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 data management and policy engine from the Mantle system 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. Our exploration of this space results in a dynamically sized cache policy that, for our initial conditions, sacrifices negligible performance while using only 28% of the memory required by our hand-tuned cache.**

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 caches and a single persistent 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, so we need more effective data management techniques, such as cache management or load balancing across a cluster.

In this paper, we design cache management policies for ParSplice, driven by a detailed analysis of the key-value accesses over the course of a long running simulation across a variety of initial conditions. The default ParSplice implementation uses an unlimited sized cache, shown by the "No Cache Management" line Figure 1. This solution is unacceptable for HPC environments, where memory is precious and a common goal is to keep memory for such data structures below 3%[1]. While users can configure ParSplice to resize the cache when it reaches a certain threshold, this solution requires tuning and parameter sweeps; the dashed "Cache (too small)" curve in

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

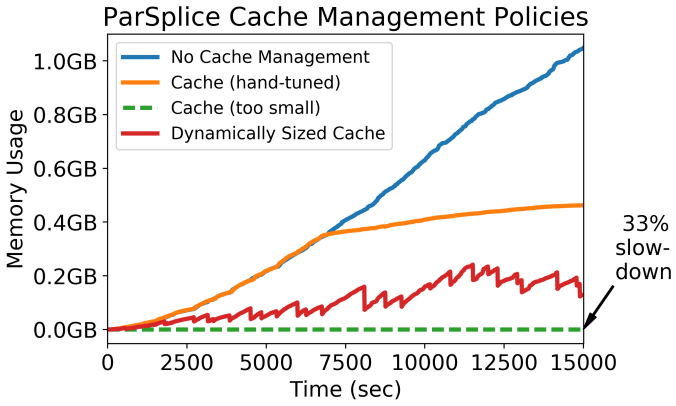

Fig. 1: Using our data management language and Mantle policy engine, we designed a dynamically sized caching policy for the ParSplice application. The solution uses knowledge about the application to detect key access patterns and adjust the cache accordingly.

Figure 1 shows how a poorly configured cache hurts performance. Our final solution, shown by the "Dynamically Sized Cache" line in Figure 1, detects keyspace access patterns and re-sizes the cache accordingly. Without tuning or parameter sweeps, our solution saves more memory than a hand-tuned cache without sacrificing performance. More importantly, it works for a variety initial conditions without changing the policy itself.

To design more flexible cache management policies, like our "Dynamically Sized Cache" policy, we use the data management language and policy engine from the Mantle paper [2]. This framework allows us to dynamically explore the effects of different software-defined cache management strategies for the changing key-value workloads generated by ParSplice. Mantle was originally touted as a programmable file system metadata load balancer, but we realize now that the collection of abstractions designed for file systems was a control plane that improved metadata access. So in this paper we refer to Mantle as a policy engine that injects policies written in our data management language directly into a running service and show how this approach is useful for reasoning about and designing different cache management

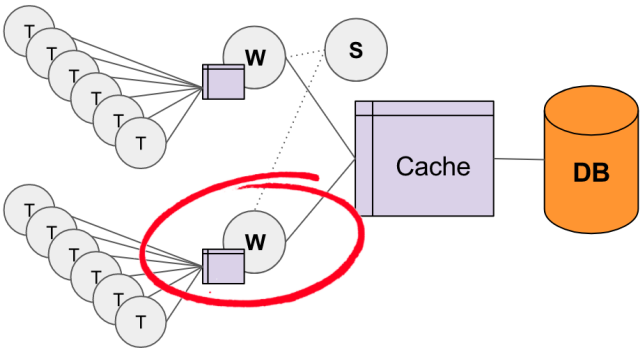

Fig. 2: The ParSplice architecture has a storage hierarchy of caches and a dedicated cache node (boxes) backed by a persistent database (DB). A splicer (S) tells worker (W) to generate segments and workers employ tasks (T) for more parallelization. We focus on the worker's cache (circled), which facilitates communication and segment exchange between the worker and its

strategies in ParSplice. By service, we mean a system that manages data and responds requests, such as a file system or key-value store. 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 abstractions 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 used to load balance metadata in the Ceph file system (CephFS) 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[2] and is important to LANL's Materials for the Future initiative. As shown in Figure 2, the 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 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 processes to act as caches that sit in front of a single persistent database. Values are written to each tier and reads traverse up the hierarchy until they find the data. Caches also reside on the workers to service reads/writes from its tasks.

We use ParSplice to simulate the evolution of metallic nanoparticles that grow from the vapor phase. This simulation stresses the storage hierarchy more than other input decks because it uses a cheap potential, has a small number of atoms, and operates in a complex energy landscape with many accessible states. As the run progresses, the energy landscape of the system becomes more complex and more states are visited. 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 but temperatures that are too high result in meaningless simulations because trajectories have so much energy that they are equally likely to visit any random state.

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. In Figure 5, the Δ_1 growth rate adds atoms every 100K microseconds while the Δ_2 growth rate adds atoms

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

every 1 million microseconds. So Δ_2 has a smaller growth rate resulting in hotter keys (line on $y1$ axis) and a smaller keyspace (dots on the $y2$ axis). Values smaller than Δ_2's growth rate or a temperature of 400 degrees results in very little database activity because state transitions take too long. Similarly, values larger than Δ_1's growth rate or a temperature of 4000 degrees result in an equally meaningless simulation as transitions are unrealistic.

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.

Experimental Setup: 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 . 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 our 10 node cluster at UCSC. 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.

The scalability experiment uses 1 splicer, 1 persistent database, 1 cache process, and up to 2 workers. We scale up to 1024 tasks, which spans 32 nodes and disable hyper-threading because we experience unacceptable variability in performance. For the rest of the experiments, we use 8 nodes, 1 splicer, 1 persistent database, 1 cache process, 1 worker, and up to 256 tasks. The keyspace analysis that follows is for the cache on the worker node, which is circled in Figure 2. 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[3] version, which comes instrumented with its own set of performance counters.

Scalability: Figure 3 shows the keyspace size (text 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: Figure 5 shows how ParSplice tasks read key-value pairs from the worker's cache for two different initial conditions of Δ, which is the rate that new atoms enter the simulation. The line is the read request rate ($y1$ axis) and the dots along the bottom are the number of unique keys accessed ($y2$ axis). This figure demonstrates that

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

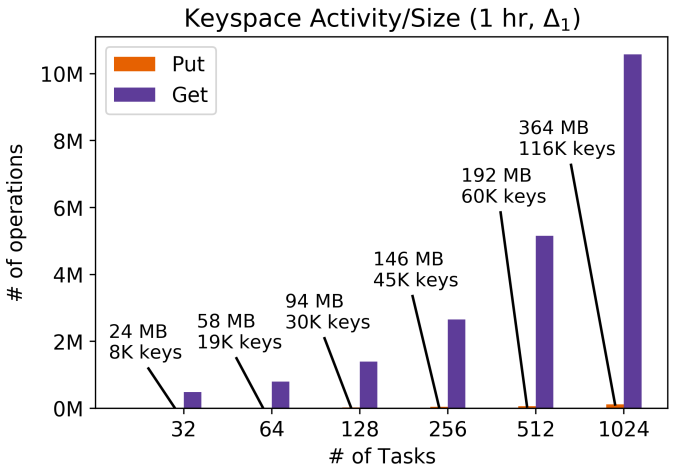

Fig. 3: The keyspace size is small but must satisfy many reads as workers calculate new segments. It is likely that we will need more than one node to manage segment coordinates when we scale the system or jobs up.

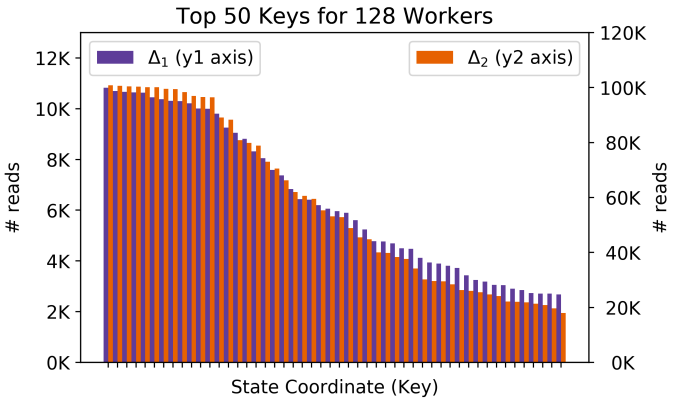

Fig. 4: The keyspace imbalance is due to workers generating deep trajectories and reading the same coordinates. Over time, the accesses get dispersed across different coordinates resulting in some keys being more popular than others.

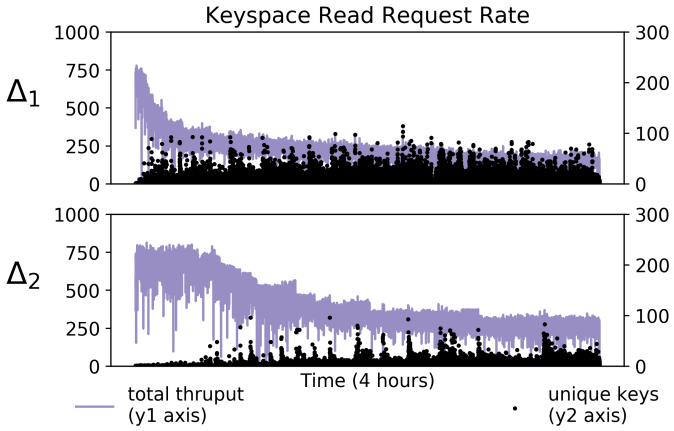

Fig. 5: The keyspace activity for ParSplice using two different growth rates. The line shows the rate that EOM minima values are retrieved from the key-value store ($y1$ axis) and the points along the bottom show the number of keys accessed in a 1 second sliding window ($y2$ axis).

small changes to 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 in Figure 5 suggests that a small cache of EOM minima should be sufficient.

The bars in Figure 3 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 §V).

Entropy increases over time: The reads per second in Figure 5 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 5 are shown in Figure 4, 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 §V). The growth rate, temperature, and number of workers have a predictable effect on the structure of the keyspace. Figure 4 shows that the number of reads changes with different initial conditions (Δ), but that the spatial locality of key accesses is similar (*e.g.*, some keys are still $5\times$ more popular than others). Figure 5 shows how entropy for different growth rates has temporal locality, as the reads per second for

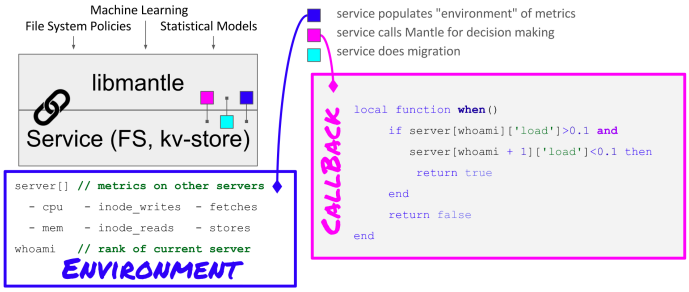

Fig. 6: Extracting Mantle as library.

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 §V).

III. Methodology

To explore dynamic cache management policies that change during the run, we use the data management language and policy engine presented in [2]. The prototype in that paper was called Mantle and it was originally built on the CephFS so administrators could control file system metadata data management policies. We now refer to Mantle as a policy engine that supports our data management language. The basic premise is that data management policies can be expressed with a simple API consisting of "when", "where", and "how much". "when" controls how aggressive or conservative the decisions are; "where" lets the policy control how distributed or concentrated the data should be; and "how much" controls the amount of data that should be sent.

The succinctness of the API lets users inject multiple, dynamic policies. In this work we focus on a single node, so the "where" policy is not used. When we move ParSplice to a distributed key-value store back-end, the "where" policy can be used determine which key-value pairs should be moved to which node.

A. Extracting Mantle as a Library

We extracted Mantle as a library and Figure 6 shows how it is linked into a service. Administrators write policies from whatever domain they choose (e.g., statistics, machine learning, storage system) and the policies are embedded into the runtime by Mantle. When the service makes decisions it executes the administrator-defined policies for when/where/how much and returns a decision. To do this, the service needs to be modified to (1) provide an environment of metrics and (2) identify where policies are set. These modification points are shown by the colored boxes in Figure 6 and described below.

1) Policies Written as Callbacks: administrators write policies in Lua that are executed whenever the service needs to make a data management decision. We chose Lua for simplicity, performance, and portability; it is a scripting language with simple syntax, which allows administrators to focus on

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 reads
	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.

the policies themselves; it was designed as an embeddable language, so it is lightweight and does less type checking; and it interfaces nicely with C/C++. The "callback" box in Figure 6 shows an example policy for the "when()" callback in a distributed service, 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`). The load is calculated using the metrics provided by the environment.

Mantle also provides functions for persisting state across decisions. `WRState(s)` saves state `s`, which can be a number or boolean value, and `RDState()` returns the state saved by a previous iteration. For example, a "when" policy can avoid trimming a cache or migrating data if it had performed that operation in the previous decision using:

```
if RDState() == 1 then
  WRState(0)   -- the next decision will return true
  return false
else then
  WRState(1)   -- the next decision will return false
  return true
end
```

2) Environment of Metrics: services can expose cluster metrics for describing resource utilization and time series metrics for describing accesses to some data structure over time. Table I shows how these metrics are accessed from the policies written by administrators.

For cluster metrics, the service passes a dictionary to Mantle, indexed by server name and metric. Policies access the cluster metric values by indexing into a Lua table using `server` and `metric`, where server is a node identifier (e.g., MPI Rank, metadata server name) and metric is a resource name. For example, a policy written for an MPI-based service can access the CPU utilization of the first rank in a communication group using:

```
load = servers[0]['cpu']
```

Other examples metrics used for file system metadata load balancing are shown by the "environment" box in Figure 6. The measurements and exchange of metrics between servers is done by the service; Mantle in CephFS leverages metrics from other servers collected using CephFS's heartbeats.

For time series metrics, the service passes an array of `(timestamp, value)` pairs to Mantle and the policies can iterate over the accesses. For example, a policy that uses accesses to a directory in a file system as a metric for load collects that information using:

```
ts = array()
for i=1,arraysize() do
  for time,value in string.gmatch(ts, (%w+=%w)) do
    if value == 'mydirectory' then
      count = count + 1
    end
  end
end
```

The service uses a pointer to the time series to facilitate time series with many values, like accesses to a database or directory in the file system namespace. This decision limits the time series metrics to only include values from the *current* node, although this is not a limitation of Mantle itself.

B. Integrating Mantle into ParSplice

Using Mantle cluster metrics, we expose cache size, CPU utilization, and memory pressure of the worker node to the cache management policies. In Section §II-A we only end up using the cache size although the other metrics proved to be valuable debugging tools. Using Mantle time series metrics, we expose accesses to the cache as a list of `timestamp, keyid` pairs. In Section §V, we explore a regime detection algorithm that uses this metric.

We link Mantle directly into the worker cache circled in red in Figure 2. We put the "when" and "how much" callbacks alongside code that checks for memory pressure. It is executed right before the worker processes incoming and outgoing put/get transactions to the cache. As stated previously, we do not use the "where" part of Mantle because we focus on a single node, but this part of the API will be used when we move the caches and storage nodes to a key-values store back-end that uses key load balancing and repartitioning.

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 enough to run as often as we want to detect regimes. The goal of the following sections is not to find an optimal solution, as this can be done with parameter sweeps for thresholds; rather, we try to find techniques that work for a range of inputs and system setups.

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 capabilities. For example, we know that LevelDB cannot handle high IO request rates.

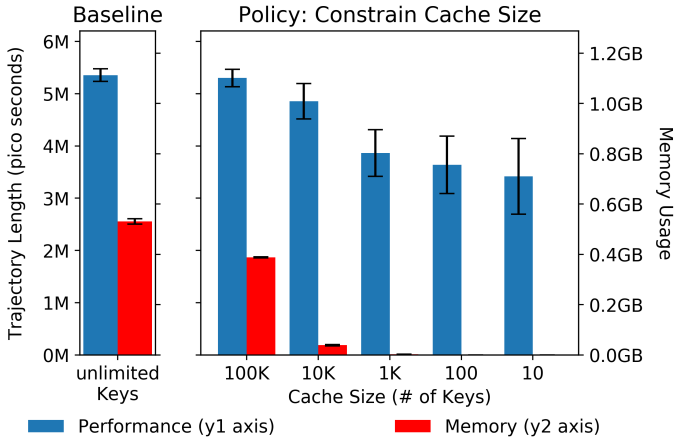

Fig. 7: The performance and resource utilization trade-off for different cache sizes. "Baseline" is ParSplice unmodified and the "Policy: Constrain Cache Size" graph limits the size of the cache to save memory.

```
function when()
  if server[whoami]['cachesize'] > n then
    return true
  end
  return false
end

function howmuch()
  if servers[whoami]['cachesize'] > n
    return servers[whoami]['cachesize'] - n
  end
  return 0
end
```

Fig. 8: Policy that implements a basic LRU cache. The performance and memory utilization for different values of n are graphed in Figure 7.

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 7. "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 graph shares the y axis and shows the trade-off of using a basic LRU-style cache for different cache sizes, implemented using the "when" and "how much" policies in Figure 8. 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. Despite the memory savings, our results suggest that dynamic cache management

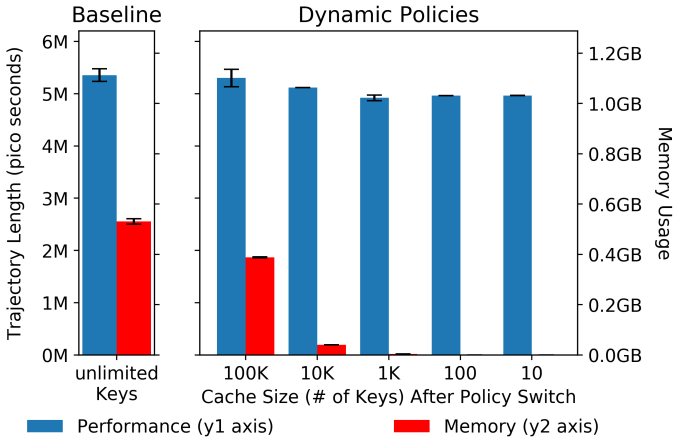

Fig. 9: The performance and resource utilization trade-off of using a dynamic cache management policy that switches to a constrained cache policy after absorbing the initial burstiness of the workload. The sizes of these smaller caches are on the x axis.

policies could save even more memory. Figure 7 show that a 100K key cache is sufficient as a static policy but the top graph in Figure 5 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 5, 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 cache management policy.

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 9 shows the results of using Mantle to program a dynamic cache management policy into ParSplice that switches between two policies:

```
1  function when()
2    if server[whoami]['cachesize'] > n then
3      if server[whoami]['cachesize'] > 100K then
4        WRstate(1)
5      end
6      if RDstate() == 1 then
7        return true
8      end
9    end
10   return false
11 end
```

Fig. 10: A more complicated LRU-based cache policy that has two parts. After absorbing the initial burstiness of the workload, the policy switches to a more constrained cache to limit memory usage. The performance/utilization for different values of `n` is shown in Figure 9.

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

The actual policy is shown in Figure 10 where the only difference between the code snippet in Figure 8 is the addition of lines 2-7. Note that policy also uses the `RDState` and `WRState` functions described in Section §III-A.

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 9 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 9 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 11; 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 7 and 9 we plot the final value. For (3), Figure 11 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 cache management policy.

Despite these caveats, the result is still valid: we found a dynamic cache management 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

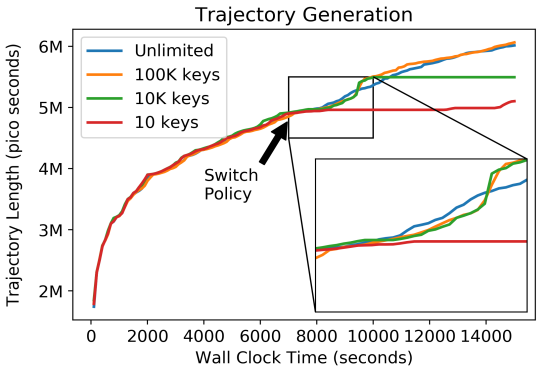

Fig. 11: The rate that the trajectory is computed decays over time (which is expected) but this skews the performance improvements in Figure 9. Our dynamic policy works for 2.5 hour jobs but not for 4 hour jobs.

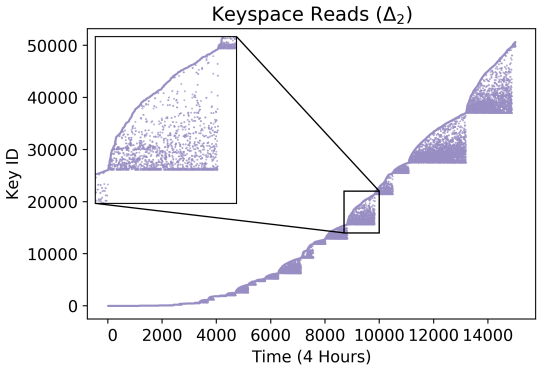

Fig. 12: Key activity for a 4 hour run shows groups of accesses to the same subset of keys. Detecting these "access regimes" leads to a more accurate cache management strategy.

effectiveness of the 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 12 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 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 12 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.

VI. General Data Management Policies

In the previous section, we used our data management language and the Mantle policy engine to design effective cache management strategies for a new service and domain. In this section, we compare and contrast the policies examined for file system metadata load balancing in [2] with the ones designed for cache management in ParSplice. The similarities show how the "when"/"where"/"how much" abstractions, data management language, and policy engine may be widely applicable to other data management techniques, such as QoS, scheduling, and batching.

```
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. 13: CephFS file system metadata load balancer.

A. Using File System Policies for ParSplice

From a high-level the cache management policy we designed in Figure 10 trims the cache if the cache reaches a certain size *and* if it has already absorbed the initial burstiness of the workload. Much of this implementation was inspired by the file system metadata load balancing policy in Figure 13, which was presented in [2]. That policy migrates load if the metadata load is higher than the average load in the cluster *and* the current load has been overloaded for more than two iterations. The two policies have the following in common:

Condition for "Overloaded" (Fig. 10: Line 2; Fig. 13: Line 2) - these lines detect whether the node is overloaded using the load calculated in the load callback (not shown). While the calculations and thresholds are different, the way the loads are used is exactly the same; the ParSplice policy flags the node as overloaded if the cache reaches a certain size while the CephFS policy compares the load to other nodes in the system.

State Persisted Across Decisions (Fig. 10: Lines 4,6; Fig 13: Lines 3,4,10) - these lines use Mantle 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.

Two-Part Policies (Fig. 10: Line 6; Fig. 13: Line 5) - after determining that the node is overloaded, these lines add an additional condition before the policy enters a data management state. 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.

B. Using ParSplice Policies for File Systems

Mantle was designed for file system metadata load balancing across a cluster of dedicated metadata servers, where spreading requests across servers improves performance. But another technique to reduce request load is caching. If clients and servers maintain a consistent cache, the client can do

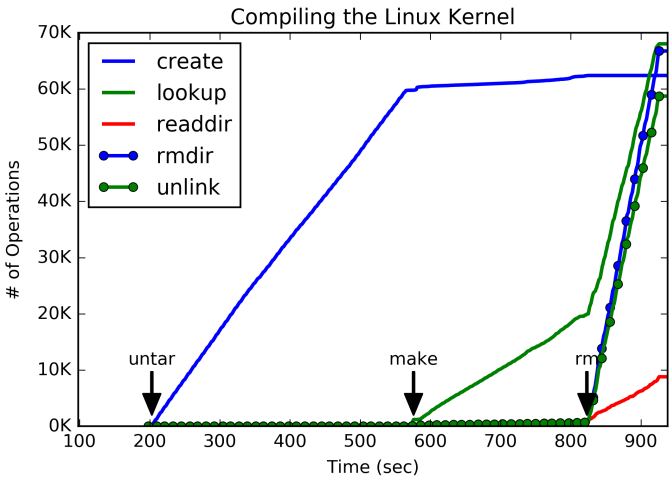

Fig. 14: The file system metadata requests for a compile job shows distinct workload phases, characterized by a dominant request type (*e.g.*, creates for "untar", lookups for "make", etc.). Using a ParSplice cache management policy for a workload like this in file systems could reduce memory pressure without sacrificing performance.

operation locally without contacting the metadata server. The caches in CephFS have the same performance and utilization trade-off as ParSplice, where large caches improve performance at the expense of a higher memory footprint. High memory utilization is problematic when a single metadata server maintains consistent caches with many clients . Using cache management policies from the previous two sections has two benefits for load balancing: (1) increases the capacity of a single metadata server, and (2) helps identify which parts of the cache to keep local to a server. For example, applying the regime detection algorithm from Section §V-B to a similar file system access pattern has the potential to identify when to migrate keys and how many keys to migrate.

To explore this example in more detail, consider the trace of metadata requests for compiling code in CephFS shown in Figure 14. That trace shows the number of file system metadata requests serviced by the metadata server when uncompressing (`untar`), compiling (`make`), and deleting (`rm`) the source code for the Linux kernel. If the system knows that the job phases would progress from many creates, to many lookups, to many deletes, then it could size its caches accordingly. For example, the file system could cache none of the metadatas from the `untar` phase and run regime detection during the `make` phase, resulting in the metadata server/clients only caching metadata that is repeatedly used. For the job in Figure 14, this would fill up the cache to only 20K inodes (the unit of metadata for file systems) instead of 60K, resulting in almost 40MB (since an inode is about 1KB[4]) of memory savings without sacrificing performance.

[4]http://docs.ceph.com/docs/master/dev/mds_internals/data-structures/

Ceph has many other data management techniques that would benefit from the caching policies developed in Sections §IV and V. Administrators can use the policies in ParSplice to automatically size and manage cache tiers [5], caching on object storage devices, or in the distributed block devices[6]. Integration with Mantle would be straightforward as it is merged into Ceph's mainline[7] and the three caching subsystems mentioned above already maintain keyspace access traces. We hypothesize that since this is all software defined caching, something more clever than LRU would improve cache utilization and without sacrificing too much performance.

C. Other Use Cases

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

[5]http://docs.ceph.com/docs/master/rados/operations/cache-tiering/
[6]http://docs.ceph.com/docs/master/rbd/rbd-config-ref/
[7]http://docs.ceph.com/docs/master/cephfs/mantle/

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

Data management encompasses a wide range of techniques that vary by domain and service. Yet, the techniques require policies that shape the decision making and finding the best policies is a difficult, multi-dimensional problem. We observe that many of the primitives and resulting strategies have enough in common that they can be expressed with similar semantics. We present a data management language and policy engine, called Mantle that is general enough to express complicated, dynamic policies for two different domains and services. Rather than attempting to construct a single, complex load balancing policy that works for a variety of scenarios, we instead use the Mantle framework to enable 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 Mantle is flexible enough to support domain-specific knowledge, which lays the groundwork for future work in adaptable policies and policy generation.

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.