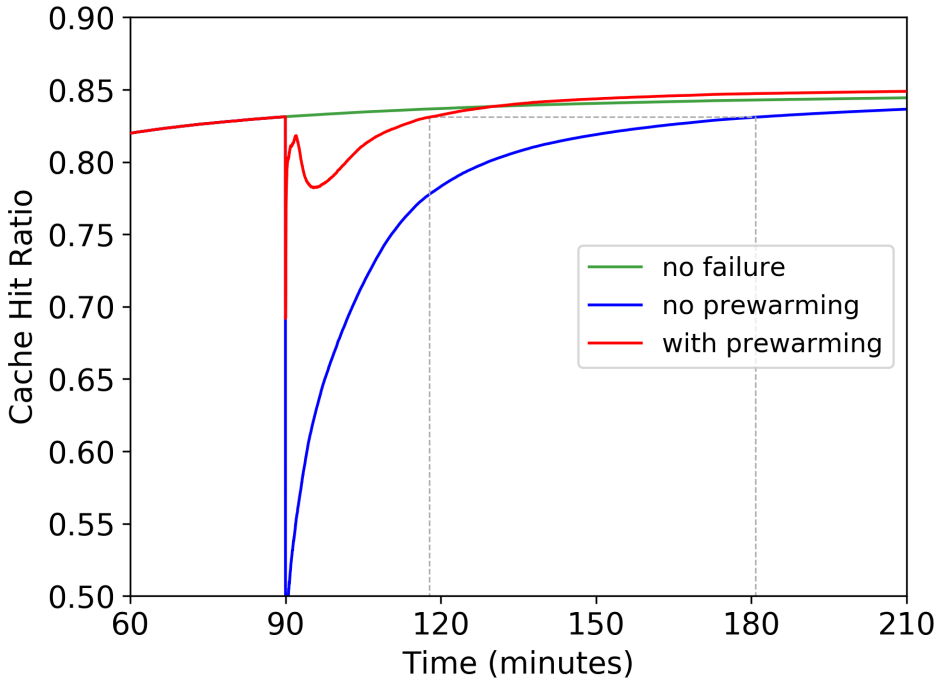

Figure 4.20: Syn4GB: Impact of prewarming on the (cumulative) cache hit ratio after failure

In the figure above, we can see the overall impact prewarming has on the cache. The parameter combinations used for prewarming here are the same as in the previous figure. For this workload as well, we can see that the time it takes to reach the hit ratio value that was just before failure is cut by more than an hour.

4.3.3 Workload: RealCSE

Mentioned below are the hit ratio values we obtained on a cold cache for this workload.

HR before failure	Instantaneous HR	5-min avg. HR	15-min avg. HR	30-min avg. HR
0.95	0.33	0.89	0.91	0.92

Table 4.17: RealCSE: Values for metrics recorded in the cold cache

Implications of prewarm set size limit on the destination

Snapshot Rate (requests)	Prewarm Set Size Limit (% 32 MiB)	Prewarm Rate	Heuristic	Instant. HR	5-min avg. HR	15-min avg. HR	30-min avg. HR
100 K	5%	50 MBps	k-Recent	0.5	0.92	0.93	0.94
100 K	10%	50 MBps	k-Recent	0.67	0.93	0.94	0.94
100 K	25%	50 MBps	k-Recent	0.83	0.94	0.95	0.95
100 K	50%	50 MBps	k-Recent	1	0.95	0.95	0.95
100 K	75%	50 MBps	k-Recent	1	0.95	0.95	0.95
100 K	100%	50 MBps	k-Recent	1	0.95	0.94	0.95

Table 4.18: RealCSE: Effect of various Prewarm Set Size Limits on the cache

We also observe that most inferences from the results previous workloads (Syn6GB and Syn4GB) apply here as well. Since the cold cache hit ratios are quite high already, prewarming does only little help in increasing the hit ratio for the first few minutes. We can infer from the table above that increasing the prewarm set limit to 50% or more gives a little benefit over the cold cache values.

The figure below compares the hit ratio metrics for various values of prewarm set limits, keeping snapshot rate fixed at 100K, heuristic as k-Recent and prewarm rate at 50 MBps. The lower part of each bar with hatch pattern shows the corresponding hit ratio value for cold cache.

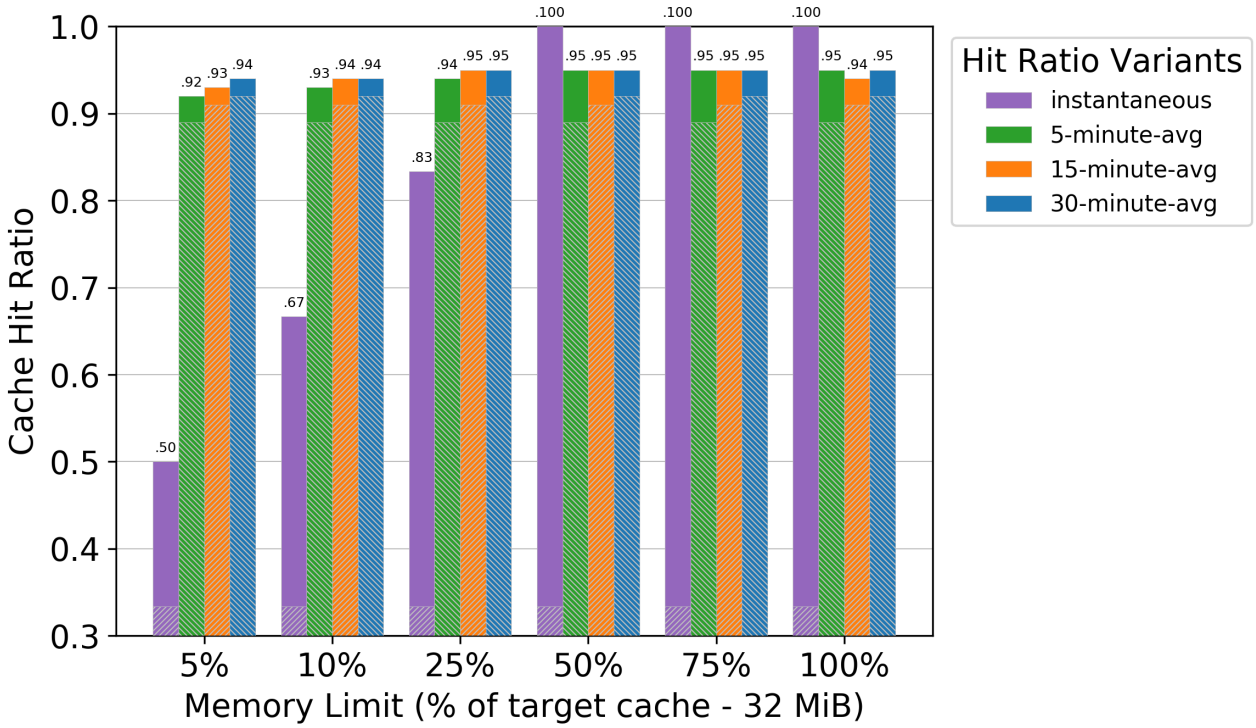

Figure 4.21: RealCSE: Comparison of various Prewarm Set Size Limits for Hit Ratio metrics

Implications of prewarm rate on the destination

Snapshot Rate (requests)	Prewarm Set Size Limit (% 32 MiB)	Prewarm Rate	Heuristic	Instant. HR	5-min avg. HR	15-min avg. HR	30-min avg. HR
100 K	25%	50 MBps	k-Recent	0.83	0.94	0.95	0.95
100 K	25%	100 MBps	k-Recent	0.83	0.94	0.95	0.95
100 K	25%	500 MBps	k-Recent	0.83	0.94	0.95	0.95
100 K	25%	∞	k-Recent	0.83	0.94	0.95	0.95

Table 4.19: RealCSE: Effect of various Prewarm Rates on the cache

Unlike the results for this experiment with the previous workloads, the results do not change here at all with the prewarm rates. The reason is that since the destination cache size is also 32 MB, we can load everything in the prewarm set at only 50 MBps. We do not require throttling of prewarming for this workload.

The figure below compares the Hit Ratio metrics for various values of prewarm rates, keeping the snapshot rate fixed at 100K, heuristic as k-Recent and prewarm set limit at 25%. The lower part of each bar with hatch pattern shows the corresponding hit ratio value for cold cache.

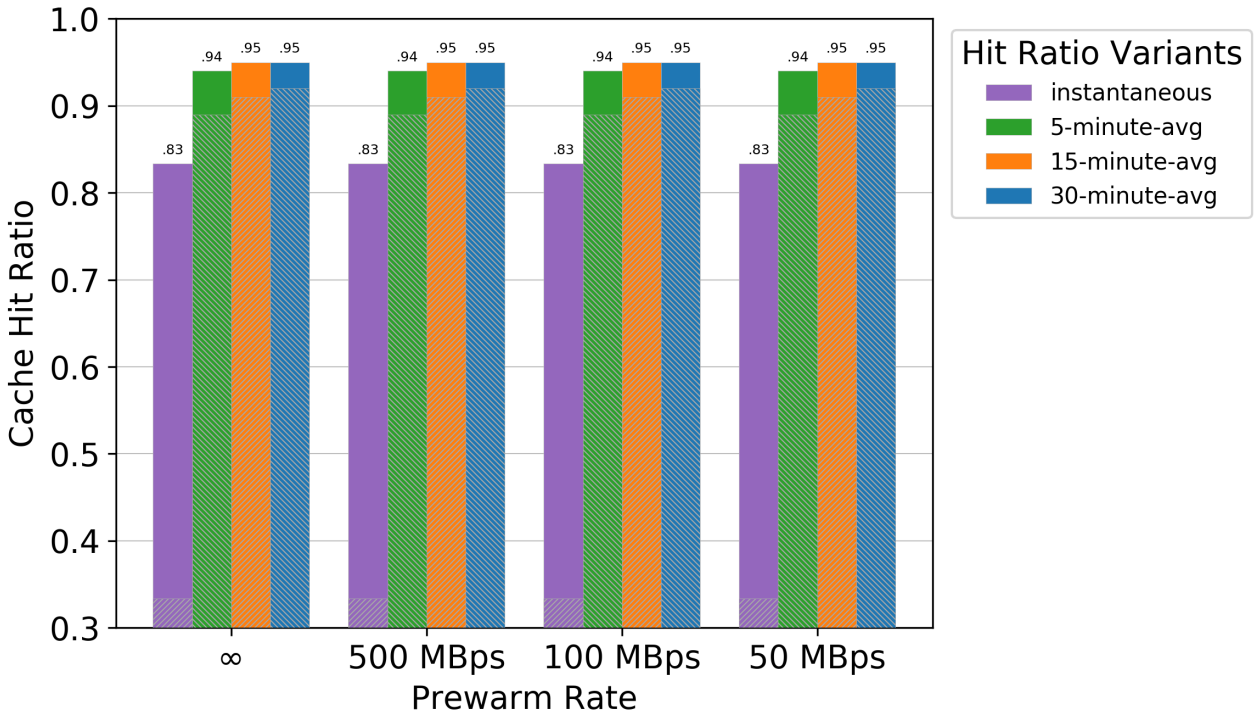

Figure 4.22: RealCSE: Comparison of various Prewarm Rates for Hit Ratio metrics

Implications of snapshot rate used for saving the cache state

Snapshot Rate (requests)	Prewarm Set Size Limit (% 6 GiB)	Prewarm Rate	Heuristic	Instant. HR	5-min avg. HR	15-min avg. HR	30-min avg. HR
10 K	25%	50 MBps	k-Frequent	0.83	0.94	0.94	0.95
50 K	25%	50 MBps	k-Frequent	1	0.94	0.95	0.95
100 K	25%	50 MBps	k-Frequent	0.83	0.94	0.95	0.95

Table 4.20: RealCSE: Effect of various Snapshot Rates on the cache

The snapshot rate does not have a significant effect for this workload as well. Since we see a very small difference in the hit ratios, we proceed with using 100K as the rate for other experiments using this workload.

Implications of heuristic used for snapshot analysis

Snapshot Rate (requests)	Prewarm Set Size Limit (% 6 GiB)	Prewarm Rate	Heuristic	Instant. HR	5-min avg. HR	15-min avg. HR	30-min avg. HR
100 K	25%	100 MBps	k-Recent	0.83	0.94	0.95	0.95
100 K	25%	100 MBps	k-Frequent	0.83	0.94	0.95	0.95
100 K	25%	100 MBps	k-Frerecent (LNR)	0.83	0.94	0.95	0.95
100 K	25%	100 MBps	k-Frerecent (QDR)	0.83	0.94	0.95	0.95

Table 4.21: RealCSE: Effect of various Heuristics on the cache

The effect of the choice of heuristic on the hit ratios is not noticeable here. This is the same for all combinations of the other parameters as well. This behavior is indicative of the fact that this workload has a very high locality of reference, both temporally and spatially.

Per-vDisk analysis

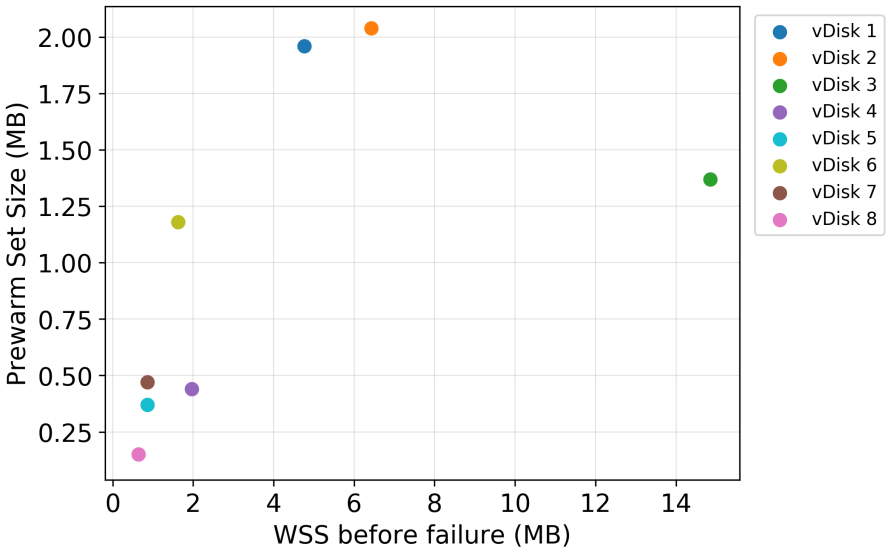

Figure 4.23: RealCSE: Correlation between WSS and share in Prewarm Set for vDisks

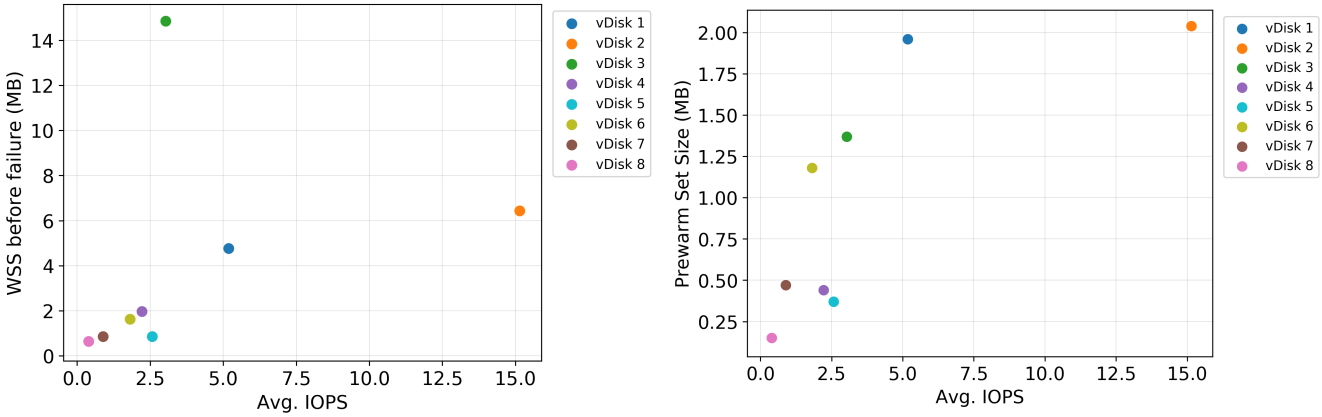

Figure 4.24: RealCSE: Correlation between avg. IOPS and WSS (left), avg. IOPS and Prewarm Set share (right) for vDisks

We have plotted the correlation between the IOPS of vDisks, and the WSS and Prewarm Set Share obtained during the experiment for this workload. These numbers are obtained from the experiment run with 25% prewarm set size limit and k-Recent heuristic for snapshot analysis. The WSS was estimated using a moving 10-minute window with a stride of 5 minutes, same as before.

From the graph on top, we can infer that for all vDisks, prewarm set share is (almost) proportional to their WSS before failure. The vDisk towards far right (vDisk 3) has a very high WSS as it had the highest (predetermined) wss80, and thus the lowest locality of reference among the vDisks. The individual values are in Table 4.3. The graphs at the bottom compare how the IOPS of each vDisk affect the WSS and prewarm set share of each vDisk. Here, we see that all vDisks except vDisk 3 are having proportional WSS values, and all of them are having similar

Prewarm Set shares. Even though vDisk 3 has highest IOPS value among all the vDisks, its (predetermined) wss80 value is very low.

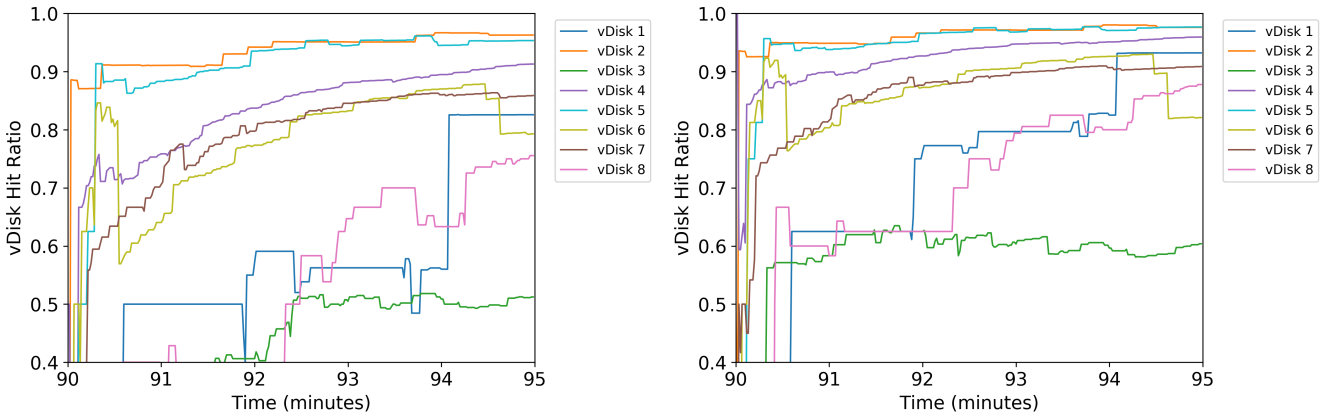

Figure 4.25: RealCSE: Impact of prewarming on the vDisk hit ratio for 5 minutes after failure: cold cache (left) and prewarmed cache (right)

The two figures above show the prewarming benefits for each vDisk. All of the vDisks in this workload experience an increased hit ratio within the first 5 minutes. In the results, we have also observed that increasing the prewarm set size (after 5% limit) does not help in reaching a higher hit ratio. We can easily achieve a reasonable hit ratio for all vDisks within the first few minutes after failure by prewarming only 5% (or less) of the cache.

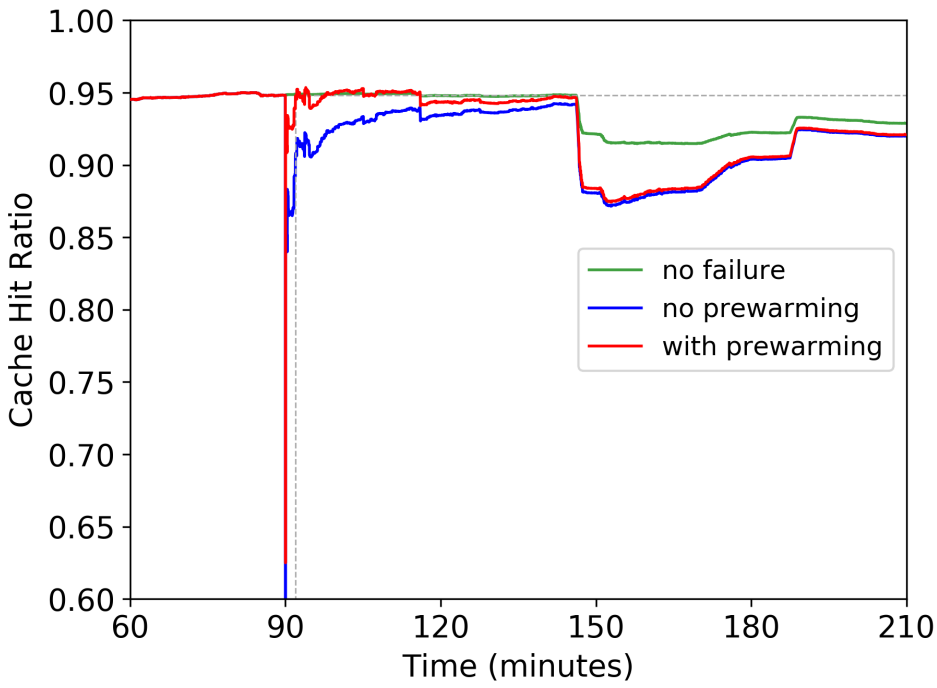

Figure 4.26: RealCSE: Impact of prewarming on the (cumulative) cache hit ratio after failure

In the figure above, we can see the overall impact prewarming has on the cache. The parameter combinations used for prewarming here are the same as in the previous figure. For this workload as well, the time it takes to reach the hit ratio value that was just before failure was reached within a few minutes. But, the hit ratio continues to fluctuate for a while after that. The prewarming does not help this workload much as it already is able to reach high hit ratios in the first few minutes with cold cache.

Summary across workloads

This experiment was run with 3 workloads, 2 of which are synthetic. We observe that the vDisks in real workloads are not able to utilize a large portion of the disk. Most of them have their I/O activity focused on multiple small sized hotspots. Regardless of the portion of disk they utilize, their accesses also have a certain pattern which is repeated over time. Our synthetic workloads have high spatial locality in some of the vDisks, but the temporal locality was not explicitly considered while generating them. From the various experiment results we can infer that the workloads Syn6GB and Syn4GB exhibit similar behavior for the same parameter combinations, with minor differences due to their vDisk configurations. But those results do not apply to the workload RealCSE. We saw that prewarming the cache benefited both Syn6GB and Syn4GB to a great extent, but it does not help much in the case of RealCSE since it had very high temporal locality as well.

Moreover, we have inferred that the various parameters related to a vDisk such as IOPS and WSS affect the size of its prewarm set share. For most vDisks, we get a slightly higher hit ratio when this share size is increased. The benefit in hit ratio with increasing prewarm set size depends heavily on the disk block access pattern of the vDisk as well. For the vDisks that do not benefit from an increased prewarm set set share, we should assign only a small share to them. For the same reason, we need a policy for partitioning the prewarm set according to these attributes of each vDisk.

Another important observation made across these workloads is that even when prewarming the cache with low prewarm set size limit and with prewarm low rate, we are able to see considerable improvements over the cold cache. Unless the workload has very high locality of accesses, prewarming is needed to alleviate the drastic drop in the hit ratio after failure.

4.3.4 Online estimation of parameters

In all the experiments we have performed, we first fix the values for each parameter and then measure the hit ratio on the destination node. But we have no way of knowing in advance which of combinations of these parameter values to use. In a practical setup, we should be able to determine the parameter value while the storage traffic is running and then use them if the node fails and we need prewarming at the destination.

At runtime, we have the following information for each vDisk available to us:

- Hit ratios of each vDisk
- Working set size of each vDisk
- Average IOPS for each vDisk

For prewarming at the destination, we need values for two parameters: prewarm set size and prewarm rate. Prewarm set size indicates how many objects should be considered for loading into the cache, and prewarm rate indicates how fast these objects need to be loaded into the cache. Also, we assume here that we can specify the values for these parameters for each vDisk separately.

There are several questions that arise from this online style of estimation for the parameters:

- Should we consider prewarming a vDisk at the destination cache?

  If a vDisk was able to reach a high hit ratio quickly (and maintains it) on the source node, then it has high locality of accesses. We can skip prewarming it as there will be little or no gains in the hit ratio on the destination.
  For vDisks in a workload having lower IOPS values than the others, there is high chance that their WSS is low as well. Since we can have a small prewarm set for that vDisk, we can choose to prewarm it. Similarly, we can always prewarm vDisks that have low WSS value (regardless of IOPS), such as only 1-5% of the destination cache.
  Even if a vDisk has high IOPS values and low WSS, we should always try to prewarm it. In general, if there is a vDisk which can perform well even without prewarming, we can choose to not prewarm it.

- What is the size of the prewarm set for a vDisk that we should load into into the cache?

  If we choose to prewarm a vDisk, we can partition the total cache memory available for prewarming proportionally based on the WSS of vDisks. Another way to partition the prewarm set is on the basis of their average IOPS value.
  We can also skip the partitioning completely and select a fixed size (such as 10% available cache memory on the destination) and start the prewarming process, with a certain prewarm rate. If the vDisk gets to a high hit ratio (such as its hit ratio before failure), stop the prewarming process for the vDisk. Similarly, if prewarming this vDisk is causing evictions, stop the prewarming.

- What is the rate at which we should prewarm a vDisk?

  We need an upper bound on prewarming rate for all vDisks, and this bound can be the same for all vDisks. The prewarm rate should depend on the rate at which vDisk loads objects into the cache due to its normal storage I/O traffic. Ideally, we should load the objects into the cache faster than the vDisks requests them.
  For vDisks with higher IOPS value, we should try loading the objects faster and vice-versa. If we have an available bandwidth for prewarming, we can divide it proportionally based on the vDisks' IOPS and use that as the prewarming rate. Since we load the most important objects first, this rate can be changed dynamically based on the time elapsed in prewarming. For instance, we can start with a high prewarm rate and halve it each second.

5. Conclusion & Future Work

From the results obtained from the experiment, we conclude that prewarming the cache does help mitigate the drop in hit ratio due to cold cache. The extent to which this drop is reduced depends on the combination of parameters (especially the snapshot rate, heuristic, memory limit and prewarm rate) we use. In general, we saw in the results for all workloads that snaphotting the cache at 100 K requests shows the most promising results. Moreover, (constrained) k-Recent heuristic performs well for all of these workloads, but for higher prewarm memory limits, (constrained) k-Frequent performs slightly better than the k-Recent.

In the first workload (Syn6GB), we were able to achieve an average hit ratio of 0.8 within 30 minutes using only 50% of cache for prewarming at 100MBps, when the cold cache was at 0.6. Moreover, since the hit ratio before failure was also 0.8, we were able to get to this value in a short time period.

In the results for second workload (Syn4GB), we saw similar improvements as well. With only 50% cache available for prewarming at 500MBps, we were able to get a 30-minute average hit ratio of 0.83. This is an improvement as well over the 0.69 hit ratio value for the cold cache. Similar to the previous workload, we were able to reach the hit ratio before failure (0.83) within 30 minutes after failure.

In the results for third workload (RealCSE), we observed that the benefits of prewarming apply not only to synthetic workloads, but to the real workloads as well. This workload had vDisks with a very high locality of references. Even with the cold cache, we had the 5-minute average hit ratio at 0.92. Our prewarming process did help in achieving a higher hit ratio in the first few minutes. With only 25% cache available for prewarming at 50MBps, we saw an improved 5-minute average of 0.94. Due to high locality in disk block accesses, prewarming the cache with only a few objects helps us significantly.

While the current setup for these experiments yielded promising results, we have to find ways to achieve better results which are indicative of improved storage performance.
The first step would be to run the same experiments for more diverse workloads. The workloads should be either be real or generated using some benchmark. These workloads should cover a wide range of combinations of various attributes such as locality of accesses, IOPS distributions and WSSes.
Moreover, we had fixed a few parameters for all these experiments, and the next step should be to try varying some of them for the experiments. Some important parameters (currently fixed) that we need to consider for tuning are:

- Sizes of HM1 and HM3 objects (sampled from a distribution)
- Partitioning of prewarm set for HM1 and HM3 objects
- Time instant at which failover happens (and the total duration)
- Number of vDisks migrating and number of vDisks already running on the destination

Apart from expanding the parameter space, we also need to add the notion of time into the simulator, where miss penalty being composed of disk and network accesses in units of time. We also need to consider the arrival and completion times of individual requests for a more elaborate empirical analysis. We need these properties to ensure a more realistic simulation and to create a better model of the Nutanix DSF Infrastructure.

Bibliography

[1] The Nutanix Bible. https://nutanixbible.com/

[2] Suryanarayana, V., Balasubramanya, K. M., & Pendse, R. (2012, October). Cache isolation and thin provisioning of hypervisor caches. In 37th Annual IEEE Conference on Local Computer Networks (pp. 240-243). IEEE.

[3] Tarasov, V., Jain, D., Hildebrand, D., Tewari, R., Kuenning, G., & Zadok, E. (2013). Improving I/O performance using virtual disk introspection. In Presented as part of the 5th USENIX Workshop on Hot Topics in Storage and File Systems.

[4] Zheng, J., Ng, T. S. E., & Sripanidkulchai, K. (2011, March). Workload-aware live storage migration for clouds. In Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (pp. 133-144).

[5] UMass Trace Repository. http://traces.cs.umass.edu/index.php/Storage/Storage

[6] FIU SyLab Resources. http://sylab-srv.cs.fiu.edu/doku.php?id=resources:start

[7] Flexible I/O Tester. https://github.com/axboe/fio

[8] Tarasov, V., Zadok, E., & Shepler, S. (2016). Filebench: A flexible framework for file system benchmarking. USENIX; login, 41(1), 6-12.

[9] Filebench. https://github.com/filebench/filebench

[10] Denning, P. J. (1968). The working set model for program behavior. Communications of the ACM, 11(5), 323-333.

[11] Block I/O Layer Tracing: blktrace. https://www.mimuw.edu.pl/ lichota/09-10/Optymalizacja-open-source/Materialy/10%20-%20Dysk/gelato_ICE06apr_blktrace_brunelle_hp.pdf

[12] https://en.wikipedia.org/wiki/Hyper-converged_infrastructure

[13] https://en.wikipedia.org/wiki/Storage_virtualization

[14] https://en.wikipedia.org/wiki/Software-defined_storage