Towards Dynamic Load Balancing Policies in Software-Defined Storage

Michael A. Sevilla, Carlos Maltzahn
University of California, Santa Cruz
{msevilla,carlosm}@soe.ucsc.edu

Bradley W. Settlemyer, Danny Perez, David Rich, Galen M. Shipman (LANL)
{bws,danny_perez,dor,gshipman}@lanl.gov

ABSTRACT

Our analysis of the key-value activity generated under multiple initial conditions demonstrates the need for a distributed, load balancing key-value store for the ParSplice molecular dynamics simulation. Our analysis indicates the presence of clear access regimes and hot spots that offer significant opportunity for optimization. We leverage the Mantle load balancing framework, which was originally designed for distributed file systems, to dynamically switch policies and present a two policy scheme that achieves 96% efficiency while using only 7.6% of the memory resources required by the base case. Finally, we demonstrate how a machine learning clustering technique is an effective method for detecting access patterns within the keyspace over time.

1 INTRODUCTION

The fine-grained data annotation capabilities provided by key-value storage is a natural match for many types of scientific simulation. Simulations relying a mesh-based decomposition of a physical region may result in millions or billions of mesh cells. Each cell contains materials, pressures, temperatures and other characteristics that are required to accurately simulate phenomena of interest. In our target application, the ParSplice [11] molecular dynamics simulation, a hierarchy of cache nodes and a single node key-value store are used to store both observed minima across a molecule's equation of motion (EOM) and the hundreds or thousands of partial trajectories calculated each second during a parallel job. Unfortunately, if we scale the system the IO to the storage hierarchy will quickly saturate both the storage and bandwidth capacity of a single node.

To motivate the need for load balancing data across a distributed key-value store, we show how limiting the cache size saves memory and sacrifices negligible performance. This type of analysis will help inform our load balancing policies for when we switch to a distributed key-value store back-end to store EOM minima 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 pressure.

In this paper we present a detailed analysis of how the ParSplice application accesses 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 (Δ) can have a strong effect on the timing and frequency with which new EOM minima are discovered and referenced.

,
2017. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

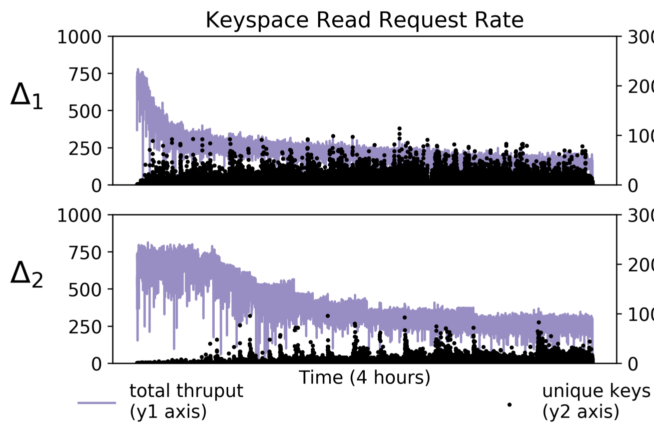

Figure 1: The keyspace activity for ParSplice using two different growth rates. The line shows the rate that EOM minima values are retrieved from the key-value store ($y1$ axis) and the points along the bottom show the number of minima keys accessed in a 1 second sliding window ($y2$ axis).

We also demonstrate the Mantle [15] approach to load balancing for software-defined storage systems. Although Mantle was initially developed for the Ceph distributed file system, the flexible policy-based approach to load balancing is also applicable to the changing key-value workloads generated by ParSplice. The Mantle API also allows the storage system to change the active load balancing policy in use. We show that this technique is critical in applications such as ParSplice that have alternating stable and chaotic simulation "access regimes" over the course of a long-running simulation.

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* and *how* 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.

2 PARSPLICE BACKGROUND

ParSplice [11] is an accelerated molecular dynamics (MD) simulation package developed at LANL. It is part of the Exascale Computing Project[1] and is important to LANL's Materials for the Future initiative. Its phases are:

[1] http://www.exascale.org/bdec/

(1) a splicer tells workers to generate segments (short MD trajectory) for specific states
(2) workers read initial coordinates for their assigned segment from data store; the key-value pair is (state ID, coordinate)
(3) upon completion, workers insert final coordinates for each segment into data store, 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 data store each time they begin 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 a single node persistent database. Values are written to each tier and reads traverse up the hierarchy until they find the data.

ParSplice simulates 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 domain factors control the number of entries in the data store: the growth rate and the temperature. The growth rate controls how quickly new atoms are added to the nanoparticle: fast growth 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 data store 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 growth rates and temperature alters the size, shape, and locality of the data store 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 ParSplice. At ideal efficiency, the trajectory length should increase with the square root of the wall-clock time, since the wall-clock cost of time-stepping 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 progress while keyspace counters track which keys are 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 persistent database node, we replaced BerkeleyDB on NFS with LevelDB on Lustre. Original ParSplice experiments showed that BerkeleyDB's syncs caused reads/writes to bottleneck on the persistent database. We also use Riak's customized LevelDB[2] version, which comes instrumented with its own set of performance counters.

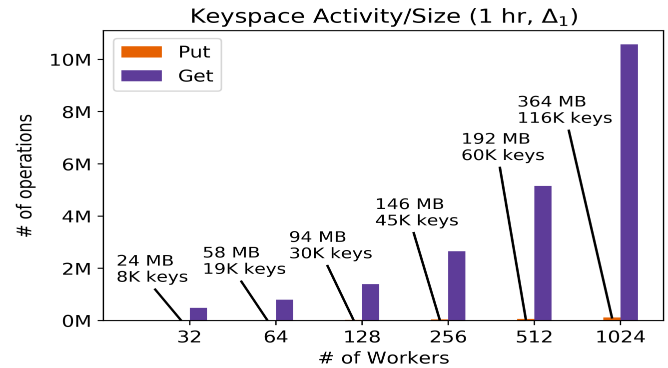

Figure 2: The keyspace size is small but must satisfy many reads as workers calculate new segments. It is likely that we will need more than one node to manage segment coordinates when we scale the system or jobs up.

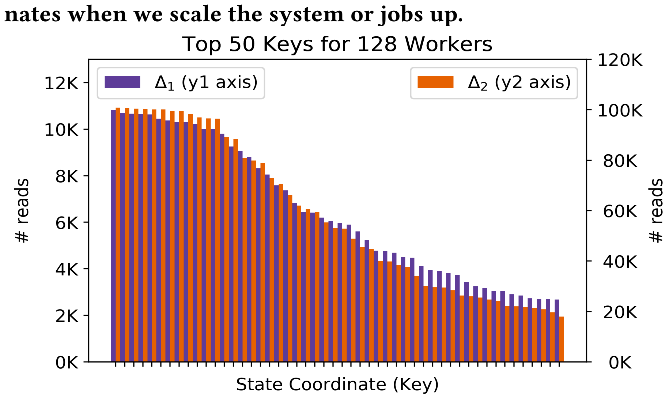

Figure 3: The keyspace imbalance is due to workers generating deep trajectories and reading the same coordinates. Over time, the accesses get dispersed across different coordinates resulting in some keys being more popular than others.

All experiments ran on Trinitite, a Cray XC40 with 32 Intel Haswell 2.3GHz cores per node. Each node has 128GB of RAM and our goal is to limit the size of the cache to 3% of RAM[3]. Note that this is an addition to the 30GB that ParSplice uses to manage other ranks on the same node. A single Cray node produced trajectories that are 5× times longer than our 10 node CloudLab clusters and 25× longer than UCSC's 10 node 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 have more to do with memory/PCI bandwidth than network.

Scalability. Figure 2 shows the keyspace size (black annotations) and request load (bars) after a one hour run with a different number of workers (x axis). While the keyspace size and capacity is relatively modest the memory usage scales linearly with the number

[2]https://github.com/basho/leveldb
[3]Empirically, this is a threshold that we find to work well for most applications

of workers. This is a problem if we want to scale to Trinitite's 6000 cores. Furthermore, the size of the keyspace also increases linearly with the length of the run. Extrapolating these results puts an 8 hour run across all 100 Trinitite nodes at 20GB for the cache. This memory utilization easily eclipses the 3% threshold we set earlier, even without factoring in the memory usage from other workers.

An active but small keyspace. The bars in Figure 2 show $50-100\times$ as many reads (`get()`) as writes (`put()`). Worker tasks read the same key for extended periods because the trajectory can remain stuck in so-called superbasins composed of tightly connected sets of states. In this case, many trajectory segments with the same 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 computationally cheap (this motivates Section §4).

Entropy increases over time. The reads per second in Figure 1 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 1 are shown in Figure 3, where reads are plotted for each unique state, or key, along the x axis. Keys are more popular than others (up to $5\times$) because worker tasks start generating states with different coordinates later in the run (this motivates Section §5). The growth rate, temperature, and number of workers have a predictable effect on the structure of the keyspace. Figure 3 shows that the number of reads changes with different growth rates, but that the spatial locality of key accesses is similar (*e.g.*, some keys are still $5\times$ more popular than others). Figure 1 shows how entropy for different growth rates has temporal locality, as the reads per second for Δ_2 looks like the reads per second for Δ_1 stretched out along the time axis. Trends also exist for temperature and number of workers but are omitted here for space. This structure means that we can learn the regimes and adapt the storage system (this motivates Section §6).

4 STATIC LOAD BALANCING

In the original ParSplice implementation, each cache node uses as an unlimited amount of memory 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.

The results for different cache sizes for a growth rate of Δ_1 over a 2.5 hour run across 256 workers is shown in 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. The other graph shares the y axis and shows the trade-off of constraining the cache to different 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 performance. Decreasing the cache degrades performance and predictability. While this result is not unexpected, it

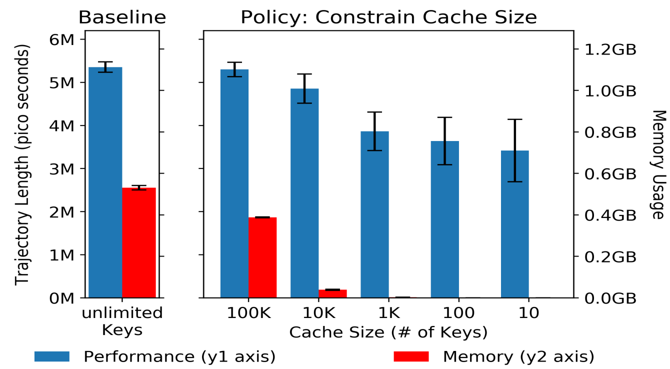

Figure 4: The performance and resource utilization trade-off for different cache sizes, which are enumerated along the x axis. "Baseline" is ParSplice unmodified and the "Policy: Constrain Cache Size" graph limits the size of the cache to save memory.

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. Figure 4 show that a 100K key cache is sufficient as a static policy but the top graph in Figure 1 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.

After analyzing traces, we see that the 100 key cache is insufficient because the persistent database cannot service the read-write traffic. According to Figure 1, 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, resulting in very slow progress. Traces verify this hypothesis and show reads getting backed up as the read/write ratio increases. To recap, small caches incur too much load on the persistent database at the beginning of the run but 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 built on the Ceph file system that lets administrators 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 multiple, dynamic policies.

Although ParSplice does not use a distributed file system, its workload is very similar because the minima key-value store responds to small and frequent requests, which results in hot spots

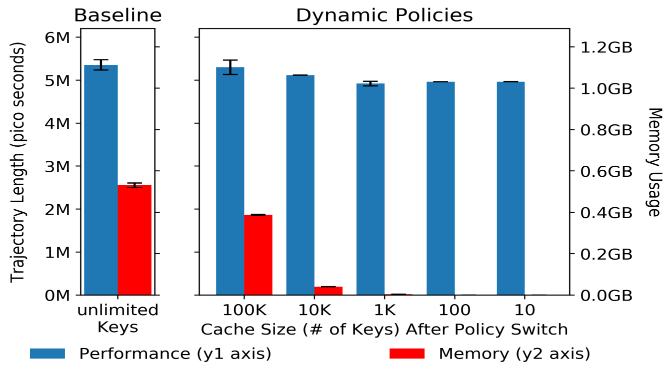

Figure 5: The performance and resource utilization trade-off of using a dynamic load balancing policy that switches to a constrained cache policy after absorbing the initial burstiness of the workload. The sizes of these smaller caches are on the x axis.

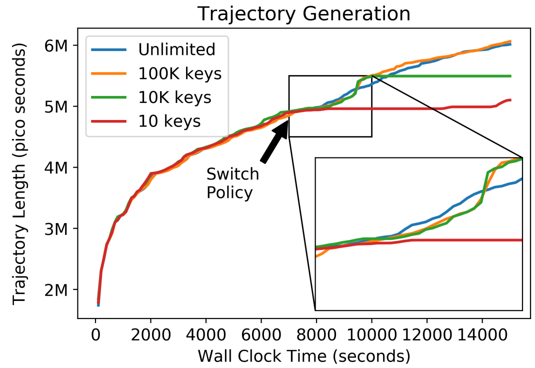

Figure 6: The rate that the trajectory is computed decays over time (which is expected) but this skews the performance improvements in Figure 5. Our dynamic policy works for 2.5 hour jobs but not for 4 hour jobs.

and flash crowds. Modern distributed file systems have found efficient ways to measure, migrate, and partition metadata load and have shown large performance gains and better scalability [3, 5, 9, 12, 18, 19]. Previous work quantified the speedups achieved with Mantle and formalized balancers that were good for file systems.

Figure 5 shows the results of using the Mantle API to program a dynamic load balancing policy into ParSplice:

- unlimited growth policy: cache increases on every write
- n key limit policy: cache constrained to n keys

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 used keys are evicted. In the bar chart, the cache sizes for the n key limit policy are along the x axis.

The dynamic policies show better performance than the single n key policies. The performance and memory utilization for a 100K key cache size 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. We see the worst performance when the engine switches to the 10 key limit policy, which achieves 94% of the performance while only using 40KB of memory.

Caveats. The results in Figure 5 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 6; 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 switch threshold. In Figures 4 and 5 we plot the final value. For (3), Figure 6 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.

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 tasks, 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 efficient policy, based on the read burstiness at the beginning of the run, exists for Δ_1 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. Rather than finding policies by hand again for every cluster size, growth rate, and temperature, in this section we use machine learning to inform the Mantle policy engine. Machine learning excels at both handling large design spaces and matching patterns. Our keyspace analysis in Section 3 demonstrated a large design space and Figure 1 shows 4 workload regimes: one plateau of redundant reads at the beginning, decreasing requests per second, and then two plateaus of steady requests per second. We start with a simple clustering algorithm to attack this multi-dimensional design space problem and detect workload regimes.

We feed the read request rate from the Δ_1 and Δ_2 runs in Figure 1 into the K-means clustering algorithm as (timestamp, ops/second) tuples[4]. 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. 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

[4]Note that the magnitudes are different because Figure 1 was run with keyspace tracing on and across less nodes which reduces performance

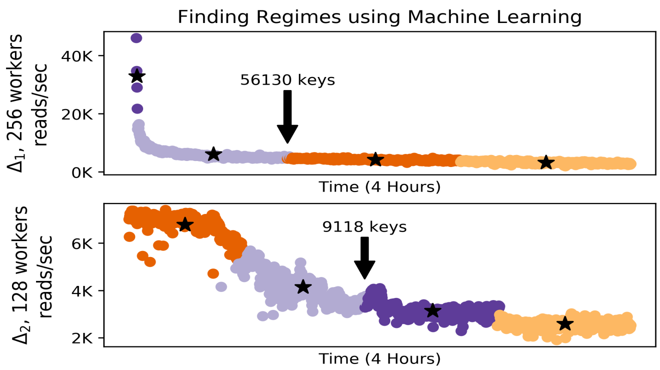

Figure 7: Learning the workload regimes with K-means clustering helps pick cache size thresholds that can be fed into a dynamic load balancing policy engine, like Mantle. Specifying 4 clusters and selecting the third for informing the policy switch returns keyspace size thresholds similar to the values we found by hand in Section §5.

point to switch to a fixed sized cache because the request rate has lowered to sustainable levels for the persistent database.

We plot throughput over time in Figure 7 and color each point with its assigned group. The black stars 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. Our implementation also selects 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 chooses reasonable values for the key cache size. The Δ_1 growth rate selects a 55K key cache size, which is between the threshold we chose using brute force for the high watermark (100K) and lower cache size (10K) in our dynamic policy. Note that this result does not validate our claims about ML, rather it is a promising development and suggests that we may avoid lengthy parameters sweeps for ParSplice in the future.

7 RELATED WORK

Key-value storage organizations for scientific applications is a field gaining rapid interest. In particular, the analysis of the ParSplice keyspace and the development of an appropriate scheme for load balancing is a direct response to a case study for computation caching in scientific applications [7]. In that work the authors motivated the need for a flexible load balancing *microservice* to efficiently scale a memoization microservice. Our work is also heavily influenced by the Malacology project [14] which seeks to provide fundamental services from within the storage system (*e.g.*, consensus) to the application.

State-of-the-art distributed file systems partition write-heavy workloads and replicate read-heavy workloads, similar to the approach we are advocating here. IndexFS [12] 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. The Ceph file system (CephFS) [16, 17] employs both techniques to a lesser extent; directories can be replicated or sharded but the caching and replication policies are controlled with tunable parameters. 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/CephFS use the GIGA+ [10] 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) to create these policies.

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 algorithm used in [2]. We cannot drop these techniques into ParSplice because the magnitude and speed of the workload hotspots/flash crowds makes existing approaches less applicable.

Our plan is to use MDHIM [4] as our back-end 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 [8]. It has cursor types for walking the key-value store, bulk operations for exploiting data locality, per-job server spawning, and pluggable back-ends for its local database and network type.

8 CONCLUSION

Load balancing is a well-know technique for improving storage system performance, yet finding the best policies is a difficult, multi-dimensional problem. Rather than attempting to construct a single, complex load balancing policy that works for a variety of scenarios, we instead use the Mantle framework as a *microservice* approach to load balancing that enables software-defined storage systems to flexibly change policies as the workload changes over time. In our analysis of the ParSplice key-value workload we have detected clear workload regimes that are sensitive to the initial conditions and the scale and duration of the simulation. We have also demonstrated that changing load balancing policies at runtime in response to the current workload is an effective mechanism to providing better load distribution. Finally, we have demonstrated that the pattern matching strengths of many machine learning algorithms is an effective mechanism for detecting access pattern regimes within the ParSplice application's key-value usage.

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] John Jenkins, Galen M. Shipman, Jamaludin Mohd-Yusof, Kipton Barros, Philip H. Carns, and Robert B. Ross. 2017. A Case Study in Computational Caching Microservices for HPC. In *IPDPS Workshops*. IEEE Computer Society, 1309–1316.

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

[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] 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)*.

[11] 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* (????).

[12] 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)*.

[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, Ivo Jimenez, Peter Alvaro, Shel Finkelstein, Jeff LeFevre, and Carlos Maltzahn. 2017. Malacology: A Programmable Storage System. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*.

[15] 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)*.

[16] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design & Implementation (OSDI'06)*.

[17] Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, and Ethan L. Miller. 2004. Dynamic Metadata Management for Petabyte-Scale File Systems. In *Proceedings of the 17th ACM/IEEE Conference on Supercomputing (SC'04)*.

[18] 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)*.

[19] 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)*.