Contents

List of Figures

1 Abstract

2 Introduction

With the advent of cloud computing, datacenters are making use of distributed applications more than ever. Companies like Google use software such as MapReduce to generate over 20 petabytes of data per day using very

large numbers of commodity servers [3]. Many other companies use large scale clusters to perform various computational tasks via the the open-source MapReduce implementation, Hadoop [4], 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/capacity nodes being mixed with lower performance/capacity HDD nodes. This thesis presents an adaptive data placement method in the Nutanix distributed file system (ADSF) which will attempt to remedy the common problems found in many heterogeneous clustered file systems.

2.1 Motivation

A number of scenarios arise in heterogeneous Nutanix clusters that can degrade performance for an entire cluster. The currently replica disk selection logic in Stargate uses does not take into account a number of variables such as disparities in tier size, CPU power, workloads, and disk health among other things.

Considering that a write is not complete until all replicas are written, the write's performance is at the mercy of the slowest disk and node. There are several scenarios, both pathological and daily occurences, where a more

robust replica placement heuristic is required. For the work in this thesis, I will focus on two orthogonal cases described below.

2.1.1 Interfering Workloads

An example of interfering workloads can take the form of a 3-node homogeneous cluster with only 2 nodes hosting active workloads as shown in Figure 1. In the current random selection scheme in use by the ADSF, writes are equally likely to place their replica on the other node with an active workload as they would be to place it on the idle node. This can impact performance on both the local and remote workloads as secondary writes will be slower on nodes whose resources are being utilized by their primary workloads. An adaptive replica placement scheme is needed to avoid the busy node and bias secondary replica placement on an idle node.

2.1.2 Nodes with Tier Size Disparities

A cluster containing nodes with a tier size disparity are susceptible to a skew in node fullness, even if the workload on each node is identical. This can be illustrated via Figure 2 where we have a 3-node heterogeneous cluster with 2 high-end nodes and a single weak node. Suppose these high-end nodes have 500GB of SSD tier and 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

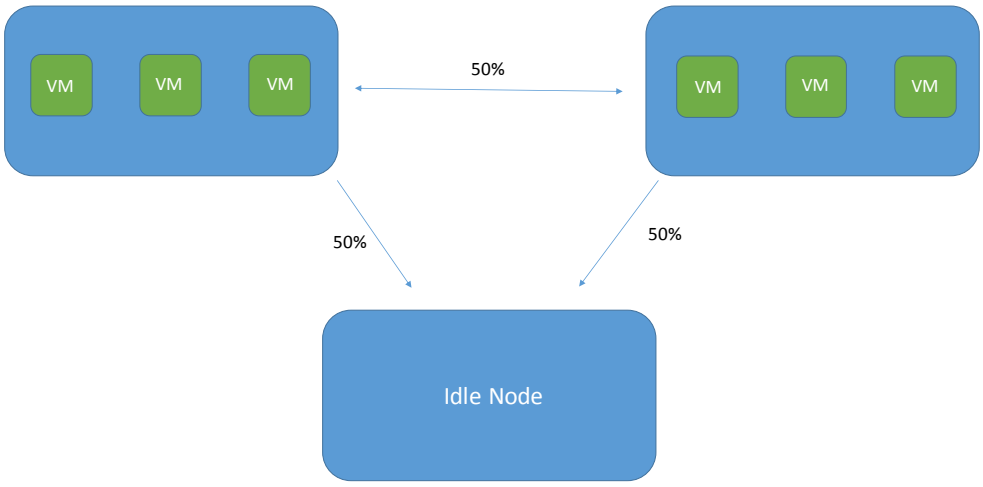

Figure 1: A cluster with identical nodes running a heterogeneous workload.

to generate data such that the working sets of the workloads are 50% of the local SSD tier, the weaker node is at a significant disadvantage. Given the current NDFS replica selection algorithm, we can expect 500GB of replica traffic to flood the weak node and fill up its SSD tier well before the workload is finished. 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, increasing latency. An adaptive replica placement heuristic would mitigate this issue by taking disk usages into consideration during the placement of secondary replicas and biasing placement of secondary replicas on the nodes with more free capacity.

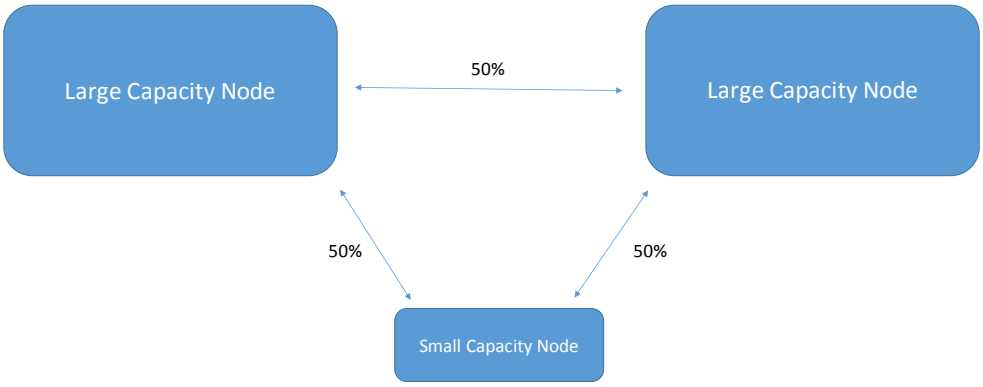

Figure 2: A cluster with nodes of varying resource capacity.

2.2 Acropolis Base System

NDFS is facilitated by a clustering of controller virtual machines (CVMs) which reside, one per node, on each server in the cluster. The CVM presents via NFS (for VMWare's ESXi [14]), SMB (for Microsoft's Hyper-V [17]), or iSCSI (for Nutanix's AHV [1]) an interface to each hypervisor that they reside on. For example, the interface provided by the CVMs to VMware's ESXi hypervisor [14] will be interfaced with as a datastore. The virtual machines' virtual disk files will reside on the Nutanix datastore and be accessed via NFS through the CVM sharing a host with the user VM. Within the CVM exists an ecosystem of process that make up the ADSF. This work is scoped

specifically to the I/O manager process, Stargate.

2.3 Stargate

The Stargate process is responsible for all data management and I/O operations. The NFS/SMB/iSCSI interface presented to the hypervisor is also presented by Stargate. All file allocations and data replica placement decisions are made by this process.

As the Stargate process facilitates writes 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 average time to complete an operation on the disk. These statistics are only gathered on the local disks; however, they are then stored in a distributed database provided by another ADSF service along with the statistics gathered by every other Stargate in the cluster. These disk statistics stored in the database and are pulled periodically and are then used to make decisions on data placement when performing writes.

2.3.1 Oplog and Extent Store

2.3.2 Arithmos Interactions

2.3.3 Storage Tiering

2.3.4 Data Replication and Fault Tolerance

2.3.5 Replica Selection

When a Stargate write operation wants to select disks for data placement, it is done on a per-tier basis. The interface for a replica selection function call, DetermineNewReplicas, takes the following arguments:

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 we prefer to choose disks from. This means we 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 evaluates to false for any disk, 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, we first attempt to select replicas

from the preferred nodes. This involves finding the set of disks that belong to each node on the specified tier, shuffling the 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% 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, we simply continue searching the shuffled set of disks belonging to the specified tier. If a disk is found to be suitable, we must add the disk to the replic_vec and add the node that the disk belongs to in the exclude_nod_vec. This prevents us from considering other disks on that node to maintain a node fault tolerance guaranteed by the cluster's replication factor.

3 Prior Work

4 Implementation

4.1 Stargate Disk Stats Collection

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

Stargate maintains a mapping, henceforth referred to as 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 stats from Arithmos, the information is used to create a disk fitness value for each disk in the cluster.

4.2 Fitness Values and Functions

To calculate a value to represent the desirability of a disk for replica placement, we'll use a function, $f_{fitness}$ that takes as its argument disk stats and returns a positive number we will call 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 thesis, I eval-

uate two fitness functions that are comprised of terms that utilize a disk's average queue length over some stretch of time, t_q, and a disk's percentage utilization, t_u.

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

$q_{ceiling}$ is defined as the maximum observed queue length such that beyond this value, t_q does not contribute to the fitness value. This ensures that as the queue length grows, t_q approaches zero.

$$t_u = \frac{1}{a^u} \tag{2}$$

u is the 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 used in two different fitness functions evaluated in this thesis:

$$f_{add} = t_u + t_q \tag{3}$$

$$f_{mult} = t_u t_q \tag{4}$$

4.3 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. Since the file system only supports replication factors of 2 or 3, the investigation was limited to algorithms that allow for a weighted N choose Y, where Y is the data replication factor.

This section provides an overview of the algorithms investigated via simulations to compare each algorithm at different orders of magnitude.

4.3.1 Scalability Simulation Methodology

To test the scalability of the weighted random selection algorithms evaluated in the next section, a single-threaded Python script was written to to evaluate the change in run time as sample sets increase. Each algorithm's time to select is calculated for each of a fixed number of iterations for multiple sample sets. pseudo-code for the simulations can be written as follows:

```
for each selection_algorithm in algorithm_list:
    run_time_values = empty_list()
    for each sample_set_size in all_sample_set_sizes:
        sample_set = generate_sample_set(sample_set_size)
        all_elapsed_times = empty_list()
        for each iteration:
            start_time = time.now()
            selection_algorithm(sample_set)
            elapsed_time = time.now() - start_time
            all_elapsed_times.append(elapsed_time)
        calculate_avg_elapsed_time(all_elapsed_times)
        calculate_std_error(all_elapsed_times)
```

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) are performing their preprocessing steps for each selection. This gives information about the worst-case behavior for each algorithm in comparison with others'.

4.3.2 Herding Behavior Evaluation Methodology

Herding behaviors can be seen in some weighted random selection algorithms when (TODO: cite some papers) 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. We can 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. 1e3, 1e4, and 1e5 iterations 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

identical to the high-skew sets, except there was not a single object with an exaggerated weight of 1000. All element weights were their array indices.

4.3.3 Stochastic Universal Sampling (SUS) Simulations

SUS is another sampling technique first introduced by Baker in 1987 [10]. 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 could also be unevenly sized which would indicate a weighted probability. SUS uses this same concept except allows for N evenly spaced pointers corresponding to the selection of N items. Key things to note are that the set, or "bins" in my roulette analogy, must be shuffled prior to selection. Also, there is a minimum spacing allowed for the pointers to prevent selection of the same bin.

Figure 3 shows the evolution of the distribution of selection frequencies for SUS as the number of samples increases. We can see 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 4. We can see that the high weight object's selection frequency eclipses all other objects in the selection pool which can lead to extreme herding behaviors.

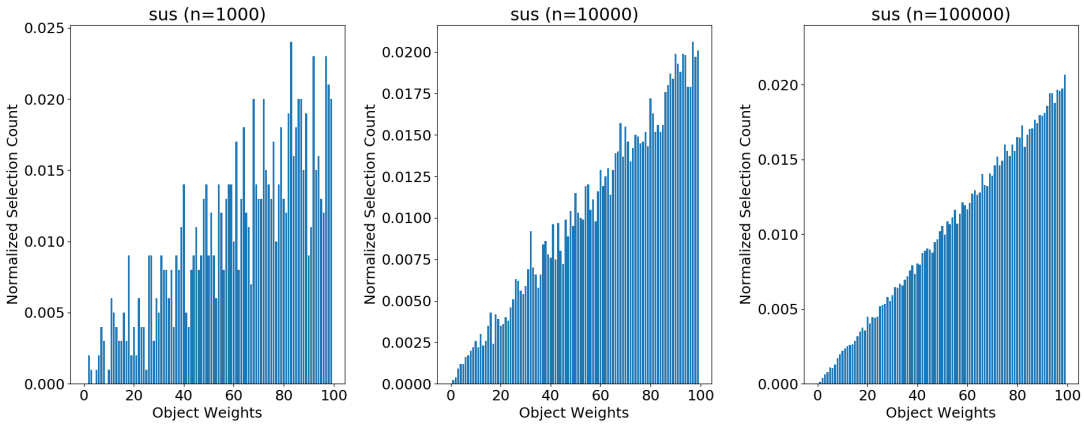

Figure 3: Histograms generated by simulation of Stochastic Universal Sampling using sample sizes of 1e3, 1e4, and 1e5. Object weights for the histograms are in the range [1,100].

4.3.4 Tructation Selection Simulations

Truncation selection[TODO: CITE] does not consider any objects for selection below some threshold, T. In figure 5, 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 may 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 6.

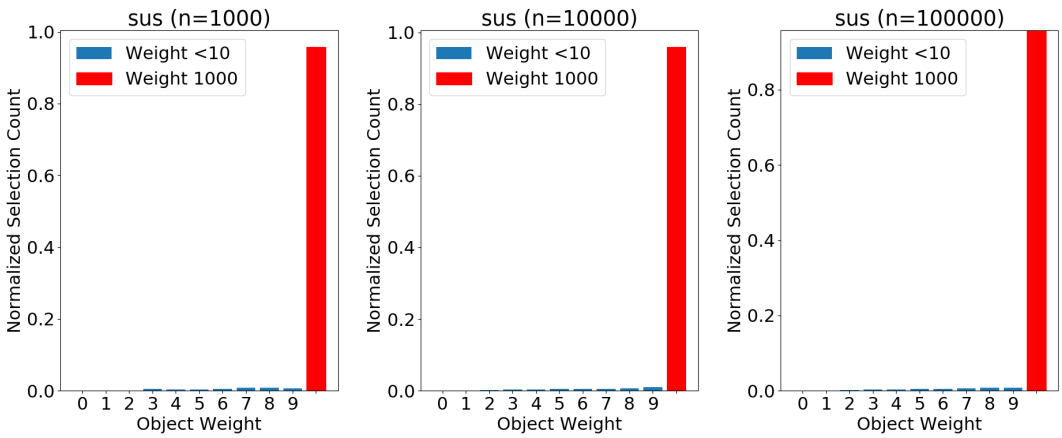

Figure 4: Histograms generated by simulation of Stochastic Universal Sampling using sample sizes of 1e3, 1e4, and 1e5. Object weights are in the range [0,9] with a single outlier of weight 1000 to illustrate the effect of herding behavior.

4.3.5 Two-choice Sampling Simulations

Two-choice sampling, first introduced by has proven to be extremely resilient to herding behavior as shown in Figure 8 and selection frequencies for all objects are influenced by object weights in a way similar to SUS. While an object with a higher weight is more likely to be selected, it is not selected with a probability proportional to its fitness value. This makes the algorithm resistant to any herding behaviors, but will not be a good candidate if we desire the WeightedVector to select objects with probability proportional to its weight.

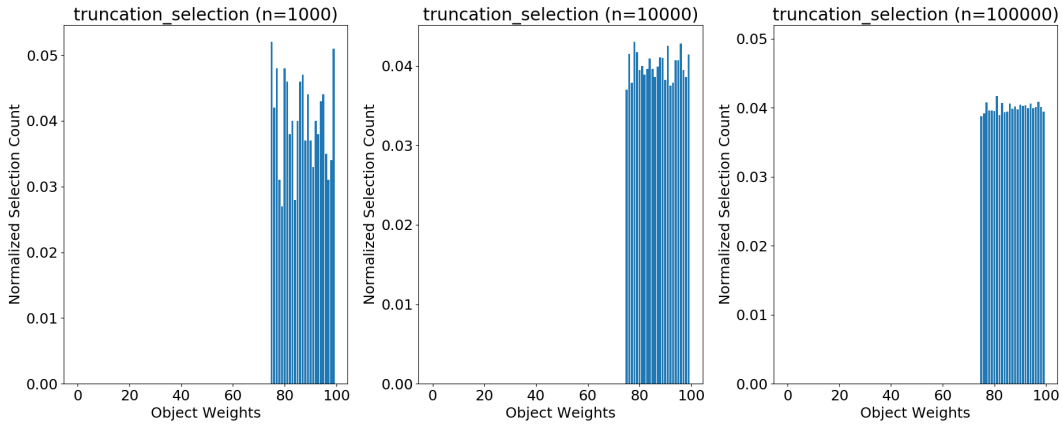

Figure 5: Histograms generated by simulation of truncation selection using sample sizes of 1e3, 1e4, and 1e5. Object weights for the histograms are in the range [1,100].

4.3.6 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 it is important to include its analysis for comparison with other selection algorithms.

Figure 6 shows that we can expect uniform selection probabilities and Figure 10 confirms that we can expect no herding behavior with a uniform random selection scheme.

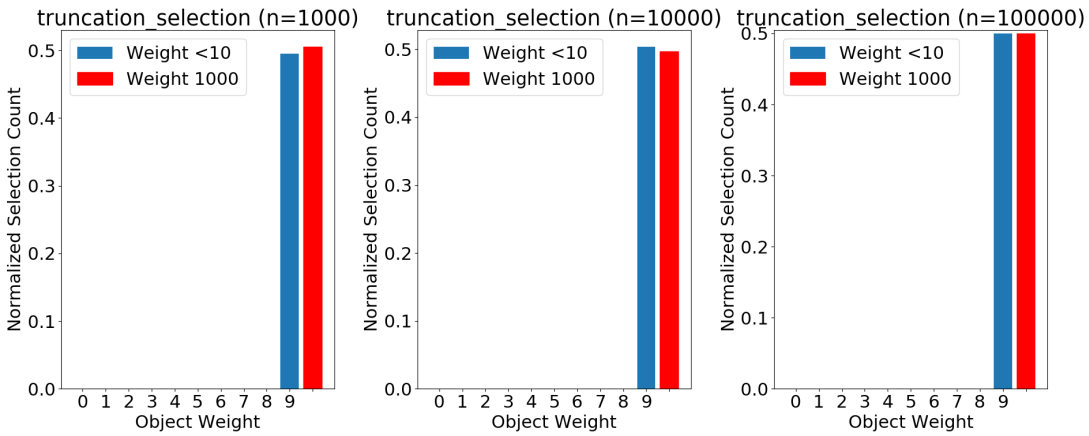

Figure 6: Histograms generated by simulation of truncation selection using sample sizes of 1e3, 1e4, and 1e5. Object weights are in the range [0,9] with a single outlier of weight 1000 to illustrate the effect of herding behavior.

4.3.7 Weighted Random Algorithm Scalability Simulations

It's clearly seen that even though both SUS and truncation selection are [TODO: cite runtimes for each] exponential in run time, truncation selection is slower than SUS by an order of magnitude. This is mainly due to the need for the truncation selection algorithm to calculate the top T% of the set for every selection performed. As expected, two-choice and random selections are observed to be constant-time algorithms. Two-choice is slightly slower than random selection due to the second selection and comparison operation that must occur.

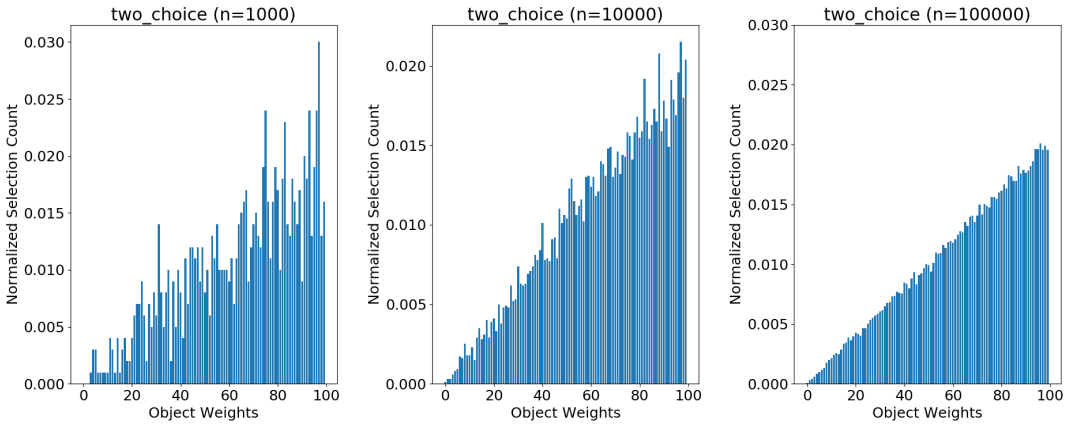

Figure 7: Histograms generated by simulation of two-choice selection using sample sizes of 1e3, 1e4, and 1e5. Object weights for the histograms are in the range [1,100].

4.4 WeightedVector Class

It's necessary to perform repeated weighted sampling of disk IDs, easily, and given their fitness values. An object container class is needed similar to a C++ STL container [XXX CITE XXX] that can also 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

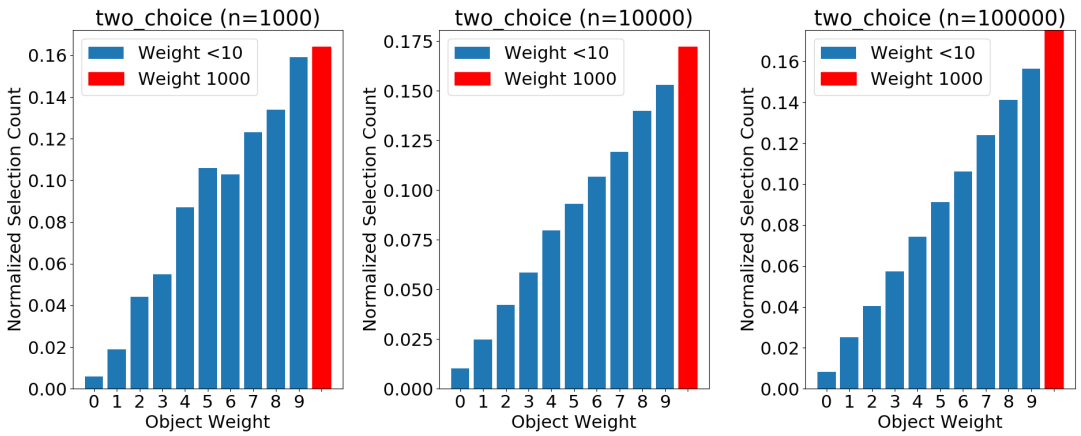

Figure 8: Histograms generated by simulation of two-choice selection using sample sizes of 1e3, 1e4, and 1e5. Object weights are in the range [0,9] with a single outlier of weight 1000 to illustrate the effect of herding behavior.

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 share an index into their respective internal vectors.

4.4.1 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

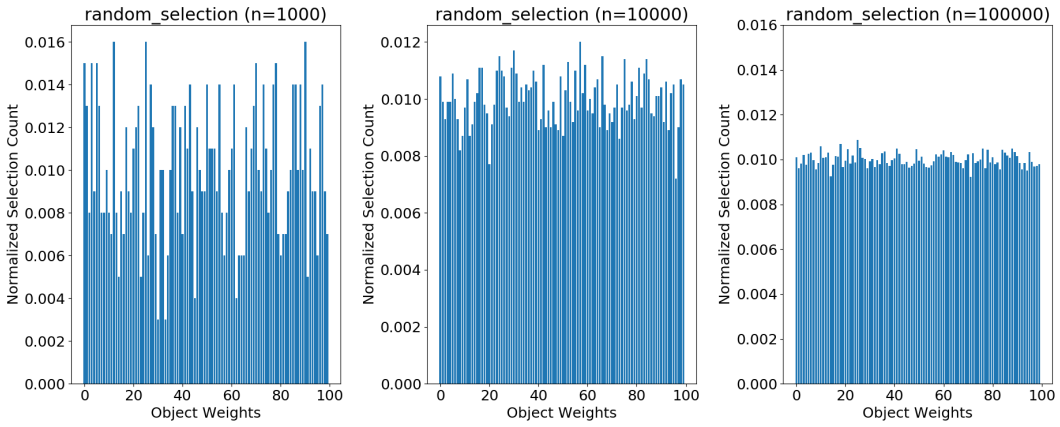

Figure 9: Histograms generated by simulation of uniform random selection using sample sizes of 1e3, 1e4, and 1e5. Object weights for the histograms are in the range [1,100].

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()
```

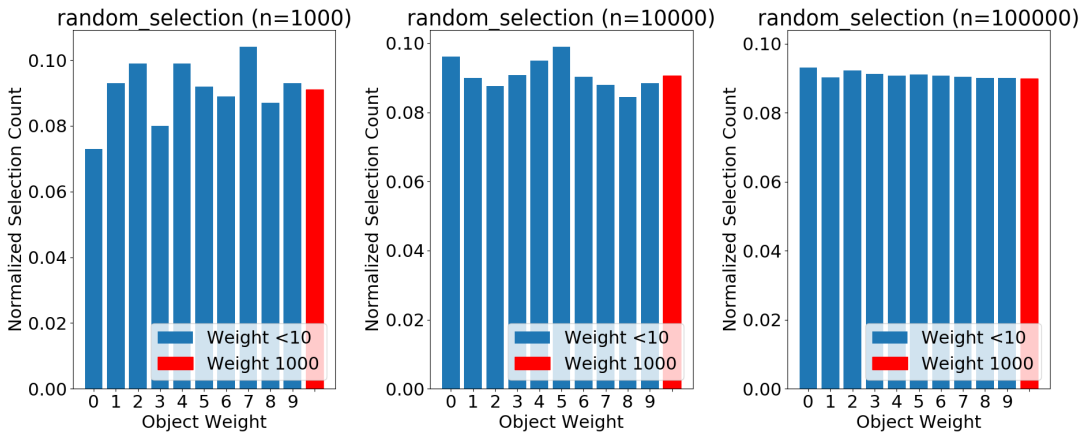

Figure 10: Histograms generated by simulation of uniform random selection using sample sizes of 1e3, 1e4, and 1e5. Object weights are in the range [0,9] with a single outlier of weight 1000 to illustrate the effect of herding behavior.

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()
```

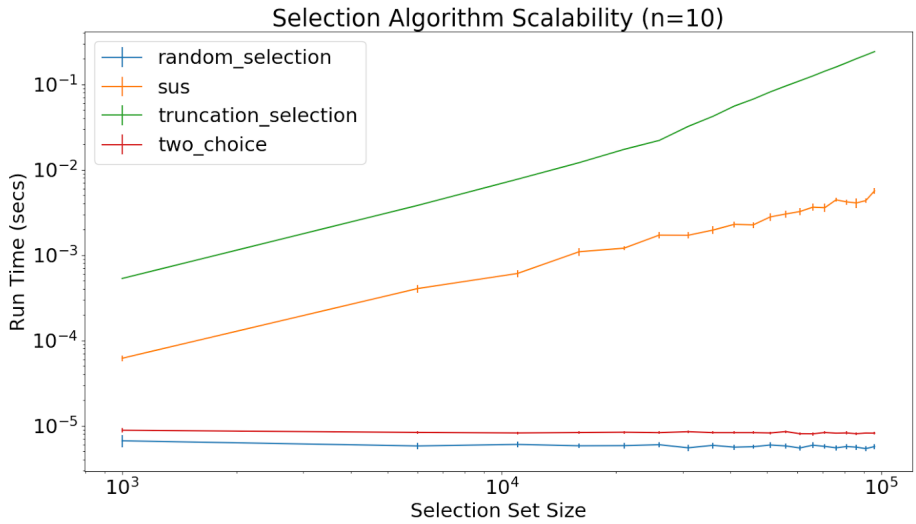

Figure 11: Running times of various weight random selection algorithms. Each algorithm is run for 10 iterations at each object pool size.

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.

4.4.2 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. There is the inner_vector_ variable which stores the objects inserted via EmplaceBack(), the weights_ variable which 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 us to exclude objects from being sampled multiple times with time complexity $O(1)$, since a weight of 0 gives a probability of 0 for sampling using Stochastic Universal Sampling as shown in the example below.

Before sampling, we have identical weights_ and sampling_weights_ accompanying an inner_vector_ of objects:

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

Suppose we then call Sample() on the WeightedVector and it 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 zero:

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

If we call Sample() again, it is not possible to return D since its sampling weight is now zero. 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 following state:

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

We may only call Sample() one more time without triggering an assertion failure and the WeightedVector is obligated to return D. The only way to restore sampling state is to call Reset(). A Reset call simply copies weights_ into sampling_weights_ and all objects are eligible for sampling again:

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

4.4.3 Weighted Vector Unit Testing

The WeightedVector class' unit test has four phases:

1. Test Average() functionality
2. Test 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, we verify 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 add monotonically increasing integers in the range $[0, N]$ to the WeightedVector. The fitness function 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 several orders of magnitude larger

than the number of elements. Each sampled element's number of times being selected is tracked in a test-local hash map. We then calculate the actual selection probability of each element in the WeightedVector with the expected value and verify the difference is within an acceptable tolerance. For the uniform test, we expect all integers to be sampled roughly the same amount and for the non-uniform test, we expect larger integers to be sampled an amount of times proportional to their value.

4.5 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 iterating through each disk until enough replica targets are found, we call `WeightedVector::Sample()` until enough replica targets are found. All of Stargate's replica placement logic is untouched and I've simply modified the order in which candidate disks are considered.

5 Evaluation and Results

The experiments below seek to measure the effect of the additive and multiplicative term fitness functions on both the tier utilization of each node and the queue lengths of disks residing on those nodes compared with uniform

random selection.

5.1 Experimental Setup

The replica selection schemes were evaluated using a NX-1350 for evaluating the disk fullness and a NX-3-node 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 are 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.

5.2 Fio and Write Patterns

When generating I/O in these experiments, Fio is 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 is used to generate a sequential write workload that completely fills the cluster's hot-tier. I choose to use exclusively sequential writes for all tests because they are the default for fio tests and the purpose of these experiments is to generate new 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 generatred an extent group.

5.3 Tier Utilization Experiments

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

$$d_{hot\ tier} = \frac{u_{w1} + u_{w2}}{u_o} \tag{5}$$

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 due to the fact that it does not have to service a local workload. In a uniform random replica selection scheme, we expect the nodes running a workload have to bear 100% of their own primary replicas and 50% of secondary replicas from the other worker node. This causes total SSD utilization to be skewed towards the worker nodes and for this skew to grow as the tests run. This is indicated by higher $d_{hot\ tier}$ values. We expect a more sophisticated replica selection scheme to minimize $d_{hot\ tier}$ by biasing

secondary replicas towards the idle node and limiting the skew.

5.3.1 Low Outstanding Operation Results

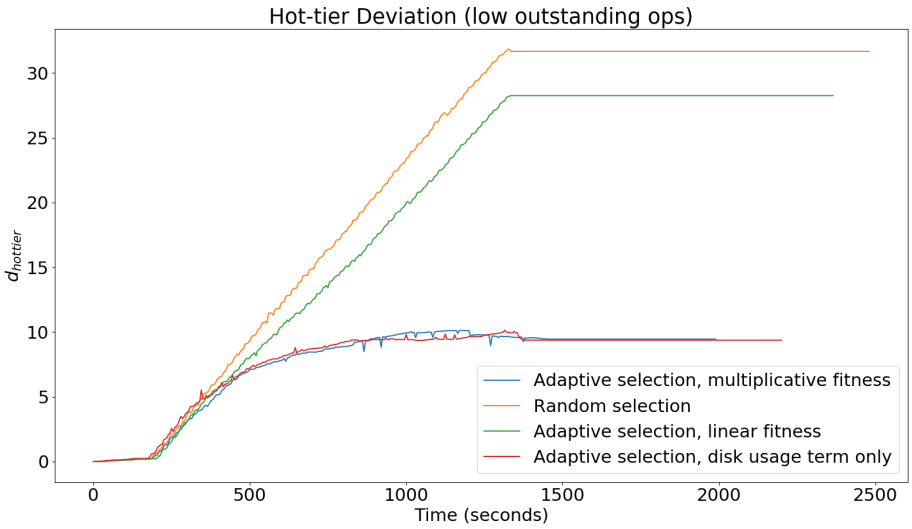

Figure 12: $d_{hot\ tier}$ values over time for low outstanding I/O operations.

Figure 12 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.

We can see that the additive fitness function does not minimize the hot-tier deviation as well as the multiplicative. This is because an 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 both 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 very much while giving no weight to the queue length term keeps all nodes' usages within 10% of each other. This is because the queue length term is contributing the maximum amount possible to the fitness value due to the consistently low queue length values. We can illustrate this 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, we can calculate b_r as follows:

$$b_r = \frac{f(d)}{f(d) + f(d')} \tag{6}$$

Figures 13 and 14 show the disk selection biases for very large and very small queue lengths respectively.

We can see that for an additive fitness function, the bias towards a less utilized disk decreases as the disk fullness percentages for d and d' increase, even though they still only differ by 10% in the figure above. This is because the entire linear fitness function does not scale with each term, so we can conclude that the multiplicative fitness function is superior.

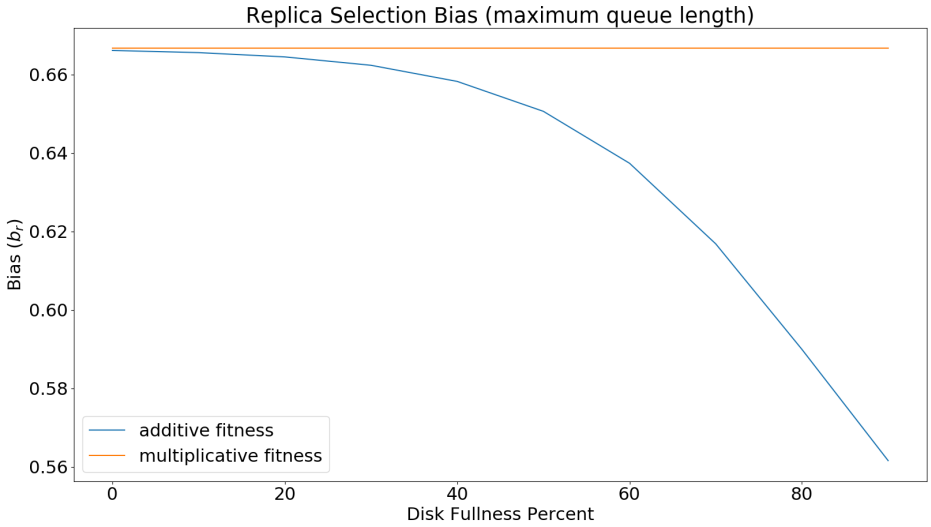

Figure 13: b_r values with static queue lengths at the fitness function ceiling values.

5.3.2 High Outstanding Operation Results

In Figure 15, we can see 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.

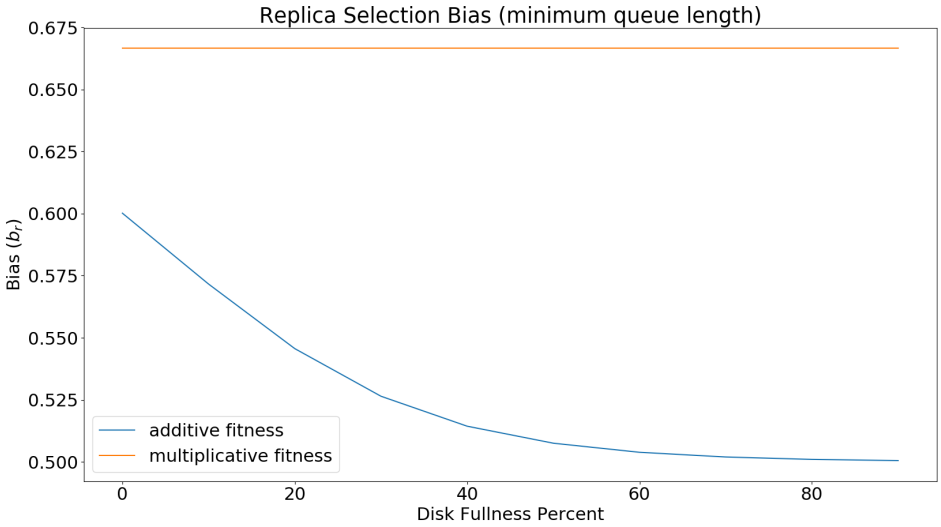

Figure 14: b_r values with static queue lengths at 1.

5.4 Disk Queue Length Experiments

Since a low outstanding operation workload would not give useful information for measuring the effects of fitness-based replica selection on disk queue lengths, the high outstanding I/O operation experiment in the previous section was re-run for all fitness function types and for fitness function queue length term ceilings of 200 and 100. Figures 16 and 17 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 are observed to provide better results in reducing the queue lengths for the worker nodes.

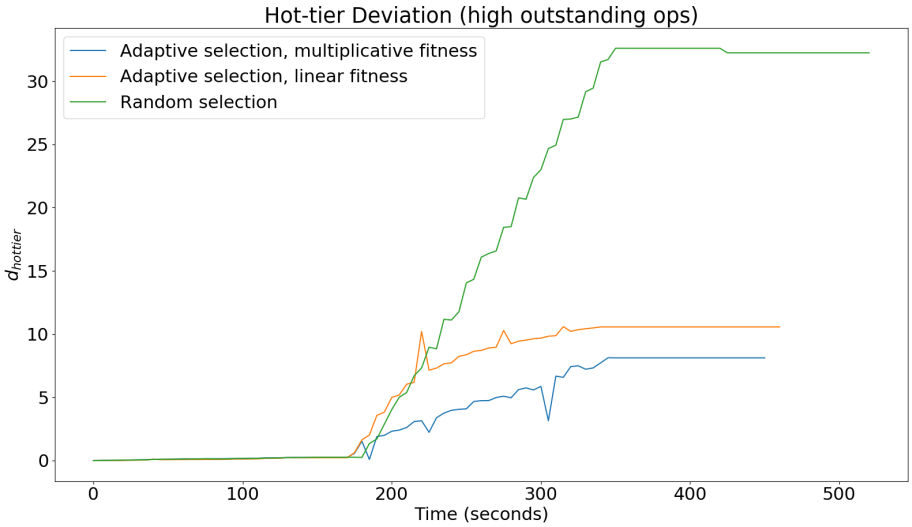

Figure 15: $d_{hot\ tier}$ values over time for low outstanding I/O operations.

6 User Guide

6.1 Re-creating Experiments

6.2 Scraping Data From the Nutanix Cluster

7 Future Work

7.1 Real-time Fitness Feedback

7.2 Read Replica Selection

7.3 More Fitness Function Variables

8 Appendix

8.1 Herding Behavior Due to Implementation Bug

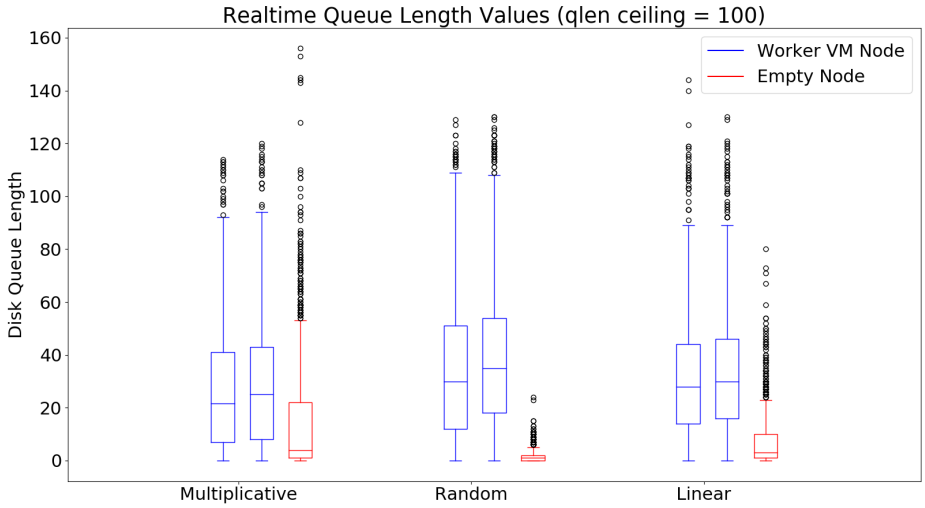

Figure 16: Queue lengths for all SSDs on the specified nodes sampled every 1 second.

in Figure 18. By default, disk usage and performance stats are supposed to be refreshed every 10 seconds. This is frequent enough to avoid herding behavior, but the 128 outstanding op experiments exhibited herding.

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

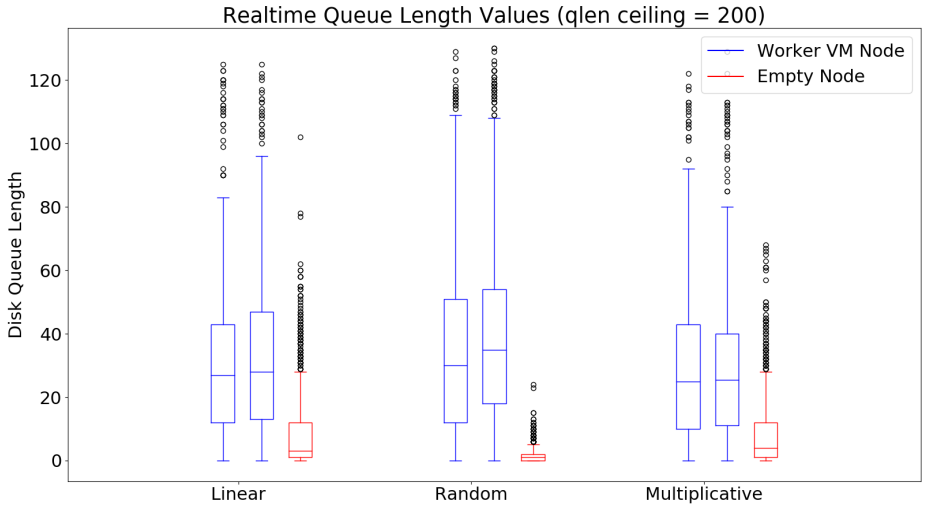

Figure 17: Queue lengths for all SSDs on the specified nodes sampled every 1 second.

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

Within Stargate, we keep a mapping 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 stats 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 stats and bind a callback to each query to be executed when a response is received. Disk

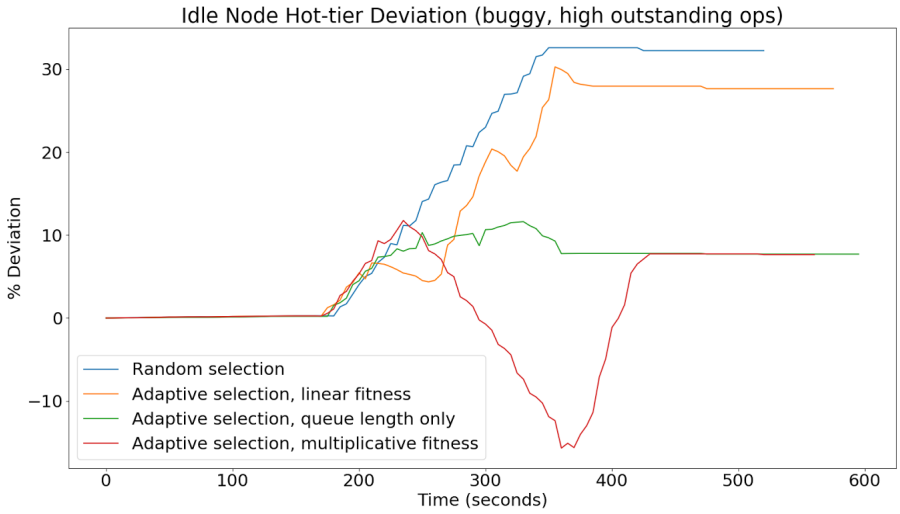

Figure 18: $d_{hot\ tier}$ values over time for low outstanding I/O operations. This set of experiments contains the stats update bug.

usage and performance lookups each have their own callback functions:

Function Name	Description
`UsageStatLookupCallback`	Decrement the outstanding stats lookup counter, acquire lock and populate performance stats in `disk_map_`, and leave performance stats untouched.
`PerformanceStatLookupCallback`	Decrement outstanding stats lookup counter, lock and populate performance stats in `disk_map_`, and leave usage stats 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 stats, all performance stats will be cleared and cause the fitness function to assume worst-case values for the queue length term.

This problem is fixed by simply serializing our usage and performance stats lookups.

8.2 Glossary

8.3 References