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

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, and Carlos Maltzahn. 1997. Towards Dynamic Load Balancing Policies in Software-Defined Storage. In *Proceedings of ACM Woodstock conference, El Paso, Texas USA, July 1997 (WOODSTOCK'97),* 5 pages.
DOI: 10.475/123_4

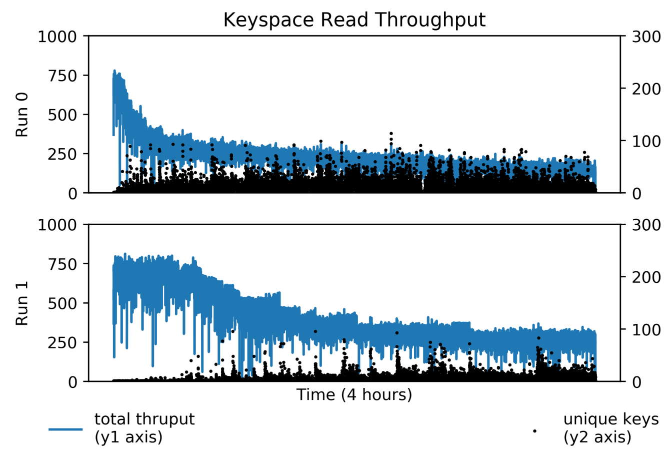

Figure 1: The keyspace activity for Parsplice runs using two different growth rates. The blue line show the rate at which EOM minima values are retrieved from the Key-value store (y1 axis) and the black points show the number of unique minima keys accessed in a 1 second sliding window.

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 [4] 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. Figure 2 shows the architecture of the ParSplice application.

In this paper we present a detailed analysis of how the ParSplice application accesses EOM minima KV pairs over the course of a long running simulation across a variety of initial conditions. Figure **??** 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 [7] 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 relatively stable and relatively chaotic simulation 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 thus determining *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.

2 PARSPLICE BACKGROUND

ParSplice [4] is a molecular dynamics simulation developed at LANL. Its phases are depicted in Figure 2:

(1) splicer tells producers to compute segments for state s_i
(2) producers pull initial coordinates $\{x_i, y_i\}$ from database

*Work done while interning at LANL.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
WOODSTOCK'97, El Paso, Texas USA
© 2016 Copyright held by the owner/author(s). 123-4567-24-567/08/06...$15.00
DOI: 10.475/123_4

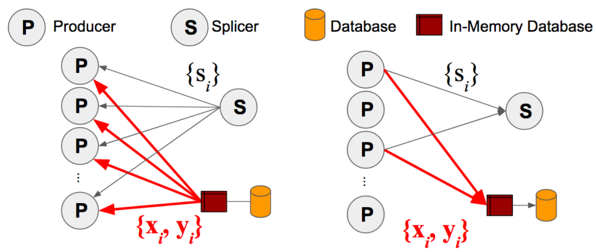

Figure 2: ParSplice is a ready'-heavy HPC application where producers use a database for consistency.

(3) a producer inserts completed coordinates for segment s_i into database and splicer broadcasts next segment(s) s_j
(4) producer pull new segment coordinates $\{x_j, y_j\}$

The producers are stateless and pull intitial segment coordinates from the database every time they start generating segments. The computation can be parallelized by adding more LAMMPS engines or by adding workers to paralleilize a single LAMMPS engine. Since workers do not maintain their own history, they end up pulling the same coordinates repeatedly.

The database runs on a single node but ParSplice provisions 2 other nodes to act as caches. Producers contact this hierarchy of caches for their segment coordinates. 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 states that molecules visit as nanoparticles grow. As the run progresses, particles become more unstable and visit more states. The number of states in the simulation is controlled with growth rate and temperature parameters. The growth rate controls how big the nanoparticle is; smaller values grow faster but lengthen the time to generate segments. When the nanoparticles reach large sizes, the simulation slows down because the calculation is more expensive. The temperature controls the molecules' ability to transition to new states; higher temperatures transition more frequently because molecules have more energy. Making the temperature too high has diminishing returns as molecules can visit any state with equal probability, reducing the size and number of the basins they explore.

The nanoparticle simulation stresses the database architecture of ParSplice. It visits more states than other input decks (e.g., UO2) because the system uses a cheap potential. Changing the growth rate and temperature changes the size, shape, and locality of the database keyspace. Larger temperatures and smaller growth rates create hotter keys with smaller keyspaces.

Our evaluation uses "trajectory length" as the goodness metric. This value is the duration of the overall trajectory produced by the system. The relationship between wall clock time and trajectory length should be linear because it means the system is keeping up with the simulation load and is mirroring real time. We tested our own private cluster and a cluster of beefy nodes on CloudLab but the trajectory lengths were much smaller than our in-house Cray at LANL.

3 METHODOLOGY

3.1 ParSplice Instrumentation

We instrument ParSplice with performance counters and keyspace counters. The performance counters track ParSplice activities while keyspace counters track which keys are being accessed by the ranks. Because the keyspace counters have high overhead, we only turn them on for the keyspace analysis in Section §4.1.

The cache hierarchy is unmodified but for the backend persistent database, we replace 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.

3.2 Testbed: Cray

All experiments are run on Trinitite, a Cray XC40 with 100 nodes each with 32 Intell Haswell 2.3GHz cores. We disable hyperthreading as this has caused problems in the past. 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.

Inititial runs on commodity hardware, 10 CloudLab nodes and 10 nodes in our private cluster, show poor performance compared to the Cray supercomputer. A single Cray node produces trajectories that are 45×) 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.

3.3 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 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 check every operation instead of when segments complete because (1) the cache fills up too quickly, and (2) it reduces the overhead of key eviction.

3.4 Mantle: Dynamic Load Balancing

To explore dynamic load balancing policies, 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 polcies can be expressed with a simple API consisting of when, where, and how much callbacks. The succinctness of the API lets users inject muitiple, dynamic policies.

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

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 [6], 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 [1–3, 5, 8]. 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.

3.5 Machine Learning Keyspace Activity

We can't find the optimal keypsace size for every permuation, finding the key threshold changes with

- number of nodes
- delay: Figuree 1
  - unique keys increase over time, throughput of keys goes down
  - smaller delay has bigger keyspace but keys are way colder

4 RESULTS

4.1 ParSplice Keyspace Analysis

We examine the keyspace size and access patterns using Figures 1 and 3. These results are collected using the performance and keyspace counters described in Section §3.1.

An active but small keyspace. The black text annotations in Figure 3 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()`). Workers 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 workers; their magnitude is smaller than reads because the caches ignore redundant put requests. The number of read and write requests are highest at the beginning of the run when workers generate segements 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 1 show that the number of requests decreases and the number of active keys increases over time. The resulting imbalance for the two growth rates in Figure 1 are shown in Figure 4, where reads are plotted for each unique state (x axis). Keys are more popular than others (up to 5×) because workers start generating states with different coordinates later in the run.

Entropy growth is structured. The access patterns reflect the locality of computation: workers 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 4 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 1 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 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.

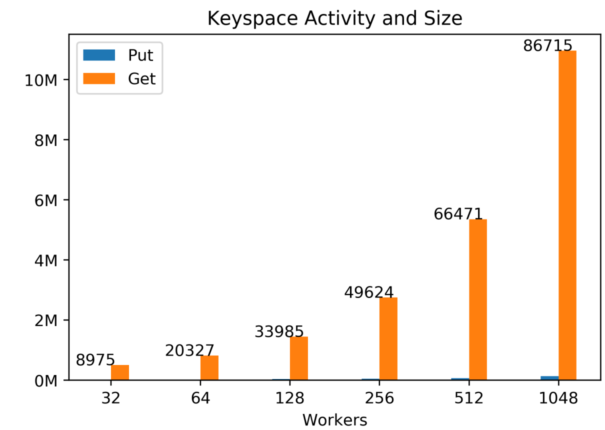

Figure 3: The keyspace size is small (numbers above bars) but must satisfy many reads as workers calculate new segments. The active keyspace is difficult to predict a priori but the optimal load balancing strategy strikes a good balance between preformance and utilization.

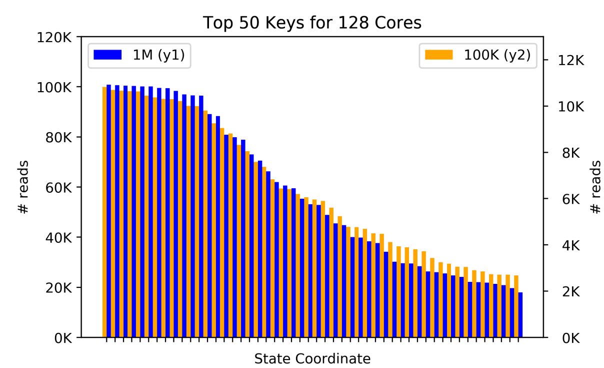

Figure 4: The keyspace imbalance is due to workers generating deep trajectories and reading the same coordinates. Over time, the accesses get dispersed across different coordinates resulting in some keys being more popular than others.

4.2 Positive Effects of Load Balancing

The results for different cache sizes, described in Section §3.3, for a growth rate of 100K over a 2.5 hour run is shown in the left two plots of Figure 5. "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.

4.3 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 5 show that a 100K key cache is sufficient as a static policy but the top graph in Figure 1 indicates that the value should 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 are 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 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 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 LevelDB 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, so we need a two part load balancing policy.

The right most graph of Figure 5 shows the results of using the Mantle API, described in Section §3.4 to program a dynamic load balancing policy with two phases into ParSplice:

- unlimited growth: cache increases on every put
- 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 figure, 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 the read/write traffic on the persistent database and reduces the amount 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 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 2.5 hours, over 90% of the trajectory is already generated. For (2), the memory footprint is around 0.4GB until we reach 100K keys. In Figure 5 we plot the final value. For (3), Figure 6 shows that the cache fills up with 100K keys at time x 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.

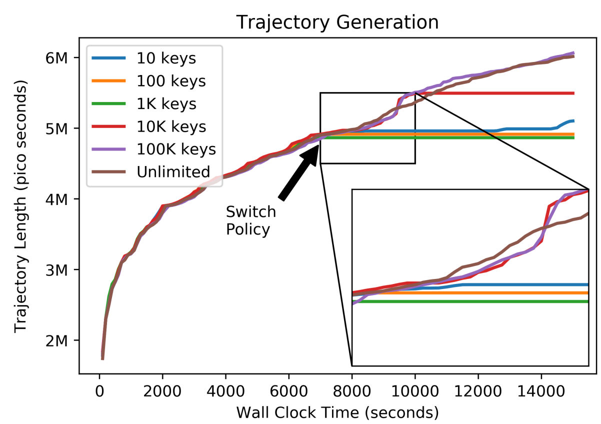

Figure 6: The optimal cache size must strike a balance between performance and the keyspace being 250K keys.

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. The problem is that the thresholds in these policies does not work for different setups (*i.e.* different ParSplice parameters, number of worker nodes, and job lengths). We need a way to identify what thresholds we should use for different job permutations.

4.4 Machine Learning Keyspace Activity

5 CONCLUSION

(1) analysis of Parsplice keyspace
(2) using a modern distributed kv store
(3) positive effects of Mantle

REFERENCES

[1] 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).*

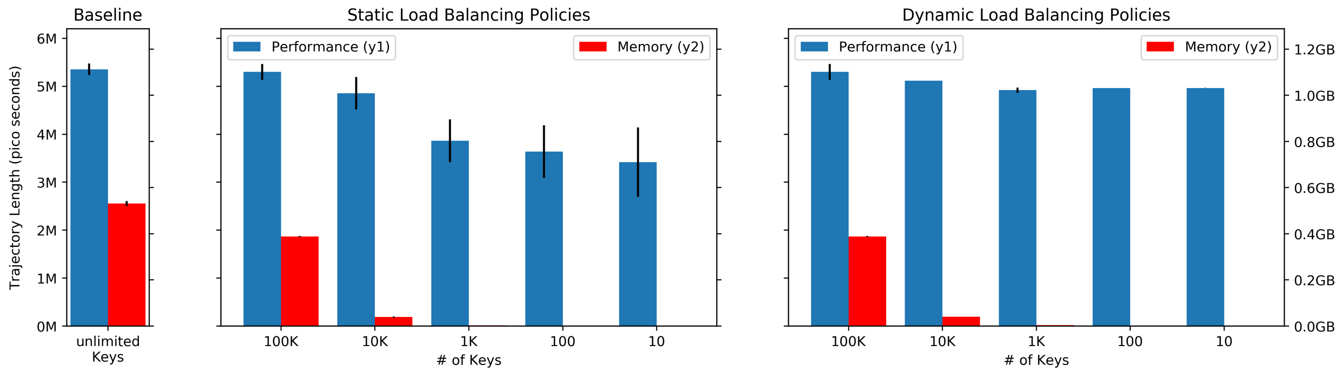

Figure 5: The optimal cache size must strike a balance between performance and resoure utilization. Here we show the trade-off for a static load balancing policy that evicts keys when the cache reaches a certain size. For this configuration, a 100K key cache has the best performance/utilziation, despite the keyspace being 250K keys.

[2] 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)*.

[3] 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)*.

[4] 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* (????).

[5] 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)*.

[6] 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.

[7] 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)*.

[8] 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)*.