Programmable Caches with a Data Management Language & 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 key access patterns and hot spots that offer significant opportunity for optimization. We use the data management language 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

Storage systems use software-based caches to improve performance but the policies that guide what data to evict and when to evict vary with the use case. For example, caching file system metadata on clients and servers reduces the number of remote procedure calls and improves the performance of create-heavy workloads common in HPC [1], [2]. But the policies for when to evict and what data to evict are specific to the application's behavior and the system's configuration (hardware, settings, etc.) so a new use case may prove to be a poor match for the selected caching policy [3]–[7]. In this paper, we evaluate a variety of caching policies using a data management language/policy engine and arrive at a customized policy that works well for our target application. This process of trying general policies and quickly iterating to a customized solution shows that our prototype can adapt to different workloads.

Our target application, ParSplice [8], is representative of applications that extensively use software-based caches. ParSplice is a molecular dynamics simulation that uses a hierarchy of caches and a single persistent key-value store 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. The fine-grained data annotation capabilities provided by key-value storage is a natural match for scientific simulations like ParSplice. But these simulations rely on a mesh-based decomposition of a physical region and result in millions or billions of mesh cells, where each cell contains materials, pressures, temperatures and other characteristics that are required to accurately simulate phenomena of interest. Unfortunately, simulations of this size saturate the capacity and bandwidth capabilities of a single node so we need more effective data management techniques, such as cache management or load balancing across a cluster.

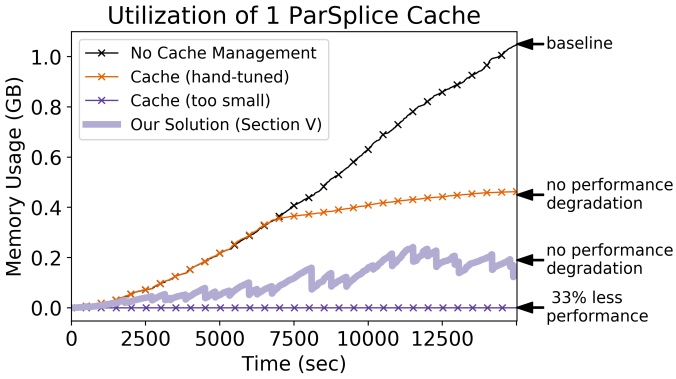


Fig. 1: Using our data management language and policy engine, we designed a dynamically sized caching policy (thick line) for ParSplice. Compared to existing configurations (thin lines with ×'s), our solution saves the most memory without sacrificing performance and works for a variety of inputs.

The biggest challenge for ParSplice is properly sizing the caches in the storage hierarchy. The memory usage for a single cache that stores molecule coordinates is shown in Figure 1, where the thin solid lines marked with ×'s are the existing configurations in ParSplice. The default configuration uses an unlimited sized cache, shown by the "No Cache Management" line, but using this much memory for one cache is unacceptable for HPC environments, where a common goal is to keep memory for such data structures below 3%[1]. ParSplice deploys a cache per 300 worker processes, so large simulations will not scale because they need multiple caches. Users can configure ParSplice to evict data when the cache reaches a threshold but this solution requires tuning and parameter sweeps; the "Cache (too small)" curve in Figure 1 shows how a poorly configured cache can save memory but at the expense of performance, which is shown by the annotation to the

[1]Anecdotally, we find this threshold works well for HPC applications. For reference, a 1GB cache for a distributed file system is too large in LANL deployments.

right. Even worse, this threshold changes with different initial configurations and cluster setups so tuning needs to be done for all system permutations. Our dynamically sized cache, shown by the thick line in Figure 1, detects key access patterns and re-sizes the cache accordingly. Without tuning or parameter sweeps, our solution saves more memory than a hand-tuned cache with a negligible performance degradation (within the baseline's standard deviation) and works for a variety of initial conditions.

To design more flexible cache management policies, like our solution in Figure 1, we use the data management language and policy engine from the Mantle paper [5]. While Mantle was designed for the narrow purpose of file system metadata load balancing, this paper provides evidence that the approach is more broad. Specifically, the collection of abstractions in Mantle provides a general control plane that improves the performance of 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, such as a file system or key-value store. Rather than co-designing a policy directly into the service, which requires domain-specific knowledge to find and change policies, we expose the policies in a general way so even developers unfamiliar with the domain can quickly deploy solutions. We show that our framework:

- decomposes cache management into independent policies that can be dynamically changed, making the problem more manageable and easier to reason about.
- can 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 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 cache management in ParSplice 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 [8] 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.

A. Background

As shown in Figure 2, the phases are:

1) a splicer tells workers to generate segments (short MD trajectory) for specific states

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

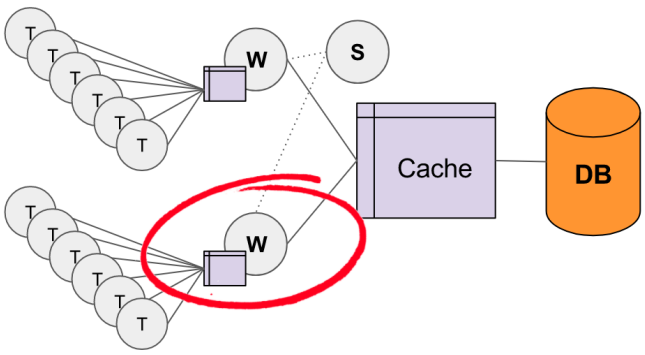


Fig. 2: The ParSplice architecture has a storage hierarchy of caches (boxes) and a dedicated cache process (large box) backed by a persistent database (DB). A splicer (S) tells workers (W) to generate segments and workers employ tasks (T) for more parallelization. We focus on the worker's cache (circled),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

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.

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.

B. Results and Conclusions

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 UC Santa Cruz. 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 hyperthreading 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 tasks(x axis). While the keyspace size and capacity is relatively modest the memory usage scales linearly with the number of tasks. This is a problem if we want to scale to Trinitite's 3000 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

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

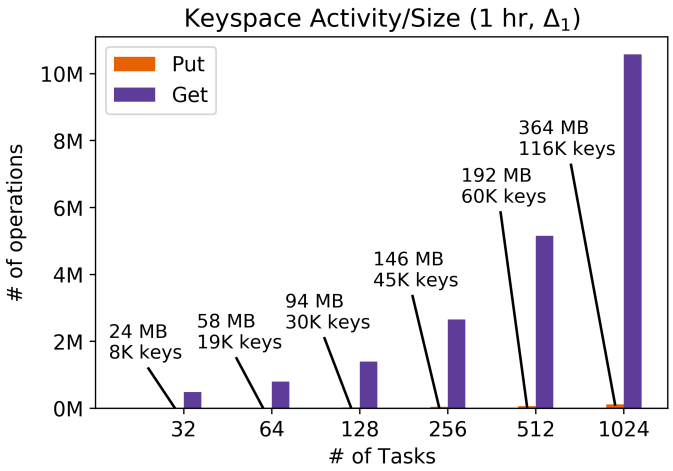


Fig. 3: The keyspace size is small but must satisfy many reads as workers calculate new segments. It is likely that we will need more than one node to manage segment coordinates when we scale the system or jobs up.

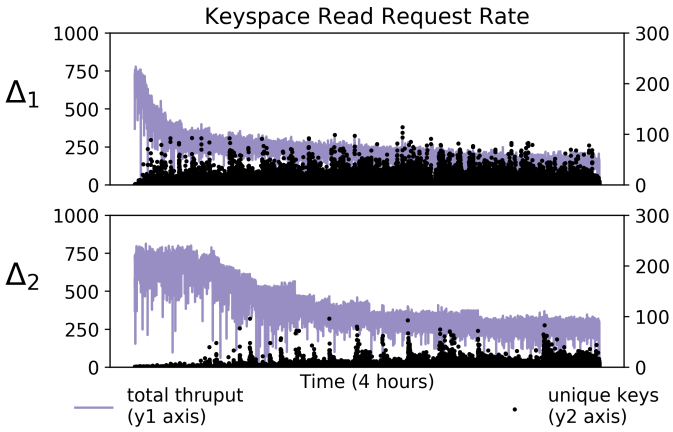


Fig. 4: The keyspace activity for ParSplice using two different growth rates. The line shows the rate that EOM minima values are retrieved from the key-value store ($y1$ axis) and the points along the bottom show the number of keys accessed in a 1 second sliding window ($y2$ axis).

at 20GB for the cache. This memory utilization easily eclipses the 3% memory usage per node threshold we set earlier, even without factoring in the usage from other workers.

An active but small keyspace: Figure 4 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). The Δ_1 growth rate adds atoms every 100K microseconds while the Δ_2 growth rate adds atoms every 1 million microseconds. So Δ_2 has a smaller growth

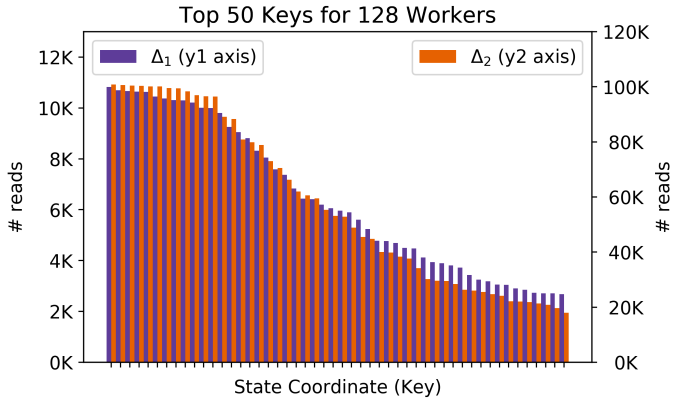


Fig. 5: The keyspace imbalance is due to workers generating deep trajectories and reading the same coordinates. Over time, the accesses get dispersed across different coordinates resulting in some keys being more popular than others.

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. This figure demonstrates that small changes to Δ can have a strong effect on the timing and frequency with which new EOM minima are discovered and referenced. This finding suggests that we need a flexible policy language and engine to explore these trade-offs.

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 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 tasks generate segments for the same state, which is computationally cheap (this motivates Section §IV).

Entropy increases over time: The reads per second in Figure 4 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 4 are shown in Figure 5, 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. The growth rate, temperature, and number of workers have a predictable effect on the structure of the keyspace. Figure 5 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 4 shows how entropy for different growth rates has temporal locality, as the reads per second for Δ_2 look 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 key access patterns and adapt the storage system accordingly (this motivates Section §V).

III. Methodology

To explore software-defined cache management, we use the data management language and policy engine presented in [5]. The prototype in that paper was called Mantle and it was originally built on 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" controls how distributed or concentrated the data should be; and "how much" controls the amount of data that should be sent. There is also a "load" policy that lets administrators specify how to collapse many metrics into a single load metric (*e.g.*, $2 \times$ `cpu` $+ 3 \times$ `memory usage`).

The succinctness of the API lets users inject multiple, possibly 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 will 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.

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 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++.

1) Environment of Metrics: services 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.

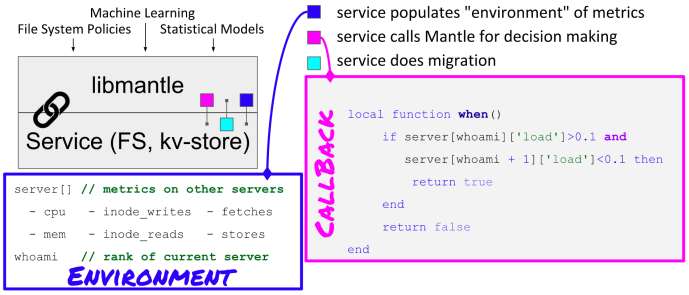


Fig. 6: Extracting Mantle as library.

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.

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.

2) Policies Written as Callbacks: the "callback" box in Figure 6 shows an example policy for "when()", 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.

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-B 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, key` pairs. In Section §V, we explore a key access pattern 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 the worker using the keyspace analysis in Section §II-B. The evaluation metric is the accuracy and runtime of each strategy; the policy should be accurate enough so as to sacrifice negligible performance and fast enough to run as often as we want to detect key access patterns. 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.

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 middle graph shares the y axis and shows the trade-off of using a basic LRU-style cache for different cache sizes, where the penalty for a cache miss is retrieving the data from the persistent

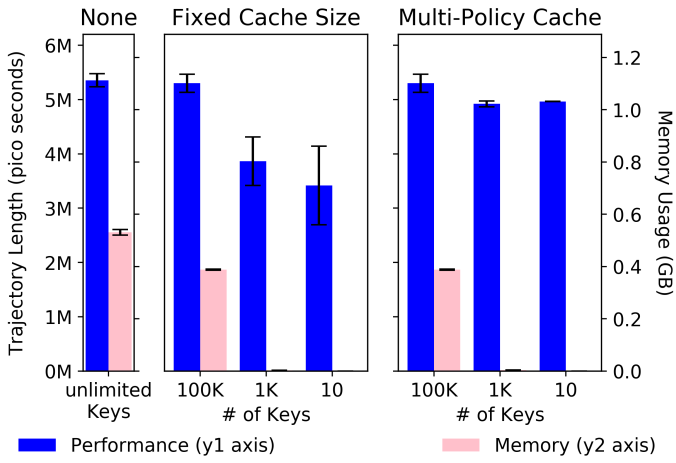


Fig. 7: The performance and resource utilization trade-off for different cache management policies. "None" is ParSplice unmodified, "Fixed Sized Cache" limits the size of the cache by evicting the least recently used items, and "Multi-Policy Cache" switches to a fixed sized cache after absorbing the initial burstiness of the workload. The size of the fixed sized cache for each experiment is on the x axis.

database. We evict keys (if necessary) at every operation instead of when segments complete because the cache fills up too quickly otherwise. This is implemented using a "when" policy of `server[whoami]['cachesize']>n` and a "how much" policy of `servers[whoami]['cachesize']-n`. 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.

But the top graph in Figure 4 suggests that a smaller cache size should suffice, as only 100 keys seem to be active at any one time. It turns out that the unique keys plotted in Figure 4 are per second and are not representative of the actual active keyspace; the number of active keys is larger than 100, as some keys may be accessed at time t_0, not in t_1, and then again in t_2. Because the cache is too small, reads and writes hit the persistent database, where the excessive traffic triggers a LevelDB compaction and reads block. To avoid overloading the storage hierarchy with requests, we design a multi-part policy that switches between:

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

The key observation is that 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. We program Mantle to 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. The actual policy is shown and described in more detail in the next section in Figure 10a. The plot on the right side of Figure 7 shows the performance/utilization trade-off of the multi-part policy, where the cache sizes for the n key limit policy are along the x axis. They show better performance than the single n key policies. The performance and memory utilization for a 100K key cache size is the same as the 100K bar in the "Multi-Part Policy" graph in Figure 7 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.

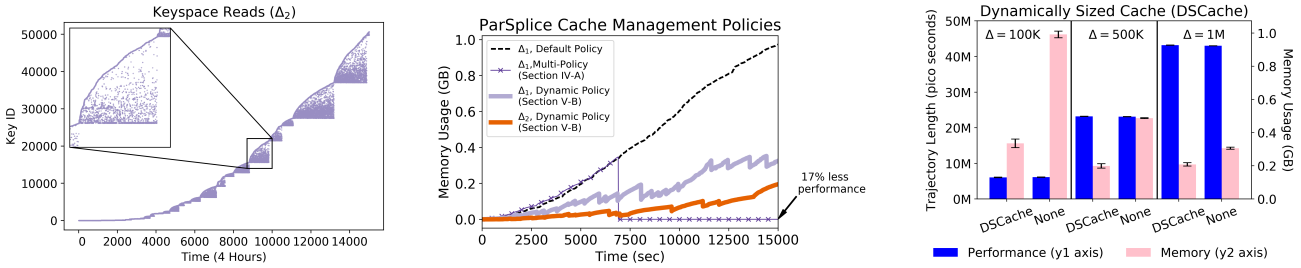
Caveats: The results in Figure 7 are slightly deceiving for two reasons: (1) segments take longer to generate later in the run and (2) the memory footprint is the value at the end of 2.5 hours. For (1), the trajectory length vs. wall-clock time curves down over time; as the nanoparticle grows it takes 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 rises until we reach 100K key switch threshold at 0.4GB, as shown by the "Δ_1, Multi-Policy" curve in Figure 8b but in Figure 7 we plot the final value. As a result, we can only conclude that this policy only works well for this 2.5 hour run.

Despite these caveats, the result is still valid: we found a multi-policy cache management strategy 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 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. 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.

Figure 8a shows which keys (y axis) are accessed by tasks over time (x axis). The groups of accesses to a subset of keys occurs because molecules are stuck in deep trajectories. Recall that the cache 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. In Figure 8a, superbasins are never revisited 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:



(a) Key activity for a 4 hour run shows groups of accesses to the same subset of keys. Detecting these access patterns leads to a more accurate cache management strategy, which is discussed in Section §V-B.

(b) Memory utilization for "No Cache Management" (unlimited cache growth), "Multi-Policy" (absorbs initial burstiness of workload) , and "Dynamic Policy" (sizes cache according to key access patterns).

(c) Performance/Utilization for dynamically sized cache (DSCache) policy. With negligible performance degradation, DSCache adjusts to different Δs over $2\times$ memory in the best case.

Fig. 8: Different cache management policies tested over the Mantle policy engine.

Detecting these superbasins can lead to more effective cache management strategies because the height of the groups of key accesses is "how much" of the cache to evict and the width of the groups of key accesses is "when" to evict values from the cache. The zoomed portion of Figure 8a 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.

- 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

To detect the access patterns in Figure 8a, we try a variety of techniques using Mantle. Unfortunately, we found that the following techniques proliferate more parameters that need to be tuned per hardware/software configuration. Furthermore, we find that many of the metrics do not signal a new set of key accesses. Below, we indicate with quotes which parameters we need to add for each technique with and the value we find to work best, via tuning and parameter sweeps, for one set of initial conditions.

- Statistics: decay on each key counts down until 0; 0-valued keys are evicted. "history-of-key-accesses", set to 10 seconds, to evict keys.
- Calculus: use derivative to strip away magnitudes; use large positive slopes followed by large negative slope as signal for new set of key accesses. "Zero-crossing", set to 40 seconds, for distance between small/large spikes to avoid false positives; 'window size", set to 200 seconds, for the size of the moving average
- K-Means Clustering fails because "K" is not known *a-priori* and groups of key accesses are different size. "K", set to 4, for the number of clusters in the data using the sum of the distances to the centroid
- DBScan: finds clusters using density as a metric. "Eps", set to 20, for max distance between 2 samples in same neighborhood; "Min", set to 5, for the number of samples per core
- Anomaly Detection: size of the image is too big and bottom edges are not thick enough

B. Dynamically Sized Cache: Access Pattern Detection

After trying these techniques we found that the basic algorithm in Figure 9 works best. The algorithm detects groups of key access patterns, which we call "fans", by iterating backwards through the key access trace, finding the lowest key ID, and comparing against the lowest key ID we have seen so far (Line 7). We also maintain the top and bottom of each group of key accesses (Line 13) so we can tell the "how much" policy the number of keys to evict (Line 19). The algorithm iterates backwards over the key access trace because a change in the minimum value signals a new group of key accesses. No signal exists iterating left to right, as the maximum value always increases and the minimum values at the bottom of each group of key accesses are sparse. For example, the maximum distance between values along the bottom edge of the zoomed group of key accesses in Figure 8a is 125 seconds, while the maximum distance between minimum values for the group of key accesses before is 0 seconds. As a result of this sparseness, iterating left to right requires a "window size" parameter to determine when we think a minimum value will not show up again. We exchange runtime, in the form of an O(n) algorithm where n is the number events, for simplicity because this approach avoids adding new thresholds for key access pattern detection (*e.g.*, space between key accesses, space between key IDs, and window size of consecutive key accesses).

The performance and memory utilization is shown by the "DSCache" bars in Figure 8c. Without sacrificing performance (trajectory length), the dynamically sized cache policy uses between 32%-66% less memory than the default ParSplice

configuration (no cache management) for the 3 initial conditions we test. The memory usage is shown by the "Dynamic Policy" curves in Figure 8b, where the behavior resembles the key access patterns in Figure 8a[4]. We also show a Δ_2 growth rate to demonstrate the dynamic policy's ability to adjust to a different set of initial conditions.

```
1  d = timeseries()           -- provided by mantle
2  ts, id = d:get(d:size())   -- provided by mantle
3  fan  = {start=nil, finish=ts, top=0, bottom=id}
4  fans = {}
5  for i=d:size(),1,-1 do     -- detect all fans
6    ts, id = d:get(i)
7    if id < fan['bottom'] then
8      fan['start']  = ts
9      fans[#fans+1] = fan
10     fan = {start=nil, finish=ts, top=0, bottom=id}
11   end
12
13   if id > fan['top'] then fan['top'] = id end
14 end
15 fan['start'] = 0
16 fans[#fans+1] = fan
17
18 if #fans < 2 then return false else
19   WRstate(fans[#fans-1]['top']-fans[1]['bottom'])
20   return true
21 end
```

Fig. 9: The dynamically sized cache policy, whose performance is shown in Figure 1, iterates backwards over timestamp-key pairs and detects when key accesses move on to a new subset of keys. We refer to groups of key accesses as "fans".

VI. Towards 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 [5] with the ones we designed and evaluated above 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.

A. Using Load Balancing Policies for Cache Management

From a high-level the cache management policy we designed in Figure 10a 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 10b, which was presented in [5]. That policy migrates file system metadata if the load is higher than the average load in the cluster *and* the current server has been overloaded for more than two iterations. The two policies have the following in common:

[4]The memory usage is not *exactly* the same because these are two different runs; Figure 8a has key activity tracing turned on, which reduces performance.

Condition for "Overloaded" (Fig. 10a: Line 2; Fig. 10b: 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. 10a: Lines 4,6; Fig 10b: Lines 3,4,9) - 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.

Multi-Policy Strategy (Fig. 10a: Line 6; Fig. 10b: 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.

These similarities among effective policies for two very different domains suggests that the heuristics and techniques in other load balancers can be used for cache management. This generalization may reduce the work that needs to be done for cache management as ideas may have already been explored and could work "out-of-the-box".

B. Using Cache Management Policies for Load Balancing

The abstractions in Mantle were 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 operations 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 key access pattern 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 11. 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,

```
function when()
  if server[whoami]['cachesize'] > n then
    if server[whoami]['cachesize'] > 100K then
      WRstate(1)
    end
    if RDstate() == 1 then
      return true
    end
  end
  return false
end
```

(a) ParSplice cache management policy that absorbs the burstiness of the workload before switching to a constrained cache. The performance/utilization for different `n` is in Figure 7.

```
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
```

(b) CephFS file system metadata load balancer, designed in 2004 in [6], reimplemented in Lua in [5]. This policy has many similarities to the ParSplice cache management policy.

Fig. 10: ParSplice's cache management policy has the same components as CephFS's load balancing policy.

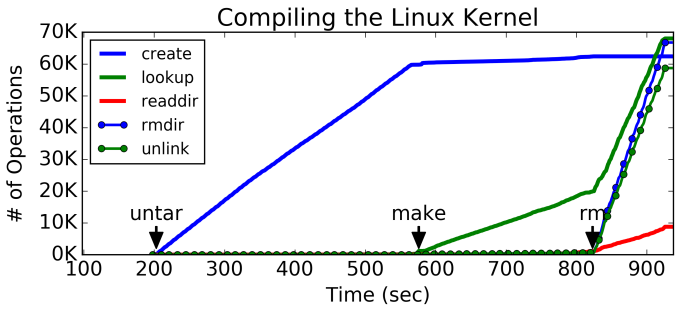


Fig. 11: File system metadata requests for a compile job show workload phases characterized by a dominant request type (*e.g.*, creates for "untar"). Using ParSplice cache management policies for these types of workloads would reduce memory pressure without sacrificing performance.

to many deletes, then it could size its caches accordingly. For example, the file system could cache none of the metadata from the `untar` phase and run key access pattern detection during the `make` phase, resulting in the metadata server/clients only caching metadata that is repeatedly used. For the job in Figure 11, this would fill up the cache to only 20K inodes (the unit of metadata for file systems) instead of 60K, resulting in almost 40MB of memory savings (since an inode is about 1KB[5]) without sacrificing performance.

Storage systems have many other data management techniques that would benefit from the caching policies developed in Sections §IV and V. For example, Ceph administrators can use the policies in ParSplice to automatically size and manage cache tiers[6], caching on object storage devices, or in the distributed block devices[7]. Integration with Mantle would be straightforward as it is merged into Ceph's mainline[8] and the three caching subsystems mentioned above already maintain key access traces. We hypothesize that since this is all software defined caching, something more clever than LRU would improve cache utilization 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. Our plan is to use MDHIM [11] as our back-end key-value store because it was designed for HPC and has the proper mechanisms for migration already implemented.

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 [1] 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) [6], [7] 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+ [12] 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 [13], [14], big data systems systems [15], and databases [16]. Like our work, these systems focus on the physical design of the storage (*e.g.* cache size) but since we focused on

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

a relatively small set of parameters (cache size, migration thresholds), we did not need anything as sophisticated as the genetic algorithm used in [13]. We cannot drop these techniques into ParSplice because the magnitude and speed of the workload hotspots/flash crowds makes existing approaches less applicable.

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. Ultimately, we hope that this automation enables control of policies by machines instead of administrators.

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 iterate to a custom solution for our target application that uses workload access patterns to size its caches. Without tuning or parameter sweeps, our solution saves memory without sacrificing performance for a variety of initial conditions, including the scale, duration, configuration, and hardware of the simulation. More importantly, rather than attempting to construct a single, complex 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. We also observe that many of the primitives and strategies have enough in common with data management in file systems that they both can be expressed with similar semantics and a general policy engine. Our prototype lays the groundwork for future work in adaptable policies and policy generation, which ushers in an era where policies are controlled by machines instead of administrators.

References

[1] 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 Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '14, 2014.

[2] 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.

[3] L. Xiao, K. Ren, Q. Zheng, and G. A. Gibson, "ShardFS vs. IndexFS: Replication vs. Caching Strategies for Distributed Metadata Management in Cloud Storage Systems," in *Proceedings of the Symposium on Cloud Computing*, ser. SoCC '15.

[4] 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.

[5] 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 Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '15.

[6] 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 Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '04.

[7] 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 Symposium on Operating Systems Design and Implementation*, ser. OSDI '06.

[8] 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*.

[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] 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.

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

[13] 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.

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

[15] 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.*

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