Impact of Data Placement on Resilience in Large-Scale Object Storage Systems

Philip Carns, Kevin Harms, John Jenkins, Misbah Mubarak, Robert Ross
Argonne National Laboratory
Lemont, IL 60439
carns@mcs.anl.gov

Christopher Carothers
Rensselaer Polytechnic Institute
Troy, NY 12180

***Abstract*—Distributed object storage architectures have become the de facto standard for high-performance storage in big data, cloud, and HPC computing. Object storage deployments using commodity hardware to reduce costs often employ object replication as a method to achieve data resilience. Repairing object replicas after failure is a daunting task for systems with thousands of servers and billions of objects, however, and it is increasingly difficult to evaluate such scenarios at scale on real-world systems. Resilience and availability are both compromised if objects are not repaired in a timely manner.**

In this work we leverage a high-fidelity discrete-event simulation model to investigate replica reconstruction on large-scale object storage systems with thousands of servers, billions of objects, and petabytes of data. We evaluate the behavior of CRUSH, a well-known object placement algorithm, and identify configuration scenarios in which aggregate rebuild performance is constrained by object placement policies. After determining the root cause of this bottleneck, we then propose enhancements to CRUSH and the usage policies atop it to enable scalable replica reconstruction. We use these methods to demonstrate a simulated aggregate rebuild rate of 410 GiB/s (within 5% of projected ideal linear scaling) on a 1,024-node commodity storage system. We also uncover an unexpected phenomenon in rebuild performance based on the characteristics of the data stored on the system.

I. Introduction

Distributed object storage architectures are widely adopted in large-scale storage systems for a variety of problem domains [1], [2], [3], [4], [5], [6], [7]. Object storage deployments that use commodity hardware to reduce costs often employ object replication as a method to achieve data resilience. These systems must therefore use distributed rebuild algorithms to regain resilience by replacing lost replicas after a failure event. Distributed rebuild is a daunting task for systems with thousands of servers and billions of objects because of the quantity of data and the degree of interserver coordination involved. This leads to an essential question for large-scale storage systems: how efficiently does the system recover from faults? The performance of this procedure (i.e., the aggregate rebuild rate) has a direct impact on the mean time to data loss (MTTDL), particularly for replication levels greater than two [8], [9]: the longer it takes to replace missing replicas, the higher the probability that subsequent failures will result in data loss. Perceived application response time may also be compromised while the system is in a degraded state.

The *object placement algorithm*, that is, the mechanism used to map object replicas to available servers, is a critical element of storage system design that affects performance, locality, and load balancing. The object placement algorithm also dictates replica declustering and thus the total amount of parallelism that can be achieved during rebuild. Declustering was originally proposed in the context of disk arrays as a means to distribute RAID stripe units across different subset of disks for better load distribution during rebuild [10]. In the context of replicated object storage systems, declustering is the degree to which surviving servers are engaged in replica reconstruction following a failure [11], [12]. Fully declustered algorithms lead to a rebuild in which all servers participate evenly, while fully clustered algorithms localize the rebuild load to the smallest possible number of affected servers. Most placement algorithms fall somewhere between these two extremes in practice. We focus our analysis on algorithmic placement, in which storage locations are calculated by applying a deterministic function to the object ID and candidate server IDs. Algorithmic placement is a popular choice in distributed system design because it eliminates the need to store explicit layout metadata for each object and it allows any client or server to independently (but consistently) calculate placement.

Declustering, object placement, and aggregate rebuild performance are clearly critical elements of storage system design. However, the cost and complexity of distributed object storage systems make it difficult to perform thorough experimental assessment of these components at scale. As a result, the nuances of large-scale fault recovery are not well understood; for example, what are the weakest links in performance and how will rebuild be impacted by the nature of the data stored on the system? Worse yet, distributed storage protocols may contain subtle performance or correctness flaws [13] that are difficult to discover intuitively. We address this gap by developing a high-fidelity distributed object storage simulator to investigate the behavior of storage systems with thousands of servers, billions of objects, and petabytes of storage capacity.

Our analysis uncovers subtle limitations in a well-known object placement method at scale and identifies methods that future storage systems can adopt to remedy the problem. We find that an efficient object placement algorithm can enable near-linear scaling of aggregate rebuild performance; our example model achieves over 400 GiB/s of aggregate throughput using 1,024 commodity storage servers. The contributions of this work include the following:

- A case study evaluating the rebuild efficiency of an object storage system using an existing well-known object placement method
- Identification and evaluation of object-placement optimizations that can improve rebuild efficiency at scale
- A high-fidelity object storage system simulator that can model a system with thousands of servers, billions of objects, and petabytes of storage capacity
- An analysis of large-scale data populations, a method for modeling their characteristics, and a demonstration of their impact on experimental results

The remainder of the text is organized as follows. Section II provides background information, and Section III describes our simulation platform. Section IV explores a large-scale object rebuild case study. Section V evaluates potential performance optimizations, the impact of data populations, and the effectiveness of the simulation methodology. Section VI highlights related work, and Section VII summarizes our findings.

II. Background

Our target use case is a horizontally scalable data-center storage system, consisting of hundreds or thousands of servers, built from commodity components, and capable of hosting many petabytes of replicated data for high-performance distributed and parallel applications. The servers are homogeneous, and configuration is controlled by a system administrator. We expect server failures to be frequent but not continuous [14]. It follows that group membership will be relatively static as well.

Various failure modes are possible in production, but for clarity in this work we focus on the scenario in which a single server fails completely such that its data is no longer reachable. This is a straightforward starting point for understanding baseline rebuild behavior. More complex failure modes can be studied in future work.

A. Object placement

An object placement algorithm is the mechanism used to map a given object and its replicas to a subset of servers in a distributed storage system. Many large storage systems elect to use a deterministic mathematical function for this purpose in order to simplify the design and avoid storing explicit layout metadata for each object [1], [15], [16], [6]. The most popular class of deterministic algorithms is consistent hashes [17]. Consistent hashes map object identifiers to the "K closest" server identifiers based on a numerical distance metric. The calculation is stable in that if a server fails, it will affect the mapping of only object replicas that were owned by that server. All surviving replicas remain in their previous locations. This minimizes the amount of data that must be transferred following a failure.

Figure 1 shows an example of a one-dimensional ring consistent hashing method. For simplicity in this example we limit the object ID and server ID space to the values from 0 to 100. Servers are assigned numerical identifiers that can be visualized as positions on a ring. In this case the server IDs are randomized: 12, 27, 60, 75, and 87, although some systems may explicitly assign IDs in order to more evenly distribute them in the ID space. Each object is mapped to the K closest server IDs clockwise from the object ID's position on the ring. Note that the "distance" on this ring is a virtual construct that has no relation to physical locality. In this example object 33 is 3-way replicated across servers 60, 75, and 87. If server 75 were to fail, then the two surviving replicas would remain in place, and server 12 (the next closest continuing in a clockwise direction) would be responsible for generating a new replica to take its place. This algorithm can be extended in a number of ways, most notably by adding virtual servers such that a given server appears in multiple locations in the ring to better balance average load [16].

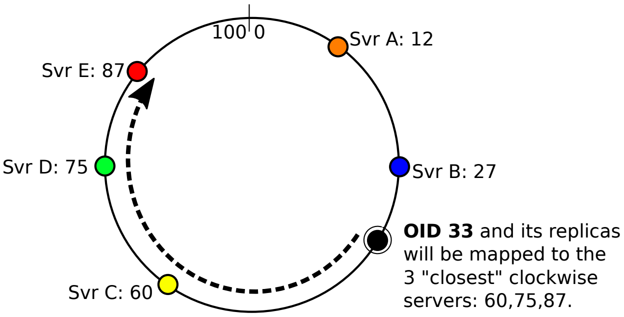
100
0
Svr A: 12
Svr B: 27
Svr C: 60
Svr D: 75
Svr E: 87
OID 33 and its replicas will be mapped to the 3 "closest" clockwise servers: 60,75,87.

Fig. 1. Object placement example: a one-dimensional consistent hashing ring.

More sophisticated placement algorithms include CRUSH [11], which was developed by Weil et al. and forms a critical component of the Ceph file system [1]. CRUSH organizes storage targets into a hierarchical *cluster map* (e.g., by rows, cabinets, shelves, and devices). *Placement rules* govern how replicas are distributed within that hierarchy. For example, a placement rule may enforce that replicas of a given object span fault domains to improve resilience, or a placement rule may enforce that replicas be placed in the same bucket to improve locality.

Each level of the cluster map hierarchy is referred to as a *bucket*, and each bucket uses a pluggable algorithm to map objects to targets. In some sense CRUSH can therefore be thought of as a suite of placement algorithms organized into a hierarchy with flexible placement rules. The most popular bucket type is the straw bucket. The straw bucket algorithm chooses a location for an object by hashing together the object ID, replica number, and target ID for each candidate target in the bucket. The target with the "longest straw," or the highest hash value, is chosen to store the object. Each target can also be assigned a scalar weight value to influence the distribution of objects across targets. The Ceph file system (which uses CRUSH) does not actually execute the CRUSH algorithm independently for each object in the storage system, however. Objects are instead translated into a smaller, fixed number of

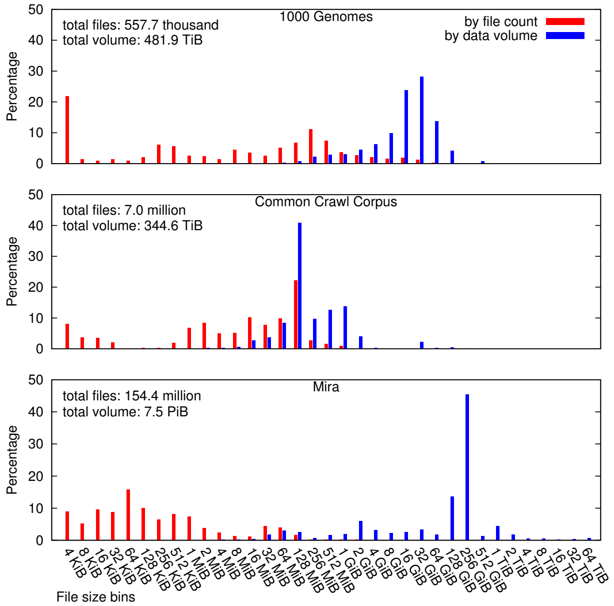

Fig. 2. Distribution of file sizes across example data sets.

placement groups, and placement is calculated per group rather than per object.

In previous work we developed a modular consistent hashing library known as `libch-placement` to evaluate tradeoffs in consistent hashing algorithms [18]. In this work, we add support for the CRUSH algorithm under the same abstract API. The rebuild simulator used in this work leverages libch-placement to interchangeably employ a variety of CRUSH or consistent hashing algorithms for the placement of data.

B. Data populations

Large-scale storage system behavior is dependent not just on algorithmic decisions but also on the nature of the data being stored. Data population characteristics such as object size can influence a variety of runtime behaviors, such as the ratio of control to data messages, disk performance (seek time vs. streaming throughput), pipeline transfer depth, and the duration of time that pairs of servers are occupied by individual object transfers.

Figure 2 compares the distribution of file sizes across three different example data populations.[1] The first two (the contents of the 1000 Genomes [19] catalog of gene sequencing data and the Common Crawl Corpus [20] catalog of web crawler data) are available as Amazon Public Data Sets [21]. They are intended to be processed by using a Hadoop [22] framework. The third example shows the contents of the GPFS parallel file system used on Mira, an IBM Blue Gene/Q system administered by the Argonne Leadership Computing Facility. Mira contains data from a diverse collection of scientific domains in support of the DOE INCITE program [23], and the data is accessed via conventional file system interfaces or high-level parallel I/O libraries. These histograms display both the number of files (red) and the volume of data (blue) for each file size bin. This is an important distinction; for example, the 1000 Genomes data set contains a large number of metadata files that are 4 KiB or less in size, but they constitute a negligible fraction of the overall data volume.

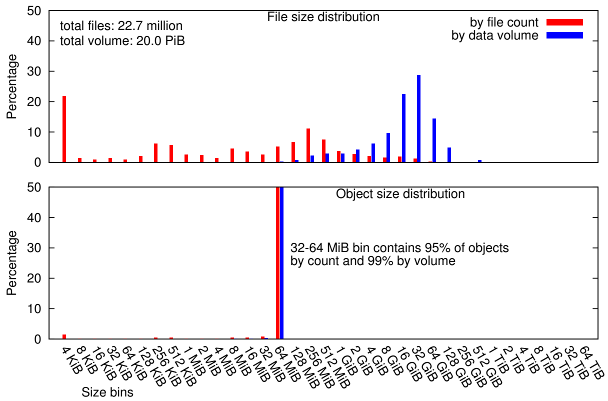

Fig. 3. Distribution of file sizes in a synthetic data set generated by weighted histogram sampling of the 1000 Genomes histogram.

In this study we focus on the 1000 Genomes use case. To generate a representative object population, we first select file sizes via weighted random sampling of the 1000 Genomes histogram. We then divide files into constituent objects according to Hadoop File System chunking policies [22], [24]: each file is split into a collection of distinct 64 MiB objects. This histogram sampling methodology can generate surrogate object populations at a variety of scales while still retaining the aggregate characteristics of a real-world data set. Figure 3 shows an example of the application of this synthetic data population generation strategy. We generated a 20 PiB data set based on the 1000 Genomes population and plot its distribution in the top portion of the figure to confirm that it matches trends observed in the top graph of Figure 2. The second graph in Figure 3 shows a histogram of *object* sizes for the same synthetic data set assuming that each file is divided into 64 MiB objects. Although smaller objects are present, the population is dominated by the 64 MiB objects that contain chunks of data for the largest files in the population.

C. Rebuild protocols

Once a fault has been detected and confirmed in a distributed object storage system, the surviving servers must generate new replicas in order to regain the original level of resilience. We use the term *rebuild protocol* to refer to the steps in this process. Rebuild protocols can be categorized by which entity (or entities) in the system coordinates the data transfers, whether data is *pushed* or *pulled*, and how the data itself is transmitted (i.e., concurrency and pipelining).

Although some storage systems elect to drive the rebuild protocol from centralized subsets of nodes [3], our simulator follows a model similar to that of Ceph [1] in which each

[1] File size data was collected from the 1000 Genomes and Common Crawl Corpus data sets via the S3 interface and from the Mira data set using the GPFS `mmapplypolicy` utility.

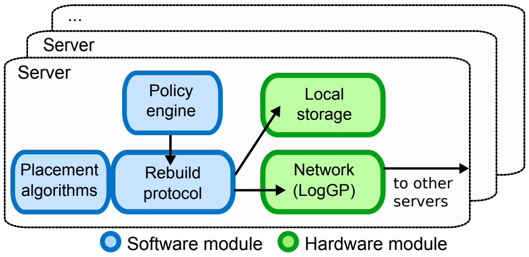

Fig. 4. Discrete event simulation components.

surviving server is independently responsible for generating its own replica copies. Each server therefore recovers at its own pace with a high degree of concurrency. This approach is naturally complemented by a "pull" transfer model in which the destination servers explicitly request data from source servers. This transfer model avoids overwhelming any one server by allowing each server to control its own rebuild rate.

The transfer itself is *pipelined* in our simulator, as in modern implementations, in order to increase utilization of both the network and disk resources. The pipelining method and parameters are described in greater detail in Section III-C. We do not explicitly limit the pipeline depth other than to constrain the memory consumption on each server to 1 GiB for buffering incoming data and 1 GiB for buffering outgoing data.

Our rebuild simulation includes distributed coordination, pull-based transfers, and data pipelining. It does not model the auxiliary fault-response steps of fault detection or calculation of which objects to repair. The simulation instead begins at time zero with the assumption that the fault and its impact have already been assessed, and we focus our attention on the data transfer component of fault recovery. We do not ignore computational overhead, however; it is evaluated separately in Section V.

III. Simulation Methodology

Discrete event simulation is a powerful tool for evaluating large-scale storage systems. It enables the observation of subtle, time-varying, and workload-dependent behavior that cannot be captured by analytical models. It also enables the exploration of fault scenarios that are expensive or otherwise impractical to induce in real-world systems. In fact, if the simulation performs well, it can be used to execute ensemble experiments that reveal statistical trends over a large number of randomized samples.

We developed a discrete event simulation of a distributed object storage system[2] using the CODES [25] toolkit, which in turn is built on the ROSS high-performance parallel discrete event simulator [26]. The storage system model is decomposed into submodels for key hardware components and software components as illustrated in Figure 4. The policy engine is

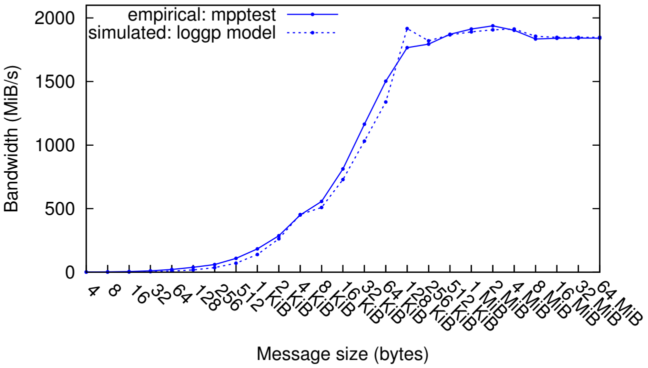

Fig. 5. Point-to-point bandwidth comparison between empirical and simulated performance on QDR InfiniBand network with MPI.

responsible for managing overall server state. The rebuild component implements a distributed data transfer protocol atop disk and network resources. It employs an object placement algorithm component to determine the location of replicas on the system. The simulation executes with network message and disk buffer granularity and tracks the state of every object in the system. Although we use Ceph's CRUSH algorithm for object placement, the simulator does not model the actual Ceph file system. It is a generalized model of a distributed object rebuild protocol.

A. Network model validation

We chose a production Linux cluster at Argonne National Laboratory to serve as a representative example of a data center platform and interconnect. Each node contains two 2 GHz AMD Opteron 6128 processors, 64 GiB of main memory, and a single-port Mellanox ConnectX 2 QDR InfiniBand NIC. The nodes are interconnected via four Mellanox IS-5600 switches that are shared with other computational resources.

We modeled communication costs for the InfiniBand network using a LogGP model [27]. We assume that each node has a full-duplex network card with independent send and receive queues and infinite buffering in the switch complex. The parameters for our LogGP model are obtained by using the `netgauge` utility [28]. Our simulation deviates from the traditional LogGP model in two ways. First, `netgauge` assumes that the overhead parameter (o) (representing the CPU time consumed during transmission) overlaps with network fabric transmission costs on modern networks, so we do not apply the o parameter to the communication time calculation. Second, we take advantage of the fact that `netgauge` calculates LogGP parameters independently for a range of message sizes by using these parameters in a lookup table in our model. This approach allows the model to more accurately reflect protocol crossover points and other fabric-specific characteristics.

Figure 5 compares the empirically measured point-to-point bandwidth on the Linux cluster (measured by using `mpptest` [29]) with a simulation of the point-to-point performance using our simulation framework. We see that the simulated performance closely matches the performance trends on

[2] The source code for this simulation and corresponding CRUSH enhancements will be released under an open source license concurrent with the publication of this paper.

the example system, including an apparent protocol crossover point between 4 KiB and 8 KiB as well as an unexpected decline in performance between 4 MiB and 8 MiB. The overall RMSE of the model is 58.128 MiB/s for this experiment.

B. Disk model validation

We assume that each storage server is attached to a commodity storage array (JBOD). We use an LSI SAS 9207-8i controller and LSI SAS2x36 expander as found in a DataON DNS-9470 product as an example of this class of storage device. Our example JBOD was configured with 10 Seagate Constellation ES3 3 GB SATA drives, each with an advertised peak throughput of 175 MB/s, an average latency of 4.16 ms, and 128 MiB of cache. The disks were combined into a single volume using software RAID (mdraid) in a RAID0 configuration with a 512 KiB chunk size.

The design of our disk model was driven by two requirements. The first requirement was ROSS simulation framework compatibility. ROSS achieves its highest performance at scale when operating in optimistic mode via the Time Warp [30] protocol: each model entity independently and speculatively processes its own local event population. If it receives an event with a time stamp that is out of order with respect to its current state, then it uses reverse computation to "roll back" to a coherent point in time. The popular DiskSim [31] model is incompatible with this approach because it does not provide a mechanism to reverse state. We will investigate the possibility of adding this capability to DiskSim in future work. The second requirement for our disk model was that it be based on real-world device characteristics such as seek time and bandwidth. This approach enables "what-if" exploration by altering model parameters to reflect hypothetical hardware configurations, but it precludes the use of black-box disk models [32], [33]. We instead elected to construct a disk model based on the analytical techniques described by Ruemmler and Wilkes [34].

The input parameters used in this study were collected by executing the `fio` [35] benchmark on the DataOn test system with 10 disks. We measured I/O performance for 60 seconds at each access size as the access size was varied from 4 KiB to 8 MiB for each of sequential read, sequential write, random read, and random write access patterns. The `libaio` engine was used in all cases with an iodepth of 4 and direct I/O. We set the model's rate parameter to the maximum rate achieved by the sequential read and sequential write measurements. The overhead parameter was set to the median completion latency minus the transfer rate for the 4K request size. The seek parameters were set to the difference between the sequential and random latency values divided by the iodepth (4) at each request size. Read and write operations therefore use different rate and overhead values in our model, while the seek time for random operations is determined via a lookup table based on the access size. We define a random I/O operation as any operation that does not begin within 512 bytes of the previous operation. The Ruemmler and Wilkes model calls for a fixed seek parameter, but we were unable to achieve a good model fit using this approach. We believe the reason is caching behavior, which is not reflected in the Reummler and Wilkes model.

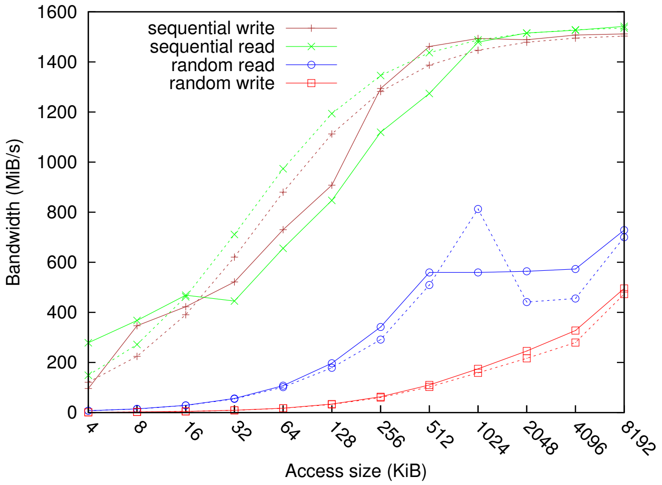

Fig. 6. Disk bandwidth comparison between `fio` benchmark (solid lines) and simulation results (dashed lines) as the access size is varied.

TABLE I
ROOT MEAN SQUARE ERROR OF DISK SIMULATION.

Mode	RMSE (MB/s)	RMSE (%)
random read	10.75	8.69
random write	11.95	3.84
sequential read	90.75	9.45
sequential write	13.32	1.36

Figure 6 compares the empirically measured disk bandwidth (measured using `fio`) with results from our simulation environment when configured to continually issue four concurrent I/O operations. Table I shows the overall RMSE for each simulation mode (also known as the demerit figure [34] for the model). We have a cumulative error of less than 10% in each mode.

C. Pipelining protocol

Pipelining is a common technique for optimizing large inter-server data transfers [36], [24], [22] and is therefore critical to performance when rebuilding replicas. We implement pipelining in our model by dividing objects into smaller buffers and then transferring those buffers with asynchronous network operations and multithreaded disk I/O. The pipeline buffer size should be large enough to amortize startup costs but small enough to maximize concurrency. Figure 7 shows the results of a simulated pipelined transfer of a single 1 TiB object between two servers using the network and disk parameters obtained in Sections III-A and III-B. Each server was configured to commit a maximum of 1 GiB of RAM to receiving/writing data and 1 GiB of RAM to reading/sending data. We use separate memory buffer pools for sending and receiving in order to ensure that servers are always able to make forward progress on rebuilding their own local replica even while servicing pull requests from peers. The pipelined transfer reaches a near-peak rate of 1.5 GiB/s using a 16 MiB buffer size; both servers are bottlenecked by storage throughput. We

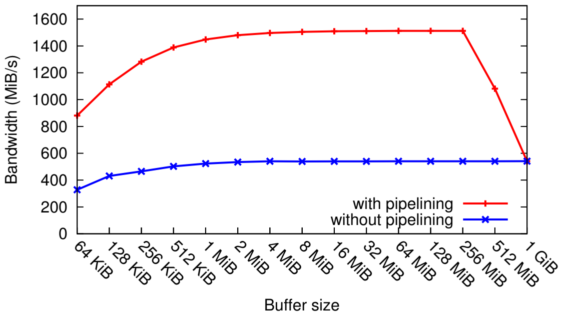

Fig. 7. Sustained server-to-server rebuild bandwidth for a 1 TiB object as the pipeline buffer size is varied from 64 KiB to 1 GiB.

TABLE II
TOTAL NUMBER OF PLACEMENT GROUPS AND OBJECTS IN EACH CONFIGURATION

No. of Servers	Placement Groups	Objects
4	128	1,373,187
8	512	2,745,833
16	4096	5,494,324
32	4096	10,990,294
64	4096	21,984,426
128	8192	43,971,596
256	16384	87,949,846
512	32768	175,897,623
1024	65536	351,765,871

also plot the transfer bandwidth with pipelining disabled and observe that the servers are unable to saturate their hardware resources in this configuration. Based on these results, we configured our storage system model to enable pipelining with a buffer size of 16 MiB. Note that the peak aggregate transfer bandwidth is roughly approximated by the point-to-point peak bandwidth for $n/2$ server pairs. This hypothetical configuration would not take advantage of full-duplex network capability, but it would produce sequential streaming access patterns at each storage device. For 512 server pairs (1,024 total servers) at 1.5 GiB/s, this results in a possible aggregate rate of 768 GiB/s. This rate is well within the capabilities of a large-scale InfiniBand switch complex.

IV. CRUSH CASE STUDY

We begin our experiments by evaluating the rebuild behavior of a hypothetical object storage system with the following configuration. Objects are mapped to servers by using the CRUSH algorithm as implemented in Ceph 0.94.3. The object population is generated from a histogram of 1000 Genomes file sizes as described in Section II-B. The network, disk, and pipelining parameters are set to reflect the hardware parameters as described and validated in Section III.

Each object is 3-way replicated. The overall object population size is scaled to produce at least 20 TiB of data (60 TiB when accounting for 3-way replication) and at least one million objects per server. CRUSH was configured with a flat CRUSH map (i.e., all servers belong to a single bucket with equal weights) using a straw bucket type. Furthermore, we followed the placement group parameters recommended in the Ceph documentation [37] such that the number of placement groups in the storage system increases with the number of servers rather than the number of objects. Table II shows the number of placement groups and number of objects (not counting replicas) used in each configuration.

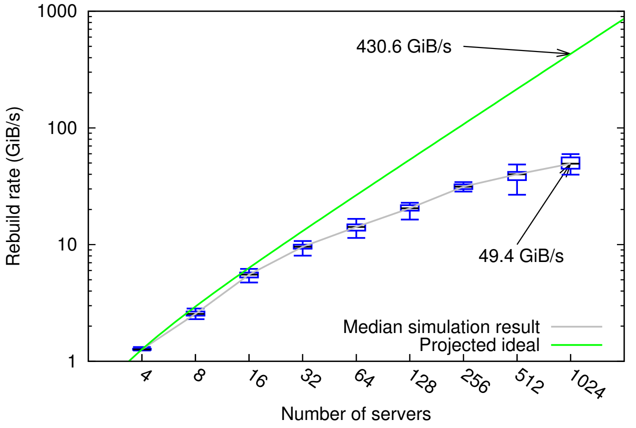

Fig. 8. Simulated aggregate rebuild rate following a single server failure using an example CRUSH configuration.

The simulation begins with the failure of a single randomly chosen server. We then measure the elapsed time for all affected replicas to be reconstructed on surviving servers. We do not simulate any auxiliary procedures such as fault detection or synchronization; this model focuses exclusively on reading, transferring, and writing replica data, the most time-consuming aspect of fault recovery in our problem domain. We divide the aggregate amount of data transferred by the elapsed time to produce an *aggregate rebuild rate*. Figure 8 shows the aggregate rebuild rate for a range of system sizes from 4 to 1,024. Both axes use a logarithmic scale. We gathered 15 random samples for each configuration by choosing a random server to fail. The only exceptions are the 4 and 8 server data points, in which we gathered 4 and 8 samples respectively because there are not 15 distinct servers to select for failure. Box-and-whiskers plots illustrate the minimum, 1st quartile, median, 4th quartile, and maximum for each set of samples.

We also annotate the graph by extrapolating ideal, linear scaling from the 4-server median value; this is shown in green. For up to 16 servers, we see that the simulated rebuild rate tracks the ideal rebuild rate reasonably well. It tapers off as the system increases in size, however, ultimately reaching a median rebuild rate of 49.4 GiB/s with 1,024 servers. This aggregate rebuild rate is an order of magnitude slower than the projected ideal rate.

We selected the slowest 64-server configuration for further investigation: this example clearly shows suboptimal performance yet is small enough for clarity in visualization. We instrumented the simulation to record the exact volume of data transferred between pairs of servers and plotted those values using the Circos software package [38]. The result is shown in Figure 9. Servers are represented by color-coded rectangles around the perimeter, and each is labeled with a server index

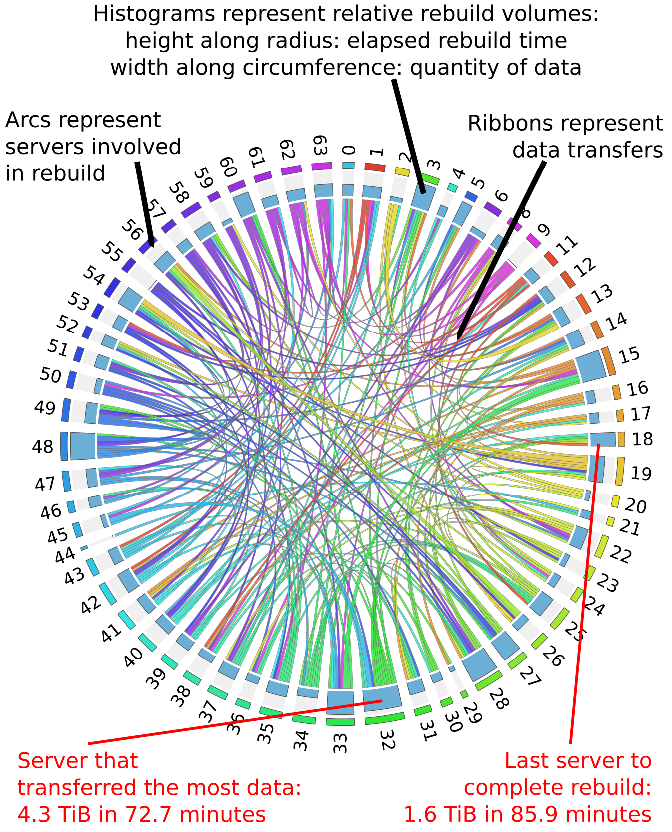
Histograms represent relative rebuild volumes:
height along radius: elapsed rebuild time
width along circumference: quantity of data
Arcs represent servers involved in rebuild
Ribbons represent data transfers
Server that transferred the most data: 4.3 TiB in 72.7 minutes
Last server to complete rebuild: 1.6 TiB in 85.9 minutes
0 1 2 3 4 5 6 7 8 9 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

Fig. 9. Circos diagram of data transfers between servers in the slowest 64-server configuration from Figure 8.

number from 0 to 63. Server 10 is omitted because it is the server that failed in this example. A small histogram is also associated with each server: its width indicates the relative volume of data transferred (both sending and receiving) by that server, while its height indicates the amount of time that the server took to complete its rebuild procedure. Ribbons connecting pairs of servers represent the flow of data between those servers.

This figure illustrates a critical phenomenon: the rebuild load is not well balanced among surviving servers. The annotations at the bottom highlight an additional nonintuitive finding: the server that transferred the most total data was *not* the server that took the longest to complete its rebuild procedure. Server 32 exchanged data with many more peers than did server 18, allowing it to diversify its traffic and complete its rebuild procedure sooner despite transferring more data. This result indicates the presence of hot spots or data dependencies that hindered some servers more than others. While advanced scheduling policies beyond the scope of this paper could help mitigate such hindrances, the underlying imbalance is an unavoidable shortcoming of the placement algorithm when used at this scale.

Figure 10 investigates the root cause of the imbalance by showing a histogram, sorted by count, of the number of new replicas generated on each of the 63 surviving servers. The newly generated replicas are not evenly distributed; in fact, this example reveals a stair-stepped pattern with the most active server receiving over 35,000 objects and the least active servers receiving no objects at all. We use the right side axis to overlay a plot of the number of new placement groups each server joined as a result of the fault. The range of the per-server placement group count varies from 0 to 7 but clearly matches the trend in the per-server object count.

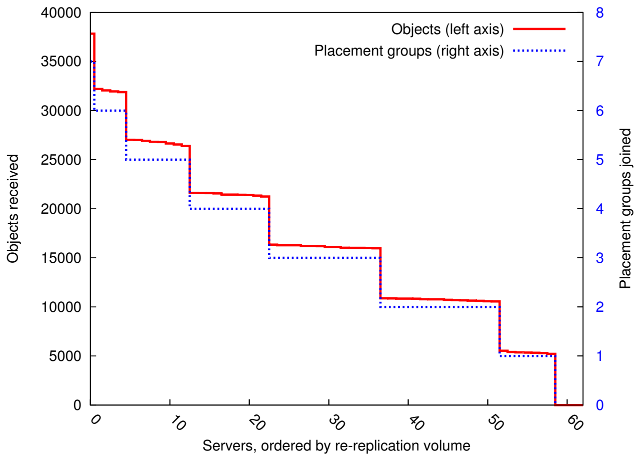

Fig. 10. Histogram of the number of new replicas received by each server from Figure 9, sorted by volume. The corresponding PG layout is overlayed for comparison on the 2nd y axis.

Recall from Table II that this 64-server system is using 4,096 placement groups. With 3-way replication, each server therefore participates in an average of 192 groups. The failed server in this specific example (server 10) participates in 190 placement groups. When server 10 fails, each of those 190 placement groups must promote exactly one new server to compensate for the loss of server 10. Because the CRUSH algorithm is pseudo-random, however, those 190 replica targets are not guaranteed to be evenly distributed across the 63 surviving servers. In fact, one server (server 28) was added to 7 placement groups, while four other servers (servers 7, 9, and 55) were not added to any new placement groups at all.

From this analysis it is clear that **subtle high-level placement policies, such as the use of placement groups in Ceph, can degrade recovery time by restricting replica declustering in unexpected ways**. In this case aggregate rebuild performance is constrained in a way that is not obvious at modest scale. The placement policy is too course-grained to take advantage of the available aggregate bandwidth on larger systems.

A. *Hierarchical CRUSH maps*

One of the distinguishing features of CRUSH is that it enables the construction of hierarchical maps that organize storage targets according to their physical layout. A production storage system with hundreds or thousands of servers would likely utilize this functionality rather than placing all servers in a single flat bucket. We revisited the three largest configurations from Figure 8 using a hierarchical cluster map in order to

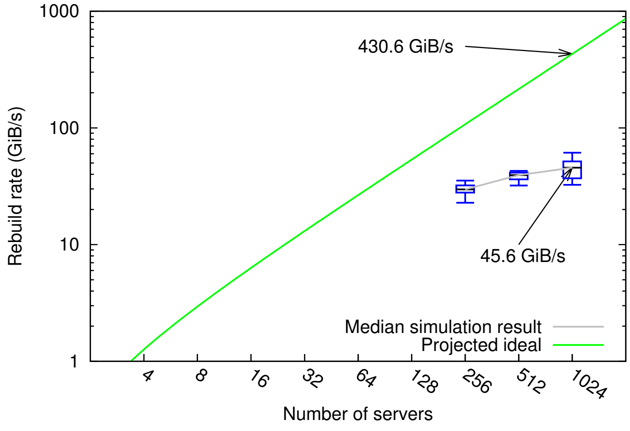

Fig. 11. Simulated aggregate rebuild rate following a single server failure using a CRUSH configuration with a hierarchical cluster map.

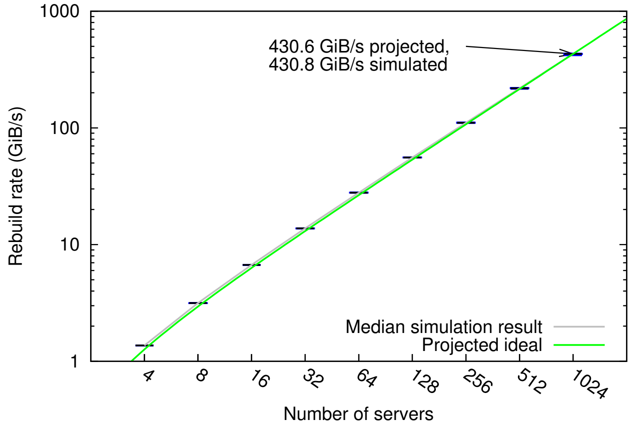

Fig. 12. Simulated aggregate rebuild rate following a single server failure by using an example CRUSH configuration with no placement groups.

investigate how this configuration would impact placement and rebuild behavior. We assumed that servers were organized as follows: 16 servers per rack, 8 racks per row, and either 2, 4, or 8 rows depending on whether there were 256, 512, or 1,024 total servers in the system. We then constructed a CRUSH placement rule enforcing that each replica for an object must reside on a different rack.

Figure 11 shows the result of repeating the simulation in this configuration. The median aggregate rebuild rate with 1,024 servers was slightly reduced in comparison with the flat bucket configuration (down to 45.6 GiB/s rather than 49.4 GiB/s). The reason that the rack-aware placement rule slightly reduces the declustering relative to the flat bucket topology by limiting the number of valid replica set permutations in order to protect against expected correlated failure modes. We focus our analysis on flat topologies for clarity in the remainder of this work.

V. Improving rebuild performance

In this section we revisit the simulations from Section IV to evaluate strategies for improving the aggregate rebuild rate. Although we use the CRUSH placement algorithm as our starting point, our goal is not to prescribe Ceph-specific tuning parameters but rather to explore how object placement algorithms in general can be architected to improve rebuild time and MTTDL. Recall from Section III that our simulation does not model the Ceph file system and therefore does not necessarily capture the ramifications of changes to recommended Ceph parameters.

A. Strategy: eliminating placement groups

One straightforward potential strategy to improve declustering in the CRUSH algorithm would be to simply eliminate placement groups and instead calculate the placement of each object independently using the CRUSH algorithm. Figure 12 shows the aggregate rebuild rate that can be achieved using this approach. As in Figure 8, we plot the ideal scaling alongside the simulated rate for comparison. In this case we find that the throughput closely matches the projected ideal at all tested scales because of a very even distribution of rebuild traffic among surviving servers. We observe that **distributed rebuild protocols are capable of near-ideal scalability when using object-granular replica placement policies**. This level of granularity also opens up the possibility of finer-grained scheduling or prioritization of reconstruction to further improve MTTDL [12]. Note that the hypothetical removal of placement groups might have significant ramifications in the Ceph storage system as a whole. The placement group construct serves a variety of purposes in fault detection, write-ahead logging, and peering. Analysis of such Ceph-specific design issues is beyond the scope of this paper, however; we limit our analysis to the more general problem of data layout in object storage systems.

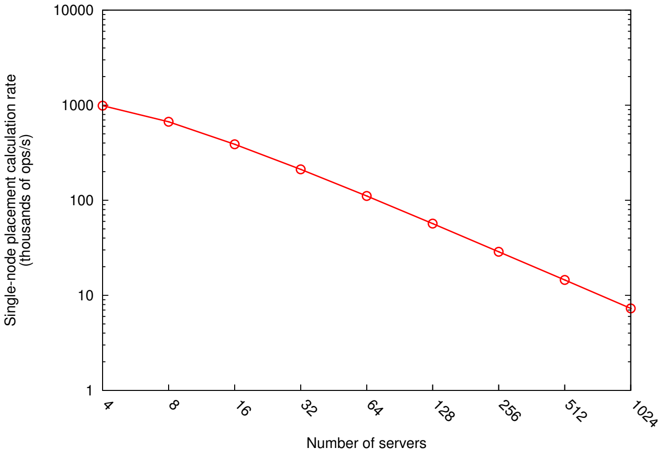

Fig. 13. CRUSH straw bucket calculation rate as the number of servers in the bucket is increased.

Object-granular placement has a downside, however. Our simulation does not model the CPU cost of the object placement calculation, just the data movement cost assuming that the new replica locations are already known. The straw bucket algorithm is inherently computationally expensive it requires an $O(n)$ calculation per object, where n is the number of servers in the system. Recall from Section II-A that the target server for a given object ID cannot be chosen without first hashing that ID against the ID of every server in the system. This situation is illustrated in Figure 13, which shows the placement calculation rate in a CRUSH straw bucket as the number of servers is increased. This measurement was performed on a Linux node equipped with two 2.4 GHz Intel Haswell E5-2620 v3 processors and a randomly generated collection of object ID values. The CRUSH calculation can be performed at a rate of nearly 1 million objects per second with 4 servers in the bucket. The rate falls to 7 thousand objects per second with 1,024 servers in the bucket. This would be prohibitively expensive for object-granularity placement in systems with billions of objects, especially when invoking procedures that call for bulk placement calculation such as rebuild, file system consistency checking, and data scrubbing. In some cases this computation could be overlapped with other activity, but efficient bulk calculation would offer more flexibility in how those operations can be performed.

B. Strategy: employing consistent hashes as bucket algorithms

The CRUSH algorithm also provides `uniform`, `list` and `tree` bucket types as alternatives to the straw bucket type, but those algorithms require superfluous reshuffling of unaffected objects when servers are added or removed, a property that makes them undesirable for use in large-scale storage systems. An ideal CRUSH bucket algorithm would avoid superfluous replica movement while striking a balance between replica declustering capability and computational efficiency. In previous work we demonstrated that multiple consistent hashing algorithms can meet this goal [18], the most straightforward of which is a one-dimensional ring with a Euclidian distance metric, similar to that illustrated in Figure 1, but with a high ratio of virtual nodes for each physical node. If the virtual node ID values are pseudo-randomly generated, then they can dramatically increase the number of replica ordering permutations on the ring.

This ring-based consistent hashing algorithm can be adapted for use as a CRUSH bucket algorithm as follows. When the CRUSH bucket is initialized, $N \times V$ pseudo-random virtual IDs are generated by hashing each of the physical server numbers (or *items* in CRUSH terminology) with a sequence of values from 0 to V. The resulting virtual IDs (along with mappings back to their original underlying bucket items) are placed in an array and sorted by virtual ID value. New virtual IDs can be generated and added to the array if the bucket is expanded. Each virtual ID element consumes an additional 32 bytes of memory in our current implementation (one integer each for the virtual ID, underlying physical ID, array index, and total array size) though that could be optimized for more compact memory use.

The CRUSH lookup function performs an $O(\log n)$ binary search through the sorted virtual ID array to find the numerically closest virtual ID to a given object ID. CRUSH weights can be applied by scaling the number of virtual IDs generated for a given item to alter the frequency that its virtual nodes appear on the ring, thereby altering the probability that it will be chosen in the lookup routine.

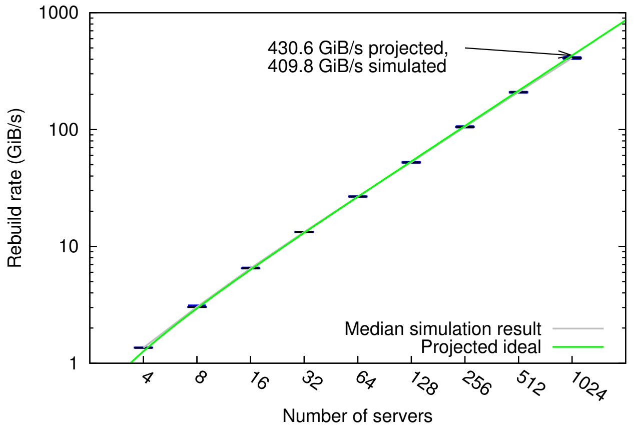

Fig. 14. Simulated aggregate rebuild rate following a single server failure by using the CRUSH vring bucket type and no placement groups.

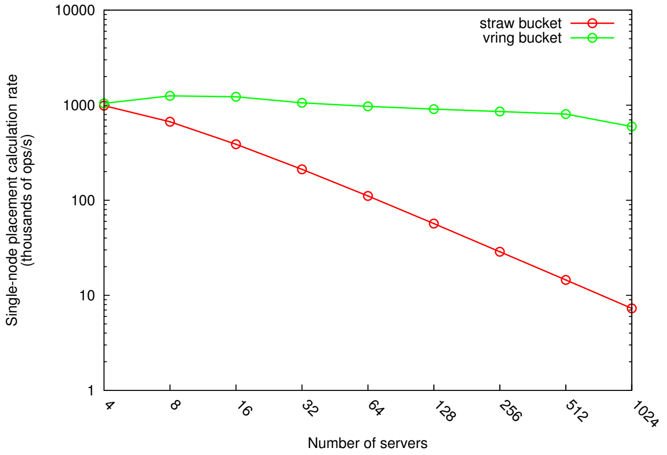

Fig. 15. CRUSH vring bucket and straw bucket calculation rate as the number of servers in the bucket is increased.

We implemented this algorithm in a new CRUSH bucket type (called the `vring` bucket type) and set the default value of V to 1,024 (i.e., 1,024 virtual IDs per item). Figure 14 shows the result of repeating the aggregate rebuild rate measurement of Figure 12 using this new bucket algorithm. At 1,024 servers it is within 5% of the ideal scaling target. This is slightly slower than the rate achieved using the `straw` bucket type with per-object placement (i.e., no placement groups), but in Figure 15 we see that it achieves this rebuild rate with a much lower computational overhead. At 1,024 servers it can calculate object placement at nearly 597 thousand objects per second, an 82x improvement over the straw bucket algorithm. This improvement in computational efficiency would reduce overall latency as well as the time needed to recalculate placement for objects following a failure. We therefore find that **consistent hashing algorithms can be adapted for use within CRUSH to significantly reduce CPU cost without**

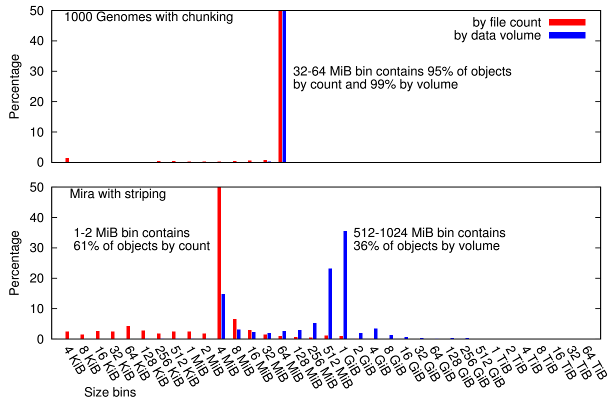

Fig. 16. Comparison of object sizes in synthetic data sets generated by weighted histogram sampling.

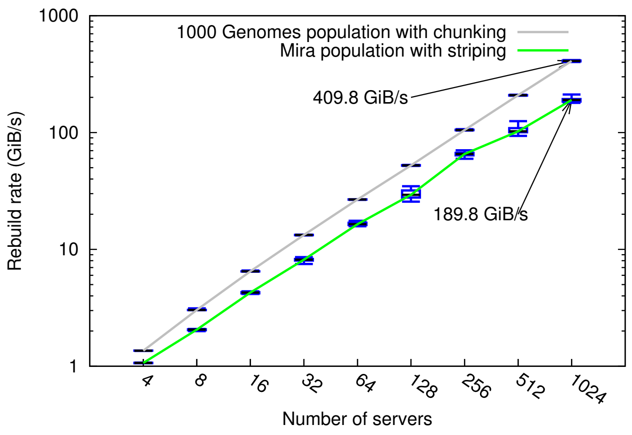

Fig. 17. Simulated aggregate rebuild rate following a single server failure using the CRUSH vring bucket type with two different data population examples.

compromising declustering.

C. Impact of data population

We observed in Section II-B that different data sets can exhibit substantially different distributions of file and object sizes. We repeated the experiments shown in Figure 14 with a different object population to illustrate how this can impact an experimental evaluation. The new object population was generated by weighted statistical sampling of the Mira file system histogram. Each file was then decomposed into striped (rather than chunked) object sets of up to $N/3$ objects, each of which in turn used 3-way replication. The size of the objects was set according to a round-robin striping policy with a 4 MiB stripe unit as might be found in a PVFS [39] or Lustre [2] parallel file system.

Figure 2 already compared the 1000 Genomes and Mira data sets in terms of distribution of file sizes. When we compare the distribution of underlying object sizes, the discrepancy is even more pronounced, as seen in Figure 16. The Mira data set includes a large number of small files that in turn map to small objects in the synthetic object population. The largest objects in the Mira population are also far larger than the largest objects in the 1000 Genomes population. In this example, the largest individual object contains 186 GiB of data. The reason is that the Hadoop-style chunking imposes an object size limit of 64 MiB, while a round-robin striped object layout has no such limit on underlying object size. The objects continue to grow indefinitely as the file is appended.

Figure 17 illustrates how this difference in object populations translates into aggregate rebuild performance by comparing rebuild rates for the 1000 Genomes data set and the Mira data set. The scaling trend is similar, but the object population based an HPC file system with striping achieves a median aggregate rebuild rate of only 189.8 GiB/s, less than half of the aggregate rebuild rate of the same experiment with the 1000 Genomes object population. There are two reasons for this. The first is that the greater number of small objects reduces efficiency by increasing the ratio of control message traffic to data transfer traffic during rebuild. A more significant factor, however, is that the presence of much larger objects exacerbates contention during rebuild. Longer object transfers will dominate server activity and potentially delay the transfer of smaller objects unless the system implements policies to prevent this phenomenon. A general solution for maintaining rebuild efficiency when faced with a wide variety of data-set properties is beyond the scope of this work. For now we simply observe that **data population characteristics can have a significant impact on the efficiency of distributed rebuild algorithms**. Data uniformity assumptions could potentially skew the results of experimental evaluations.

D. Assessment of PDES methodology

All the simulations in this study were performed by using optimistic parallel discrete simulation in the CODES simulation toolkit. The most complex of these were the 1,024-server simulations using the striped Mira population example in Figure 17. The first sample in that set tracked the state of roughly 3.9 billion replicas (60 PiB), 3.8 million of which (61 TiB) were rebuilt during the simulation and modeled with message-level granularity. This effort required processing over 200 million discrete events. We executed the simulation using 256 MPI processes spread across 22 nodes of a Linux cluster equipped with 2.4 GHz Intel Haswell E5-2620 v3 processors and an InfiniBand network. It completed in 30.2 seconds, yielding an effective simulation rate of 6.7 million events per second. This performance not only enabled rapid turnaround time on experimental questions but also allowed us to execute ensemble simulations with random failure permutations in order to distinguish aggregate trends from outliers. We found that **high-fidelity parallel discrete event simulation enables otherwise intractable evaluation of fault scenarios at scale**. The largest scenarios evaluated in this work not only are impractical for real-world experimentation but also exceed the capabilities of sequential simulation in some cases. If the example highlighted above is executed serially, it exhausts the

384 GiB of RAM available on any individual node of our simulation platform and thus fails to complete.

VI. Related work

This work relies heavily upon the CRUSH algorithm developed by Weil et al. [11]. The CRUSH framework offers considerable flexibility in object placement by providing a generalized method to express hierarchical organizations, decoupling placement rules from the topology description, and providing a modular mechanism to experiment with bucket algorithms. It is also a valuable testbed for experimentation with placement strategies as shown in this work.

Venaktesan et al. contrasted clustered and declustered placement in distributed storage [9], [12] using both analytical and simulation methods. Their work revealed previously unknown relationships between declustering methods, replication levels, and the effective MTTDL of a storage system. They did not evaluate object-based storage specifically, however, or identify placement algorithms that could be used to achieve declustered placement.

Cidon et al. [40] explored placement strategies that reduce the probability of data loss events in distributed storage by reducing the number of server combinations (and thus correlated failure scenarios) that hold all copies of a given object. Their work assumes a high fixed cost of backup recovery upon data loss, in which case reducing the probable frequency of data loss is much more important than reducing the scope of data loss. Our work in contrast focuses on efficient handling of faults that do not result in data loss. Future storage systems will likely need to strike a balance between these two goals according to their expected failure modes.

Wozniak et al. studied the rebuild behavior in object storage systems using a coarse-grained simulation [41]. They calculated the fraction of the overall workload serviced by the most heavily loaded servers during rebuild but did not consider the impact of declustering.

Welch et al. described distributed object reconstruction in the Panasas Parallel File System [3]. Their approach uses uniform random placement rather than an algorithmic placement function, uses parity encoding for large files, and drives reconstruction from metadata managers rather than independently at each server.

VII. Conclusions and future work

We evaluated the impact of data placement policies on resilience using a parallel discrete event simulation of large-scale replicated object storage rebuild protocols. Our findings include the following:

- Subtle high-level placement policies, such as the use of placement groups in Ceph, can degrade recovery time by restricting replica declustering in unexpected ways.
- Distributed rebuild protocols are capable of near-ideal scalability when using object-granular replica placement.
- Consistent hashing algorithms can be adapted for use within CRUSH to significantly reduce CPU cost without compromising declustering.
- Data population characteristics can have a significant impact on the efficiency of distributed rebuild algorithms.
- High-fidelity parallel discrete event simulation enables otherwise intractable evaluation of fault scenarios at scale.

These findings highlight both algorithmic and evaluation lessons that can be adopted in future storage systems to improve resilience and performance.

In future work we would like to explore more complex failure modes. We would also like to evaluate the reconstruction of erasure coded data in addition to replicated data. The rebuild protocol model described in this work can also be combined with workload models [42], [43], fault detection protocol models [44], and other components to produce a holistic view of large-scale storage system behavior. We would also like to leverage the variety of submodels available in CODES [45], [46] to evaluate rebuild strategies for different system architectures, such as those that are anticipated in upcoming exascale HPC systems.

Acknowledgments

This material was based upon work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computer Research, under contract DE-AC02-06CH11357. The research used resources from the Argonne Leadership Computing Facility (ALCF).

References

[1] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 307–320.

[2] Open Scalable File Systems, Inc., "Lustre File System." [Online]. Available: http://opensfs.org/lustre/

[3] B. Welch, M. Unangst, Z. Abbasi, G. A. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable performance of the Panasas parallel file system." in *FAST*, vol. 8, 2008, pp. 1–17.

[4] Scality, "The Scality RING." [Online]. Available: http://www.scality.com/ring/

[5] A. W. S. Inc., "Amazon Simple Storage Service (S3)." [Online]. Available: https://aws.amazon.com/s3/

[6] J. Arnold, *OpenStack Swift: Using, Administering, and Developing for Swift Object Storage*. O'Reilly Media, Inc., 2014.

[7] Data Direct Networks, "WOS: Object Storage." [Online]. Available: http://www.ddn.com/products/object-storage-web-object-scaler-wos/

[8] K. K. Rao, J. Hafner, and R. Golding, "Reliability for networked storage nodes," in *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, June 2006, pp. 237–248.

[9] V. Venkatesan, I. Iliadis, X.-Y. Hu, R. Haas, and C. Fragouli, "Effect of replica placement on the reliability of large-scale data storage systems," in *2010 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2010, pp. 79–88.

[10] M. Holland and G. A. Gibson, "Parity declustering for continuous operation in redundant disk arrays," in *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS V. New York, NY, USA: ACM, 1992, pp. 23–35. [Online]. Available: http://doi.acm.org/10.1145/143365.143383

[11] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "CRUSH: Controlled, scalable, decentralized placement of replicated data," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1188455.1188582

[12] V. Venkatesan, I. Iliadis, C. Fragouli, and R. Urbanke, "Reliability of clustered vs. declustered replica placement in data storage systems," in *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on*, July 2011, pp. 307–317.

[13] P. Zave, "Using lightweight modeling to understand chord," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 2, pp. 49–57, Mar. 2012. [Online]. Available: http://doi.acm.org/10.1145/2185376.2185383

[14] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 337–350, 2010.

[15] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[16] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '01. New York, NY, USA: ACM, 2001, pp. 149–160. [Online]. Available: http://doi.acm.org/10.1145/383059.383071

[17] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web," in *Proceedings of the twenty-ninth annual ACM symposium on Theory of Computing*. ACM, 1997, pp. 654–663.

[18] P. Carns, K. Harms, J. Jenkins, M. Mubarak, R. B. Ross, and C. Carothers, "Consistent hashing distance metrics for large-scale object storage (poster)," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC15)*, 2015.

[19] 1000 Genomes Project Consortium and others, "A map of human genome variation from population-scale sequencing," *Nature*, vol. 467, no. 7319, pp. 1061–1073, 2010.

[20] Common Crawl Foundation, "Common Crawl." [Online]. Available: http://www.commoncrawl.org/

[21] A. W. S. Inc., "AWS Public Data Sets." [Online]. Available: https://aws.amazon.com/public-data-sets/

[22] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*, ser. MSST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/MSST.2010.5496972

[23] U.S. Department of Energy, "INCITE program." [Online]. Available: http://www.er.doe.gov/ascr/incite/

[24] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 29–43. [Online]. Available: http://doi.acm.org/10.1145/945445.945450

[25] J. Cope, N. Liu, S. Lang, P. Carns, C. Carothers, and R. Ross, "Codes: Enabling co-design of multi-layer exascale storage architectures," in *Proceedings of the Workshop on Emerging Supercomputing Technologies 2011*, 2011.

[26] P. D. Barnes, Jr., C. D. Carothers, D. R. Jefferson, and J. M. LaPre, "Warp speed: Executing time warp on 1,966,080 cores," in *Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, ser. SIGSIM-PADS '13. New York, NY, USA: ACM, 2013, pp. 327–336. [Online]. Available: http://doi.acm.org/10.1145/2486092.2486134

[27] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, "LogGP: Incorporating long messages into the LogP model – one step closer towards a realistic model for parallel computation," in *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '95. New York, NY, USA: ACM, 1995, pp. 95–105. [Online]. Available: http://doi.acm.org/10.1145/215399.215427

[28] T. Hoefler, T. Mehlan, A. Lumsdaine, and W. Rehm, "NetGauge: A network performance measurement framework," in *High Performance Computing and Communications*. Springer, 2007, pp. 659–671.

[29] W. Gropp and E. Lusk, "Reproducible measurements of MPI performance characteristics," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 1999, pp. 11–18.

[30] D. R. Jefferson, "Virtual time," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 3, pp. 404–425, Jul. 1985. [Online]. Available: http://doi.acm.org/10.1145/3916.3988

[31] J. S. Bucy, J. Schindler, S. W. Schlosser, and G. R. Ganger, "The DiskSim simulation environment version 4.0 reference manual (cmu-pdl-08-101)," *Parallel Data Laboratory, Carnegie Mellon University*, p. 26, 2008.

[32] J. Garcia, L. Prada, J. Fernandez, A. Nunez, and J. Carretero, "Using black-box modeling techniques for modern disk drives service time simulation," in *Simulation Symposium, 2008. ANSS 2008. 41st Annual*, April 2008, pp. 139–145.

[33] A. Crume, C. Maltzahn, L. Ward, T. Kroeger, M. Curry, and R. Oldfield, "Fourier-assisted machine learning of hard disk drive access time models," in *Proceedings of the 8th Parallel Data Storage Workshop*. ACM, 2013, pp. 45–51.

[34] C. Ruemmler and J. Wilkes, "An introduction to disk drive modeling," *Computer*, vol. 27, no. 3, pp. 17–28, 1994. [Online]. Available: http://dx.doi.org/10.1109/2.268881

[35] J. Axboe, "FIO Git repository." [Online]. Available: http://git.kernel.dk/?p=fio.git

[36] R. Haskin, "The Shark continuous-media file server," in *Compcon Spring '93, Digest of Papers.*, Feb. 1993, pp. 12–15.

[37] Inktank Storage Inc., "Ceph documentation: Choosing the number of placement groups," retrieved Feb 2016. [Online]. Available: http://docs.ceph.com/docs/master/rados/operations/placement-groups/#choosing-the-number-of-placement-groups

[38] M. Krzywinski, J. Schein, İ. Birol, J. Connors, R. Gascoyne, D. Horsman, S. J. Jones, and M. A. Marra, "Circos: An information aesthetic for comparative genomics," *Genome Research*, vol. 19, no. 9, pp. 1639–1645, 2009. [Online]. Available: http://genome.cshlp.org/content/19/9/1639.abstract

[39] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur, "PVFS: A parallel file system for Linux clusters," in *Proceedings of the 4th annual Linux Showcase & Conference-Volume 4*. USENIX Association, 2000, p. 28.

[40] A. Cidon, S. M. Rumble, R. Stutsman, S. Katti, J. Ousterhout, and M. Rosenblum, "Copysets: Reducing the frequency of data loss in cloud storage," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 37–48. [Online]. Available: http://dl.acm.org/citation.cfm?id=2535461.2535467

[41] J. Wozniak, S. W. Son, and R. Ross, "Distributed object storage rebuild analysis via simulation with GOBS," in *2010 International Conference on Dependable Systems and Networks Workshops (DSN-W)*, June 2010, pp. 23–28.

[42] S. Snyder, P. Carns, R. Latham, M. Mubarak, R. Ross, C. Carothers, B. Behzad, H. V. T. Luu, S. Byna, and Prabhat, "Techniques for modeling large-scale HPC I/O workloads," in *Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS15)*, 2015.

[43] X. Luo, F. Mueller, P. Carns, J. Jenkins, R. Latham, R. Ross, and S. Snyder, "HPC I/O trace extrapolation," in *Workshop on Extreme-Scale Programming Tools (ESPT 2015)*, 2015.

[44] S. Snyder, P. Carns, J. Jenkins, K. Harms, R. Ross, M. Mubarak, and C. Carothers, "A case for epidemic fault detection and group membership in HPC storage systems," in *Proceedings of the 5th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS14)*. Springer, 2014.

[45] M. Mubarak, C. D. Carothers, R. B. Ross, and P. Carns, "Modeling a million-node dragonfly network using massively parallel discrete-event simulation," in *3rd International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems (PMBS12)*, 2012.

[46] ——, "A case study in using massively parallel simulation for extreme-scale torus network codesign," in *Proceedings of the 2nd ACM SIGSIM/PADS conference on Principles of advanced discrete simulation*. ACM, 2014, pp. 27–38.