Towards Dynamic Load Balancing Policies in Software-Defined Storage

Michael A. Sevilla*
University of California, Santa Cruz
msevilla@soe.ucsc.edu

Brad Settlemyer
Los Alamos National Lab
bws@lanl.gov

Carlos Maltzahn
University of California, Santa Cruz
carlosm@soe.ucsc.edu

Danny Perez
Los Alamos National Lab
danny_perez@lanl.gov

David Rich
Los Alamos National Lab
dor@lanl.gov

Galen Shipman
Los Alamos National Lab
gshipman@lanl.gov

ABSTRACT

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed enim libero, vestibulum quis efficitur et, commodo vitae lorem. Proin metus odio, molestie sit amet magna at, eleifend lobortis neque. Mauris placerat arcu ac lacus venenatis auctor. Aliquam rutrum non tellus vitae tempor. Aliquam erat volutpat. Duis enim leo, condimentum tincidunt posuere nec, ultricies vitae nisi. Aliquam efficitur massa felis, in tempus ante pretium ac. Cras eget luctus felis, id eleifend nulla.

ACM Reference format:

Michael A. Sevilla, Brad Settlemyer, Carlos Maltzahn, Danny Perez, David Rich, and Galen Shipman. 1997. Towards Dynamic Load Balancing Policies in Software-Defined Storage. In *Proceedings of ACM Woodstock conference, El Paso, Texas USA, July 1997 (WOODSTOCK'97),* 6 pages.

DOI: 10.475/123_4

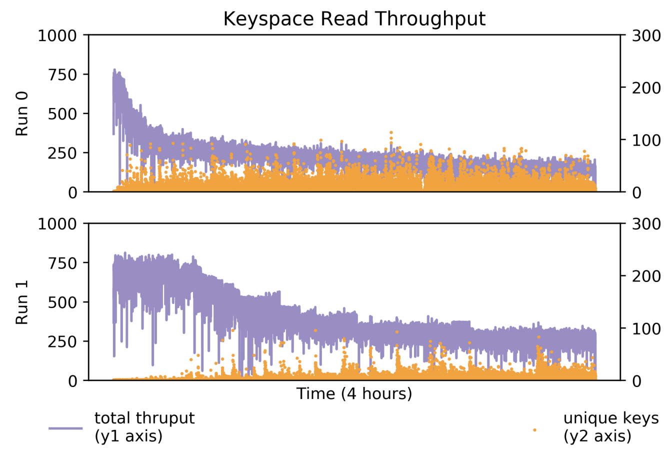

Figure 1: The keyspace activity for ParSplice runs using two different growth rates. The blue line shows the rate at which EOM minima values are retrieved from the key-value store ($y1$-axis) and the black points show the number of unique minima keys accessed in a 1 second sliding window ($y2$-axis).

1 INTRODUCTION

The fine-grained data annotation capabilities provided by key-value storage is a natural match for many types of scientific simulation. In simulations relying on the finite element method, a mesh-based decomposition of a physical region, may result in millions or billions of mesh cells each containing materials, pressures, temperatures and other characteristics that are required to accurately simulate phenomena of interest. In our target application, the ParSplice [10] molecular dynamics simulation, a key-value store is used to store both observed minima across a molecule's equation of motion (EOM) and the hundreds or thousands of unique trajectories calculated each second during a parallel job.

In this paper we present a detailed analysis of how the ParSplice application accesses EOM minima key-value pairs over the course of a long running simulation across a variety of initial conditions. Figure 1 shows that small changes to the rates at which new atoms enter the simulation, or the initial temperature at which the simulation begins can have a strong effect on the timing and frequency with which new EOM minima are discovered and referenced.

We also demonstrate the Mantle [14] approach to load balancing for storage systems. Although Mantle was intially developed for the Ceph distributed file system, the flexible policy-based approach to load balancing provided by Mantle is also applicable to the changing workloads generated by ParSplice. The Mantle load-balancing API enables the storage system to change the active load-balancing policy in use, a technique we will show is critical in applications such as ParSplice that have multiple stable and chaotic simulation "access regimes" over the course of a long-running simulation. Effectively, Mantle provides us the ability to choose among several load-balancing policies as needed.

Finally, we explore the use of simple machine learning (ML) techniques to identify access regimes within the ParSplice application's creation and access of key-value pairs. The ML component of this work drives the Mantle policy-switching API and helps determine *when* to change policies.

By understanding the regime-based key-value access patterns generated by ParSplice, leveraging the dynamic load balancing capabilities of the Mantle API, and using ML to identify key-value access regime changes we are working toward a flexible load balancing capability for software-defined storage systems.

*Work done while interning at LANL.

WOODSTOCK'97, El Paso, Texas USA
2016. 123-4567-24-567/08/06…$15.00
DOI: 10.475/123_4

2 PARSPLICE BACKGROUND

ParSplice [10] is an accelerated molecular dynamics (MD) simulation tool developed at LANL. Its phases are:

(1) splicer tells workers to generate segments (short MD trajectory) for specific states
(2) workers read initial coordinates for their assigned segment from database
(3) upon completion, workers insert final coordinates for each segment into database, and wait for new segment assignment

The computation can be parallelized by adding more workers or by adding worker tasks to parallelize individual workers. The workers are stateless and read initial coordinates from the database every time they start generating segments. Since worker tasks do not maintain their own history, they can end up reading the same coordinates repeatedly. To mitigate the consequences of these repeated reads, ParSplice provisions a hierarchy of nodes to act as caches that sit in front of the persistent database. Writes are stored on each tier and reads traverse up the hierarchy until they find their data.

Input Deck

We use ParSplice to simulate the evolution of metallic nanoparticles that grow from the vapor phase. As the run progresses, the energy landscape of the system becomes more complex. Two factors control the number of entries in the database: the growth rate and the temperature. The growth rate controls how quickly new atoms are added to the nanoparticle: fast rates lead to non-equilibrium conditions, and hence increase the number of states that can be visited. However, as the particle grows, the simulation slows down because the calculations become more expensive, limiting the rate at which new states are visited. On the other hand, the temperature controls how easily a trajectory can jump from state to state; higher temperatures lead to more frequent transitions.

The nanoparticle simulation stresses the database architecture of ParSplice. It visits more states than other input decks because the system uses a cheap potential, has a small number of atoms, and a complex energy landscape with many accessible states. Changing the growth rate and temperature changes the size, shape, and locality of the database keyspace. Lower temperatures and smaller growth rates create hotter keys with smaller keyspaces as many segments are generated in the same set of states, before the trajectory can escape to a new region of state space.

Our evaluation uses the total "trajectory length" as the goodness metric. This value is the duration of the overall trajectory produced by the system. At ideal efficiency, the trajectory length should increase with the square root of the wall-clock time, since the wall-clock cost of timestepping the system by one simulation time unit increases in proportion of the total number of atoms.

3 PARSPLICE KEYSPACE ANALYSIS

We instrumented ParSplice with performance counters and keyspace counters. The performance counters track ParSplice activities while keyspace counters track which keys were being accessed by the ParSplice ranks. Because the keyspace counters have high overhead we only turn them on for the keyspace analysis.

The cache hierarchy was unmodified but for the backend persistent database, we replaced BerkeleyDB on NFS with LevelDB on Lustre. Original ParSplice experiments showed that BerkeleyDB's syncs caused reads/writes to pile up on the persistent database node. We also use Riak's customized LevelDB[1] version, which comes instrumented with its own set of performance counters.

Testbed: Cray XC40

All experiments were run on Trinitite, a Cray XC40 with 100 nodes each with 32 Intel Haswell 2.3GHz cores. Each node has 128GB of RAM and our goal is to limit the size of the database to 3% of RAM. Note that this is an addition to the 30GB that ParSplice uses to manage other ranks on the same node.

Initial runs on commodity hardware, 10 CloudLab nodes and 10 nodes in our private cluster, show poor performance compared to the Cray. A single Cray node produces trajectories that are 5×) longer than our CloudLab clusters and 24× longer than our own cluster. As a result, it reaches different job phases faster and gives us a more comprehensive view of the workload. The performance gains compared to the commodity clusters has more to do with memory/PCI bandwidth than network.

Results. We examine the keyspace size and access patterns using Figures 6 and 2.

An active but small keyspace. The black text annotations in Figure 2 show that the keyspace size ranges from about 10K keys for 32 workers to 100K keys for 1048 workers. The bars show $50 - 100\times$ as many reads (`get()`) as writes (`put()`). Worker tasks read the same key for extended periods because the trajectory segment is stuck in a superbasin composed of local minima, so many coordinates are needed before the trajectory moves on. Writes only occur for the final state of segments generated by worker tasks; their magnitude is smaller than reads because the caches ignore redundant write requests. The number of read and write requests are highest at the beginning of the run when worker tasks generate segments for the same state, which is cheap. This type of keyspace encourages replication across a cluster.

Entropy increases over time. The reads per second in Figure 6 show that the number of requests decreases and the number of active keys increases over time. The resulting key access imbalance for the two growth rates in Figure 6 are shown in Figure 3, where reads are plotted for each unique state (x axis). Keys are more popular than others (up to 5×) because worker tasks start generating states with different coordinates later in the run.

Entropy growth is structured. The access patterns reflect the locality of computation: worker tasks stuck in state basins generate segments with similar coordinates. The growth rate, temperature, and number of workers changes that locality, which has an effect on the structure of the keyspace. Figure 3 shows that the number of reads changes with different growth rates, but that spatial locality is similiar (*e.g.*, some keys are stil 5× more popular than others). Figure 6 shows how entropy for different growth rates has temporal locality, as the reads per second for 1M looks like the reads per second for 100K stretched out along the time axis. Trends also exist

[1]https://github.com/basho/leveldb

for temperature and number of workers but are ommitted here for space. This structure means that we can learn the regimes and adapt the storage system to it.

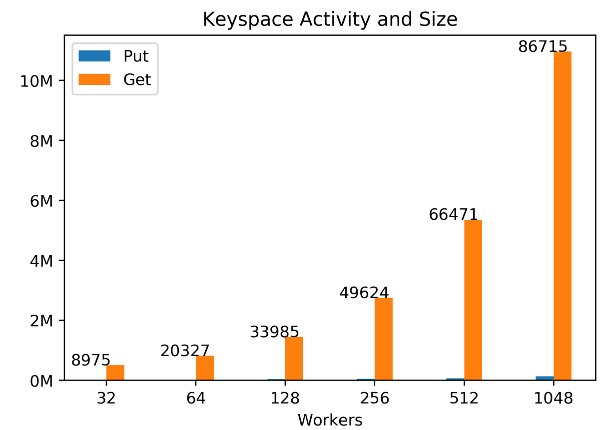

Figure 2: The keyspace size is small (numbers above bars) but must satisfy many reads as workers calculate new segments. The active keyspace is difficult to predict *a priori* but the optimal load balancing strategy strikes a good balance between preformance and utilization.

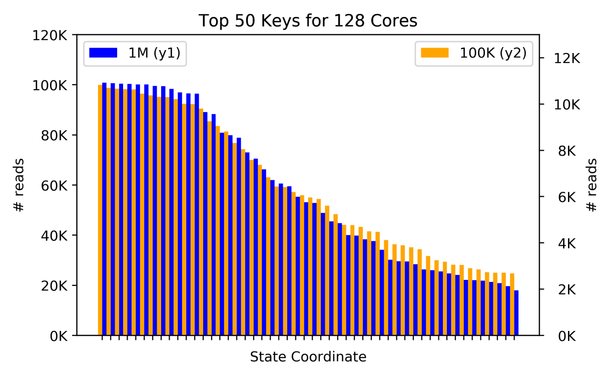

Figure 3: The keyspace imbalance is due to workers generating deep trajectories and reading the same coordinates. Over time, the accesses get dispersed across different coordinates resulting in some keys being more popular than others.

4 STATIC LOAD BALANCING

To motivate the need for load balancing we show how limiting the cache size saves memory and sacrifices neglible performance. This type of analysis will help inform our load balancing policies for when we switch to a distributed key-value store backend to store segment coordinates. We need to know when and how to partition the keyspace: a smaller cache hurts performance because key-value pairs need to be retrieved from other nodes while a larger cache has higher memory usage.

On each cache node, ParSplice uses an infinitely large cache to store segment coordinates. We limit the size of the cache using an LRU eviction policy, where the penalty for a cache miss is retrieving the data from the persistent database. We evict keys (if necessary) at every operation instead of when segments complete because the cache fills up too quickly otherwise and it reduces the overhead of key eviction.

Results. The results for different cache sizes for a growth rate of 100K over a 2.5 hour run is shown in the left two plots of Figure 4. "Baseline" is the performance of unmodified ParSplice measured in trajectory duration (y-axis) and utilization is measured with memory footprint ($y2$ axis) of just the cache. "Static Load Balancing Policies" shares the y-axis and shows the trade-off for different cache sizes. The error bars are the standard deviation of 3 runs.

Although the keyspace grows to 150K, a 100K key cache achieves 99% of the peformance. Decreasing the cache degrades performance and predictability. Not suprisingly, the memory usage all decreases with the cache size and although we only save 0.4GB, larger and more complicated runs use up to 4GB, which is 3% of the 128GB on each node. While this result is not unexpected, it nonetheless achieves our goal of showing the benefits of load balancing keys across nodes and that smaller caches on each node are an effective way to save memory without completely sacrificing performance.

5 THE NEED FOR DYNAMIC LOAD BALANCING POLICIES

Despite the memory savings, our results suggest that dynamic load balancing policies could save even more memory. The left two plots in Figure 4 show that a 100K key cache is sufficient as a static policy but the top graph in Figure 6 indicates that the cache size could be much smaller. That graph shows that the beginning of the run is characterized by many reads to a small set of keys and the end sees much lower reads per second to a larger keyspace. Specifically, it shows only about 100 keys as active in the latter half of the run, so a smaller cache should indeed suffice.

After analyzing traces, we see that the 100 key cache is insufficient because LevelDB cannot service the read-write traffic. By limiting the size of the cache, some reads must traverse up the ParSplice cache hierarchy to the persistent database. According to Figure 6, the read requests arrive at 750 reads per second in addition to the writes that land in each tier (about 300 puts/second, some redundant). This traffic triggers a LevelDB compaction and reads block and eventually pile up, resulting in very slow progress. Traces verify this hypothesis and show reads getting backed up as the read/write ratio explodes. To recap, small caches incurr too much load on persistent database at the beginning of the run but smaller caches should suffice after the initial read flash crowd passes because the keyspace is far less active. This suggests a two-part load balancing policy.

To explore dynamic load balancing policies (*i.e.* policies that change during the run), we use the Mantle approach. Mantle is a framework buit on the Ceph file system that lets admnistrators control file system metadata load balancing policies. The basic premise is that load balancing policies can be expressed with a simple API consisting of "when", "where", and "how much". The succinctness of the API lets users inject muitiple, dynamic policies.

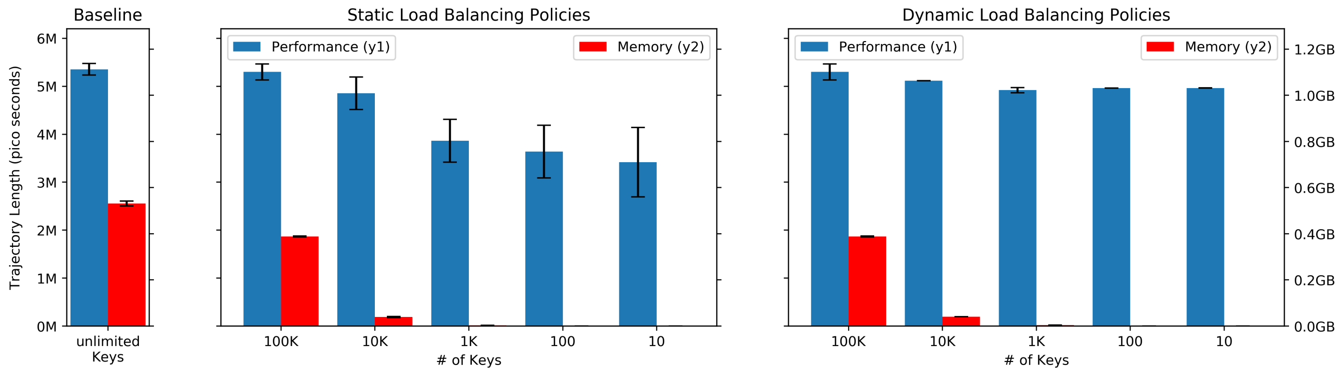

Figure 4: The performance and resource utilzation trade-off for different cache sizes, which are enumerated along the x-axes. "Baseline" is ParSplice unmodified, "Static Load Balancing Policies" limits the size of the cache to demonstrate the benefits of smaller keyspaces on cache nodes (Section §4), and "Dynamic Load Balancing Policies" switches to a smaller cache after absorbing the initial burstiness of the workload (Section §5).

Although ParSplice does not use a distibuted file system, its workload is very similiar because the minima key-value store responds to small and frequent requests, which results in hot spots and flash crowds. Distributed file systems solve similiar issues: since data IO does not scale like metadata IO [12], finding optimal ways to measure, migrate, and partition metadata load is a relatively new field, but has been shown to lead to large performance increases and more scalable file systems [3, 5, 8, 11, 15, 16]. Both workloads also have data locality so the storage should have mechanisms for leveraging requests with similar semantic meaning. Previous work quantified the speedups achieved with Mantle and formalized balancers that were good for file systems.

Results. The right most graph of Figure 4 shows the results of using the Mantle API to program a dynamic load balancing policy with two phases into ParSplice:

- unlimited growth: cache increases on every write
- n key limit: cache maintained at this size

We trigger the policy switch at 100K keys to absorb the flash crowd at the beginning of the run. Once triggered, keys are evicted to bring the size of the cache down to the threshold and the least recently keys are actively evicted. In that bar chart, the cache sizes are along the x-axis.

The dynamic policies show better performance than the single n key policies. The performance and memory utilization for 100K is the same as the 100K bar in the middle graphs but the rest reduce the size of the keyspace after the read flash crowd. This reduced read/write traffic on the persistent database and lowers the number of stalls. The worst performing policy is the 10 key cache, which achieves 94% of the performance while only using 40KB of memory.

Caveats. The results from the right most graph in Figure 4 are slightly deceiving for three reason: (1) segments take longer to generate later in the run, (2) the memory footprint is the value at the end of 2.5 hours, and (3) this policy only works well for the 2.5 hour run. For (1), the curving down of the simulation vs. wall-clock time is shown in Figure 5; as the nanoparticle grows it takes longer to generate segments so by the time we reach 2 hours, over 90% of the trajectory is already generated. For (2), the memory footprint is around 0.4GB until we reach 100K key threshold. In Figure 4 we plot the final value. For (3), Figure 5 shows that the cache fills up with 100K keys at time 7200 seconds and its size is reduced to the size listed in the legend. The curves stay close to "Unlimited" for up to an hour after the cache is reduced but eventually flatten out as the persistent database gets overloaded. 10K and 100K follow the "Unlimited" curve the longest and are sufficient policies for the 2.5 hour runs but anything longer would need a different dynamic load balancing policy.

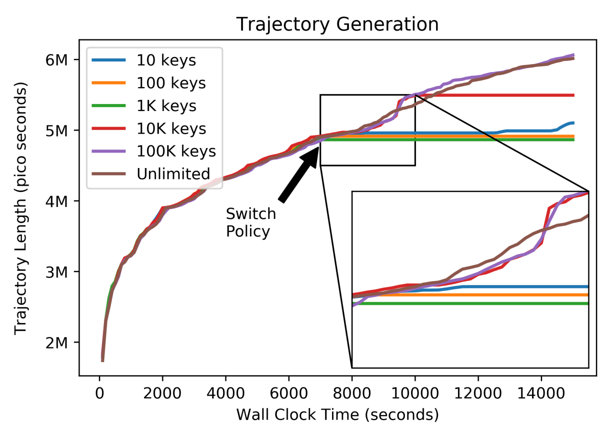

Figure 5: The rate that the trajectory is computed decays over time (which is expected) but this skews the performance improvements in Figure 4. While the dynamic policy we designed does not work for 4 hours, it does work for our target runtime of 2.5 hours.

Despite these caveats, the result is still valid: we found a dynamic load balancing policy that absorbs the cost of a high read throughput on a small keyspace and reduces the memory pressure for a 2.5 hour run. Our experiments show the effectiveness of the load balancing policy engine we integrated into ParSplice, not that we were able to identify the best policy for all system setups (*i.e.* different ParSplice parameters, number of worker nodes, and job lengths). To solve that problem, we need a way to identify what thresholds we should use for different job permutations.

6 USING ML FOR THE KEYSPACE

We proved in the previous section that an optimial policy, based on the read burstiness at the beginning of the run, exists for a 100K growth rate on 8 nodes, but we cannot re-do this analysis for every workload, system, and parameter permutation. Fortunately, Section §3 shows that the keyspace size and activity is structured. so rather than finding policies hand again for every cluster size, growth rate, and temperature, in this section we use machine learning to inform the Mantle policy engine.

We feed the read request rate from the 100K and 1M runs in Figure 1 into the K-means clustering algorithm as (timestamp, ops/second) tuples. We weight the timestamp and ops/second equally and set the number of clusters to be 4. We chose this initial K based on visual inspection of Figure 1, where it appears that there are 4 workload phases: one plateau of redundant reads at the beginning, decreasing requests per second, and then two plateaus of steady requests per second. Knowing that the setup parameters transform the request rates temporally or spatially, this same initial K should work for all setups. Once the algorithm identifies the workload regimes, we select the start of the third regime as the point to switch to a fixed sized cache because the request rate has lowered to sustainable levels for the persistent database.

Results. We plot throughput over time in Figure **??** and color each point with its assigned group. The black squares are the centroids, also known as the center of K-means groups. We run the algorithm for a variety of request rate traces but only show the setups from Figure 1. We also annotate the graphs with the suggested cache size, which is calculated by looking up the timestamp for the third regime that corresponds to the keyspace size in our performance counters.

The algorithm properly identifies the 4 workload phases: the plateau of redundant reads, the phase with a large decrease in request rate, and the two plateaus of steady read requests. It also picks different timestamps for the start of the third regime, which aligns with our keyspace analysis and our assertion that the growth parameters affect how long it takes the workload to reach a certain phase. Finally, the algorithm picks reasonable values for the key cache size. The 100K growth rate selects a 55K cache size, which is between our benchmarked optimal threshold for the high watermark value chosen to absorb the read burst (100K) and the lower cache size we limit the system to after the initial burst (10K). This result both reaffirms the results from the previous section and provides hope that we can avoid lengthy paramaters sweeps for ParSplice in the future.

7 RELATED WORK

The motivation for this work is the Mochi project, which is trying to move away from application-specific sofware stacks by pushing for re-use and composable services. Applications need more services to work and they are demanding a richer set of functionality, so the hope is to share these building blocks (e.g., Margo transport layer,

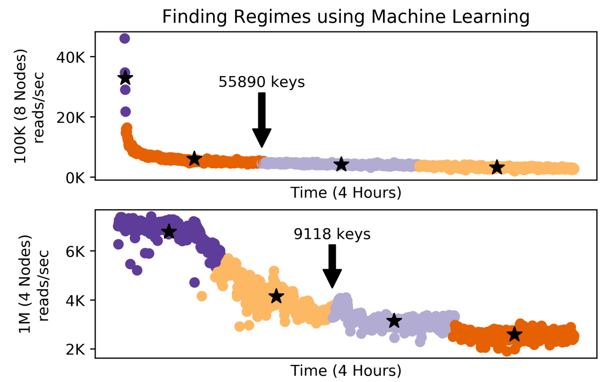

Figure 6: Learning the workload regimes with K-Means clustering helps pick keyspace size thresholds that can be fed into a dynamic load balancing policy engine, like Mantle. Specifying 4 clusters and selecting the third for informing the policy switch returns keyspace size thresholds similar to the values we found by hand in Section §5.

SPINDLE library, MDHIM kv-store, etc.) across HPC applications. Malacology takes a similar approach to storage, where internal functionality of the storage system is exposed to applications so they do not have to bolt-on services or re-implement functionality themselves. Load balancing is a microservice that we think would be a useful part of the Mochi suite and Mantle seems like a good fit because it has already identified balancers that work well for file systems. Here we try to show that the balancers and the Mantle approach are also applicable to key-value store workloads.

File System Metadata. Our focus is the load balancing of a distributed key-value store. For inspiration, we focused on file system metadata load balancing because the hotspots and flash crowds of redundant operations we were seeing did not match well with the keyspace partitioning techniques of traditional databases or key-value stores.

State-of-the-art distributed file systems partition write-heavy workloads and replicate read-heavy worklods. IndexFS [11] partitions directories and clients write to different partitions by grabbing leases and caching ancestor metadata for path traversal. ShardFS takes the replication approach to the extreme by copying all directory state to all nodes. CephFS **??** employs both techniques to a lesser extent; directories can be replicated or sharded but the caching and replication policies are controlled with tunable parameters. Despite the obvious benefits of these protocols, these systems still need to be tuned by hand with *ad-hoc* policies designed for specific applications.

Setting policies for migrations is arguably more difficult than adding the migration mechanisms themselves. For example, IndexFS and CephFS use the GIGA+ [9] technique for partitioning directories at a predefined threshold. Mantle makes headway in this space by providing a framework for exploring these policies, but does not attempt anything more sophisticated (e.g., machine learning or autotuning) to create these policies.

Auto-tuning. Auto-tuning is a well-known technique used in HPC [1, 2], big data systems systems [6], and databases [13]. Like our work, these systems focus on the physical design of the storage (*e.g.* cache size) but since we focused on a relatively small set of parameters (cache size, migration tresholds), we did not need anything as sophisticated as the genetic algortihm used in **??**. We cannot drop these techniques into ParSplice because the magnitude and speed of the workload hotspots/flash crowds makes existing approaches less applicable.

Distributed KV Stores. Our goal is to use MDHIM [4] as our backend key-value store because it was designed for HPC and has the proper mechanisms for migration already implemented. MDHIM tailors its mechanisms and policies to HPC, showing improved performance over cloud-based key-value stores like Cassandra [7]. It has cursor types for walking the key-value store, bulk operations for exploiting data locality, per-job server spawning, and pluggable backends for its local database and network type (iband/RDMA). Its policies are extenible, supporting customized partitioning strategies and user-decondary indices.

8 CONCLUSION

Load balancing is a well-known technique for alleviating load and improving performance, yet finding the best policies for applications is still a hands-on, tedious process. If a load balancing module were to be accepted into a suite of services like the Mochi project, it must transcend domains and be useful to a wide-range of applications. In this paper, we present the framework for such a service which (1) helps administrators explore the techniques for informing load balancing, (2) supports dynamic policies for quickly changing workloads and higher level intelligence engines like machine learning, and (3) comes packaged with balancers that work well for both key-value stores and file system metadata. We demonstrate (1) with a keyspace analysis for ParSplice and use our findings to design a load balancing policy that improves resource utilization. Finally, to drive the policy engine designed in (2), we show that a single policy is inadequate and that Mantle is flexible enough to explore policies generated with machine learning. It is our hope that the introduction of the Mantle approach encourages the use of machine learning and auto-tuning for policy design in future storage systems.

REFERENCES

[1] Babak Behzad, Surendra Byna, Stefan M Wild, and Marc Snir. 2014. Improving Parallel I/O Autotuning with Performance Modeling. (2014).

[2] Babak Behzad, Huong Vu Thanh Luu, Joseph Huchette, Surendra Byna, Ruth Aydt, Quincey Koziol, Marc Snir, and others. 2013. Taming parallel I/O complexity with auto-tuning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 68.

[3] Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Lan Xue. 2003. Efficient Metadata Management in Large Distributed Storage Systems. In *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST '03)*.

[4] Hugh Greenberg, John Bent, and Gary Grider. 2015. MDHIM: A Parallel Key/Value Framework for HPC. In *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*.

[5] Gary Grider, Dave Montoya, Hsing-bung Chen, Brett Kettering, Jeff Inman, Chris DeJager, Alfred Torrez, Kyle Lamb, Chris Hoffman, David Bonnie, Ronald Croonenberg, Matthew Broomfield, Sean Leffler, Parks Fields, Jeff Kuehn, and John Bent. 2015. MarFS - A Scalable Near-Posix Metadata File System with Cloud Based Object Backend. In *Work-in-Progress at Proceedings of the 10th Workshop on Parallel Data Storage (PDSW'15)*.

[6] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F.B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *Proc. of the Fifth CIDR Conf* (2011).

[7] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. 44 (????).

[8] Swapnil V. Patil and Garth A. Gibson. 2011. Scale and Concurrency of GIGA+: File System Directories with Millions of Files. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST '11)*.

[9] Swapnil V. Patil and Garth A. Gibson. 2011. Scale and Concurrency of GIGA+: File System Directories with Millions of Files. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST '11)*.

[10] Danny Perez, Ekin D Cubuk, Amos Waterland, Efthimios Kaxiras, and Arthur F Voter. Long-Time Dynamics Through Parallel Trajectory Splicing. *Journal of chemical theory and computation* (????).

[11] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. 2014. IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion. In *Proceedings of the 20th ACM/IEEE Conference on Supercomputing (SC '14)*.

[12] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. 2000. A Comparison of File System Workloads. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '00)*. 4–4.

[13] Karl Schnaitter, Neoklis Polyzotis, and Lise Getoor. Index interactions in physical design tuning: modeling, analysis, and applications. 2 (????).

[14] Michael A. Sevilla, Noah Watkins, Carlos Maltzahn, Ike Nassi, Scott A. Brandt, Sage A. Weil, Greg Farnum, and Sam Fineberg. 2015. Mantle: A Programmable Metadata Load Balancer for the Ceph File System. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*.

[15] Qing Zheng, Kai Ren, and Garth Gibson. 2014. BatchFS: Scaling the File System Control Plane with Client-funded Metadata Servers. In *Proceedings of the 9th Workshop on Parallel Data Storage (PDSW' 14)*.

[16] Qing Zheng, Kai Ren, Garth Gibson, Bradley W. Settlemyer, and Gary Grider. 2015. DeltaFS: Exascale File Systems Scale Better Without Dedicated Servers. In *Proceedings of the 10th Workshop on Parallel Data Storage (PDSW' 15)*.