Improving Data Placement Decisions for Heterogeneous Clustered File Systems

Cyril Allen

A Thesis in the Field of Information Technology

for the Degree of Master of Liberal Arts in Extension Studies

Harvard University

March 2018

Abstract

With the advent of cloud computing, datacenters are using distributed applications more than ever. MapReduce is used to generate over 20 petabytes of data per day by using prodigious numbers of commodity servers (Dean & Ghemawat, 2008). Many companies use large scale clusters to perform various computational tasks via the open-source MapReduce implementation, Hadoop (White, 2012), or they can possess a virtualized datacenter, allowing them to migrate virtual machines between various machines for high-availability reasons. As economics change for hardware, it is likely that a scalable cloud will have the requirement to mix node types, which will lead to higher performance and higher capacity nodes to be mixed with lower performance, lower capacity nodes. This thesis presents an adaptive data placement method in the Nutanix distributed file system which will remedy some common problems found in many heterogeneous clustered file systems.

Acknowledgements

I doubt that I would be writing this document were it not for my father, Dr. Cyril Allen. Not only did he spark my initial interest in computing when I was a child, but also he convinced me to pursue a graduate degree.

Thanks are also due to my amazing mother, Dr. Hengameh Allen-Schaal, for her unconditional love and support. She has always been there for me when I needed her. I could not have done any of this without her.

I would like to thank my thesis director, Professor Jamie Frankel, for his patience and guidance through multiple drafts of this document. Working with him has been a remarkable learning experience.

None of the work in this thesis would have been possible without my employer, Nutanix. They have helped me tremendously by providing hardware resources and the chance to work on thought-provoking problems.

Contents

List of Figures

List of Tables

1 Introduction

The work in this thesis was performed in the context of a distributed file system designed by Nutanix for enterprise clusters. The file system is hosted on some number of virtual machine hosts (commonly referred to as hypervisors) connected in a full mesh network. Figure 1 shows an example full mesh network topology. This full mesh configuration allows for the clustering of virtual machines (VMs) that provide file system services by allowing each VM to send messages to any other VM.

The basic purpose of a file system is to store and retrieve data. Storage and retrieval of data is performed via write and read operations on the file system. A write operation in the Nutanix file system requires multiple copies, or replicas, of data to be written to disk on multiple physical servers distributed across a network partition. Given this, a write operation is not complete until all replicas of the data are written to disk. A write operation's performance is at the mercy of the slowest disk in the set chosen to host replicas. Variables that affect a disk's performance could include the hosting server's average CPU utilization and the number of operations already in flight on a chosen disk. To mitigate the negative impact these variables could have on a write, the file system should consider how these variables will affect a disk's performance before choosing to target it for a write operation. This results in biasing toward disks that will give better performance overall.

1.1 Nutanix and the Acropolis Base System

1.1.1 Architecture

A Nutanix cluster is facilitated by a clustering of controller virtual machines (CVMs) that reside, one per node, on each server in the cluster, as shown in Figure 2. CVMs work together to form a distributed system that provides an NFS (for VMware's ESXi

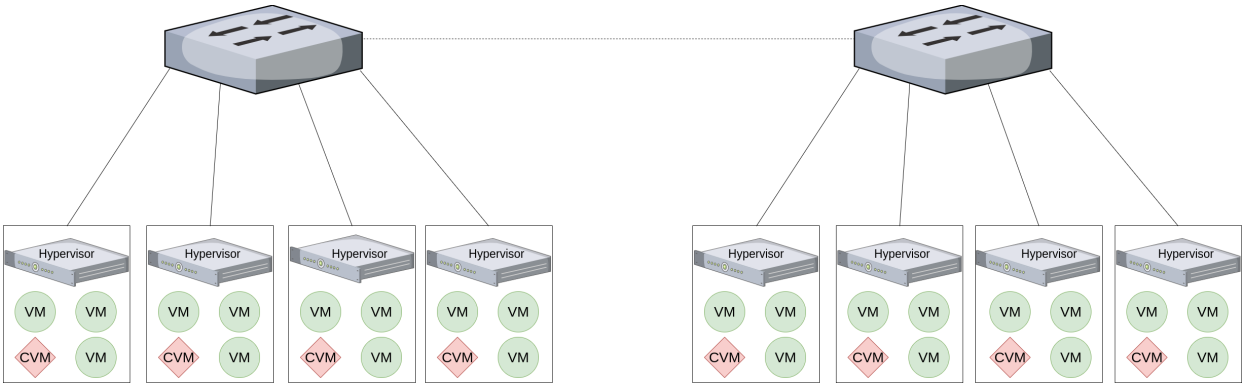

Figure 1: An example network for an 8-node Nutanix cluster.

(Chaubal, 2008)), SMB (for Microsoft's Hyper-V (Velte & Velte, 2009)), or iSCSI (for Nutanix's AHV (Poitras, n.d.)) interface to each hypervisor that they reside on. For example, the interface provided by the CVMs to VMware's ESXi hypervisor will be interfaced as an NFS mount, or in ESXi terminology, an NFS datastore. The virtual machines' virtual disk files will reside on the Nutanix datastore and be accessed via NFS through the CVM, thus sharing a host with the other hosted VMs.

Users of a Nutanix cluster will typically have virtual machines (VMs) hosted by one of the previously mentioned hypervisors. These VMs will have their storage requests forwarded to the local CVM that is hosted on the same node, or another CVM in the cluster if the local CVM is down. This fault tolerance is one of the advantages of using a clustered file system such as Nutanix as opposed to a monolithic storage array.

The CVMs expose some number of block devices, known as vDisks, that are used by the VMs hosted by the hypervisor. Within each CVM exists an ecosystem of processes that are responsible for the services provided by the Nutanix cluster. The work in this thesis is performed on the process that is directly responsible for disk I/O in the CVM ecosystem, Stargate.

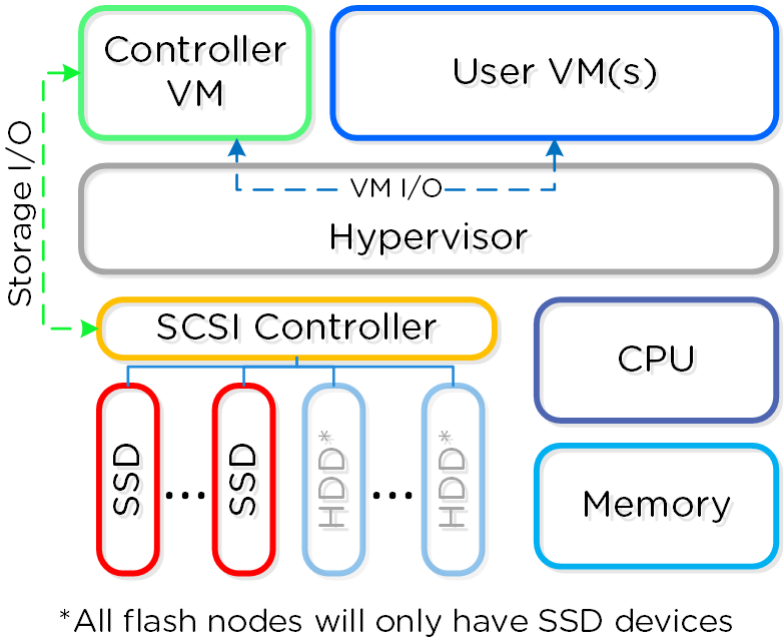

Figure 2: Nutanix node architecture diagram from the Nutanix Bible.

1.1.2 Stargate

The Stargate process is responsible for all data management and I/O operations. The messages sent via NFS/SMB/iSCSI from the hypervisor to the CVM are consumed and acted upon by Stargate. All file allocations and data replica placement decisions are made by this process.

As the Stargate process facilitates read/write I/O to physical disks, it gathers statistics for each disk such as the number of operations currently in flight on the disk (queue length), how much data in bytes currently resides on the disk, and the average time to complete an operation on the disk. These statistics are only gathered on the local disks; however, they are then stored via the cluster's Cassandra database service as a separate table. The Cassandra table that stores the statistics backs another service, Arithmos. The disk statistics stored in Arithmos are pulled periodically and are then used to make decisions on whether a disk has enough space available to make

it a write target. The work in this thesis uses other available statistics to make better data replica placement decisions when performing file system writes.

1.1.3 Cassandra

The Cassandra process stores and manages all cluster metadata in a distributed manner. It is also the distributed database used by Arithmos to store cluster statistics. The version of Cassandra that is running in NDFS is a heavily modified Apache Cassandra (Lakshman & Malik, 2010). One of the main differences between Nutanix Cassandra and Apache Cassandra is that Nutanix has implemented strict consistency via the Paxos (Lamport, 1998) family of protocols, which provide consensus in an unreliable network of compute nodes.

1.1.4 The Extent Store

The Extent Store is a sub-component of Stargate that serves as the persistent bulk storage. Data stored within the Extent Store are referred to as extent groups (also abbreviated as egroups), or 4MB pieces of physically contiguous data. An extent group is replicated a number of times dependent on the cluster replication factor, and each replica is placed on a different CVM except for a single replica that resides on the local node. The current replica selection algorithm will be explained further in section 1.1.6.

1.1.5 Storage Tiering and Data Replication

Data in the system is physically stored on disks of varying type. Within a given cluster, one can find NVMe, PCIe SSD, SATA SSD, and HDD drive types. These disks are separated into groups of drives with similar performance characteristics, referred to as storage tiers. For example, PCIe SSDs and SATA SSDs might belong

together in the same storage tier, but hard drives would not be included due to their relatively poor random access performance. Extent groups and their replicas are created on a particular storage tier, but might be migrated between tiers depending on data access patterns.

Data replicas for each extent group are written on separate fault domains on the same tier to ensure that, in the event a failure occurs and a replica becomes unavailable, that piece of data can still be accessed. An example of a fault domain can be a node (single CVM) or a block (a rackable unit of hardware containing multiple nodes).

1.1.6 Replica Selection

Stargate is inherently limited in its choices of disk to service reads because it must select a disk that contains the desired data. If the data is on a local disk, this disk is always selected by Stargate to avoid the unnecessary overhead of data traversal over the network when selecting a remote disk. However, when selecting disks to service writes, Stargate must decide whether to perform an in-place overwrite for writes over existing ranges of data or to write the data to a new extent group. In-place overwrites are similar to reads in that the choice of target disk is limited by where the relevant data resides. When Stargate enables a common data compression feature such as data deduplication, in-place overwrites are not performed because the metadata for multiple logical ranges of data point to the same physical data. When overwriting deduplicated data and all scenarios where data is being created for the first time, Stargate has the freedom to choose any disk in the cluster.

When a Stargate write operation selects appropriate disks for data placement upon entrance into the Extent Store, it is done on a per-tier basis. Table 1 shows the interface for a replica selection function call via the DetermineNewReplicas function.

Table 1: Argument descriptions for DetermineNewReplicas.

Argument Name	**Description**
`replica_vec`	A `std::vector` that is populated with the selected disk IDs. The number of selections made is the size of this vector.
`storage_tier`	The name of the storage tier (set of disks) to select for data placement.
`storage_pool_vec`	The collection of all disks in the cluster.
`preferred_node_vec`	Nodes Stargate prefers to choose disks from. This means it will consider disks from these nodes first and only consider other disks if the ones belonging to these nodes are not suitable.
`predicate_func`	A function that accepts a disk ID and returns a boolean. If this function returns false for any disk ID, it is not considered for selection.
`exclude_replica_vec`	Disk IDs that will be excluded from selection.
`exclude_node_vec`	Node IDs whose disks will be excluded from selection.
`exclude_rackable_unit_vec`	Rack IDs whose disks will be excluded from selection.

Upon calling DetermineNewReplicas, Stargate first attempts to select replicas from the preferred nodes. Because replicas must reside on the same tier, Stargate must first find the set of disks that belong to each node on the specified tier so that it may select a disk from that set. This is followed by a pseudorandom reordering of the recently found candidate disk vector, to prevent over-selection of the same set of disks, and sequentially evaluating each disk until a suitable one is found (the evaluation step). If a suitable disk is not found on a preferred node, all other disks belonging to the specified tier are shuffled and considered sequentially until enough disks are found to satisfy the requirement set by replica_vec.

A suitable disk is one that:

1. Contains enough space to accept new data. This is less than 95% full by default for Nutanix clusters.

2. Returns "true" when evaluated by the predicate_func.

3. Is not included in the exclude_replic_vec, exclude_node_vec, and exclude_rackable_unit_vec.

If a disk does not meet the criteria above, Stargate simply continues searching the shuffled set of disks belonging to the specified tier. If a disk is found to be suitable, Stargate must add the disk to the replic_vec and add the node that the disk belongs to in the exclude_node_vec. This prevents it from considering other disks on that node to maintain a node fault tolerance guaranteed by the cluster's replication factor.

1.2 Motivation

The current replica disk selection logic in Stargate does not take into account a number of variables such as disparities in tier size, CPU power, workloads, and disk health among other metrics. There are several scenarios, across both pathological and daily occurrences, where a more robust replica placement heuristic is required. The work in this thesis will focus on two orthogonal cases, which are described in the next section.

1.2.1 Interfering Workloads

As shown in Figure 3, interfering workloads can take the form of a homogeneous cluster of 3 servers in a full mesh network (also referred to as a 3 node cluster) with only 2 nodes hosting active workloads. In the current random selection scheme used by Nutanix clusters, writes in this example are equally likely to place their replica

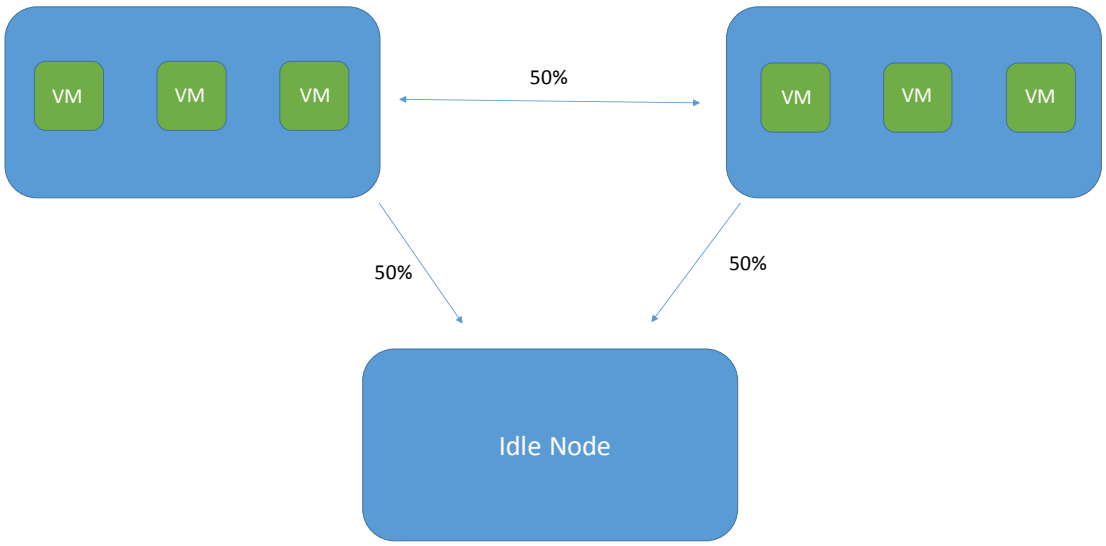

Figure 3: A cluster with identical nodes running a heterogeneous workload.

on the remote node with an active workload as they would be to place it on the remote node that is idle. This can impact performance via reduction of throughput or increasing latency on both the local and remote workloads because secondary writes will be slower on nodes whose resources are being used by their primary workloads. A data replica placement scheme is needed that adapts its behavior based on how idle the target nodes are. This would result in a bias toward nodes that are more idle.

1.2.2 Nodes with Tier Size Disparities

A cluster containing nodes with a tier size disparity are susceptible to a large skew in space utilization on the node, even if the workload on each node is identical. This is illustrated in Figure 4, which shows a 3-node heterogeneous cluster with 2 high-capacity nodes and a single low-capacity node. Suppose these high-capacity nodes have 500GB of SSD tier, 6TB of HDD tier, and the single weak node has only 128GB of SSD tier and 1TB of HDD tier. If 3 simultaneous workloads were to generate

data such that the working sets of these workloads are 50% of the local SSD tier, the weaker node is at a significant disadvantage. Given the replica selection algorithm in Nutanix cluster versions prior to 5.0, one can expect 500GB of replica traffic to flood the weak node and fill up its SSD tier well before the workload is able to write its data to the local disks. Because this results in an inability for the workload on the smaller node to place its primary replicas locally and forces the workload to rely on remote CVMs, latency is increased due to the network traversal to reach the remote nodes. An adaptive replica placement heuristic would mitigate this issue by taking disk usages into consideration during the placement of secondary replicas and bias placement of secondary replicas toward the nodes with more free capacity.

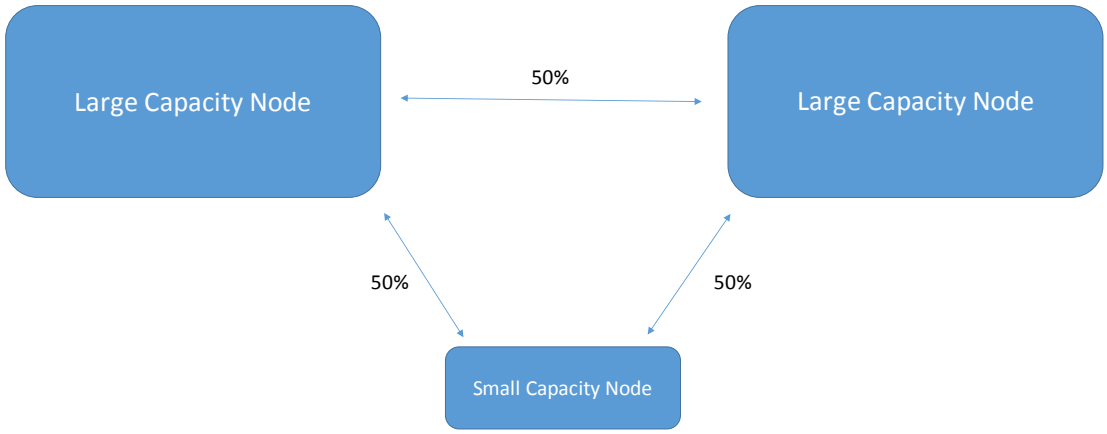

Figure 4: A cluster with nodes of varying resource capacity.

2 Prior Work

2.1 Clustered Storage Systems

Among the contemporary distributed storage systems, Apache's Hadoop Distributed File System (HDFS) (White, 2012) (Borthakur et al., 2008) is one of the most well-known. HDFS is an open source distributed file system written as a userspace library in Java that is used by Hadoop clusters for storing large volumes of data. An HDFS cluster is comprised of a single NameNode and multiple DataNodes, which respectively manage cluster metadata and store the data. HDFS stores files as a sequence of same-size 128MB blocks that are replicated across DataNodes to provide fault tolerance in the event of node failures. Decisions regarding the replication and placement of blocks are made by the NameNode. Since large HDFS clusters will span multiple racks, replica placement within HDFS is made in a rack-aware manner to improve data reliability and network utilization.

While HDFS is an open source distributed file system that is part of the greater Apache Hadoop ecosystem, the Google File System (GFS) (Ghemawat, Gobioff, & Leung, 2003) is a proprietary distributed file system developed by Google that inspired HDFS. Google's goal was to build a massive storage network from inexpensive commodity hardware (McKusick & Quinlan, 2010), so scalability, fault tolerance, and error detection were top-of-mind when designing the GFS.

A GFS cluster is comprised of a single Master node and multiple Chunkservers that store 64MB "chunks," or blocks, of data. Each file in the GFS is divided into the 64MB chunks, and each chunk is replicated by the Chunkservers multiple times throughout the GFS cluster. Master nodes store the metadata associated with the chunks such as chunk location or which processes are operating on a chunk. While the initial decision by the Google GFS team to have a single master node introduces a single point of

failure, this did not pose a problem at the time of initial implementation. The GFS was initially meant to handle data on the order of a few hundred Terabytes from the early web crawler.

As Google's data grew to the order of tens of Petabytes, the single master node became a bottleneck. The master node task of scanning metadata during chunk recovery scaled linearly with the amount of data, and each GFS client spoke to the master when performing a file open operation. Ultimately, Google took a "multi-cell" approach with GFS, which allowed for multiple GFS masters for a pool of chunk servers. Each GFS client application was then able to shard data across multiple cells, creating their own master node fault domains.

Both the HDFS and GFS architectures prioritize high sustained bandwidth over low I/O latency. One source of latencies in these architectures comes from the network partition between file system clients and the servers hosting the file system services. The Nutanix storage system differs from the HDFS and GFS in that the clients are hosted on the same servers as the file system services and data because the client VMs and CVM always share physical hardware. For client read requests, this means that there is no network traversal necessary to access data.

2.2 Data Placement Schemes in Clustered Storage Systems

Xie et al. (Xie et al., 2010) showed that data placement schemes based on the computing capacities of nodes in the HDFS significantly improved workload performance. These computing capacities are determined for each node in the cluster by profiling a target application leveraging the HDFS. Their MapReduced `wordcount` and `grep` results showed up to a 33.1% reduction in response time. Similarly, Perez, Garcia, Carretero, Calderon, and Sanchez (Perez, Garcia, Carretero, Calderon, & Sanchez, 2003) used adaptive data placement features that make use of data regarding the

available free space in their Expand parallel file system design. Though effective in their given contexts, the main drawback to this approach is that it assumes the specific application is working without interference and does not account for other workloads on the system.

One adaptive data placement approach that can account for other workloads on the system was introduced by Jin et al. (Jin, Yang, Sun, & Raicu, 2012) in their work on ADAPT. The study predicts how failure-prone a node in a MapReduce cluster is and uses the information to guide their availability-aware data placement algorithm to avoid scheduling work for those nodes. This proves useful for performance by avoiding faulty nodes that could fail mid-task and cause data transfers and re-calculation of data. Zaharia et al. (Zaharia, Konwinski, Joseph, Katz, & Stoica, 2008) perform similar predictive scheduling in their LATE scheduler for Hadoop, except that they estimate the finish time of tasks and schedule work that is expected to take longest to complete first in the pool of available nodes.

Work by Suresh, Canini, Schmid, and Feldmann (Suresh, Canini, Schmid, & Feldmann, 2015) approaches adaptive replica selection in much the same way as this thesis, though their work was mainly focused on decreasing tail latencies for Apache Cassandra (Lakshman & Malik, 2010) reads. Their load balancing algorithm, C3, incorporates the concept of a value calculated from request feedback from their servers. The value then guides decisions made regarding which servers to select for future requests. In addition to the ranking function in C3, they implemented a distributed rate control mechanism to prevent scenarios where many individual clients can bombard a single, highly-desirable server with requests. Many of the same problems that the work in this thesis seeks to remedy are addressed by the C3 algorithm, such as the reduction of I/O latency. However, the C3 algorithm is not a feasible solution when it comes to reducing each Nutanix node's disk utilzation skew relative to the

other nodes in the cluster.

The C3 algorithm considers the request queue length of certain servers similar to the way I will use disk queue lengths. In addition, service latencies of each server are factored in to the decision making process so that different ideal queue lengths are considered for each server. With this approach, longer service times will warrant a lower queue length and vice versa. This is beneficial for scenarios where multiple underlying storage technologies such as NVMe drives, SSDs, and HDDs are under consideration, but the Nutanix file system's architecture does not allow for multiple replicas to span storage tiers. This forces the ideal queue lengths for each selection pool to be the same. Therefore, this thesis does not incorporate service latencies in the fitness value calculations discussed in section 3.1.

Herding of requests to a single highly suitable server is a problem that arises in any replica ranking algorithm. C3 mitigates this issue by rate-limiting client requests to each server by a decentralized calculation by using a configured time interval and local sending rate information. I've opted to use a simpler probabilistic spreading of client requests via selecting remote nodes using weighted random selection based on the calculated fitness values. This fitness value approach was inspired by Cisco's Enhanced Interior Gateway Routing Protocol (EIGRP) (Albrightson, Garcia-Luna-Aceves, & Boyle, 1994) and its composite routing metric calculations, which is discussed further in section 2.4.

2.3 Non-Clustered Data Replication via RAID

So far, I have only discussed distributed file systems and data replica placement schemes in a clustered/distributed context. It is worth noting that not all storage replication occurs across a network partition. For example, it is common to see Redundant Arrays of Inexpensive Disks (RAID) (Patterson, Gibson, & Katz, 1988)

(Chen, Lee, Gibson, Katz, & Patterson, 1994) used on servers to increase performance or to protect data written to the disks in the event of a drive failure.

Table 2: Common RAID levels and their descriptions.

RAID Level	**Description**
RAID 0	Disk striping with no replication.
RAID 1	Disk mirroring.
RAID 5	Block-level striping with distributed parity.
RAID 6	Block-level striping with doubly distributed parity.

RAID combines multiple physical disks into a single logical disk that is then presented to the operating system as a single device. The exact data replication scheme that dictates the way data is distributed across the drives is known as the RAID level. Table 2 shows some of the standard RAID levels.

2.4 EIGRP and the Inspiration for Fitness Values

EIGRP (Albrightson et al., 1994) is Cisco's proprietary distance vector (McQuillan, Richer, Rosen, & Bertsekas, 1978) routing protocol and enhancement to their Interior Gateway Routing Protocol (IGRP) (Hedrick & Bosack, 1991). In EIGRP, each router keeps information about its adjacent neighbors, containing the address and interface of the connected neighbor. This information is needed because of the Topology Table (also referred to as the Distance Table) which contains the distance, bundled with an associated metric, to all destinations advertised by the adjacent neighbors. The metrics across all paths to the destination are stored in the distance table and are used to calculate the minimum cost path to a particular destination.

$$Term_1 = K_1 * \frac{K_2 * Bandwidth}{Bandwidth + 256 - Load} + K_3 * Delay \tag{1}$$

$$Metric = 256 * Term_1 * \frac{K_5}{K_4 + Reliability} \tag{2}$$

Equation 2 shows the formula used by Cisco routers to calculate the distance metric. Note that as bandwidth increases, the metric is increased; however, as load increases on the links, the metric decreases. There are constant values, also known as K-values, that allow network engineers to modify the behavior of the network based on their preferences. This idea of taking desirable traits and using them to calculate a metric, or desirability value, can be applied to the problem of choosing the best disk for data replica placement. This concept will be explored further in the following sections within the context of the Nutanix distributed file system.

3 Implementation

3.1 Fitness Values and Functions

To calculate a value to represent the desirability of a disk for replica placement, a function, $f_{fitness}$, is defined. The function $f_{fitness}$ takes as its argument some number of variables that can affect disk performance and returns a positive number henceforth referred to as a fitness value.

A low fitness value indicates a poor placement candidacy for a disk, and a high fitness value will indicate a highly desirable disk for replica placement. In this work, I evaluate one fitness function of a disk's average queue length, t_q, and one fitness function of a disk's percentage utilization, t_u.

$$t_q = 1 - \frac{q}{q_{ceil}} \tag{3}$$

The variable $q_{ceiling}$ holds the maximum observed queue length. Increasing t_q from 0 toward $q_{ceiling}$ decreases the fitness value, but increasing t_q beyond $q_{ceiling}$ does not continue to modify the fitness value to drop below 0. This ensures that as the queue length grows, t_q approaches 0.

$$t_u = \frac{1}{a^u} \tag{4}$$

The variable u represents disk utilization percentage and a is an aggression variable used to control the exponential decay of t_u. The larger a is, the more aggressively t_u will decay as u increases. An aggressive decay results in more preference given to less utilized disks when compared with disks that are slightly more utilized.

These terms are applied in two different fitness functions evaluated in this document:

$$f_{add} = t_u + t_q \tag{5}$$

$$f_{mult} = t_u t_q \tag{6}$$

3.2 Collection of Stargate Disk Statistics

Prior to this work, Stargate's periodic disk statistics collection was limited to solely caching disk usage statistics for all disks in the cluster. This has been expanded to now include disk performance statistics for use in disk fitness values.

Stargate maintains a mapping, the *disk_map_*, from cluster disk ID to a disk state structure. The *disk_map_*'s state structure contains disk usage and performance information that has been published to Arithmos by other Stargates in the cluster. Upon gathering fresh statistics from Arithmos, the information is used to create a disk fitness value for each disk in the cluster.

3.3 Weighted Random Selection Algorithms

After a weight is calculated for a disk in the cluster that will store a replica, the `WeightedVector` class' `Sample()` calls will perform a weighted random selection on the set of potential candidate disks. To determine the best method of weighted random selection for Stargate's `WeightedVector` class, an exploration of various weighted random selection algorithms was necessary. Because the file system only supports replication factors of 2 or 3, the investigation was limited to algorithms that allow for a weighted $\binom{N}{K}$, where K is the data replication factor.

3.3.1 Scalability Simulation Methodology

To test the scalability of the weighted random selection algorithms that are evaluated in the next section, a single-threaded Python (Van Rossum et al., 2007) script was written to evaluate the change in run time as sample sets increase. Each algorithm's mean running time is calculated for multiple sample sets. Code for the simulations is as follows:

```
def gen_runtimes(selection_func, iteration_count):
  runtimes = []
  std_devs = []
  # For each pool size.
  for pool_size in num_disks_for_tests:
      print("Measuring with pool size " + str(pool_size))
      runtime_list = []
      # Make a list of pool_size random numbers between 1 and 10.
      lst = pool_size * [1]
      # Time selections and get the average runimes.
      for it in range(iteration_count):
          start_time = time.time()
          selection_func(lst)
          elapsed_time = time.time() - start_time
          runtime_list.append(elapsed_time)
      avg_runtime = np.average(runtime_list)
      std_dev = np.std(runtime_list)
      runtimes.append(avg_runtime)
      std_devs.append(std_dev)
  return runtimes, std_devs
```

Object weights are constant throughout the simulation, so selection schemes that require some amount of preprocessing (such as the top T% calculation for truncation selection) perform their preprocessing steps for each weighted random selection. This provides information about the worst-case run time behavior for each algorithm in comparison with the worst-case runtime of the others.

3.3.2 Herding Behavior Evaluation Methodology

Herding behaviors can be seen in some weighted random selection algorithms when the weights of a subset of objects in the sampling pool cause a disproportionate amount of selections to target those objects. In the case of replica disk selections in a Nutanix cluster, this can cause too many operations to target an especially suitable disk, resulting in poor performance. It is possible to simulate an exaggerated scenario in which the susceptibility to herding behavior can be observed by having a single object with a weight that is multiple orders of magnitude heavier than the next highest object in the sampling set. It is also necessary to observe any herding behavior for a sampling set with low weight skew. This section describes the simulation methodology for each.

High-skew sampling sets of 11 objects were generated such that the array index of the first 10 objects was assigned as the object weight, and the last object was given a weight of 1000. This creates an extremely large skew in weights and makes the high-weight object a target for herding behavior. Given this sampling set, weighted random selections were performed and a histogram was kept that tracked the number of selections for each object. Iterations of 10^3, 10^4, and 10^5 were performed to observe any changes in herding behavior at larger time scales. In addition to the high-skew sampling sets, low-skew sets of 100 objects were also simulated. These low-skew sets were generated such that the array index of the first 10 objects was assigned as the object weight.

3.3.3 Stochastic Universal Sampling (SUS) Simulations

SUS is another sampling technique first introduced by James Baker in 1987 (Baker, 1987). The algorithm can be understood as follows: On a standard roulette wheel, there's a single pointer that indicates the winner. The roulette wheel's "bins" can all

be the same size, which would indicate a uniform probability of selecting any bin, and they could also be unevenly sized, which would indicate a weighted probability. SUS uses this same concept, except it allows for N evenly-spaced pointers that correspond to the selection of N items. Key factors to note are that the set, or "bins" in my roulette analogy, must be shuffled prior to selection. Also, a minimum spacing is allowed for the pointers to prevent selection of the same bin.

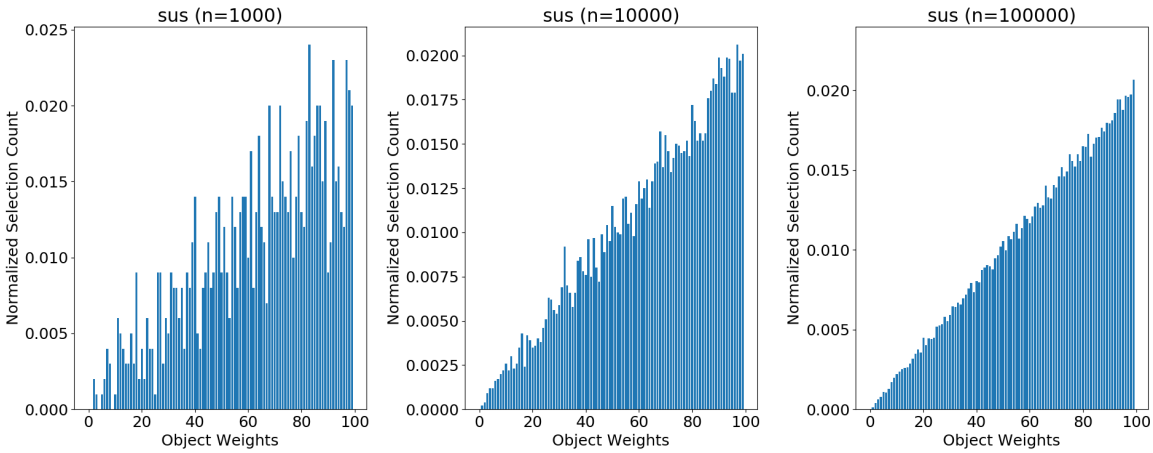

Figure 5: Histograms generated by simulation of Stochastic Universal Sampling to illustrate selection distributions.

Figure 5 shows the evolution of the distribution of selection frequencies for SUS as the number of samples increases. It is evident that the selection frequency is proportional to the object weight. This can prove problematic for outlier objects with weights that are much larger than the other objects in the set, as shown in Figure 6. The high weight object's selection frequency eclipses all other objects in the selection pool, which can lead to extreme herding behaviors.

3.3.4 Truncation Selection Simulations

Truncation selection (Crow & Kimura, 1979), typically used in evolutionary computing algorithms, does not consider any objects for selection below some threshold,

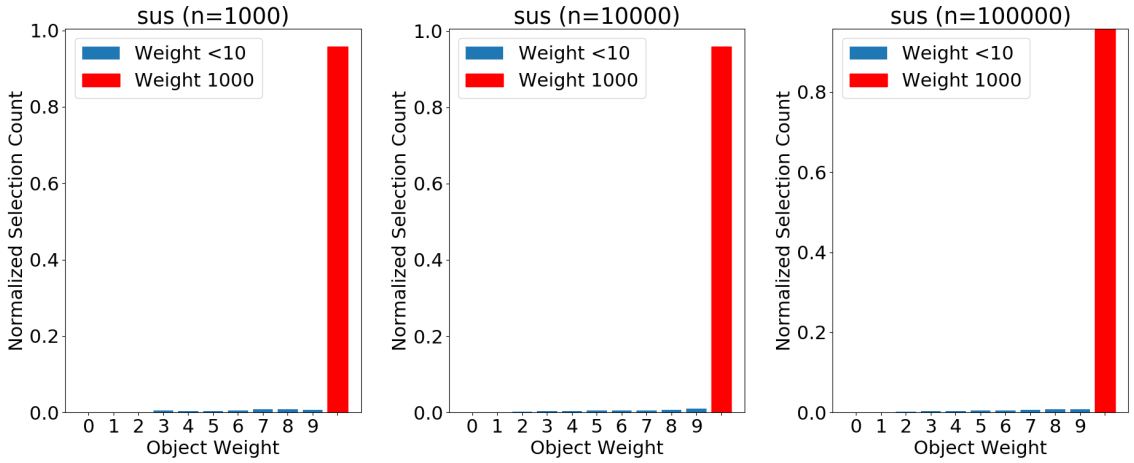

Figure 6: Histograms generated by simulation of Stochastic Universal Sampling to illustrate herding behavior.

T.

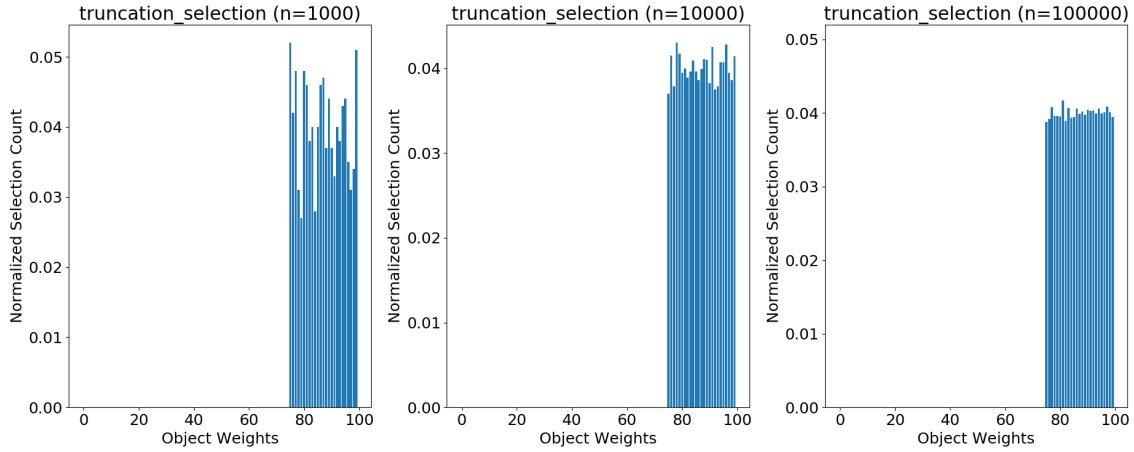

Figure 7: Histograms generated by simulation of truncation selection to illustrate selection distributions.

In Figure 7, only the top 10% of objects ranked by weight are considered for selection. Within this subset, weight has no meaning so there is a uniform distribution of selections. However, this might not be suitable for use cases where all objects must be candidates for selection. Depending on the threshold chosen, this algorithm can

be resistant to herding behavior, as shown in Figure 8.

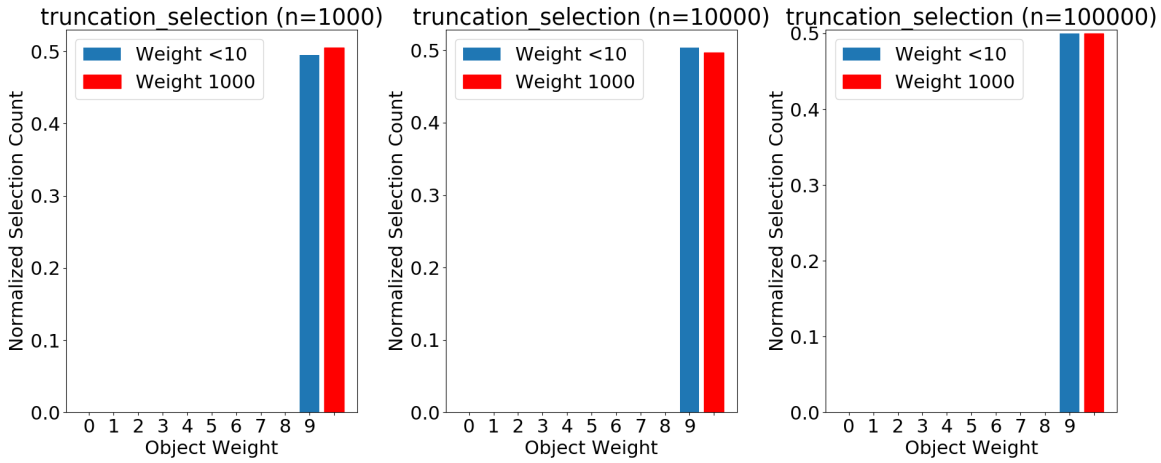

Figure 8: Histograms generated by simulation of truncation selection to illustrate herding behavior.

3.3.5 Two-Choice Sampling Simulations

Two-choice sampling (Azar, Broder, Karlin, & Upfal, 1999) (Mitzenmacher, 2001) performs a uniform random selection twice and returns the object with higher weight. This algorithm has proven to be extremely resilient to herding behavior, as shown in Figure 10, and selection frequencies for all objects are influenced by object weights in a way similar to SUS. While it is more likely an object with a higher weight will be selected, it is not selected with a probability proportional to its fitness value. This makes the algorithm resistant to any herding behaviors, but it will not be a good candidate if the desire is for the `WeightedVector` to select objects with probability proportional to its weight.

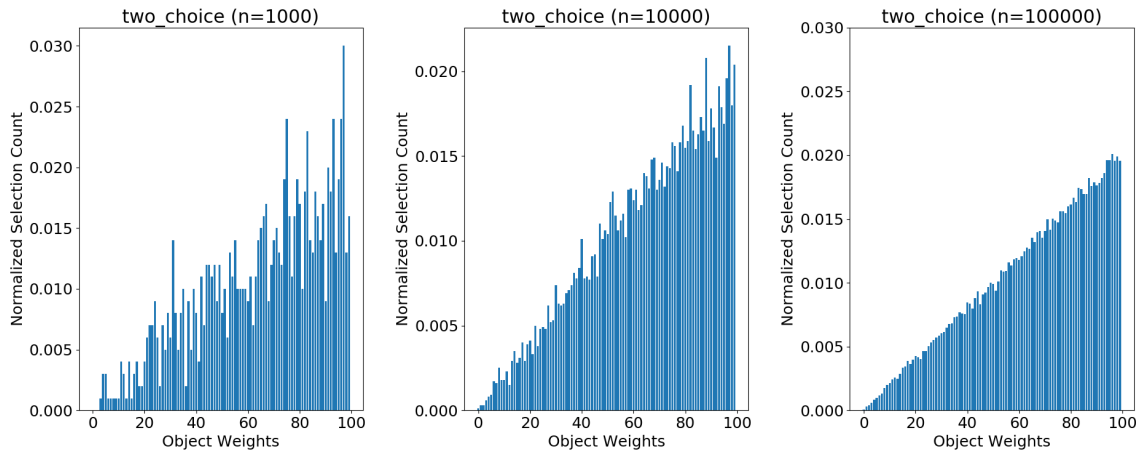

Figure 9: Histograms generated by simulation of two-choice selection to illustrate selection distributions.

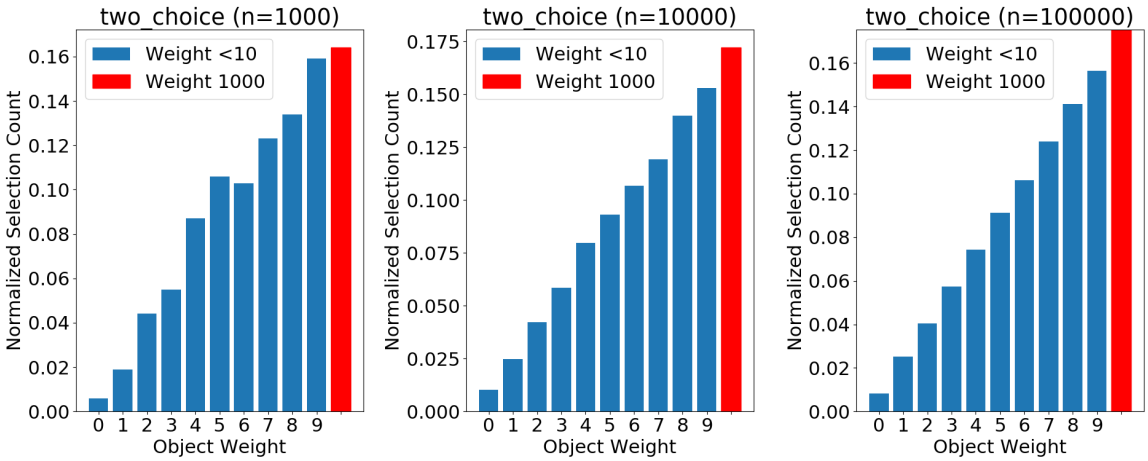

Figure 10: Histograms generated by simulation of two-choice selection to illustrate herding behavior.

3.3.6 Uniform Random Selection Simulations

Uniform random selection is completely indifferent to object weights. Therefore, it exhibits no herding behavior and no usefulness for the `WeightedVector`'s sampling, but including its analysis for comparison with other selection algorithms is important.

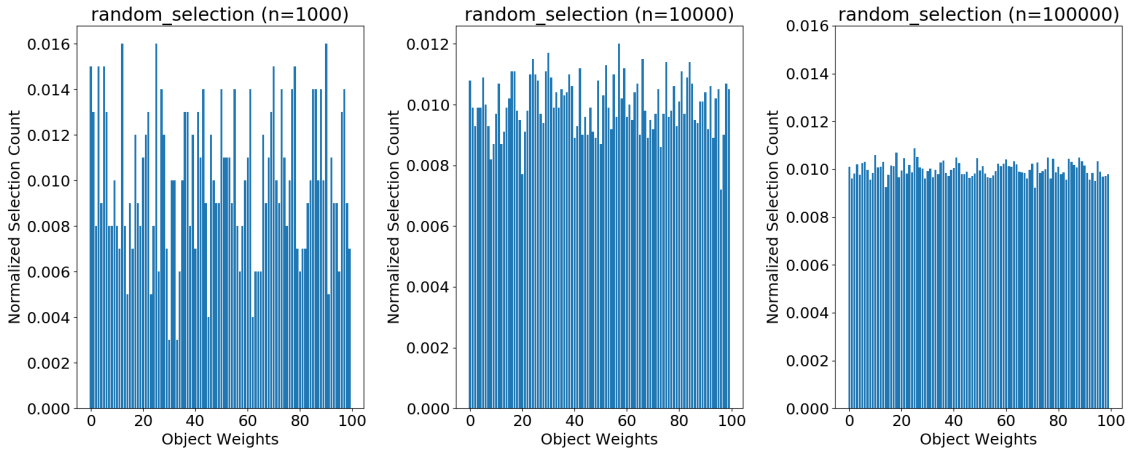

Figure 11: Histograms generated by simulation of uniform random selection to illustrate selection distributions.

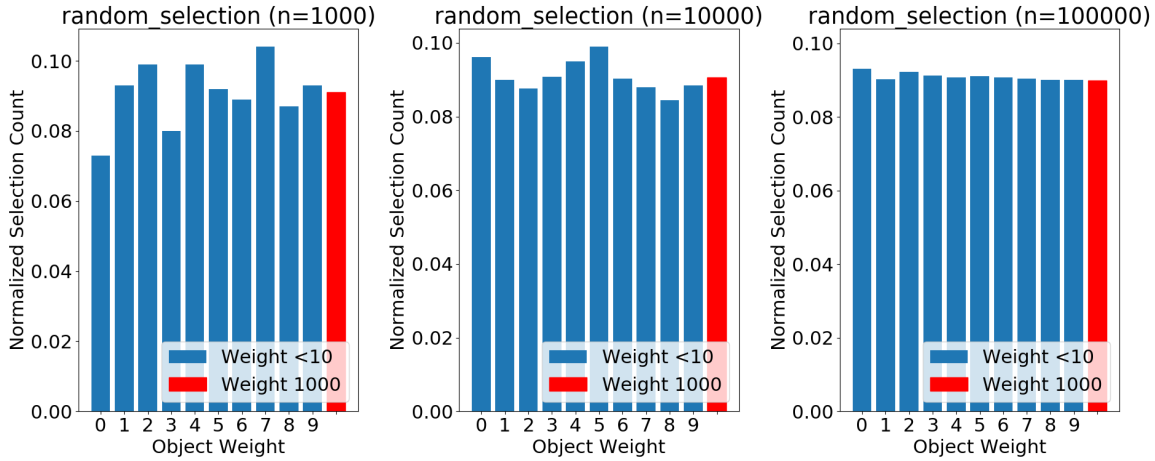

Figure 12: Histograms generated by simulation of uniform random selection to illustrate herding behavior.

Figure 8 shows that uniform selection probabilities can be expected, whereas Fig-

ure 12 confirms there is no herding behavior with a uniform random selection scheme.

3.3.7 Weighted Random Algorithm Scalability Analysis

Even though both SUS and truncation selection have $O(N * D)$ time complexity, where D is the number of objects in the set, and N is the number of selections required in one iteration, truncation selection is slower than SUS by an order of magnitude. This is mainly caused by the need for the truncation selection algorithm to calculate the top T% of the set for every selection performed in the simulations, an $O(Dlog(D))$ operation. As expected, two-choice and random selections are observed to be constant-time algorithms due to its simplicity of two uniform random selections. Two-choice is slightly slower than random selection because of the second uniform random selection and comparison operation that must occur before selecting an object.

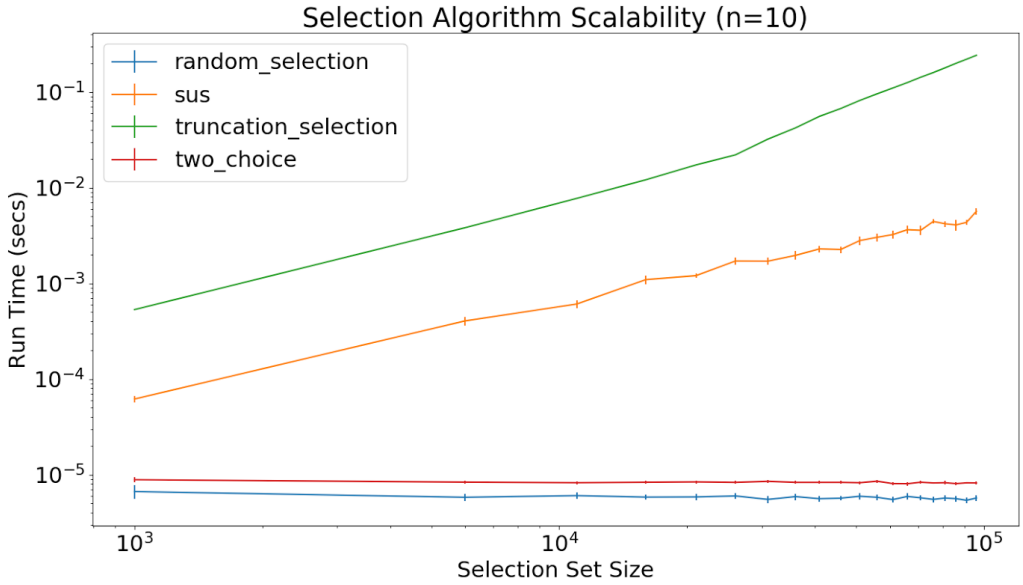

Figure 13: Running times of various weight random selection algorithms.

Figure 13 shows the running times on an exponential scale for each of the weighted

random selection algorithms discussed. It becomes clear that weighted random selections for large sets of objects is most efficient when using a two-choice sampling technique.

3.4 Weighted Vector Class

It is cleaner to implement repeated weighted sampling of disk IDs in such a way that we call a single function to return a set of unique disk IDs from a sampling set by using a weighted random sampling algorithm that uses the disks' fitness values. An object container class is needed that is similar to the data structures in the C++ Standard Template Library (Josuttis, 2012). A requirement for this object is that it must be able to access elements of arbitrary types via arbitrary weighted random sampling methods. The `WeightedVector` class was implemented to fulfill this requirement.

Upon instantiation of the `WeightedVector` class, a fitness function is provided to store the fitness values of all objects internally. The `WeightedVector` class has implemented insertion, removal, and weighted random sampling of objects of some arbitrary type, T, based on the object's fitness value as a weight.

The set of objects and their fitness values are stored in `std::vector` objects, one for the object references stored in the `WeightedVector`, and one for their corresponding fitness values. The objects and their corresponding fitness values are stored in such a way that the object at some element in the object vector has a fitness value at the same element in the fitness value vector.

Table 3 shows the `WeightedVector` public interface.

Table 3: WeightedVector public interface.

`WeightedVector(const std::function<double(const T)>)`
The `WeightedVector` class is instantiated by providing a fitness function that accepts a reference to an object of type T and returns a fitness value that is of type *double*. The provided fitness function is used to create a mapping between inserted objects and their corresponding fitness values for sampling. The `WeightedVector` class asserts that the fitness function returns positive values.
`void EmplaceBack(const T& element)`
The EmplaceBack function constructs and inserts an object of type T at the end of the vector. This increases the WeightedVector's size by 1.
`bool Empty()`
Returns true if the `WeightedVector` is of size 0.
`void Clear()`
Resets all state in the `WeightedVector` object, sets the size to 0, and clears all internal vectors.
`T& Sample()`
Returns a reference to an object stored inside of the WeightedVector. Objects are sampled via weighted random selection and subsequent calls to Sample are guaranteed not to return the same object unless Reset is called before sampling again. The `WeightedVector` class asserts that Sample is not called more times than the size of the WeightedVector.
`void Reset()`
Resets the sampling state of the WeightedVector, allowing sampled objects to be eligible for selection in subsequent Sample calls.
`size_t size()`
Returns the size of the WeightedVector.

3.4.1 Weighted Vector Internals

Internally, the `WeightedVector` class keeps 3 different std:vector objects to keep track of sampling state and the object-to-fitness value mapping. These variables are the `inner_vector_` variable that stores the objects inserted via `EmplaceBack()`, the `weights_` variable that stores the fitness values of all objects in the `inner_vector_`, and the `sampling_weights_` variable, which is identical to `weights_` except that objects' weights are set to 0 when sampled. This allows users to exclude objects from being sampled multiple times between calls to `Reset()` with time complexity $O(1)$, because a weight of 0 gives a probability of 0 for sampling using Stochastic Universal Sampling, as exemplified in Table 4.

Before sampling, there are identical `weights_` and `sampling_weights_` vectors accompanying an `inner_vector_` of objects.

Table 4: Inner vector example part 1.

`inner_vector_`	A	B	C	D
`weights_`	1	3	3	7
`sampling_weights_`	1	3	3	7

Suppose that a call to Sample() on the `WeightedVector` returns D. The probability of this event is 0.5, so to maintain sampling state within the `WeightedVector`, the weight associated with object D is set to 0 in Table 5.

Table 5: Inner vector example part 2.

`inner_vector_`	A	B	C	D
`weights_`	1	3	3	7
`sampling_weights_`	1	3	3	0

If `Sample()` is called again, it is not possible to return D because its sampling

weight is now 0. The probabilities of returning A, B, or C in subsequent calls to `Sample()` are $\frac{1}{7}$, $\frac{3}{7}$, and $\frac{3}{7}$, respectively. Suppose two more calls to `Sample()` lead to the state in Table 6.

Table 6: Inner vector example part 3.

`inner_vector_`	A	B	C	D
`weights_`	1	3	3	7
`sampling_weights_`	0	3	0	0

There can only be one more call to `Sample()` without triggering an assertion failure and the `WeightedVector` is obligated to return B. The only way to restore sampling state is to call `Reset()`. A `Reset()` call will simply copy the values from `weights_` into `sampling_weights_`, restoring all original weights and making all objects eligible for sampling again, as shown in Table 7.

Table 7: Inner vector example part 3.

`inner_vector_`	A	B	C	D
`weights_`	1	3	3	7
`sampling_weights_`	1	3	3	7

Note that the execution of `Sample()` and `Reset()` allows the user to perform a weighted $\binom{N}{K}$ by simply making K calls to `Sample()` before resetting. This is required for the Stargate use case of selecting multiple unique disks from a storage tier to host data replicas.

3.4.2 Weighted Vector Unit Testing

The `WeightedVector` class' unit test has four phases:

1. Test Average() functionality
2. Test that there are no duplicate samples
3. Test sampling with uniform probabilities
4. Test sampling with non-uniform probabilities

The testing of the Average() function's behavior includes simply adding all zeros, all ones, and monotonically increasing integers in the range $[0, N]$. For each phase, it can be verified that the reported average is 0, 1, and $\frac{N(N-1)}{2N}$ respectively.

Verification that there are no duplicate samples involves adding monotonically increasing integers in the range $[0, N]$, inserting all sampled elements into a hash set, and verifying that the size of the hash set is equal to the size of the WeightedVector.

To test sampling of objects with uniform and non-uniform weights, I added monotonically increasing integers in the range $[0, N]$ to the WeightedVector. The fitness function that was provided for the uniform test simply returns a weight of 1 for all objects inserted into the WeightedVector. Elements are then sampled and Reset() is called for a number of times that is several orders of magnitude larger than the number of elements. The number of times that each sampled element is being selected is tracked in a test-local hash map. The actual selection probability is calculated for each element in the `WeightedVector` with the expected value, and the difference is verified to be within an acceptable tolerance. For the uniform test, all integers are expected to be sampled roughly the same amount; for the non-uniform test, larger integers are expected to be sampled an amount of times proportional to their value.

3.5 Replica Selection Changes

Stargate was modified to store disk IDs in a `WeightedVector` rather than a `std::vector`. When considering a disk for candidacy, rather than shuffling the `std::vector` and sequentially evaluating each disk until enough replica targets are found, calls to `WeightedVector::Sample()` must occur to get the next disk ID so that it can be evaluated. Calls to `WeightedVector::Sample()` will stop when either enough target disks have been selected, or when all candidate disks in the set have been evaluated and found unsuitable. All of Stargate's replica placement logic is unmodified and only the order in which candidate disks are considered has been changed. This results in Stargate tending to evaluate the higher fitness disks first and selecting those disks for data placement if they are eligible.

4 Evaluation and Results

The following experiments seek to measure the effects of the different combinations of the tier utilization and queue length variables in the fitness functions. The two functions differ in their approach by either adding the terms together (Equation 5) or multiplying the terms together (Equation 6).

4.1 Experimental Setup

The replica selection schemes were evaluated using a 3-node NX-1350 cluster. Each node contains a single 300GB SSD and 4 HDDs 1TB in size. When evaluating the new replica disk selection framework, two heterogeneous workload scenarios were tested:

1. Two worker VMs on separate nodes running a workload with low outstanding ops.

2. Two worker VMs on separate nodes with running a workload with high outstanding ops.

The worker VMs were manually deployed to their respective host hypervisors and used to perform the experiments described in sections 4.3 and 4.4.

4.2 Fio and Write Patterns

When generating I/O in these experiments, Fio was used on the worker VMs. Fio, short for Flexible IO, is an I/O workload generator that can take configuration files to specify the parameters of a test. On each worker VM, Fio was used to generate a sequential write workload that completely fills the cluster's hot-tier. Sequential writes were chosen for all tests because they are the default write pattern for Fio tests and the purpose of these experiments is to generate new data replicas in a consistent

manner. For the purposes of replica placement, the Nutanix file system does not distinguish targets based on the write pattern that generated an extent group.

4.3 Tier Utilization Experiments

The tier utilization experiments define the hot-tier deviation, $d_{hot\ tier}$. The hot-tier deviation, $d_{hot\ tier}$, is represented as the average SSD utilization percentage of the nodes that are running a workload, u_w, subtracted by the SSD utilization percentage of the node without a workload, u_o:

$$d_{hot\ tier} = \frac{u_{w1} + u_{w2}}{u_o} \tag{7}$$

Ideally, the idle node would absorb the majority of secondary replicas from the running workloads. However, uniform random selection causes only 50% of secondary replicas to go to the idle node even though it can potentially handle more work because it does not have to service a local workload. In a uniform random replica selection scheme, it is expected that the nodes that are running a workload have to bear 100% of their own primary replicas and 50% of secondary replicas from the other worker node. This causes the SSD utilization for each node to be skewed. More specifically, the nodes with a local workload will have higher SSD utilizations, which will be expected to grow as the tests run. As the cluster SSD utilization skew increases, higher $d_{hot\ tier}$ values will be observed. A more sophisticated replica selection scheme is expected to minimize $d_{hot\ tier}$ by biasing secondary replicas toward the idle node and limiting the skew.

4.3.1 Low Outstanding Operation Results

Figure 14 shows the results of a workload with only a single outstanding operation. This causes the queue length reported by Stargate to be at most 1, resulting in the fitness function's queue length term to be roughly constant and approximately 1.

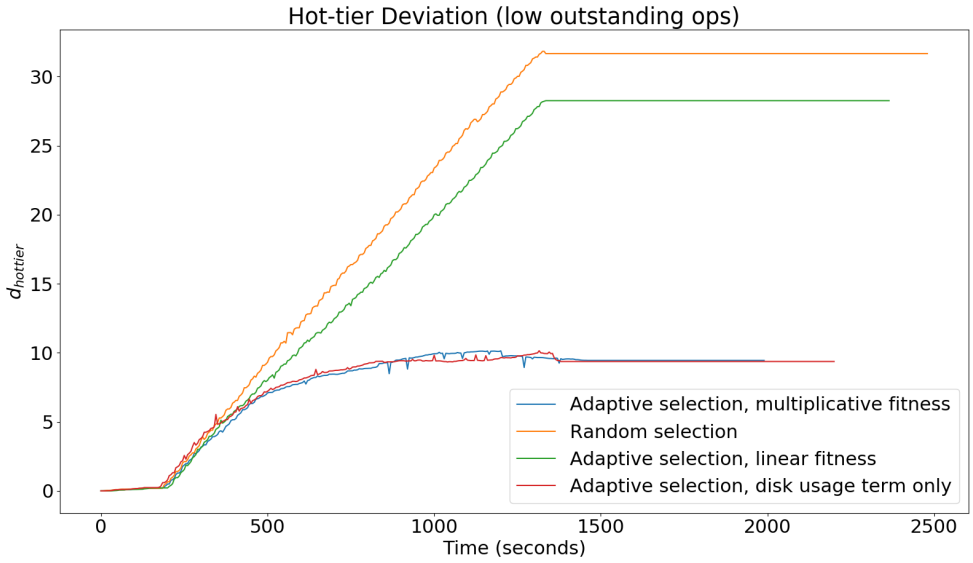

Figure 14: $d_{hot\ tier}$ values over time for low outstanding I/O operations.

It is evident that the additive fitness function does not minimize the hot-tier deviation and the multiplicative because the additive fitness function's behavior varies depending on the weight chosen for the queue length term. By default, the linear fitness function gives equal weight to the disk fullness and queue length terms; however, for one run of this experiment, the queue length term was given no weight. Linear fitness that gives equal weight to both terms does not reduce skew by much, while giving no weight to the queue length term keeps all nodes' SSD usages within 10% of each other. A linear fitness function does a poor job of reducing $d_{hot\ tier}$ values because the queue length term is contributing the maximum amount possible to the fitness value due to the consistently low queue length values. This can be illustrated

by defining a disk selection bias, b_r, as the probability some disk, d, will be selected when compared with another disk, d', whose utilization is 10% higher and queue length value is identical. Given a fitness function, f, b_r can be calculated as follows:

$$b_r = \frac{f(d)}{f(d) + f(d')} \tag{8}$$

Figures 15 and 16 show the disk selection biases for large and small queue lengths, respectively.

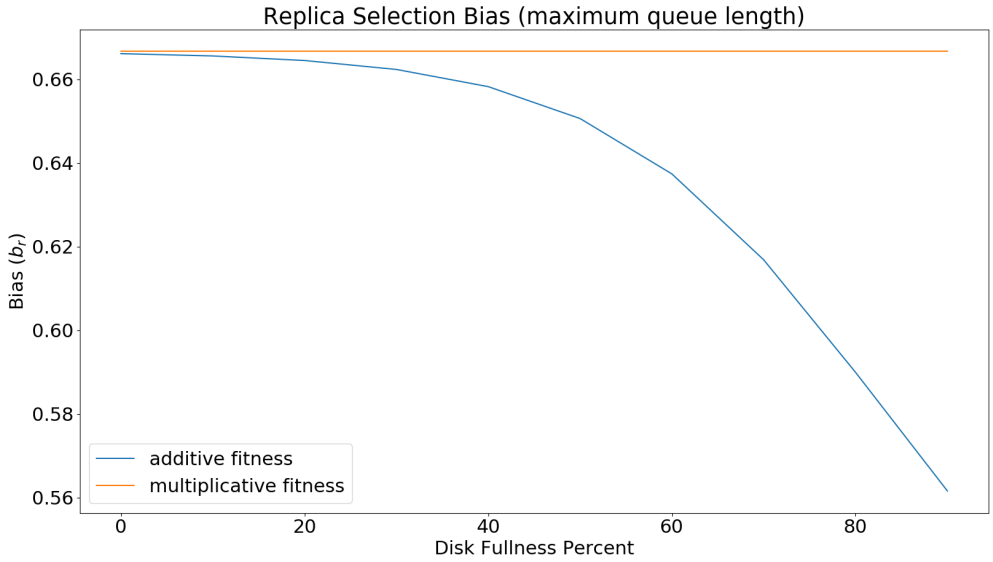

Figure 15: b_r values with static queue lengths at the fitness function ceiling values.

For an additive fitness function, the bias toward a less utilized disk decreases as the disk fullness percentages for d and d' increase, even though they still only differ by 10% in Figures 15 and 16. The entire linear fitness function does not scale with each term, so the multiplicative fitness function is superior.

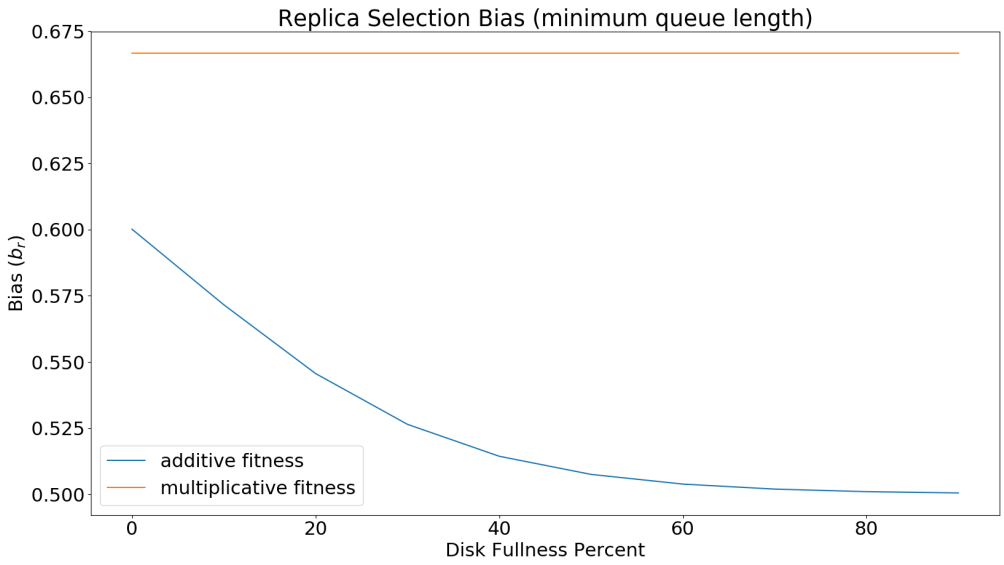

Figure 16: b_r values with static queue lengths at 1.

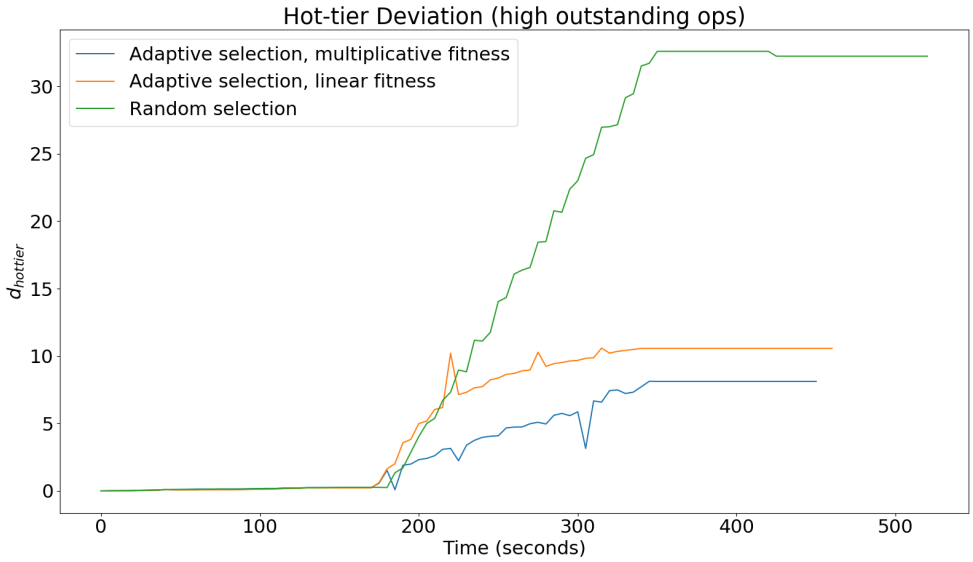

Figure 17: $d_{hot\ tier}$ values over time for low outstanding I/O operations.

4.3.2 High Outstanding Operation Results

Figure 17 shows that both additive and multiplicative fitness functions reduce the disk fullness skew from 30% to less than 10%. Multiplicative fitness performs slightly better at minimizing $d_{hot\ tier}$ than additive fitness, possibly due to scaling the fitness value by the value of both the fullness and queue length terms, rather than weights.

4.4 Disk Queue Length Experiments

Due to the nature of low outstanding I/O workloads, disk queue length values would remain low for the duration of an experiment. This does not provide useful information for measuring the effects of fitness-based replica selection on disk queue lengths. The experiments for high outstanding I/O experiments in section 4.3.2 were re-run for all fitness function types and for fitness function queue length term ceilings of 200 and 100. Figures 18 and 19 show a reduction in queue length quartiles when fitness-based selection is used for disks on nodes that host local workloads. Lower queue length ceilings were observed to provide better results in reducing the queue lengths for the worker nodes.

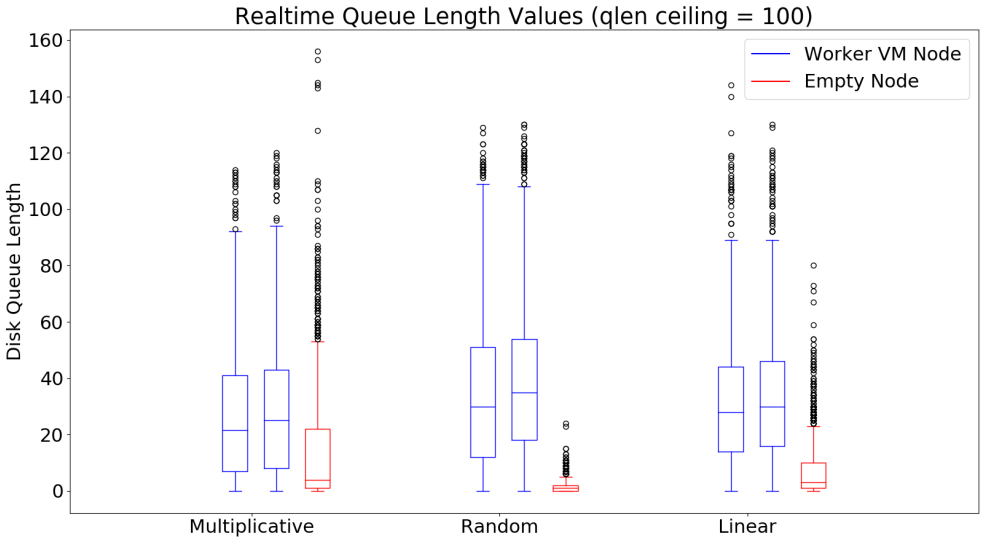

Figure 18: Queue lengths for all SSDs on the specified nodes sampled every 1 second.

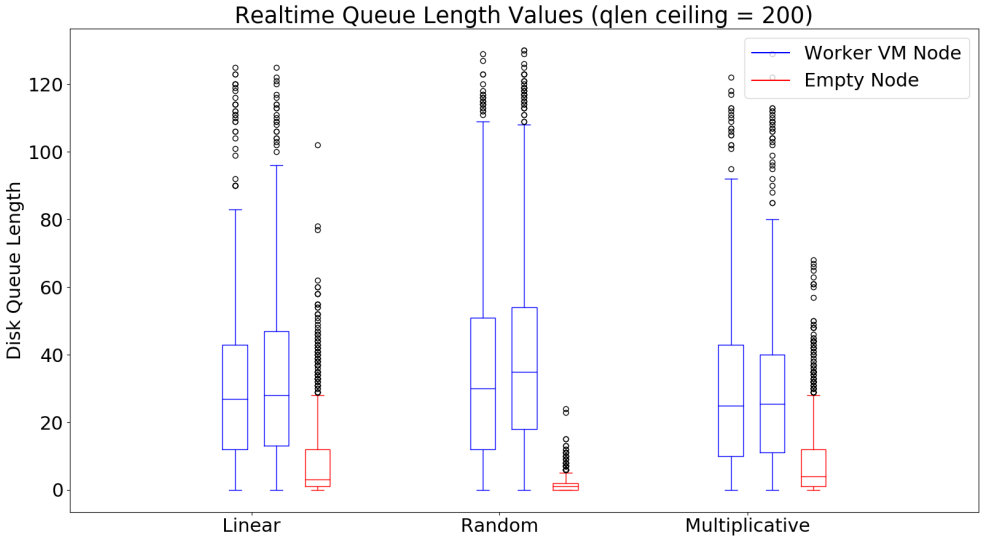

Figure 19: Queue lengths for all SSDs on the specified nodes sampled every 1 second.

5 User Guide

5.1 Scraping Data from the Nutanix Cluster

Stargate exposes a web server on the CVM, listening on port 2009, that exposes various real-time statistics such as the number of operations in flight, disk throughput,

disk queue length, and many other pieces of information. I wrote a Python script to parse and derive the following information for each node:

1. SSD tier usage
2. SSD tier availability
3. SSD tier max capacity
4. Read/Write counts for each disk
5. Queue lengths for each disk

The script will filter out all disks in the HDD tier and only pull the statistics for the SSD tier. Upon each invocation of the script, information is appended to a file for each node that is created if it is non-existent. The data is laid out in a CSV format for easy plotting via the Matplotlib library (Hunter, 2007). For each experiment, the script was called every 5 seconds via the command `watch -n 5`.

5.2 Re-creating Experiments

An NX-1350 with VMware ESXi 5.5u2 hypervisor and a single 300GB SSD on each node was used for all experiments. This means that for an RF2 cluster, there is less than 150GB of useable hot tier available because a small amount of space is reserved on each disk for other services on the CVM.

A single CentOS 6.5 worker VM was manually created on a single node with 4GB RAM, single 150GB disk, and installation of Fio. The VM was packaged as a file in Open Virtualization Format (OVF) for ease of deployment to the host hypervisor. This removed the need for me to use a tool to operate the VMware vSphere API when deploying the worker VMs as mentioned in the proposal for this thesis.

The Worker VM's 150GB disk size ensures that the entire hot tier of each node will be fully utilized when the disk is filled via workload generation on each node. Each of the other nodes then clone the worker VM so that all nodes in the cluster have an identical worker VM, worker VM disk state, and Nutanix node tier utilization.

The Fio script used to generate a sequential write workload on each node is as follows:

```
[global]
direct=1
ioengine=libaio
bs=32k
iodepth=128
randrepeat=0
group_reporting
filesize=150G

[job1]
rw=write
filename=/dev/sdb
name=sequential-write
```

The `iodepth` variable is modified depending on the number of outstanding I/Os needed.

When resetting the cluster for a new test, the commands `cluster stop ; cluster destroy` were run on the CVM and the old Stargate binary was swapped with the one required for the next tests in the `/home/nutanix/bin/` directory. When the cluster ready, it is necessary to recreate the Nutanix cluster via `cluster -s <cvm IPs> create` and deploy new worker VMs. The process is then repeated for each different test.

6 Summary and Conclusions

Prior to this work, the problem of selecting a location for the placement of data replicas in a Nutanix cluster was solved by choosing disks in a uniform random fashion. This method of data placement does not perform well in the presence of heterogeneous workloads or tier size disparities. This work has shown that candidate disks for data placement can be evaluated by using a function of observed queue length and fullness percentage values. The result of this function, the fitness value, can be used to bias disk selections toward disks that are less busy and less full. Biasing disk selections results in an overall even utilization of individual disk space and shows potential for a reduction of disk queue lengths for disks residing on busy nodes.

6.1 Limitations

The current weighted random selection approach, Stochastic Universal Sampling, is optimized for clusters that do not contain extremely large numbers of disks. While SUS allows disk selection probabilities to scale proportional to the fitness values of the disks, it is not as scalable as the two-choice method for weighted random selection.

In addition, the frequency of each Stargate statistic update will impact its view of the world and can lead to herding behaviors if updates are not frequent enough. In the absence of a Stargate's ability to update its view of the world, it will be unable to adapt to changes in workload characteristics or disk fullness, which defeats the purpose of using fitness-based disk selections in the first place.

6.2 Future Work

6.2.1 Real-time Fitness Feedback

The replica placement scheme shown in this work relies on periodic updates of usage statistics; therefore, the system is always working with older statistics for data placement decisions. One change that can be made to this system is to track the time to completion of the last N operations that place data on a disk. This data can be used similar to the C3 algorithm, which was discussed in section 2.2, to bias the replica placements toward the faster disks. This would remove the dependence on a centralized statistics repository such as Arithmos and allow each Stargate to make data placement decisions independent of reports from other nodes in the cluster. Stargate would be required to keep historical latency data for each operation on remote nodes.

6.2.2 Read Replica Selection

When choosing which replicas to read from, Stargate always selects local disks. However, if there were ever a scenario where a read from a remote disk containing the desired data is faster than servicing the read from a local disk, Stargate is not currently equipped to detect or act on this scenario. Currently, disk fitness values are not factored into the selection of disks that will service a read operation. A fitness function for read replica selection would still consider disk queue length, but disk fullness is not a helpful metric to determine the suitability of a read replica target. More experiments will be required to discover variables that would distinguish the read performance of disks that are holding data replicas before adding support for fitness-based read replica selection.

6.2.3 More Fitness Function Variables

The disk fitness values do not need to be limited to derivation from disk queue length and fullness percentage. The cluster tracks many other variables such as average node CPU utilization, number of Stargate failures in a specified time window, number of active hosted VMs, and data access patterns, among other pieces of information. More experiments should occur to see how this data can affect placement decisions and whether the result is beneficial to overall system performance.

6.2.4 Additional Testing

The experiments and testing in this work was performed on a 3-node, homogeneous hardware, Nutanix cluster, which is unsuitable for testing at large scale. Additional performance testing work on larger scale (roughly 64 node) clusters would be desirable to measure the impact of this work in larger datacenters. In addition to scale testing, heterogeneous hardware configurations in the form of hybrid (mixed SSD/HDD nodes) and all-flash nodes in the same cluster would provide intriguing results for analysis. This work focused mainly on heterogeneous synthetic workloads on homogeneous hardware to simulate the effects of realistic workloads on heterogeneous clusters. However, observing replica placement behavior on live clusters would provide additional insight into common replica placement behavior in the majority of Nutanix cluster installations.

Glossary

`CVM`

Nutanix Controller Virtual Machine.

`Data Deduplication`

A data compression technique for removing multiple copies of the same data.

`Fitness Value`

A value representing the desirability of a disk for data placement.

`HDD`

Hard Disk Drive.

`Hypervisor`

Software that runs virtual machines.

`iSCSI`

An IP based storage networking standard.

`NFS`

A distributed file system protocol.

`OVF`

A standard for packaging software to run in virtual machines.

`Secondary Replica`

Refers to any replica of data written after the first copy.

`SSD`

Solid State Drive.

`vDisk`

A file abstraction in the Nutanix distributed file system.

References

Albrightson, R., Garcia-Luna-Aceves, J., & Boyle, J. (1994). Eigrp–a fast routing protocol based on distance vectors. *Interop 94*.

Azar, Y., Broder, A. Z., Karlin, A. R., & Upfal, E. (1999). Balanced allocations. *SIAM journal on computing*, *29*(1), 180–200.

Baker, J. E. (1987). Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the second international conference on genetic algorithms* (pp. 14–21).

Borthakur, D., et al. (2008). Hdfs architecture guide. *Hadoop Apache Project*, *53*.

Chaubal, C. (2008). The architecture of vmware esxi. *VMware White Paper*, *1*(7).

Chen, P. M., Lee, E. K., Gibson, G. A., Katz, R. H., & Patterson, D. A. (1994). Raid: High-performance, reliable secondary storage. *ACM Computing Surveys (CSUR)*, *26*(2), 145–185.

Crow, J. F., & Kimura, M. (1979). Efficiency of truncation selection. In (Vol. 76, pp. 396–399). National Acad Sciences.

Dean, J., & Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, *51*(1), 107–113.

Ghemawat, S., Gobioff, H., & Leung, S.-T. (2003). The google file system. In *Acm sigops operating systems review* (Vol. 37, pp. 29–43).

Hedrick, C., & Bosack, L. (1991). An introduction to igrp. *Rutgers-The State University of New Jersey Technical Publication, Laboratory for Computer Science*.

Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, *9*(3), 90–95.

Jin, H., Yang, X., Sun, X.-H., & Raicu, I. (2012). Adapt: Availability-aware mapreduce data placement for non-dedicated distributed computing. In *Distributed*

computing systems (icdcs), 2012 ieee 32nd international conference on (pp. 516–525).

Josuttis, N. M. (2012). *The c++ standard library: a tutorial and reference.* Addison-Wesley.

Lakshman, A., & Malik, P. (2010). Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, *44*(2), 35–40.

Lamport, L. (1998). The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, *16*(2), 133–169.

McKusick, K., & Quinlan, S. (2010). Gfs: evolution on fast-forward. *Communications of the ACM*, *53*(3), 42–49.

McQuillan, J. M., Richer, I., Rosen, E. C., & Bertsekas, D. (1978). *Arpanet routing algorithm improvements* (Tech. Rep.). BOLT BERANEK AND NEWMAN INC CAMBRIDGE MA.

Mitzenmacher, M. (2001). The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, *12*(10), 1094–1104.

Patterson, D. A., Gibson, G., & Katz, R. H. (1988). *A case for redundant arrays of inexpensive disks (raid)* (Vol. 17) (No. 3). ACM.

Perez, J. M., Garcia, F., Carretero, J., Calderon, A., & Sanchez, L. M. (2003). Data allocation and load balancing for heterogeneous cluster storage systems. In *Cluster computing and the grid, 2003. proceedings. ccgrid 2003. 3rd ieee/acm international symposium on* (pp. 718–723).

Poitras, S. (n.d.). *The nutanix bible.* Retrieved from `www.nutanixbible.com`

Suresh, P. L., Canini, M., Schmid, S., & Feldmann, A. (2015). C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *Nsdi* (pp. 513–527).

Van Rossum, G., et al. (2007). Python programming language. In *Usenix annual*

technical conference (Vol. 41, p. 36).

Velte, A., & Velte, T. (2009). *Microsoft virtualization with hyper-v.* McGraw-Hill, Inc.

White, T. (2012). Hadoop: The definitive guide.

Xie, J., Yin, S., Ruan, X., Ding, Z., Tian, Y., Majors, J., ... Qin, X. (2010). Improving mapreduce performance through data placement in heterogeneous hadoop clusters. In *Parallel & distributed processing, workshops and phd forum (ipdpsw), 2010 ieee international symposium on* (pp. 1–9).

Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R. H., & Stoica, I. (2008). Improving mapreduce performance in heterogeneous environments. , *8*(4), 7.

A Appendix

A.1 Herding Behavior Due to Implementation Bug

During the 128 outstanding op experiments, herding behavior was observed unexpectedly after implementing fitness-based replica placement, as shown in Figure 20. By default, disk usage and performance statistics are supposed to be refreshed every 10 seconds. This is frequent enough to avoid herding behavior, but the 128 outstanding op experiments exhibited herding.

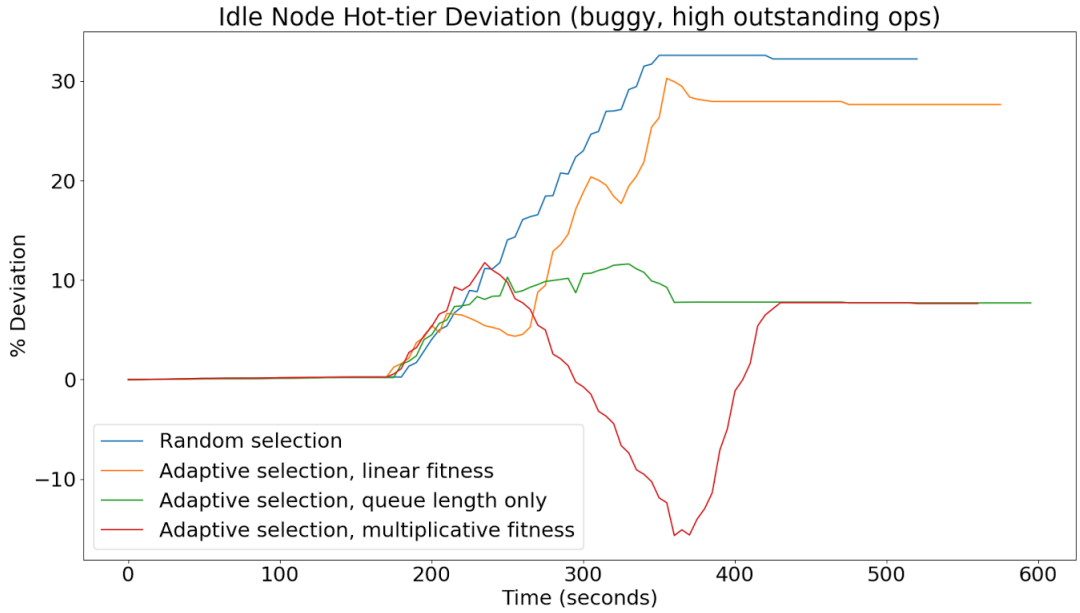

Figure 20: $d_{hot\ tier}$ values over time for low outstanding I/O operations. This set of experiments contains the disk usage statistics update bug.

The experiment with additive fitness seemed to only exhibit mild herding behavior, whereas the test with multiplicative fitness showcased much more dramatic shifts in SSD usage skews. In conjunction with the complete absence of this behavior in the single outstanding op experiments, it was thought to be highly likely that this herding behavior was caused by a bug in the queue length term of the fitness functions. An

additive fitness function is less affected by this bug due to the attenuated effect of each term in the fitness function.

Within Stargate, a mapping is kept from disk ID to a DiskState object (called `disk_map_`) containing information and cached statistics related to the disk. The disk performance and disk usage statistics are two separate elements within the DiskState structure.

Every 10 seconds, an alarm handler will execute and iterate through each active disk in the cluster and asynchronously query disk statistics and bind a callback to each query to be executed when a response is received. Disk usage and performance lookups each have their own callback functions.

Table 8: Disk usage and performance lookup callback functions.

Function Name	**Description**
`UsageStatLookupCallback`	Decrement the outstanding statistics lookup counter, acquire lock and populate performance statistics in `disk_map_`, and leave performance statistics untouched.
`PerformanceStatLookupCallback`	Decrement outstanding statistics lookup counter, lock and populate performance statistics in `disk_map_`, and leave usage statistics untouched.

The two callbacks introduce a race condition regarding `disk_map_`, even though the structure is locked. Any time PerformanceStatLookupCallback returns before the callback for usage statistics, all performance statistics will be cleared and cause the fitness function to assume worst-case values for the queue length term.

This problem was fixed by simply serializing the usage and performance statistics lookups by triggering the performance statistics lookup from inside UsageStatLookup-Callback.