Programmable Cache Management Using the Mantle Language and Policy Engine

Michael A. Sevilla, Carlos Maltzahn, Peter Alvaro, *Bradley W. Settlemyer
*Danny Perez, *David Rich, *Galen M. Shipman
University of California, Santa Cruz, *Los Alamos National Laboratory
msevilla@soe.ucsc.edu, {carlosm, palvaro}@ucsc.edu, {bws, danny_perez, dor, gshipman}@lanl.gov

***Abstract*—Our analysis of the key-value activity generated by the ParSplice molecular dynamics simulation demonstrates the need for more complex cache management strategies. Baseline measurements show clear keyspace access patterns and hot spots that offer significant opportunity for optimization. We use the Mantle language and policy engine to dynamically explore a variety of techniques, ranging from basic algorithms and heuristics to statistical models, calculus, and machine learning. While Mantle was originally designed for distributed file systems, we show how the collection of abstractions effectively decomposes the problem into manageable policies for a different domain and service (in this case, cache management). Our exploration of this space results in a two policy scheme that achieves 96% efficiency while using only 7.6% of the memory resources required by the base case.**

I. Introduction

The fine-grained data annotation capabilities provided by key-value storage is a natural match for many types of scientific simulation. Simulations relying on 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 [1] 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.

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. This type of analysis (1) shows the capacity and resource requirements of a single node and (2) will help inform our load balancing policies for when we switch to a distributed key-value store back-end to store EOM minima. For example, Figure 1 shows the cache activity of a ParSplice node and demonstrates 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. The ParSplice caches on each node are trimmed when memory pressure reaches a threshold but the number of unique keys accessed (black points along bottom of both graphs) in Figure 1 suggests that a small cache of EOM minima should be sufficient.

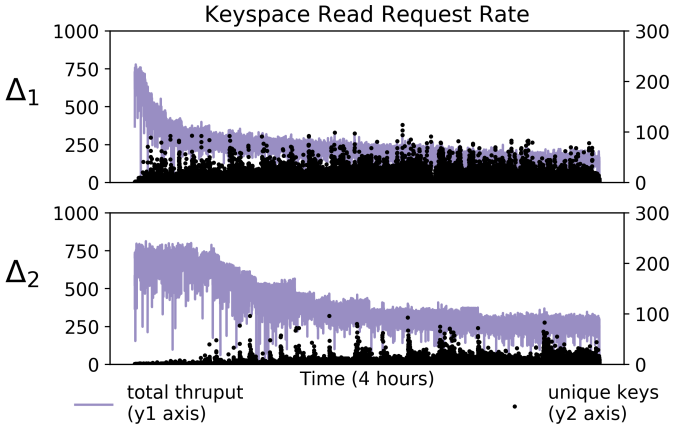

Fig. 1: The keyspace activity for ParSplice using two different growth rates. The line shows the rate that EOM minima values are retrieved from the key-value store ($y1$ axis) and the points along the bottom show the number of keys accessed in a 1 second sliding window ($y2$ axis).

We use the Mantle language and policy engine to dynamically explore the effects of different cache management strategies for the changing key-value workloads generated by ParSplice. Mantle [2] has already proven to be a critical control plane for improving file system metadata load balancing and in this work we show that the collection of abstractions provided by Mantle is useful for reasoning about and designing different cache management strategies. Developers write policies for "when" they want data moved and "how much" of the data to move, then the framework executes these policies whenever a decision needs to be made. These abstraction help developers unfamiliar with the domain quickly reason about, develop, and deploy new policies that control temporal and spatial locality. We show that Mantle:

- decomposes cache management into independent policies that can be dynamically changed, making the problem more manageable and facilitating rapid development. Changing the policy in use is critical in applications such as ParSplice that have alternating stable and chaotic

keyspace access patterns over the course of a long-running simulation.
- can be used to quickly deploy a variety of cache management strategies, ranging from basic algorithms and heuristics to statistical models and machine learning.
- has useful primitives that, while designed for file system metadata load balancing, turn out to also be effective for cache management. This finding shows how the policy engine generalizes to different domains and enables control of policies by machines instead of administrators.

This last contribution is explored in Sections §IV and §V, where we try a range of policies from different disciplines; but more importantly, in Section §VI, we conclude that the collection of policies we designed for ParSplice's cache management are very similar to the policies in the Ceph file system that are used to load balance metadata, suggesting that there is potential for automatically adapting and generating policies dynamically.

II. ParSplice Keyspace Analysis

ParSplice [1] 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) 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.

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 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 operates in 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.

A. 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 is unmodified but for the persistent database node, we replace 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.

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 $5\times$ times longer than our 10 node CloudLab clusters and $25\times$ 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 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

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

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

[3] Empirically, we find this threshold works well for most applications

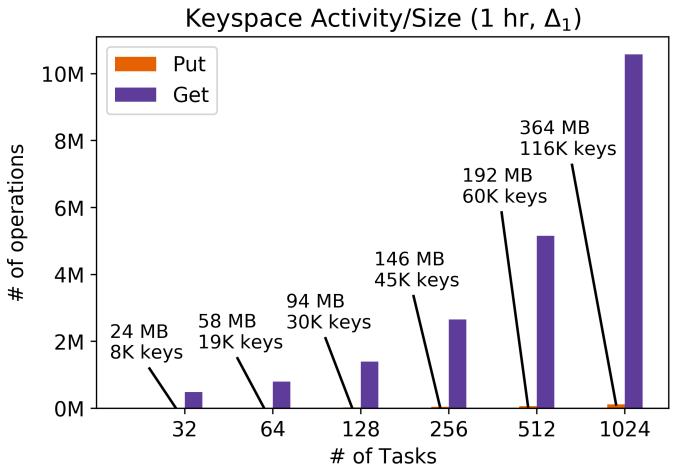

Fig. 2: The keyspace size is small but must satisfy many reads as workers calculate new segments. It is likely that we will need more than one node to manage segment coordinates when we scale the system or jobs up.

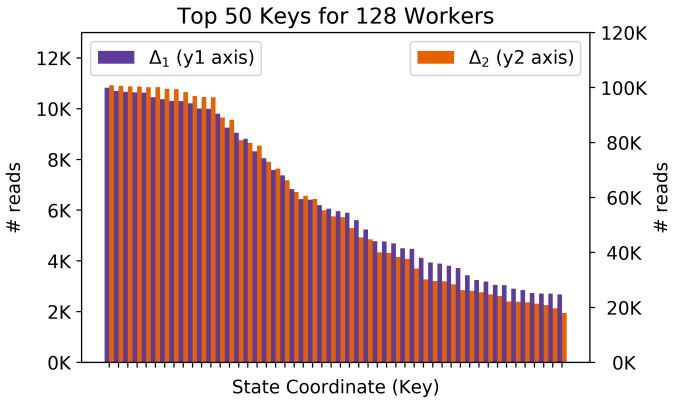

Fig. 3: The keyspace imbalance is due to workers generating deep trajectories and reading the same coordinates. Over time, the accesses get dispersed across different coordinates resulting in some keys being more popular than others.

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

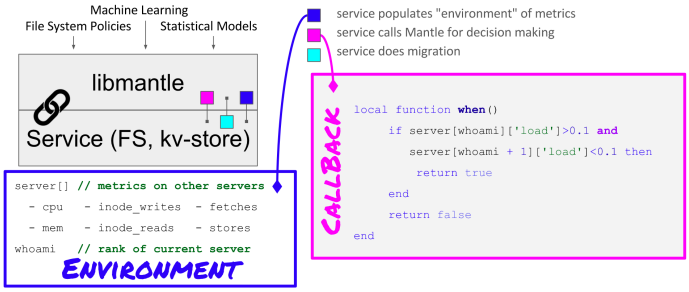

Fig. 4: Extracting Mantle as library.

the same state, which is computationally cheap (this motivates Section §**??**).

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 §**??**). 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 §**??**).

III. Mantle Engine: Methodology

A. Experimental Setup

B. Mantle: Dynamic Load Balancing Policies

We extract Mantle as a library that is linked into a service (*e.g.*, file system, key-value store, application), where administrators program policies that are embedded into the runtime. The service needs to be modified to (1) provide environment of metrics (2) identify where policies are set. At runtime, Mantle executes the admin's policies for when/where/how much and returns a decision.

Services expose two types of metrics with Mantle: cluster metrics for describing resource utilization and time series metrics for describing accesses over time. Table I shows how these metrics are accessed from the policies written by administrators. Cluster metrics are read from a dictionary indexed by server and metric name, where server is a node identifier (e.g., MPI Rank, metadata server name), metric is a resource name and val is the current reading for that metric. Examples of `(metric, value)` pairs are CPU or memory

Metrics	Data Structure	Description
Cluster	{server → {metric → val}}	resource util. for servers
Time Series	[(ts, val), ..., (ts, val)]	accesses by timestamp (ts)
	Service	Example
Cluster	File Systems	CPU util., Inode RDs
	ParSplice	CPU util., Cache Size
Time Series	File Systems	Accesses to directory
	ParSplice	Accesses to key in DB

TABLE I: Types of metrics exposed by the service to the policy engine using Mantle.

pressure, as shown in Figure 4. For time series metrics, the service passes a pointer to an array of `(timestamp, value)` pairs. We use a pointer so we can pass a large number of values, like accesses over time to a database or directory in the file system namespace, but this limits the time series metrics to only include values from the *current* node.

Administrators write policies as callbacks that use the metrics from above. Figure 4 shows an example policy for the "when()" callback, where the current server (`whoami`) migrates load if it is has load (`>0.1`) and if its neighbor server (`whoami + 1`) does not have load (`<0.1`).

C. Integrating Mantle into ParSplice

- providing environment of metrics
- identifying where policies are made

IV. Cache Management Using System Architecture Knowledge

Using the Mantle policy engine, we test a variety of cache management tools and algorithms on a single in-memory database node using the keyspace analysis in Section §II-A. These strategies are implemented as the "when" and "how much" policies from the Mantle API; the "where" callback does not make sense for a single node (segments are either in the cache or they are not). The evaluation metric is the accuracy and runtime of each strategy; the strategy should be accurate enough so as to sacrifice negligible performance and fast tenough to run as often as we want to detect regimes.

First we sized the cache according to our system specific knowledge. By "system", we mean the hardware and software of the storage hierarchy. We look at request rate, unique keys in a sliding window, and bandwidth capabilties. For example, we know that LevelDB cannot handle high IO request rates.

In the original ParSplice implementation, each cache node uses 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 5. "Baseline" is the performance of unmodified ParSplice measured in trajectory duration (y axis) and utilization is measured with memory footprint of just the cache ($y2$ axis). The other

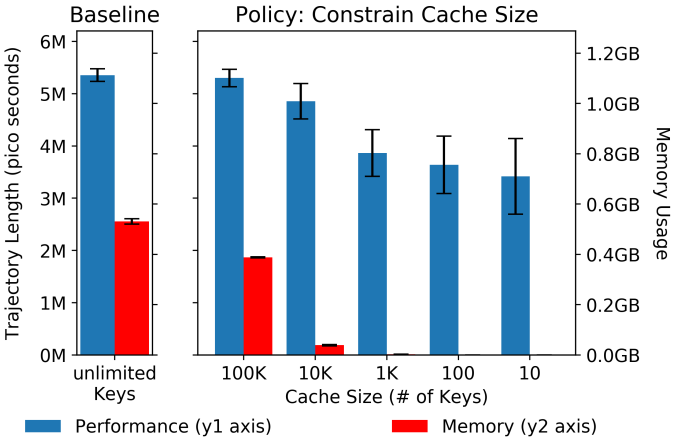

Fig. 5: The performance and resource utilization trade-off for different cache sizes. "Baseline" is ParSplice unmodified and the "Policy: Constrain Cache Size" graph limits the size of the cache to save memory.

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 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.

Despite the memory savings, our results suggest that dynamic load balancing policies could save even more memory. Figure 5 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.

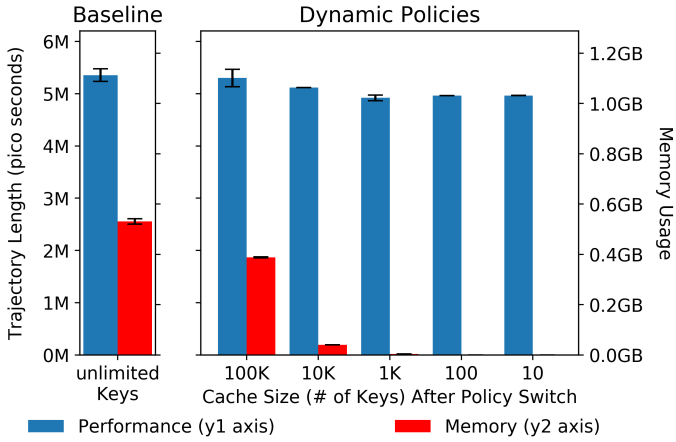

Fig. 6: The performance and resource utilization trade-off of using a dynamic load balancing policy that switches to a constrained cache policy after absorbing the initial burstiness of the workload. The sizes of these smaller caches are on the x axis.

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 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]–[8]. Previous work quantified the speedups achieved with Mantle and formalized balancers that were good for file systems.

Figure 6 shows the results of using the Mantle API to program a dynamic load balancing policy into ParSplice that switches between two policies:

- 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. 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 Figure 6 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.

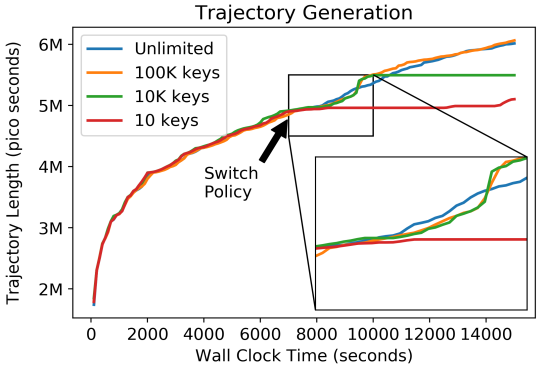

Fig. 7: The rate that the trajectory is computed decays over time (which is expected) but this skews the performance improvements in Figure 6. Our dynamic policy works for 2.5 hour jobs but not for 4 hour jobs.

Caveats: The results in Figure 6 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 7; 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 5 and 6 we plot the final value. For (3), Figure 7 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.

V. Cache Management Using Domain-Specific Application Knowledge

Feeding domain-specific knowledge about the ParSplice application into a policy leads to more a accurate cache management strategy. Figure 8 shows which keys (y axis) are accessed by the ParSplice tasks over time (x axis). The groups of accesses to a subset of the keys, or "access regimes", occur because molecules are stuck in deep trajectories. Recall that

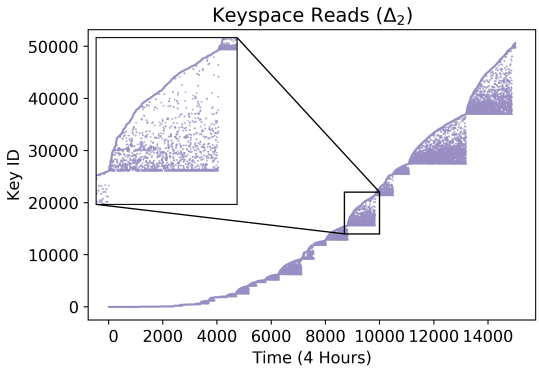

Fig. 8: Key activity for a 4 hour run shows groups of accesses to the same subset of keys. Detecting this "access regimes" leads to a more accurate cache management strategy.

the in-memory database stores the molecules' EOM minima, which is the smallest effective energy that a molecule observes during its trajectory. So molecules stuck in deep trajectories explore the same minima until they can escape to a new set of states. This exploration of the same set of states is called a superbasin.

Detecting these superbasins can lead to more effective cache management strategies because the height of the keyspace accesses is "how much" of cache to evict and the width of the keyspace accesses is "when" to evict values from the cache. The zoomed portion of Figure 8 shows how a single superbasin affects the key accesses. Moving along the x axis shows that the number of unique keys accessed over time grows while moving along the y axis shows that early keys are accessed more often. Overall, superbasins are never re-visited because the simulation only adds molecules; we can never reach a state with less molecules. This is why keys are never re-accessed. Despite these patterns, the following characteristics of superbasins make it hard to detect them:

- superbasin key accesses are random and there is no threshold "minimum distance between key access" that indicates we have moved on to a new superbasin
- superbasins change immediately
- the number of keys a superbasin accesses differs from other superbasins

Below we describe the policies we implemented and deployed using Mantle.

A. Failed Strategies

These techniques proliferated more knobs that obfuscated the problem.

- Statistics
- Calculus
- K-Means
- DBScan
- Anomaly Detection

B. Regime Detection

At each time step, we find the lowest ID and compare against the local minimum, which is the smallest ID we have seen thus far. If we move left to right, the local minimum never changes because the local minimum will start small. If we move from right to left, the local minimum changes at each access regime. For points z, y, and x, if the local minimum is the same we are in a regime. Processing y, we set the local minimum to be $min(y, m_l)$, where m_l is the local minimum of the previous time step of z. The algorithm incorrectly detects a regime change if the local minimum of y is lower than local minimum of z, since y may have points *within* z; recall that we are trying to detect the whole fan, not just the bottom edge of each fan.

C. Trajectory Length

D. Cloud Techniques: Elastic Search

What if we re-provision resources in response to events outside the application's control, such as a slow Lustre.

E. Future work

How Much: cache policy from past, regime detection, Belady's Min

VI. Relation to File Systems

The code snippets in Figures 9 and 10 are the policies used in ParSplice and CephFS, respectively. ParSplice uses policies to manage its caches and CephFS uses policies to control load balancing, but they both can be expressed with the Mantle API. From a high-level the ParSplice policy trims the cache if the cache reaches a certain size *and* if it has already absorbed the initial burstiness of the workload; the CephFS policy migrates load if the metadata load is higher than the average load *and* the current load has been overloaded for more than two iterations.

Condition for "Overloaded" (Fig. 9:Line 2; Fig. 10:Line 2) - these lines detect whether the node is overloaded using the "load" calculated in the load callback; while the load calculations and thresholds are different, the actual logic is exactly the same. Recall that this decision is made locally because there is no global scheduler or centralized intelligence.

State Persisted Across Decisions (Fig. 9:Lines 5,6; Fig 10:Lines 3,4,10) - these lines use the Mantle API to write/read state from previous decisions. For ParSplice, we save a boolean that indicates whether we have absorbed the workload's initial burstiness. For CephFS, we save the number of consecutive instances that the server has been overloaded. We also clear the count (Line 10) if the server is no longer overloaded. The underlying implementation saves the values to local disk.

Condition that Switches Policy (Fig. 9:Line 3; Fig. 10:Line 5) - these lines switch the policies using information from previous decisions. ParSplice trims its cache once it eclipses the "absorb" threshold while CephFS allows balancing when overloaded for more than two iterations. The persistent state is essential for both of these policy-switching conditions.

```
local function when()
  if servers[whoami]["load"] > target then
    if servers[whoami]["load"] > absorb_target then
      WRstate(1)
    end
    if RDstate() == 1 then
      return true
    end
  end
  return false
end
```

Fig. 9: ParSplice cache management policy.

```
local function when()
  if servers[whoami]["load"] > target then
    overloaded = RDstate() + 1
    WRstate(overloaded)
    if overloaded > 2 then
      return true
    end
  end
  else then
    WRstate(0)
  end
  return false
end
```

Fig. 10: CephFS file system metadta load balancer.

- Lustre Trace
- LinkedIn Trace
- Nathan's Trace

A. Using File System Balancers for ParSplice

B. Using ParSplice Balancers for File Systems

C. Visualizing File System Traces like ParSplice Keyspace Traces

VII. 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 [9]. 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 [10] 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 [6] 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) [11], [12] 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+ [13] 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 [14], [15], big data systems systems [16], and databases [17]. 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 thresholds), we did not need anything as sophisticated as the genetic algorithm used in [14]. 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 [18] as our back-end key-value store because it was designed for HPC and has the proper mechanisms for migration already implemented.

VIII. Future Work

This lays the foundation for future work, where we will focus on formalizing a collection of general data management policies that can be used across domains and services. The value of such a collection eases the burden of policy development and paves the way for solutions that remove the administrator from the development cycle, such as (1) adaptable policies that automatically switch to new strategies when the current strategy behaves poorly (e.g., thrashing, making no progress, etc.), and (2) policy generation, where new policies are constructed automatically by examining the collection of existing policies. Such work is made possible with Mantle's ability to dynamically change policies.

IX. Conclusion

References

[1] D. Perez, E. D. Cubuk, A. Waterland, E. Kaxiras, and A. F. Voter, "Long-Time Dynamics Through Parallel Trajectory Splicing," *Journal of chemical theory and computation*.

[2] M. A. Sevilla, N. Watkins, C. Maltzahn, I. Nassi, S. A. Brandt, S. A. Weil, G. Farnum, and S. Fineberg, "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*, ser. SC '15, 2015.

[3] Q. Zheng, K. Ren, and G. Gibson, "BatchFS: Scaling the File System Control Plane with Client-funded Metadata Servers," in *Proceedings of the 9th Workshop on Parallel Data Storage*, ser. PDSW' 14, 2014.

[4] Q. Zheng, K. Ren, G. Gibson, B. W. Settlemyer, and G. Grider, "DeltaFS: Exascale File Systems Scale Better Without Dedicated Servers," in *Proceedings of the 10th Workshop on Parallel Data Storage*, ser. PDSW' 15, 2015.

[5] G. Grider, D. Montoya, H.-b. Chen, B. Kettering, J. Inman, C. DeJager, A. Torrez, K. Lamb, C. Hoffman, D. Bonnie, R. Croonenberg, M. Broomfield, S. Leffler, P. Fields, J. Kuehn, and J. Bent, "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*, ser. PDSW'15, November 2015.

[6] K. Ren, Q. Zheng, S. Patil, and G. Gibson, "IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion," in *Proceedings of the 20th ACM/IEEE Conference on Supercomputing*, ser. SC '14, 2014.

[7] S. V. Patil and G. A. Gibson, "Scale and Concurrency of GIGA+: File System Directories with Millions of Files," in *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, ser. FAST '11, 2011.

[8] S. A. Brandt, E. L. Miller, D. D. E. Long, and L. Xue, "Efficient Metadata Management in Large Distributed Storage Systems," in *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies*, ser. MSST '03, 2003.

[9] J. Jenkins, G. M. Shipman, J. Mohd-Yusof, K. Barros, P. H. Carns, and R. B. Ross, "A Case Study in Computational Caching Microservices for HPC," in *IPDPS Workshops*. IEEE Computer Society, 2017, pp. 1309–1316.

[10] M. A. Sevilla, N. Watkins, I. Jimenez, P. Alvaro, S. Finkelstein, J. LeFevre, and C. Maltzahn, "Malacology: A Programmable Storage System," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17, 2017.

[11] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller, "Dynamic Metadata Management for Petabyte-Scale File Systems," in *Proceedings of the 17th ACM/IEEE Conference on Supercomputing*, ser. SC'04, 2004.

[12] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A Scalable, High-Performance Distributed File System," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design & Implementation*, ser. OSDI'06, 2006.

[13] S. V. Patil and G. A. Gibson, "Scale and Concurrency of GIGA+: File System Directories with Millions of Files," in *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, ser. FAST '11, 2011.

[14] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, R. Aydt, Q. Koziol, M. Snir *et al.*, "Taming parallel i/o complexity with auto-tuning," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 68.

[15] B. Behzad, S. Byna, S. M. Wild, and M. Snir, "Improving Parallel i/o Autotuning with Performance Modeling," 2014.

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

[17] K. Schnaitter, N. Polyzotis, and L. Getoor, "Index interactions in physical design tuning: modeling, analysis, and applications," vol. 2.

[18] H. Greenberg, J. Bent, and G. Grider, "MDHIM: A Parallel Key/Value Framework for HPC," in *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*, 2015.