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

Project Report
Version 1.0

submitted by

Sukrit Bhatnagar

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

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 prevalent 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, and *decentralized storage* wherein each node in the cluster consists of local storage which runs a distributed filesystem. Having such storage setups add network access latency to the existing latency incurred by the storage devices. We will consider virtualized environments with distributed storage for this study.

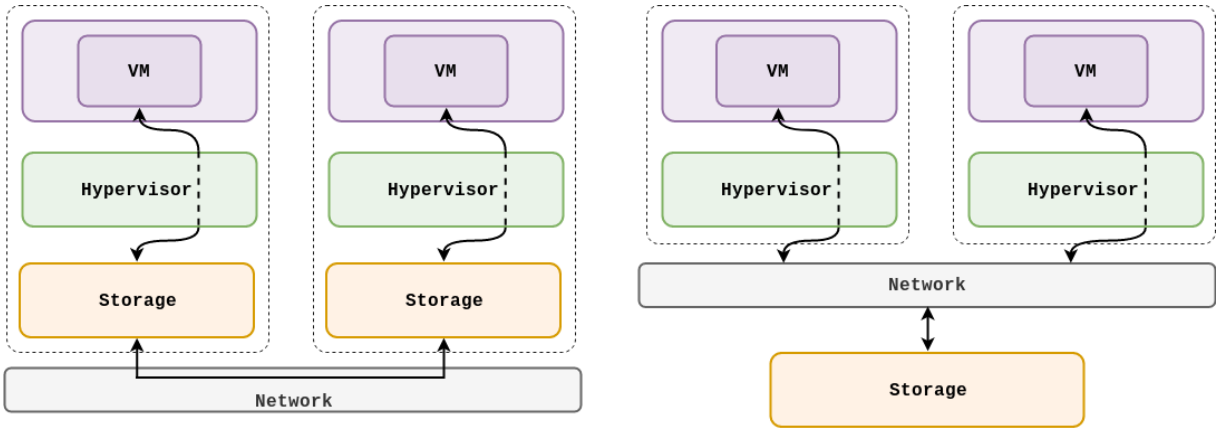

Figure 1.1: Common storage setups (distributed and centralized) in virtualized environments

Many such environments also make use of a combination of HDDs and SSDs to divide the data logically into tiers, with SSDs keeping the hot data. While SSDs provide an improvement in access latency (in the order of a few microseconds), they are expensive and are limited in size as compared to HDDs. Moreover, in networked storage setups, the network latency can still dominate the total storage I/O latency.

When a VM performs an I/O operation, the hypervisor intercepts the corresponding request and delegates it to the underlying storage mechanism, realizing a mechanism usually called storage virtualization. Hypervisor, being the manager of all system resources, is the natural choice for virtualization-oriented I/O performance improvements and optimizations. Whenever there is a need to fetch certain data from someplace, and this fetching is a costly operation,

we think of caches. A great deal of research is done in the past for making use of a cache inside the hypervisor layer. This cache is intended to work as a client-side cache for keeping storage I/O data fetched over the network.

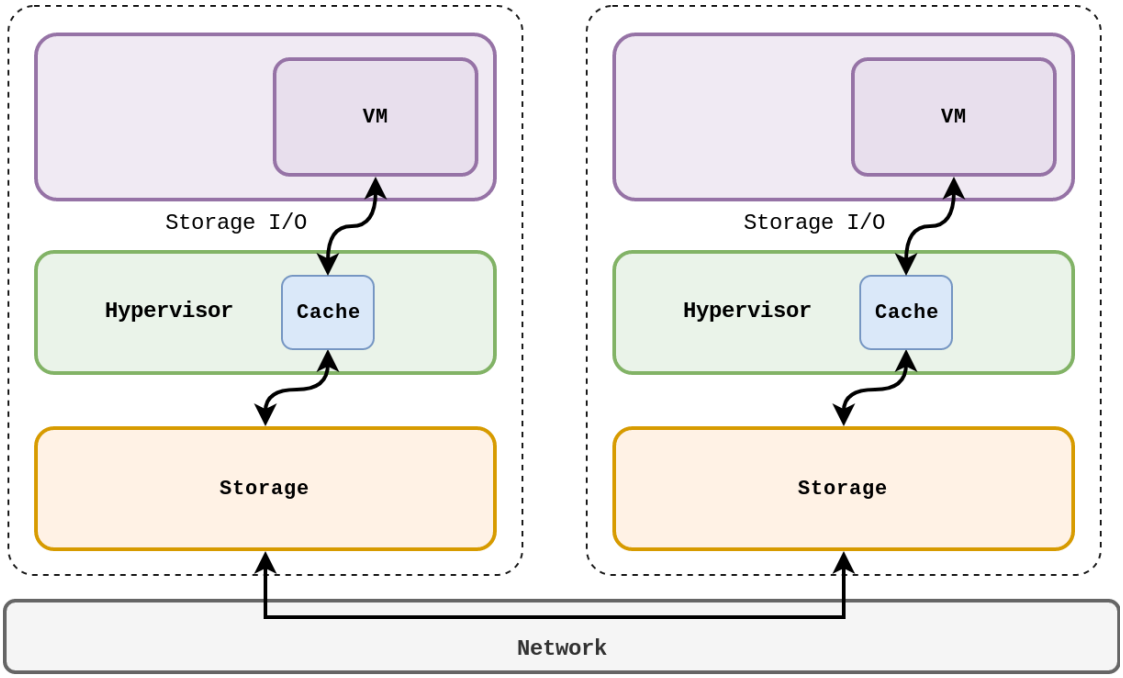

Figure 1.2: Virtualized environments with hypervisor cache

We define a few characteristics of our view of this 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 returns the same. 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 belonging to the underlying distributed filesystem. **We refer to individual entities in the cache (disk blocks as well as filesystem metadata) as objects.** Thus 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 might be contention among the VMs performing storage I/O operations when the cache is accessed.

- **Volatile**
  The cache is primarily maintained in-memory, with some part spanning over a fast storage devices such as SSD. 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 from other nodes. So, 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 memory available. While a dynamic cache seems better choice in virtualized setup, we assume its size to be fixed in our study.

Cold Cache, Warm Cache and Prewarming

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 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 cache may help in increasing the performance as a result of an increase in the overall hit ratio.

Working set of a VM in this context will refer to the set of cache objects that are actively in use by the VM. This will include objects which are frequently being accessed by the VM and/or will be accessed in the near future. Since the hypervisor cache is shared in nature, it might be possible that the working sets of all VMs cannot be accommodated and the workloads with high rate of storage I/O traffic may cause performance drop in other VM's storage I/O operations.

Prewarming refers to the process of proactively loading the cache with the relevant objects of one or more VMs. We use the term prewarming instead of prefetching 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.

Prewarm Set refers to the set of (hot) objects considered for prewarming the cache. A prewarm set can be considered as good if it is a good enough approximation of the working sets of guest VMs. We also use the term *Prewarm Time* which is the time taken by the cache to reach a certain (high) hit ratio, for instance, the hit ratio that we will get after 2 hours without prewarming.

1.1 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.

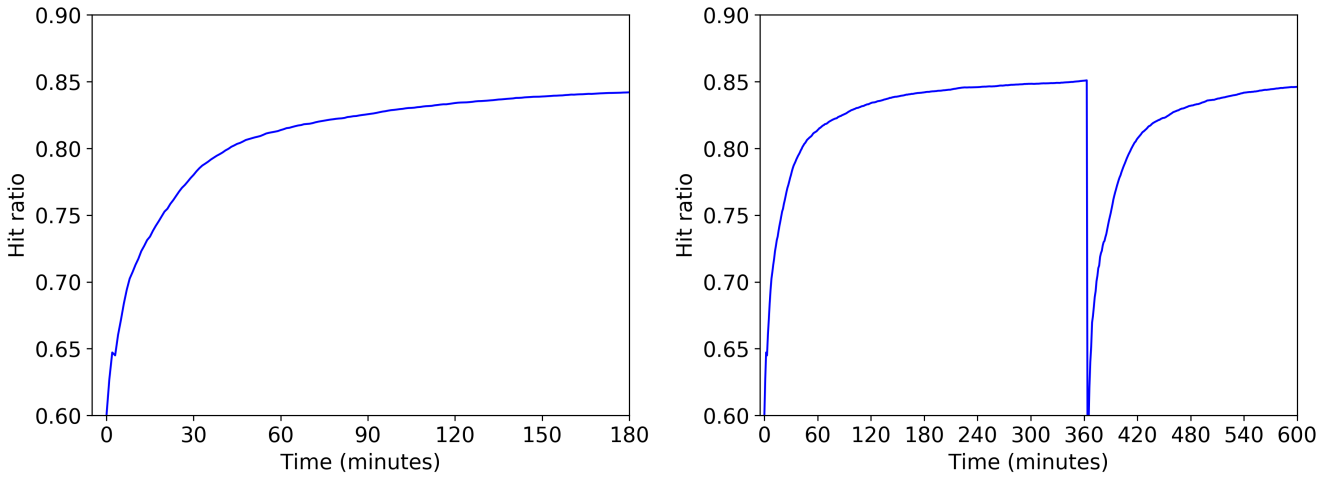

Figure 1.3: Cold cache when a node starts up (left) and when node failover happens (right)

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.

In this study we try to answer the following question:
Given a storage cache inside the hypervisor, how to reduce the time it takes to warm up the cache?
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 storage 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?

1.2 Solution Approach

A VM running on the 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. We are restricted to using only non-intrusive approaches for mitigating this problem of cold caches.

The only part visible to the hypervisor is the block I/O access requests that the VM issues to its vdisk. These requests are seen by the cache and a lookup is performed. After the required data is fetched into the cache, it is served from there. We can look inside 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 objects that are needed. But, as the cache is volatile, its state can be lost due to failures and there is no way to recover the same.

- 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 methods to establish the importance of an object to a vdisk, and we will discuss some of them in the next chapter.
- Finally, we will perform some experiments specific to a couple scenarios where the cache goes cold and we try to prewarm it to see if the performance drop (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.

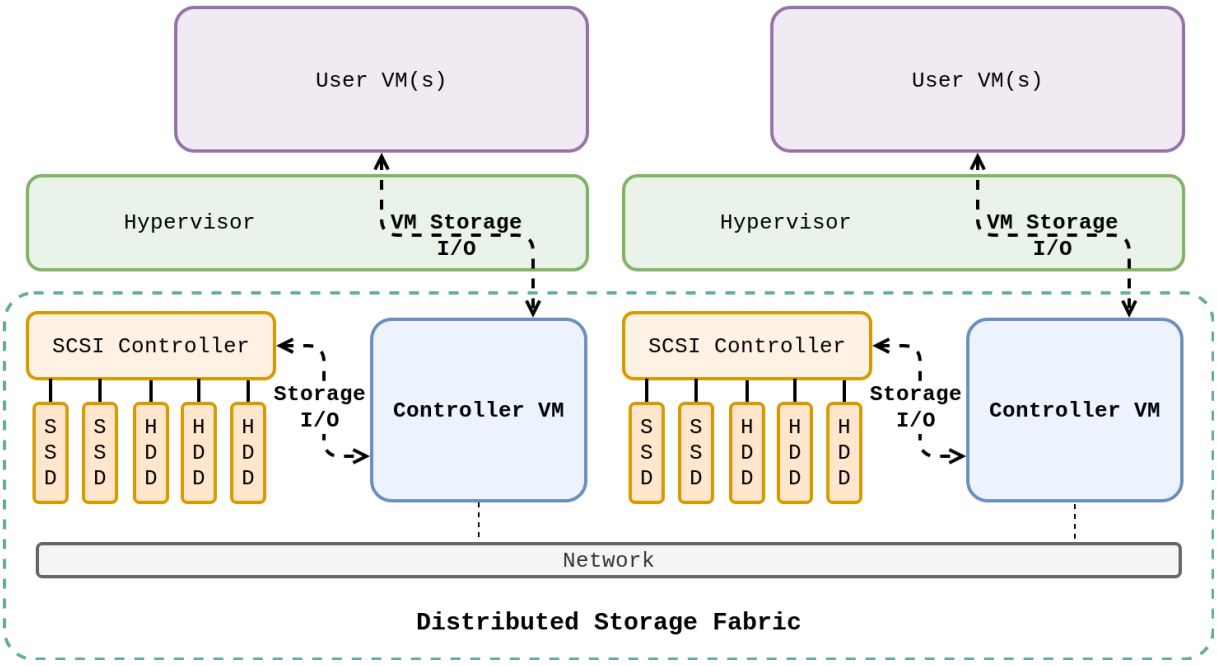

Figure 2.1: Overview of the Nutanix HCI

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.

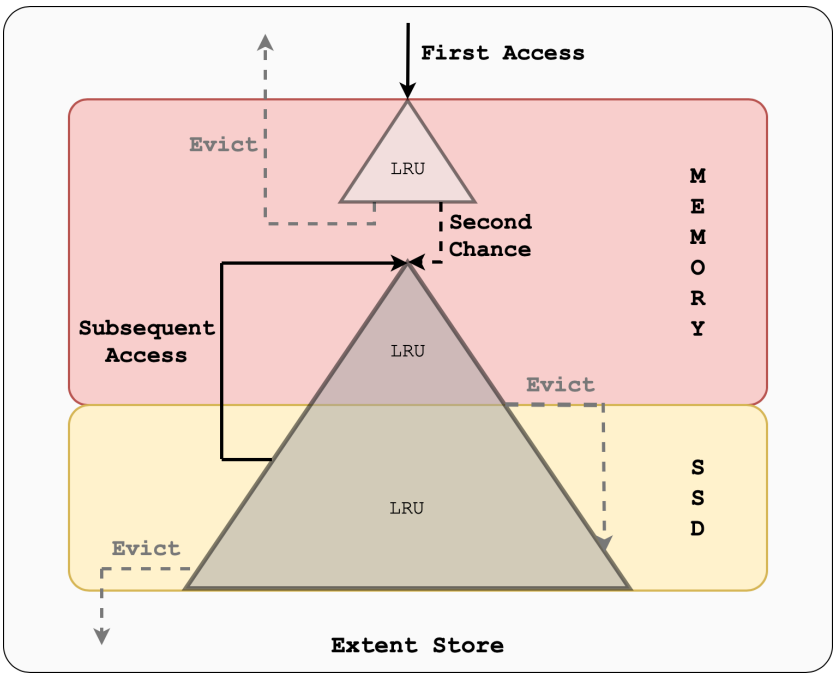

Figure 2.2: Overview of the CVM cache

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 of two tiers (memory and SSD) 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. It spans both memory and SSD. An object evicted from memory portion of this pool goes into the SSD portion, and similarly, an object promoted (on subsequent access) from SSD portion goes into the memory portion. Eviction from SSD portion results in an eviction from the cache. 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.

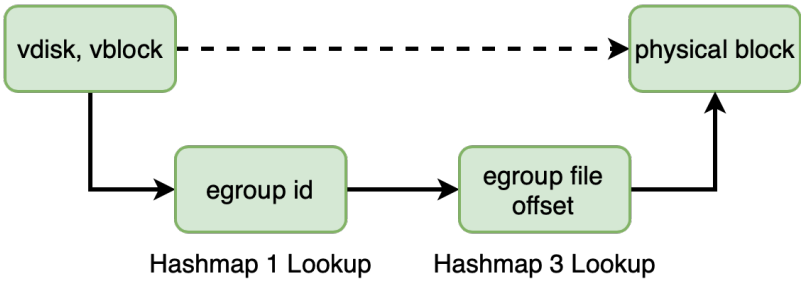

Figure 2.3: Metadata translation in Nutanix HCI

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 Hashmap, respectively.**

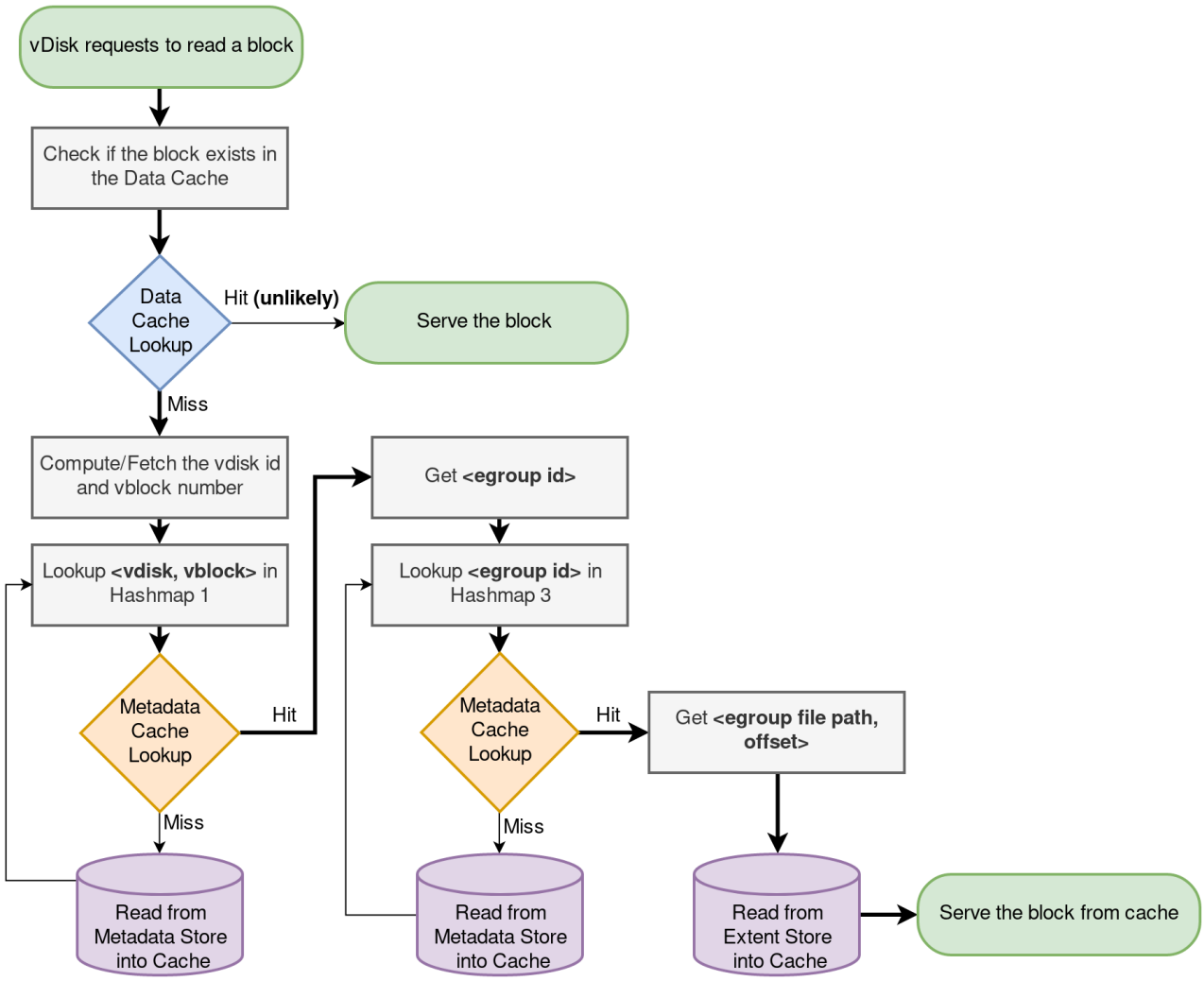

Figure 2.4: Cache Lookup Flow

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 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 3.1 shows a basic model of our system.

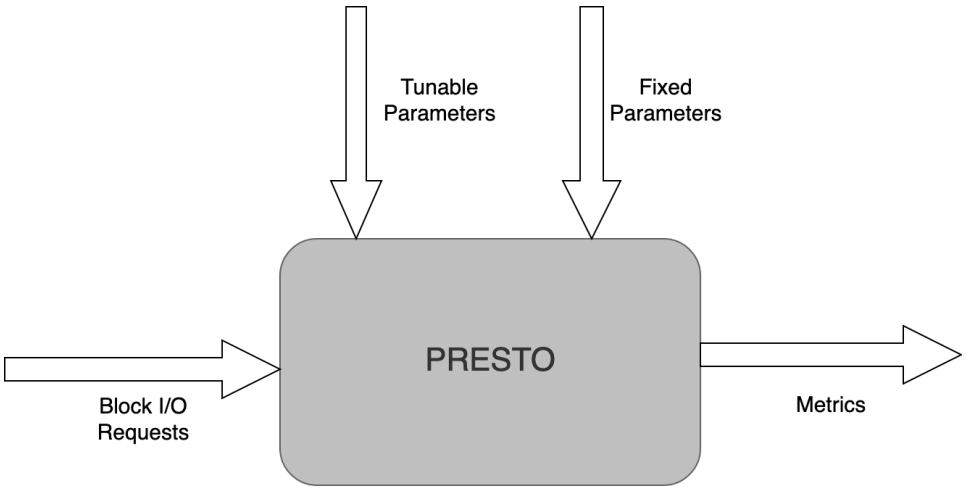

Figure 3.1: Basic overview of the simulation

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 or restart them from the last point
    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.

3.2 VM I/O Traces

To simulate VM workloads running on a hypervisor, we use a series of pre-recorded VM block I/O requests and issue them to our cache. The *blktrace* utility was used to record block I/O on various VMs running on the IITB CSE Department Infrastructure.

It 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 CSV format.

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.

The VMs on which the traces were recorded comprise of a mail server (Postfix, Dovecot, Mailman), three web servers (Apache 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. **An important thing to note here is that these traces do not capture the bootup sequence of these VMs.** The traces are recorded on up and running VMs.

We 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 is run on each VM that we consider for workload generation, as mentioned above.

```
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"
```

It was observed during the traces that most events captured on any VM 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.

The output from blktrace was formatted to include only the relevant information. All the block I/O traces used in the simulations 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. We assume vDisk IDs as consecutive whole numbers starting with 0.
- **Sector Number** is the sector number (512 KB sector size) of the vdisk which is to be accessed. The actual offest is calculated by multiplying the sector number with 512.
- **Size** is the amount of data (in bytes) that will be read or written to the vdisk.
- **Timestamp** is the time (in nanoseconds) of each request relative to the start time of our tracing utility.

The traces recorded from various VMs are merged into a single CSV file. After merging, the requests are sorted on the basis of the nanosecond fraction of their timestamp (i.e. seconds are discarded) mainly to introduce randomness because the requests when ordered by complete timestamp showed very high localities of reference in the cache and thus very high hit ratio (even without prewarming).

The trace file is read line-by-line by the simulator, and after parsing each line, a request is issued to the cache. In all our traces, there are requests with size over 1MB, and many requests span over more than one extent. We find all such cases as the traces are parsed and issue two or more requests to the cache which are extent-aligned.

For example, 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 vblock-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
```

3.3 Cache Design

3.3.1 Pools & Tiers

We model our cache as having multiple pools, and having both a memory and SSD component. The memory part consists of a single-touch and a multi-touch pool, whereas the SSD part has only the multi-touch pool. So, multi-touch pool spans over both memory and SSD, whereas the single-touch pool in completely in memory. Intuitively, the sizes of these pools increase 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, the single-touch pool is the smallest in size, SSD part of multi-touch pool is the largest in size, and the size of memory multi-touch pool sits somewhere in between theirs.

Single-touch pool stores the objects that are fetched into the cache once but are not read again. 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, it is promoted to the memory part of multi-touch pool. An object will remain in the memory multi-touch pool until it is evicted by the replacement policy, which will demote it to the SSD part of the multi-touch pool. 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. An object in the SSD pool will either be promoted to multi-touch pool upon a subsequent access, or will be evicted from the cache by the replacement policy. This is the same as shown in Figure 2.2.

Note that while the cache is divided into multiple tiers and pools, there is no partioning 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, the cache contains only metadata objects (not memory pages or the 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 |        358 |        359 |
_____________________________________________
| MULTI POOL  |       1804 |       1804 |
_____________________________________________
| SSD POOL    |       3610 |       3609 |
_____________________________________________

_____________________________________________
| (misses)    |  HASHMAP 1 |  HASHMAP 3 |
_____________________________________________
| TOTAL       |     535379 |     534928 |
_____________________________________________

_____________________________________________
| (hits)      |  HASHMAP 1 |  HASHMAP 3 |
_____________________________________________
| SINGLE POOL |     136730 |     136679 |
_____________________________________________
| MULTI POOL  |    2768238 |    2768595 |
_____________________________________________
| SSD POOL    |     274753 |     274898 |
_____________________________________________

_____________________________________________
| (evictions) |  HASHMAP 1 |  HASHMAP 3 |
_____________________________________________
| SINGLE POOL |     398291 |     397890 |
_____________________________________________
| MULTI POOL  |     409679 |     409773 |
_____________________________________________
| SSD POOL    |     131316 |     131266 |
_____________________________________________

_____________________________________________________________________________________
| (HM1 hits) |   SINGLE |    MULTI |      SSD |     HITS |   MISSES |  HIT RATIO |
_____________________________________________________________________________________
| VDISK    0 |     4527 |   810961 |    18462 |   833950 |    11835 |      0.986 |
_____________________________________________________________________________________
| VDISK    1 |    63793 |  1322921 |   141657 |  1528371 |   238055 |      0.865 |
_____________________________________________________________________________________
| VDISK    2 |    43763 |   330445 |    61626 |   435834 |   218870 |      0.666 |
_____________________________________________________________________________________
| VDISK    3 |     2176 |    92727 |     9815 |   104718 |    11679 |      0.900 |
_____________________________________________________________________________________
| VDISK    4 |     2953 |    86478 |     7708 |    97139 |    11323 |      0.896 |
_____________________________________________________________________________________
| VDISK    5 |    16544 |    85290 |    23280 |   125114 |    28186 |      0.816 |
_____________________________________________________________________________________
| VDISK    6 |     2074 |    28321 |     7150 |    37545 |    10424 |      0.783 |
_____________________________________________________________________________________
| VDISK    7 |      900 |    11095 |     5055 |    17050 |     5007 |      0.773 |
_____________________________________________________________________________________

Memory Hits : 5810242 (SINGLE + MULTI)
Total Hits  : 6359893
Total Misses: 1070307
Hit Ratio   : 0.855952
```

3.3.2 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.

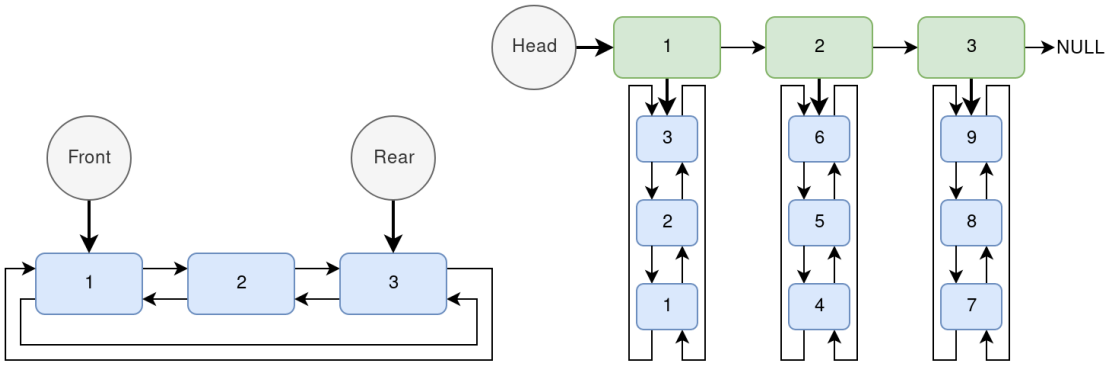

Figure 3.2: Implementation overview of LRU (left) and LFU (right)

- **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.

3.4 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.

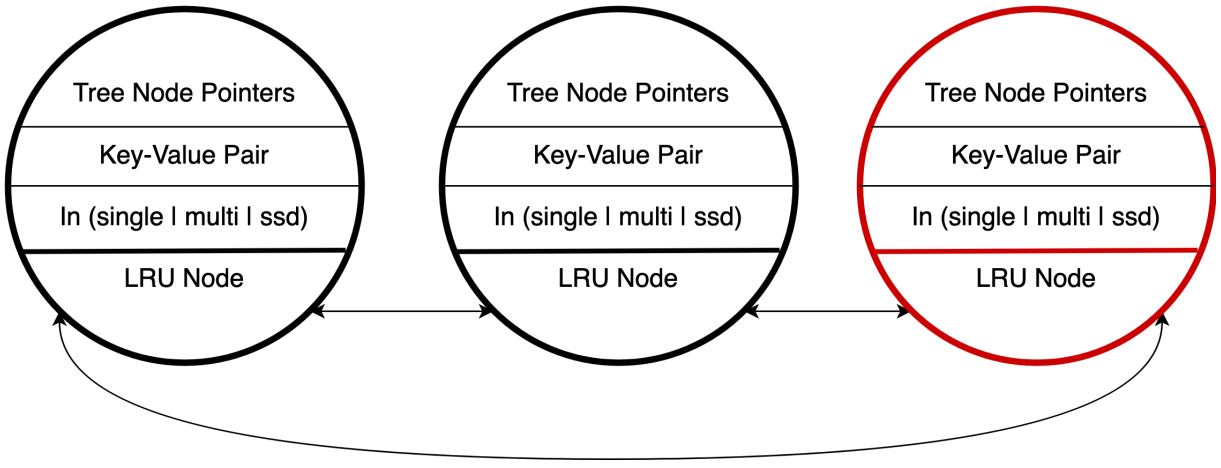

Figure 3.3: Structure of a node

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.

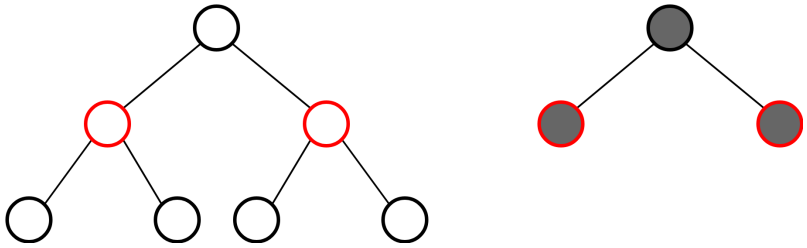

Figure 3.4: Using separate hashmaps for the metadata and the cache

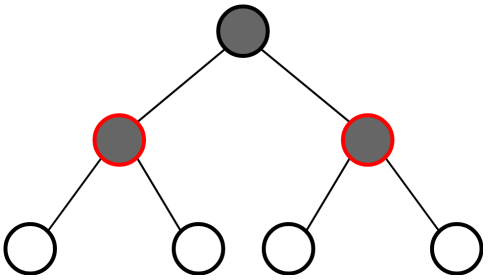

Figure 3.5: Reusing the metadata hashmap for cache lookup

Figures 3.4 and 3.5 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

3.5 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.

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

where:

- **Base Name** denotes the absolute path to the dump file for each vDisk
- **Bitmap Type** is either .csv (for Bit Set) or .bin (for Bit Array)
- **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.

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.

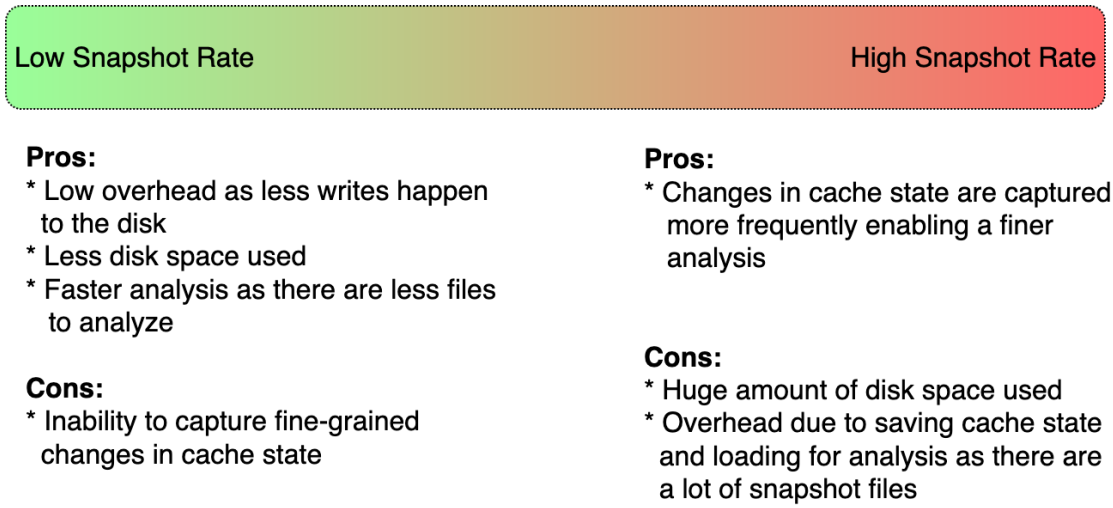

Figure 3.6: Snapshot rate tradeoffs

Figure 3.6 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.

3.6 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
_____________________________
```

3.6.1 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:
  - 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:

- Proportional Share: Split the prewarm set for vdisks in ratios calculated from the number of their currently cached objects
- 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.

3.7 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 (remember that the position of a bit in the bitmap tells us about the cache membership of that object). 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 multi-pool cache, we have a choice in which combination of the three pools to use to load these objects. We consider two such combinations: complete multi-touch pool (memory and SSD part), and memory part of multi-touch pool. The memory constraint for each of these will be the total size of pools we are using.
In both of these combinations of pools, we start loading objects in the memory part of multi-touch first. If it gets filled up, the older objects are splilled over to the SSD part and newer objects get into the memory part.

We do not load objects into single touch pool as the prewarm set contains important objects, and they may get evicted from the cache if other objects (not in prewarm set) are loaded into the cache. We keep the single touch pool free for those objects which are needed by the vdisks, but not included in the prewarm set.

4. Experiments & Results

The simulation involved 3 types of experiments which mimic specific scenarios which can render the cache cold, either completely or partially. Each experiment involves issuing block I/O requests to the cache and measuring the hit ratio. Note that the outcome of these experiments depend heavily on the nature of the block I/O requests the cache serves. We perform all our experiments using the block I/O traces described in Chapter 3.

The time for each run of our experiment is averaged from the total requests received in the total time duration, and is defined as follows.

```
1 minute = 5100 block I/O requests
```

4.1 Parameters & Metrics

Parameter Space

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

- HM1 object size: 1400 B
- HM3 object size: 27 KB (27648 B)
- Total cache size: 160 MB
  - Single-touch pool size: 10 MB
  - Multi-touch memory pool size: 50 MB
  - Multi-touch SSD pool size: 100 MB

  Note: The cache size is fixed as such to increase the memory pressure since the IOPS is very low in the block I/O traces, and the number of vdisks being served is small-scale.
- Total no. of I/O requests: ≈3.7 million (3,715,100 requests)
  - Total HM1 objects in the metadata store: 75562
  - Total HM3 objects in the metadata store: 75381
- Number of vdisks: 8 (sizes (in GB): 96, 492, 984, 19, 19, 19, 19, 16)
- Cache replacement policy: LRU

- Bitmap representation for persistent state: Bit Array
- Partition of prewarm set between HM1 and HM3 objects: Equal share
- Partition of prewarm set between vdisks (for HM1 objects): Fair share
- Time instant at which node failure happens (Experiment 2): 364th minute
- Time instant at which the VM migrates (Experiment 3): 405th minute
- Number of vdisks considered for migration: 1 (vdisk 1)

In table 4.1, some information about the vdisks used for the experiments is provided. Note that for the current study, we do not distinguish between read and write requests during cache lookup.

vdisk ID	vdisk Size	Total Requests	Read %	Write %	Min Request Size	Max Request Size
0	96 GB	843199	7.30	92.70	4 KB	1280 KB
1	492 GB	1729845	29.30	70.70	4 KB	1280 KB
2	984 GB	644935	66.64	33.36	4 KB	1280 KB
3	19 GB	115254	17.02	82.98	4 KB	512 KB
4	19 GB	106298	26.15	73.85	4 KB	512 KB
5	19 GB	136473	18.73	81.27	4 KB	512 KB
6	19 GB	47595	10.74	89.26	4 KB	504 KB
7	16 GB	21977	9.70	90.30	4 KB	512 KB

Table 4.1: Characterization of the vdisks served by the cache

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

- Snapshot rate: no. of I/O requests after which we dump the cache state
  `1 K, 5 K, 10 K, 50 K, 100 K`
- Prewarm set size limit: total size of metadata objects (in MB) considered for prewarming the cache
  `50 MB (Multi-touch pool - memory part), 150 MB (Complete multi-touch pool)`
- Heuristic for snapshot analysis:
  - k-Frequent
    - Frequency Score (k%): consider all objects appearing in at least k% of total snapshots
      `100%, 50%, 33%, 25%, 20%`
  - k-Recent
    - Recency Score (k): consider all objects from last k snapshots (first k snapshots for Experiment 1)
      `1, 2, 3, 4, 5`

- k-Frerecent
  - Score Threshold (k%): consider top k% objects by weight score
    `1%, 5%, 10%, 15%, 20%`
  - Weight function (f(x)): weight for an object if present in the snapshot number x
    x `(Linear),` x^2 `(Quadratic)`
- 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)`

Note: There is always an upper bound on the prewarm set size. In the heuristics without memory constraint, we abort the experiment run if the size limit is exceeded.

Metric Space

We capture the following metrics in our simulation:

- Cache hit ratio (recorded after each minute)
- Total size of persistent cache state written to the disk for various snapshot rates (independent of other tunable parameters)
- Time taken by prewarmed cache to reach the 90^{th}, 95^{th}, and 99^{th} percentiles of the hit ratio of the (initially) cold cache (referred to as "Time to P_{90}, P_{95} and P_{99}" in the result tables). The percentile values are calculated from the cache hit ratio value at the end of simulation.
- Total size of the prewarm set after heuristic-based snapshot anslysis
- Total number of HM1 and HM3 objects in the prewarm set

Snapshot Rate Analysis

As part of our experiments, we have also captured various metrics associated with the snapshot rates, and these metric are independent of other tunable parameters such as the memory constraint or heuristic used. We measure the total size of cache state persisted which depends on the number and sizes of vdisks, and this is the total amount which is written in multiple snapshot files. We also measure (at time of taking a snapshot) the number of objects that changed in the cache (fetched into/evicted from the cache) since the last snapshot was taken. The numbers shown in the table below for this are averaged over all the snapshots. Note that the total number of snapshots taken will depend on the total number of requests and the snapshot rate.

Snapshot Rate (one snapshot per n requests)	Total size of cache state persisted to the disk (MB)	Number of snapshots taken (per vdisk and for HM3 objects)	Average change in cache state (no. of objects changed between consecutive snapshots)
1 K	8189	3716	416
5 K	1639	744	1891
10 K	820	372	3272
50 K	165	75	4585
100 K	84	38	5038

Table 4.2: Metrics collected for various snapshot rates

From the table 4.2, we can infer that low snapshot rates are able to capture changes in objects in the cache at a finer granularity, but the total size of snapshots written to the disk is in the order of a few GBs. Moreover, this is the result when we are using only 8 vdisks in the simulation. Adding more vdisks will add to the total size. While snapshots taken at very high rates consume very less space in the disk, it misses a lot of intermediate states of objects getting in and out of the cache. We will use the snapshot rate of 1 K in the first experiment for comparison, but we omit in further experiments because it needs a large amount of state written to the disk and will incur more computational and I/O overheads for doing the same.

4.2 Experiment Scenarios

4.2.1 Node Restart

This experiment involves simulating the scenario where a node restarts and with it, all the VMs running on it. This can also be seen as the scenario where the node does not restart, but the VMs start up. In both cases, the cache will be rendered cols and when the VMs start up, they will experience many misses. We expect to see a very low cache hit ratio when the VMs start, but as they continue to run the hit ratio should stabilize as the cache will now be loaded with the metadata objects i.e., the cache will be warm.

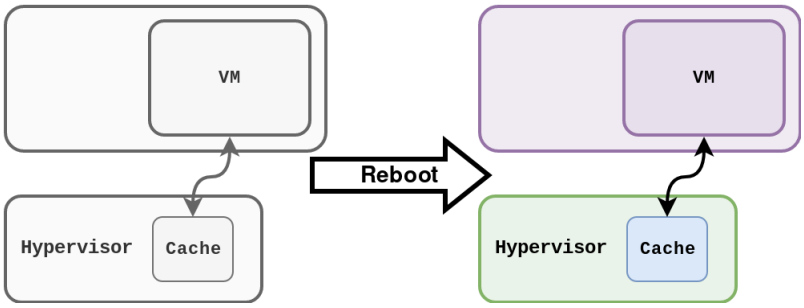

Figure 4.1: Node restart scenario

In this experiment, we answer the following question:
Does prewarming the cache helps in mitigating the drastic drop in the hit ratio when VMs boot up from a cold cache? In other words, can we reduce the time it takes to warm up a cold cache experienced at the beginning of storage I/O traffic?

The procedure for performing this experiment is as follows:

```
1. Issue the block I/O requests to the cache
2. Measure the hit ratio of the cache after each minute
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 same block I/O requests from beginning to the cache
7. Measure the hit ratio of the cache after each minute
```

At the end of this experiment, we have two sets of cache hit ratios for each minute, one where the cache was cold and the other where we prewarmed the cache with certain objects. This experiment was run multiple times for different values of the tunable parameters as mentioned previously and we plot a graph for the results obtained as shown in Figure 4.2. We show the hit ratios for only the first 3 hours as the aim of this experiment is to compare the hit ratio at the beginning of each run, with and without prewarming the cache. Tables 4.4, 4.5, 4.6, 4.7, 4.8 and 4.9 show the metrics collected for each heuristic.

Note that we have omitted those results from these tables which were exceeding the memory constraint. In all these tables, we have highlighted the top 4 combinations of parameters for each heuristic (2 for each of the size limits). These combinations offer the lowest warmup time. In the case of two combinations resulting in same warmup time, we chose the

one with lower snapshot rate (e.g. 100 K instead of 50 K).

P_{90} value of the Hit Ratio	Time taken by cold cache to reach the P_{90} value (minutes)	P_{95} value of the Hit Ratio	Time taken by cold cache to reach the P_{95} value (minutes)	P_{99} value of the Hit Ratio	Time taken by cold cache to reach the P_{99} value (minutes)
0.77	26	0.813	60	0.847	275

Table 4.3: Exp 1: Percentile values and warmup times for cold cache

Snapshot Rate (requests)	Frequency Score	Size Limit (MB)	Size (MB)	HM1 Objects	HM3 Objects	Time to P_{90} (minutes)	Time to P_{95} (minutes)	Time to P_{99} (minutes)
1 K	100%	150	0.28	10	10	26	60	275
	50%	150	64.45	463	2421	20	43	216
	33%	150	98.20	793	3684	12	27	144
	25%	150	135.18	1320	5060	8	20	113
	100%	50	0.28	10	10	26	60	275
5 K	100%	150	1.80	64	65	26	59	272
	50%	150	64.48	467	2422	20	43	216
	33%	150	98.25	798	3686	12	28	144
	25%	150	136.09	1332	5094	8	20	113
	100%	50	1.80	64	65	26	59	272
10 K	100%	150	2.30	81	83	26	59	272
	50%	150	64.86	468	2436	20	43	216
	33%	150	98.86	795	3709	12	27	144
	25%	150	136.81	1338	5121	8	20	114
	100%	50	2.30	81	83	26	59	272
50 K	100%	150	3.30	119	119	26	59	272
	50%	150	68.46	480	2572	19	41	212
	33%	150	100.51	829	3770	12	26	141
	25%	150	149.40	1522	5589	11	24	132
	100%	50	3.30	119	119	26	59	272
100 K	100%	150	3.88	140	140	26	58	271
	50%	150	70.25	501	2639	19	41	211
	33%	150	114.85	1022	4304	10	24	134
	100%	50	3.88	140	140	26	58	271

Table 4.4: Exp 1: Results for k-Frequent heuristic

Snapshot Rate (requests)	Size Limit (MB)	Size (MB)	HM1 Objects	HM3 Objects	Time to P_{90} (minutes)	Time to P_{95} (minutes)	Time to P_{99} (minutes)
1 K	150	149.86	56074	2844	0	5	40
	50	49.99	18722	948	0	7	25
5 K	150	149.86	56074	2844	0	5	40
	50	49.99	18722	948	0	7	25
10 K	150	149.34	55687	2844	0	5	40
	50	49.99	18722	948	0	7	25
50 K	150	124.39	36999	2844	0	0	26
	50	49.99	18723	948	0	7	26
100 K	150	107.05	24015	2844	0	2	30
	50	49.99	18722	948	0	9	31

Table 4.5: Exp 1: Results for Constrained-k-Frequent heuristic

Snapshot Rate (requests)	Recency Score	Size Limit (MB)	Size (MB)	HM1 Objects	HM3 Objects	Time to P_{90} (minutes)	Time to P_{95} (minutes)	Time to P_{99} (minutes)
1 K	1	150	4.96	178	179	26	59	272
	2	150	8.25	297	298	26	58	271
	3	150	11.55	416	417	25	56	266
	4	150	14.85	535	536	25	55	265
	5	150	17.56	633	634	25	55	264
	1	50	4.96	178	179	26	59	272
	2	50	8.25	297	298	26	58	271
	3	50	11.55	416	417	25	56	266
	4	50	14.85	535	536	25	55	265
	5	50	17.56	633	634	25	55	264
5 K	1	150	17.56	633	634	25	55	264
	2	150	28.92	1041	1044	23	52	252
	3	150	37.48	1349	1353	22	48	244
	4	150	46.37	1668	1674	0	46	222
	5	150	53.37	1920	1927	0	43	217
	1	50	17.56	633	634	25	55	264
	2	50	28.92	1041	1044	23	52	252
	3	50	37.48	1349	1353	22	48	244
	4	50	46.37	1668	1674	0	46	222
10 K	1	150	28.92	1041	1044	23	52	252
	2	150	46.37	1668	1674	0	46	222
	3	150	59.60	2144	2152	0	41	213
	4	150	71.52	2573	2582	0	33	188
	5	150	83.78	3010	3025	0	22	141
	1	50	28.92	1041	1044	23	52	252
	2	50	46.37	1668	1674	0	46	222
50 K	1	150	49.96	1794	1804	8	30	170
	2	150	79.82	2868	2882	1	9	85
	3	150	106.05	3809	3829	0	5	14
	4	150	128.45	4591	4639	0	2	16
	5	150	149.64	5338	5405	0	8	36
	1	50	49.96	1794	1804	8	30	170
100 K	1	150	49.96	1796	1804	9	19	134
	2	150	82.91	2948	2995	2	11	36
	3	150	111.80	3935	4041	0	7	25
	4	150	140.82	4814	5097	0	8	31
	1	50	49.96	1796	1804	9	19	134

Table 4.6: Exp 1: Results for k-Recent heuristic

Snapshot Rate (requests)	Size Limit (MB)	Size (MB)	HM1 Objects	HM3 Objects	Time to P_{90} (minutes)	Time to P_{95} (minutes)	Time to P_{99} (minutes)
1 K	150	149.86	56074	2844	1	14	87
	50	49.99	18722	948	12	24	135
5 K	150	149.86	56074	2844	1	14	87
	50	49.99	18722	948	12	24	135
10 K	150	149.30	55657	2844	1	14	77
	50	49.99	18721	948	12	24	134
50 K	150	124.13	36807	2844	0	7	43
	50	49.99	18722	948	12	24	133
100 K	150	106.70	23751	2844	1	14	62
	50	49.99	18723	948	9	20	100

Table 4.7: Exp 1: Results for Constrained-k-Recent heuristic

Snapshot Rate (requests)	Score Function	Size Limit (MB)	Size (MB)	HM1 Objects	HM3 Objects	Time to P_{90} (minutes)	Time to P_{95} (minutes)	Time to P_{99} (minutes)
1 K	LNR	150	149.86	56074	2844	0	0	20
	QDR	150	146.84	53816	2844	2	12	43
	LNR	50	49.99	18722	948	0	7	20
	QDR	50	49.99	18721	948	7	16	83
5 K	LNR	150	149.86	56074	2844	0	0	20
	QDR	150	149.86	56074	2844	0	0	12
	LNR	50	49.99	18722	948	0	7	21
	QDR	50	49.99	18722	948	0	7	20
10 K	LNR	150	149.30	55657	2844	0	0	20
	QDR	150	149.30	55657	2844	0	0	12
	LNR	50	49.99	18721	948	0	7	20
	QDR	50	49.99	18721	948	0	7	20
50 K	LNR	150	124.13	36807	2844	0	0	14
	QDR	150	124.13	36807	2844	0	0	7
	LNR	50	49.99	18722	948	0	7	21
	QDR	50	49.99	18722	948	0	7	22
100 K	LNR	150	106.70	23751	2844	0	1	19
	QDR	150	106.70	23751	2844	0	0	13
	LNR	50	49.99	18723	948	0	9	26
	QDR	50	49.99	18723	948	0	9	26

Table 4.8: Exp 1: Results for Constrained-k-Frerecent heuristic

Snapshot Rate (requests)	Score Threshold	Score Function	Size Limit (MB)	Size (MB)	HM1 Objects	HM3 Objects	Time to P_{90} (minutes)	Time to P_{95} (minutes)	Time to P_{99} (minutes)
1 K	1%	LNR	150	15.54	557	561	22	48	241
	1%	QDR	150	14.32	534	516	22	47	220
	5%	LNR	150	77.72	2800	2806	7	20	109
	5%	QDR	150	71.72	2687	2584	10	22	118
	10%	QDR	150	143.47	5378	5169	10	22	121
	1%	LNR	50	15.54	557	561	22	48	241
	1%	QDR	50	14.32	534	516	22	47	220
5 K	1%	LNR	150	15.54	557	561	22	48	240
	1%	QDR	150	15.54	557	561	22	47	223
	5%	LNR	150	77.72	2800	2806	7	20	109
	5%	QDR	150	77.72	2800	2806	7	17	95
	1%	LNR	50	15.54	557	561	22	48	240
	1%	QDR	50	15.54	557	561	22	47	223
10 K	1%	LNR	150	15.42	551	557	22	48	240
	1%	QDR	150	15.42	551	557	22	47	223
	5%	LNR	150	77.20	2779	2787	7	20	109
	5%	QDR	150	77.20	2779	2787	7	17	95
	1%	LNR	50	15.42	551	557	22	48	240
	1%	QDR	50	15.42	551	557	22	47	223
50 K	1%	LNR	150	10.74	364	389	24	54	262
	1%	QDR	150	10.74	364	389	24	52	252
	5%	LNR	150	53.73	1835	1945	13	30	156
	5%	QDR	150	53.73	1835	1945	12	27	143
	10%	LNR	150	107.48	3676	3890	5	15	65
	10%	QDR	150	107.48	3676	3890	1	8	31
	1%	LNR	50	10.74	364	389	24	54	262
	1%	QDR	50	10.74	364	389	24	52	252
100 K	1%	LNR	150	7.46	233	271	25	55	264
	1%	QDR	150	7.46	233	271	25	54	263
	5%	LNR	150	37.39	1185	1358	18	38	200
	5%	QDR	150	37.39	1185	1358	16	33	172
	10%	LNR	150	74.81	2372	2717	9	22	120
	10%	QDR	150	74.81	2372	2717	7	19	102
	15%	LNR	150	112.20	3560	4075	5	15	73
	15%	QDR	150	112.20	3560	4075	1	10	32
	20%	LNR	150	149.62	4747	5434	8	19	107
	20%	QDR	150	149.62	4747	5434	6	15	85
	1%	LNR	50	7.46	233	271	25	55	264
	1%	QDR	50	7.46	233	271	25	54	263
	5%	LNR	50	37.39	1185	1358	18	38	200
	5%	QDR	50	37.39	1185	1358	16	33	172

Table 4.9: Exp 1: Results for k-Frerecent heuristic

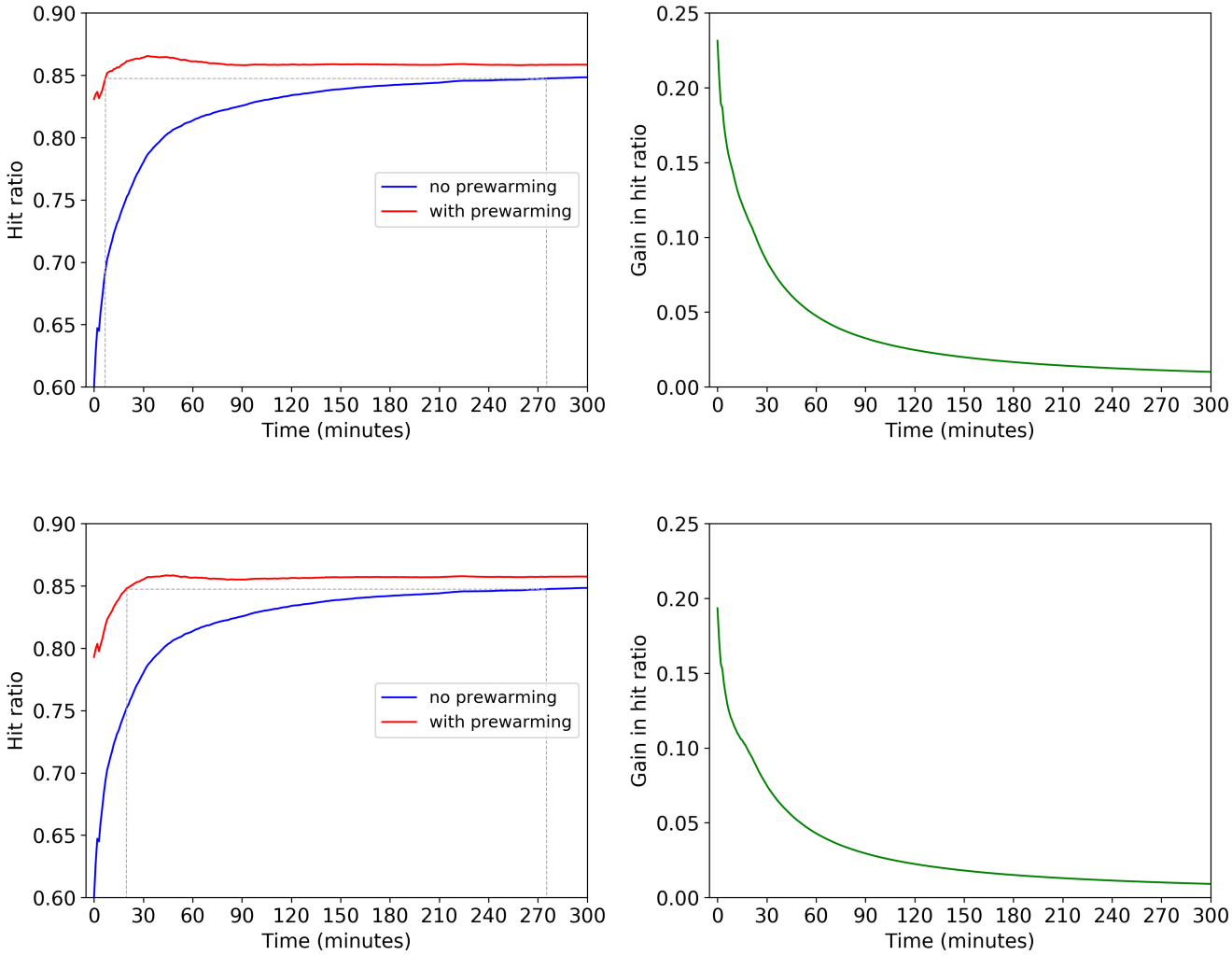

Figure 4.2: Exp 1: Impact of prewarming on hit ratio using Constrained-k-Frerecent heuristic and memory limit of 150 MB (top) and 50 MB (bottom)

We have plotted the graphs (Figure 4.2) for two combinations of parameters which yielded the most promising results. The top two plots used Constrained-k-Frereent as heuristic, 50 K as the snapshot rate, 150 MB as memory limit and Quadratic function for the scores. The bottom two plots used the same heuristic, but the snapshot rate of 10 K and the memory limit of 50 MB. It is worth mentioning here that we run the same requests again from the beginning after we reset the cache, which might not be the case in a real scenario.

The plots on the left compare the hit ratios where the horizontal grey-dotted line plots the 99^{th} percentile of the (cold cache) hit ratio. The plots on the right show the gain in hit ratio achieved with prewarming, and this gain diminishes with time. We make the following observations from these graph plots:

- In the time v/s hit ratio plots above (left), we can see that the hit ratio for the cold cache for first couple hours is very low, but it stabilizes later. When we prewarm the cache, we see that the drop in the hit ratio for initial few minutes is greatly reduced. The warmup time (for achieving the P_{99} hit ratio) is reduced to a few minutes (shown by the vertical grey-dotted lines).

- In the plots for gain in hit ratio (right), we can see that for the first few minutes, the gain in hit ratio is very large, which is indicative of the performance drop avoided due to a cold cache. This gain eventually becomes close to 0 as even the cold cache will eventually be filled with objects that are in active use.

Apart from the top results we have obtained in this experiment, there are a few worth noticing:

- In the results for Constrained-k-Frequent (Table 4.5) and Constrained-k-Frerecent (Table 4.8), even using 50 MB as the memory limit reduced the warmup time significantly. The lowest warmup time (for P_{99} hit ratio) achieved using 50 MB memory limit (20 minutes) was when Constrained-k-Frecent heuristic was used, as plotted in the bottom two graphs above.
- The score function used in the [Constrained-]k-Frerecent heuristics is very important as in most of the results (especially when using 150 MB memory limit), using the Quadratic function reduced the warmup time more than using Linear function did. The score function used affects the selection of objects for the prewarm set and even though both of these functions selected the same number of objects (HM1 and HM3), the resulting warmup times were different.
- For the memory limit of 150 MB, k-Recent performed well, reducing the time to reach P_{99} of hit ratio to just 14 minutes. For the memory limit of 50 MB, Constrained-k-Recent performed well, reducing the time to reach P_{99} of hit ratio to just 25 minutes.

Table 4.10 lists a subset of the results obtained that offer the highest speedups in the warmup time.

Heuristic Used	Snapshot Rate (requests)	Prewarm Set Size / Size Limit (MB)	Time to P_{99} for cold cache (minutes)	Time to P_{99} with prewarming (minutes)	Speedup achieved
Constrained-k-Frerecent	50 K	124.13 / 150	275	7	39x
Constrained-k-Frerecent	10 K	49.99 / 50	275	20	13x
k-Recent (k=3)	50 K	106.05 / 150	275	14	19x
Constrained-k-Recent	10 K	49.99 / 50	275	25	11x

Table 4.10: Exp 1: Speedup values for the top results

4.2.2 Node Failover

This experiment involves simulating the scenario where a node in a failover cluster experiences some problem and all its VM are started on another node. Unlike the node on which the VMs were initially running, the new node will have a cold cache, i.e. the information about the cached objects is lost. The VMs will start running from the same point at which the previous node experienced failure. We expect to see a drastic drop in the cache hit ratio when the VMs start running on the new node, but it should stabilize as the VMs continue to run.

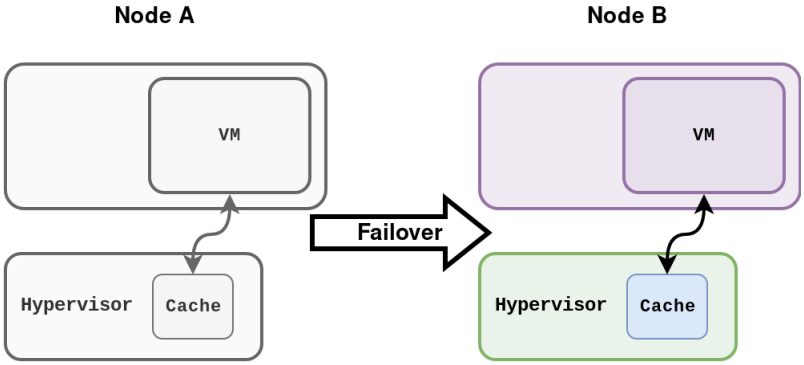

Figure 4.3: Node failover scenario

This scenario differs from the previous one in that the VMs experienced a cold cache when they were starting up in that, while in this scenario, the VMs start from the same point where they left off on the failed node.

In this experiment, we try to answer the following question:
Does prewarming the cache on the new node with some objects helps in alleviating the drastic drop in the hit ratio due to the cache being cold for the incoming VMs?

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 nth request is served
2. Measure the hit ratio of the cache after each minute
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 request
7. Measure the hit ratio of the cache after each minute
```

We issue the requests until 1857550 requests are served i.e. n is 1.856 million which is nearly half of the total requests that the cache will see. This corresponds to the 364th minute of the simulation run. At the end of this experiment, we have two sets of cache hit ratios for each minute, one where the cache was cold after nth request and the other where we prewarmed the cache with certain objects after nth request.

This experiment was run for only 4 of the 6 heuristics mentioned earlier, leaving out k-Frequent and k-Frerecent because for both of these heuristics, we needed to manually specify a score threshold which in some cases resulted in prewarm set not utilizing available space and in some cases did not properly satisfy memory constraints (as indicated by the results of Experiment 1). We have kept k-Recent in consideration as we would be interested in knowing whether prewarming the cache with a specific number of most recent snapshots holds any benefit.
Moreover, we also skip the snapshot rate of 1 K 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. Tables 4.5, 4.6, 4.7, and 4.8 show the metrics collected for each heuristic we have considered for this experiment.

Note that we have omitted those results from these tables which were exceeding the memory constraint. In all these tables, we have highlighted the top 4 combinations of parameters for each heuristic (2 for each of the size limits). These combinations offer the lowest warmup time. In the case of two combinations resulting in same warmup time, we chose the one with lower snapshot rate (e.g. 100 K instead of 50 K).

P_{90} value of the Hit Ratio	Time taken by cold cache to reach the P_{90} value (minutes)	P_{95} value of the Hit Ratio	Time taken by cold cache to reach the P_{95} value (minutes)	P_{99} value of the Hit Ratio	Time taken by cold cache to reach the P_{99} value (minutes)
0.765	31	0.808	57	0.842	180

Table 4.11: Exp 2: Percentile values and warmup times for cold cache

Snapshot Rate (requests)	Size Limit (MB)	Size (MB)	HM1 Objects	HM3 Objects	Time to P_{90} (minutes)	Time to P_{95} (minutes)	Time to P_{99} (minutes)
5 K	150	120.69	34230	2844	0	1	10
	50	49.99	18722	948	2	11	38
10 K	150	120.25	33902	2844	0	1	12
	50	49.99	18722	948	2	10	36
50 K	150	104.41	22034	2844	0	3	20
	50	49.99	18723	948	2	14	43
100 K	150	94.02	14251	2844	2	8	27
	50	44.02	14251	948	4	18	49

Table 4.12: Exp 2: Results for Constrained-k-Frequent

Snapshot Rate (requests)	Recency Score	Size Limit (MB)	Size (MB)	HM1 Objects	HM3 Objects	Time to P_{90} (minutes)	Time to P_{95} (minutes)	Time to P_{99} (minutes)
5 K	1	150	121.67	1787	4524	0	10	36
	2	150	136.10	2305	5045	0	5	29
	3	150	146.93	2694	5436	0	8	31
10 K	1	150	122.34	1793	4549	0	10	39
	2	150	145.10	2609	5371	0	9	33
50 K	1	150	122.34	1793	4549	0	10	39
100 K	1	150	131.56	1786	4899	5	15	47

Table 4.13: Exp 2: Results for k-Recent

Snapshot Rate (requests)	Size Limit (MB)	Size (MB)	HM1 Objects	HM3 Objects	Time to P_{90} (minutes)	Time to P_{95} (minutes)	Time to P_{99} (minutes)
5 K	150	120.69	34230	2844	0	1	5
	50	49.99	18723	948	1	8	32
10 K	150	120.25	33902	2844	0	1	8
	50	49.99	18722	948	2	10	33
50 K	150	103.94	21686	2844	0	2	17
	50	49.99	18722	948	2	10	34
100 K	150	93.36	13757	2844	2	8	28
	50	43.36	13757	948	5	17	47

Table 4.14: Exp 2: Results for Constrained-k-Recent

Snapshot Rate (requests)	Score Function	Size Limit (MB)	Size (MB)	HM1 Objects	HM3 Objects	Time to P_{90} (minutes)	Time to P_{95} (minutes)	Time to P_{99} (minutes)
5 K	LNR	150	120.69	34230	2844	0	1	10
	QDR	150	120.69	34230	2844	0	1	17
	LNR	50	49.99	18723	948	2	12	41
	QDR	50	49.99	18723	948	2	12	43
10 K	LNR	150	120.25	33902	2844	0	1	12
	QDR	150	120.25	33902	2844	0	2	18
	LNR	50	49.99	18722	948	2	12	41
	QDR	50	49.99	18722	948	2	13	43
50 K	LNR	150	103.94	21686	2844	0	3	20
	QDR	150	103.94	21686	2844	0	4	23
	LNR	50	49.99	18722	948	2	13	43
	QDR	50	49.99	18722	948	2	14	44
100 K	LNR	150	93.36	13757	2844	2	8	29
	QDR	150	93.36	13757	2844	2	8	28
	LNR	50	43.36	13757	948	5	21	52
	QDR	50	43.36	13757	948	5	22	53

Table 4.15: Exp 2: Results for Constrained-k-Frerecent

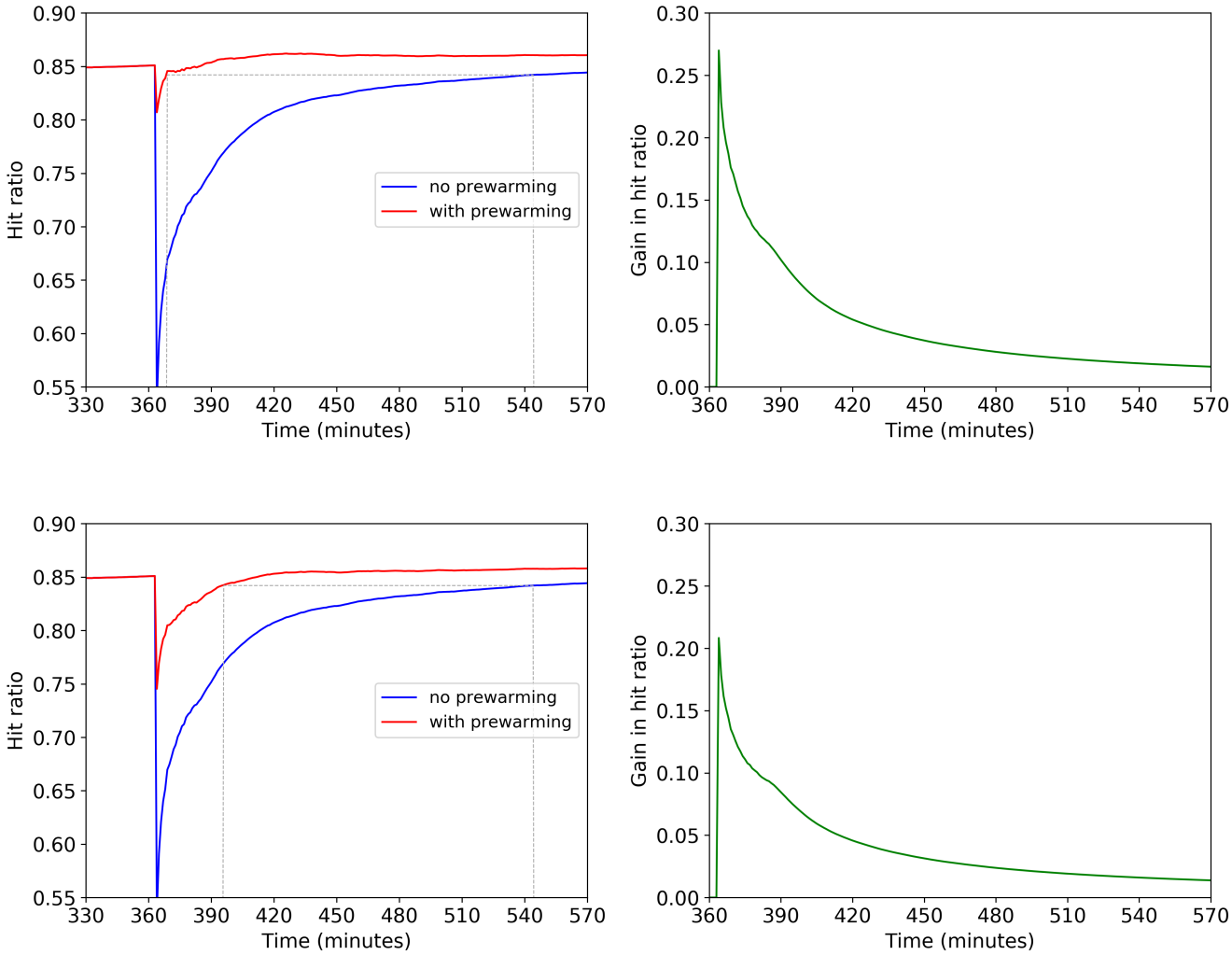

Figure 4.4: Exp 2: Impact of prewarming on hit ratio after failure using Constrained-k-Recent heuristic and memory limit of 150 MB (top) and 50 MB (bottom)

We have plotted the graphs (Figure 4.4) for two combinations of parameters which yielded the most promising results. All of the above plots use Constrained-k-Recent as heuristic and 5 K as the snapshot rate, but differ in the memory limits.
The plots on the left compare the hit ratios where the horizontal grey-dotted line plots the 99^{th} percentile of the (cold cache) hit ratio after the failure. The plot on the right show the gain in hit ratio achieved with prewarming, with the gain diminishing with time. We make the following observations from these graph plots:

- In the time v/s hit ratio plots above (left), we can see that the hit ratio for the cold cache for first few hours after failure is very low, but stabilizes later. When we prewarm the cache, we see that the drop in the hit ratio for initial few minutes is greatly reduced. The warmup time (for achieving the P_{99} hit ratio) is reduced to a few minutes (shown by the vertical grey-dotted lines).

- In the plots for gain in hit ratio (right), we can see that for the first few minutes after the failure, the gain in hit ratio is very large, which is indicative of the performance drop avoided due to a cold cache.

Apart from the top results we have obtained in this experiment, there are a few worth noticing:

- In the results obtained for Constrained-k-Frequent and Constrained-k-Frerecent, we can see that the time taken to get P_{99} hit ratio is very similar, especially with the 150 MB memory limit.
- If we compare the results for Constrained-k-Frerecent for this experiment and the previous one, we see that in this experiment, the Linear function resulted in lower warmup times as compared to Quadratic function, but it was the opposite in the previous experiment.

Table 4.16 lists a subset of the results obtained that offer the highest speedups in the warmup time.

Heuristic Used	Snapshot Rate (requests)	Prewarm Set Size / Size Limit (MB)	Time to P_{99} for cold cache (minutes)	Time to P_{99} with prewarming (minutes)	Speedup achieved
Constrained-k-Recent	5 K	120.69 / 150	275	5	55x
Constrained-k-Recent	5 K	49.99 / 50	275	32	8x

Table 4.16: Exp 2: Speedup values for the top results

4.2.3 VM Migration

This experiment involves simulating the scenario where a particular VM migrates from one node (source) to another (destination). As part of the VM migration, the state of its vdisk(s) is also transferred. But, the objects in the cache which were used by those vdisk(s) are not. On the destination node, the cache will be cold for those vdisk(s) and it may have objects cached for other VMs that are running on it. We expect to see a drop in the overall hit ratio as the migrated VM will experience cold misses, and now that the cache has to accommodate objects for the vdisk(s) of the migrated VM, the other VMs should experience more capacity misses.

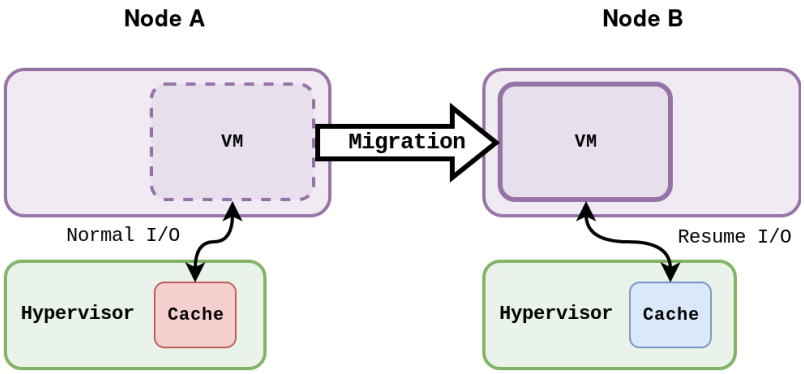

Figure 4.5: VM Migration scenario

This scenario differs from the previous two as the cache will now be partially cold for the migrated VM and all the VMs on the destination node will have to share the cache with the new VM.

In this experiment, we try ot answer the following question:
Does prewarming the cache on the destination node with some objects of the vdisk belonging to the migrating VM help in reducing the drastic drop in its hit ratio? And, with the prewarming done for this vdisk, does the hit ratio of other vdisks gets affected?

For the experiment, we assume that there is only one vdisk attached to the migrating VM.

The procedure for performing this experiment is as follows:

```
1. Run the block I/O traces on two caches (in separate threads) until nth request
2. Transfer the bitmap of migrating vdisk from first cache to the second
3. Load the objects from the migrated bitmap in the second cache
4. Resume the block I/O traces in the second cache, with additional traces for the
     migrated vdisk
5. Measure the hit ratio of the cache after each minute
```

Both caches are issued the same set of block I/O requests, but the second cache (where VM migrates to) will not see any request from the migrating vdisk until it is migrated. We issue the requests to both caches until 1034882 requests are served (the first 405 minutes). At this point we simulate that a vdisk is migrated from one node to another by sending over the prewarm set of this vdisk (in-memory bitmap of the vdisk maintained in the cache) to the second thread. The total size of objects in prewarm set of vdisk 1 just before migration was ≈1.26 MB

for 945 HM1 objects.

We do not save the cache state periodically and perform heuristic-based snapshot analysis in this experiment. Instead, we consider only the last known cache state of the vdisk (represented by its bitmap). In the second cache, we record the hit ratio after each minute when the vdisk is served from the cache, with and without prewarming. So, in both cases the second cache will see the block I/O requests for migrated vdisk from the point it was migrated, but only in the latter one we load the objects from its migrated bitmap. When the objects of migrating vdisk are loaded into the cache, some objects of other vdisks will be evicted.

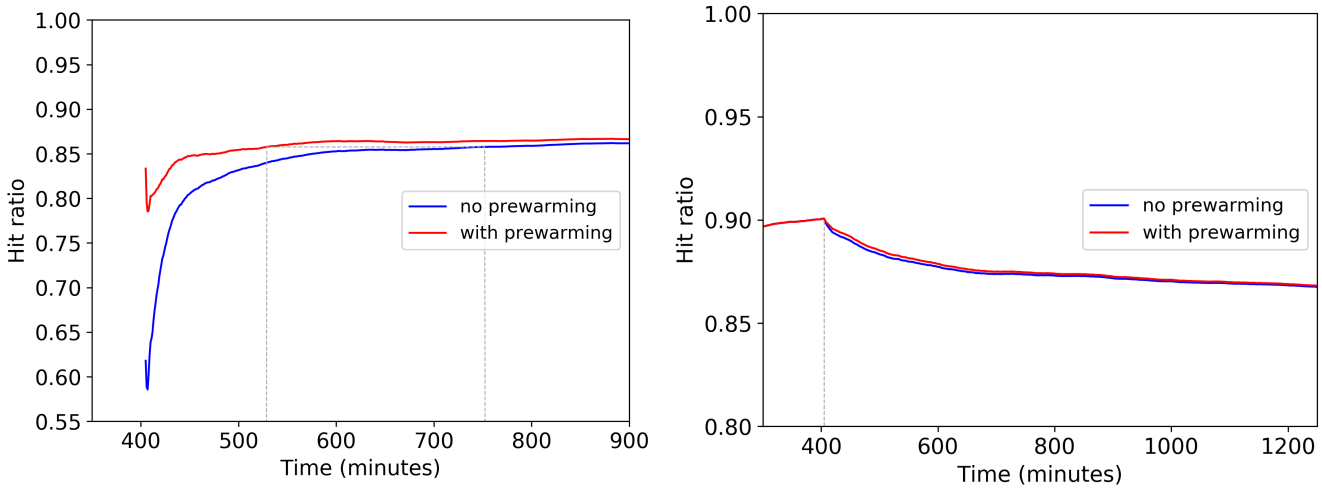

Figure 4.6: Exp 3: Impact of prewarming on hit ratio of the migrated vdisk (left) and the overall hit ratio of the destination node's cache (right)

We make the following observations from the graph plots shown above:

- From the plot on the left, we can infer that prewarming the cache with objects of a migrated vdisk helps mitigate the initial drop in the cache hit ratio for that vdisk. The warmup time (for achieving the P_{99} hit ratio) is was reduced to 124 minutes from 347 minutes, resulting in a speedup of 2.7x. The average gain in hit ratio for first 10 minutes was 0.173.

- In the plot on the right, we have the overall hit ratio of the cache on the destination node. The vertical grey-dotted line marks the point at which VM was migrated. There was no drop in the overall hit ratio when we loaded the new vdisks's objects into the cache. Instead, there was a very slight increase in the hit ratio due to a significant gain in the hit ratio of the migrated vdisk.

- We have also measured the impact of prewarming on the combined hit ratio of other vdisks served by the cache, and found that there was no significant drop in their hit ratio as well. This is mainly due to the fact that only 1.26 MB worth of cache objects were replaced with the migrated vdisk's objects.

5. Conclusion & Future Work

From the results obtained for the first two experiments, we can infer 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 (snapshot rate, heuristic, memory limit etc.) we use. In general, we see that the snaphotting the cache at 5 K, 10 K, and 50 K requests show the most promising results. Moreover, Constrained-k-Frerecent heuristic performs well in both of these experiment scenarios, but Constrained-k-Recent showed the most promising results in the second one.

In the first experiment, we were able to achieve the maximum speedup of 39x in reaching a hit ratio value of P_{99} using Constrained-k-Frerecent, where it took the prewarmed cache only 7 minutes to reach the hit ratio that we would get from an (initially) cold cache after 4.5 hours.

In the second experiment, we got a speedup of 55x in in reaching a hit ratio value of P_99 using Constrained-k-Recent, where it took the prewarmed cache only 5 minutes to reach the hit ratio that we would get from a cold cache 4.5 hours after the failure.

In the third experiment, we observed that prewarming a cache for migrated VM helps reduce the initial drop in hit ratio due to partially cold cache. Moreover, the other VMs are not significantly affected by prewarming action and the overall hit ratio does not reduce much as well.

While the current setup for these experiments yielded promising results, we would like to go into more depth and find ways to achieve better speedup values which are indicative of improved storage performance.

We had fixed a few parameters for all these experiments, and our next step will be to try varying some of them for our 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 instants at which failure/migration happens
- Score function for [Constrained-]k-Frerecent heuristics

We also need to issue a much larger set of block I/O requests which originate from more vdisks/VMs and resemble actual workloads running on VMs. For the first experiment, we need two sets of request traces which are recorded starting at the time when VMs boot up. Two sets of requests are needed so that after we reset the cache, we do not have to issue the same requests again.

Apart from expanding the parameter space, we also need to add the notion of time into the simulator, with disk and network accesses being penalized in units of time. Moreover, we need 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.