# Cache Management Using a Policy Engine for Load Balancing

Michael A. Sevilla, Carlos Maltzahn, *Bradley W. Settlemyer, *Danny Perez, *David Rich, *Galen M. Shipman
University of California, Santa Cruz, *Los Alamos National Laboratory
msevilla@soe.ucsc.edu, carlosm@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 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 it 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. We reason that limiting the size of a cache on a single node saves memory and sacrifices negligible performance. 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. 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.

To explore the effects of different cache management strategies, we link the Mantle policy engine into 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 its usefulness in cache management for the changing key-value workloads generated by ParSplice. Developers write policies for "when" they want data moved, "where" they want data moved, and "how much" of the data to move and the framework executes these policies whenever a decision needs to be made. This abstraction helps 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 Section §IV, where we try a range of policies from different disciplines; but more importantly, in Section §V, 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)

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

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.

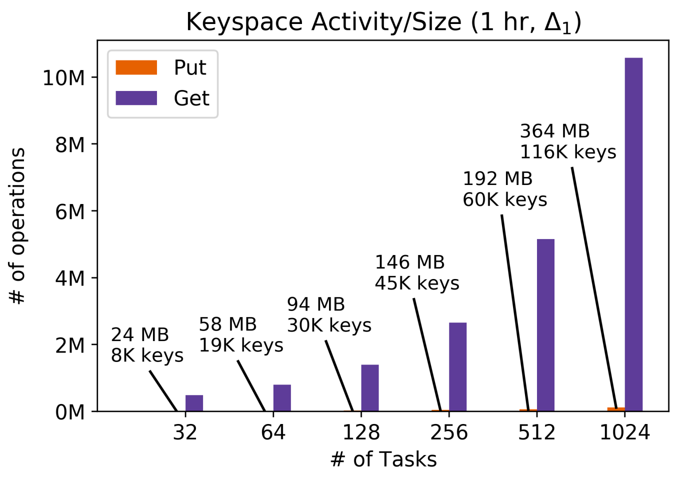

Fig. 1: The keyspace size is small but must satisfy many reads as workers calculate new segments. It is likely that we will need more than one node to manage segment coordinates when we scale the system or jobs up.

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 1 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 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 1 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

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

[3]Empirically, this is a threshold that we find to work well for most applications

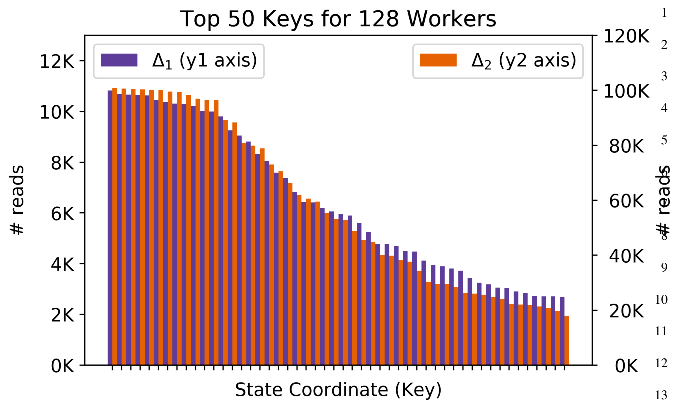

Fig. 2: The keyspace imbalance is due to workers generating deep trajectories and reading the same coordinates. Over time, the accesses get dispersed across different coordinates resulting in some keys being more popular than others.

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 §**??**).

*Entropy increases over time:* The reads per second in Figure **??** 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 **??** are shown in Figure 2, 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 2 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 **??** shows how entropy for different growth rates has temporal locality, as the reads per second for $\Delta_2$ looks like the reads per second for $\Delta_1$ 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

- motivation: Mochi load balancer microservice
- background: CephFS implementation
- library architecture, callbacks, environment

```
-- assume that (ts, keyid) are in a table
local function when()


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

Fig. 3: ParSplice cache management policy.

### C. Integrating Mantle into ParSplice

- providing environment of metrics
- identifying where policies are made

## IV. Mantle Brains: Tools We Plug In to Detect "when"

Requirements: run online, be fast enough to run as often as we want to detect regimes

### A. System Specific Knowledge

*e.g.*, request rate, unique keys in a sliding window, bandwidth capabilties. For example, we know that LevelDB cannot handle high IO request rates.

*1) Request Rate:*

*2) Belady's Min:*

### B. Domain Specific Knowledge

*e.g.*, ParSplice key access locality.

*1) 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.

*2) Trajectory Length:*

### C. Failed, Overcomplicated Brains

These techniques proliferated more, less transparent knobs

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

```
1  local function when()
2    if servers[whoami]["load"] > target then
3      if servers[whoami]["load"] > absorb_target then
4        WRstate(1)
5      end
6      if RDstate() == 1 then
7        return true
8      end
9    end
10   return false
11 end
```

Fig. 4: ParSplice cache management policy.

```
1  local function when()
2    if servers[whoami]["load"] > target then
3      overloaded = RDstate() + 1
4      WRstate(overloaded)
5      if overloaded > 2 then
6        return true
7      end
8    end
9    else then
10     WRstate(0)
11   end
12   return false
13 end
```

Fig. 5: CephFS file system metadta load balancer.

### 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

## V. Relation to File Systems

The code snippets in Figures 4 and 5 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. 4:Line 2; Fig. 5: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. 4:Lines 5,6; Fig 5: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. 4:Line 3; Fig. 5: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.

- Lustre Trace
- LinkedIn Trace
- Nathan's Trace

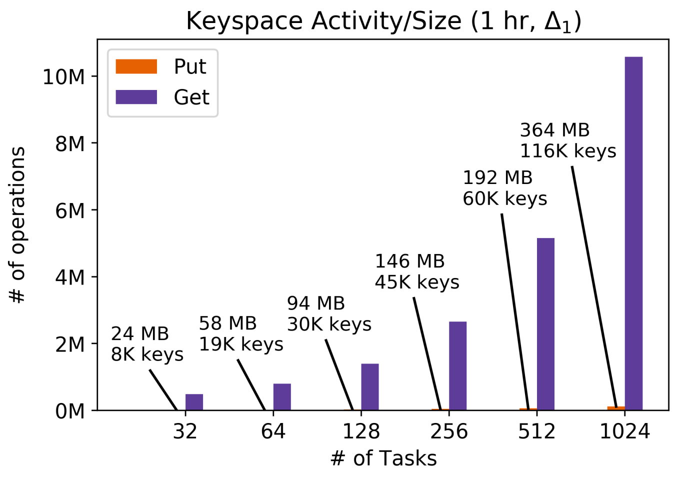

Fig. 6

### A. *Using File System Balancers for ParSplice*

### B. *Using ParSplice Balancers for File Systems*

### C. *Visualizing File System Traces like ParSplice Keyspace Traces*

## VI. 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.

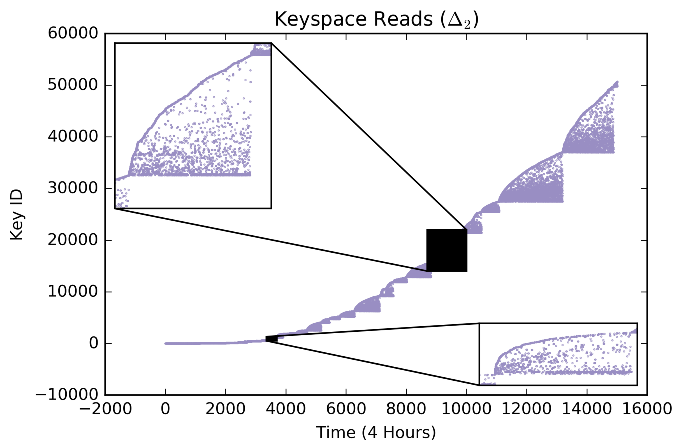

Fig. 7

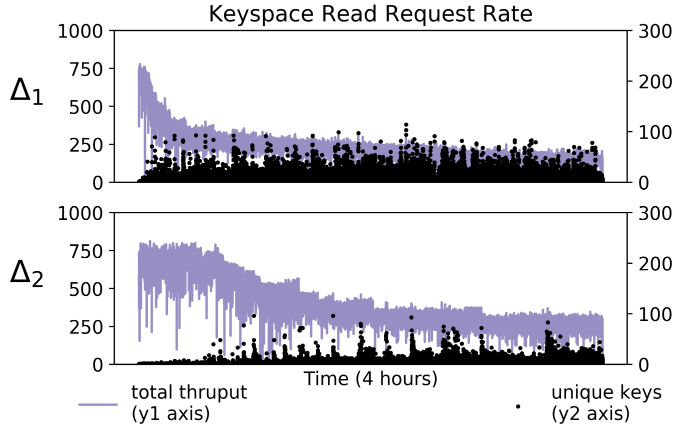

Fig. 8

## VII. 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.

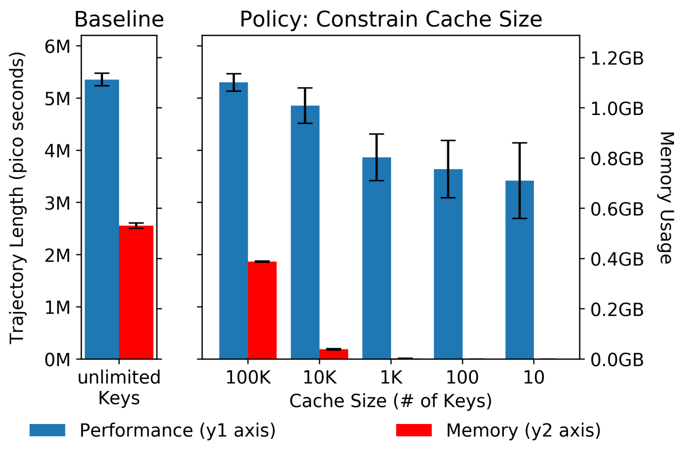

Fig. 9

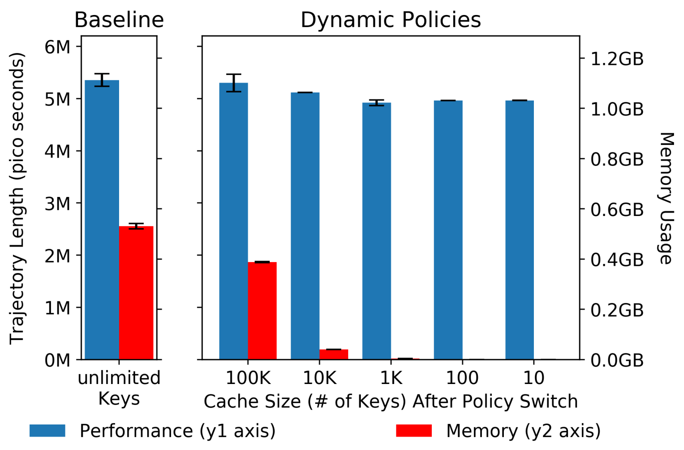

Fig. 10

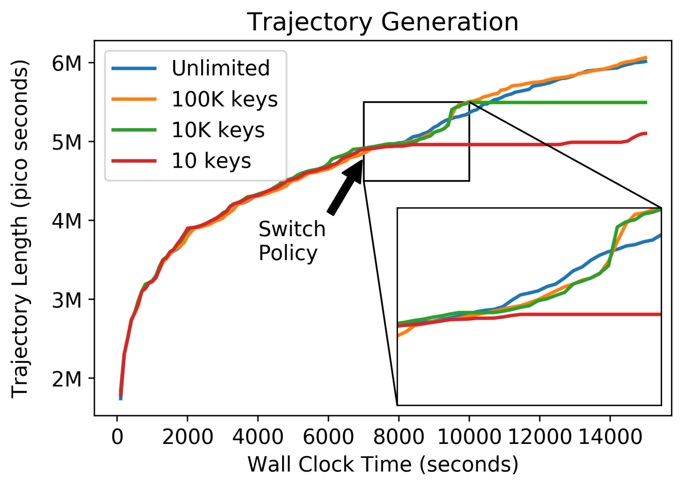

Fig. 11