Programmable Caches with a Data Management Language and Policy Engine

Michael A. Sevilla, Carlos Maltzahn, Peter Alvaro, Reza Nasirigerdeh,
*Bradley W. Settlemyer, *Danny Perez, *David Rich, *Galen M. Shipman
University of California, Santa Cruz, *Los Alamos National Laboratory
{msevilla, carlosm, palvaro, rnasirig}@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 application and storage system. Our exploration of this space results in a dynamically sized cache policy that does not sacrifice any performance while using 32-66% less memory than the default ParSplice configuration.**

***Keywords*-high performance computing; cache storage; file systems; system software**

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]–[3]. But the policies for what data to evict and when to evict are specific to the application's behavior and the hardware configuration so a new workload may prove to be a poor match for the selected caching policy [3]–[7]. We evaluate a variety of caching policies using our data management language/policy engine and arrive at a customized policy that works well for our example application, ParSplice [8].

ParSplice molecular dynamics simulations are representative of an important class of HPC applications with similar working set behaviors that extensively use software-based caches. ParSplice uses a hierarchy of caches and a single persistent key-value store to store the atomic coordinates corresponding to local energy minima (referred to simply as minima) encountered by a large number of independent dynamical trajectories. This workload is also pervasive across simulations that (1) rely on a mesh-based decomposition of a physical region and (2) 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. The fine-grained data annotation capabilities provided by key-value storage is a natural match for these types of scientific simulations. Unfortunately, simulations of this size saturate the capacity and bandwidth capabilities of a single node so we need more effective data management techniques.

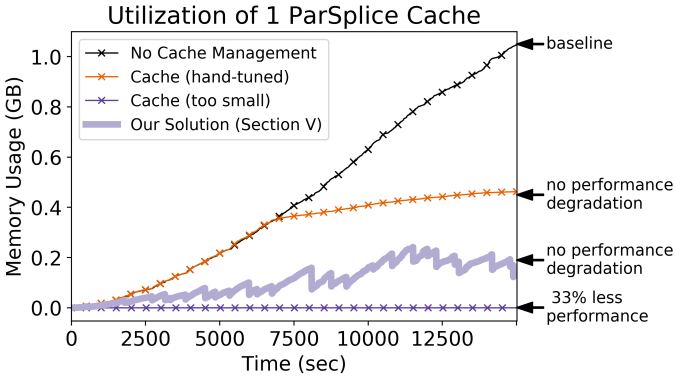


Fig. 1: Using our data management language and policy engine, we design a dynamically sized caching policy (thick line) for ParSplice. Compared to existing configurations (thin lines with ×'s), our solution saves the most memory without sacrificing performance and works for a variety of inputs. Solutions labeled "no performance degradation" are in comparison to the baseline of using a cache of unlimited size ("No Cache Management").

A challenge for ParSplice is properly sizing the caches in the storage hierarchy. The memory usage for a single cache that stores atomic 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]. Furthermore, ParSplice deploys a cache per 300 worker processes, so large simulations need more caches and will use even more memory. 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

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

a poorly configured cache can save memory but at the cost of performance, which is shown by the text annotation to the 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 without any performance degradation, works for a variety of initial conditions, and could generalize to similar applications. Triggering key eviction at a certain memory pressure (*e.g.* 3%) requires *a priori* system knowledge while our approach saves the most memory because we model the behavior of keyspace accesses.

In this paper we are presenting the successful use of our data management language and Mantle policy engine to control the behavior of ParSplice's caches. Mantle provides a control plane that injects policies into a running storage system, such as a file system or key-value store. While Mantle was originally designed for file system metadata load balancing [6], we find that it works surprisingly well for specifying cache management policies without requiring users to possess extensive knowledge about the internals of storage systems. 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. We analyze why our early policies fail for our HPC hardware and software and show how we iterated to the final solution. More importantly, in Section §VI, we conclude that the collection of policies we design for cache management in ParSplice are very similar to the policies used to load balance metadata in the Ceph file system (CephFS [7]) suggesting that there is potential for automatically adapting and generating policies dynamically. This paper does not thoroughly study cache management policies for file systems, but our findings for this specific application and related storage systems work suggests that this approach is a strong avenue for future work.

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.

[2]https://www.exascaleproject.org/

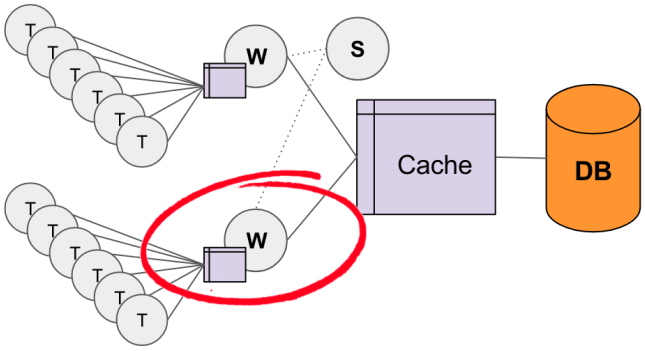


Fig. 2: The ParSplice architecture has a storage hierarchy of caches (boxes) and a dedicated cache process (large box) backed by a persistent database (DB). A splicer (S) tells workers (W) to generate segments and workers employ tasks (T) for more parallelization. We focus on the worker's cache (circled),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

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) 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 caches that sit in front of a single persistent database. Values corresponding to new keys are written to each tier and reads traverse up the hierarchy until they find the data.

We use ParSplice to simulate the evolution of metallic nanoparticles that grow from the vapor phase. 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

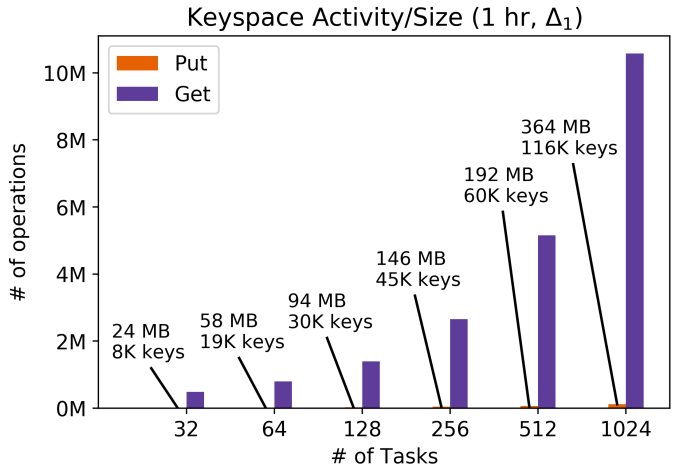


Fig. 3: The keyspace is small but must satisfy many reads as workers calculate segments. Memory usage scales linearly, so it is likely that we will need more than one node to manage segment coordinates when we scale the system or jobs up. Δ_1 is the growth rate, an application specific parameter described in Section §II-C3.

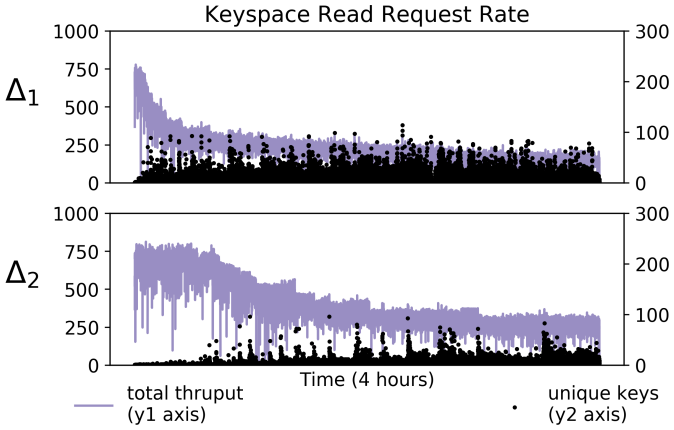


Fig. 4: Key activity for ParSplice starts with many reads to a small set of keys and progresses to less reads to a larger set of keys. The line shows the rate that EOM minima values are retrieved from the key-value store ($y1$ axis) and the points along the bottom show the number of unique keys accessed in a 1 second sliding window ($y2$ axis). Despite having different growth rates (Δ), the structure and behavior of the key activities are similar.

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 lead to the melting of the nanoparticle and hence to a complete change in the physics of the system.

B. Experimental Setup

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.

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. 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.

C. Results and Observations

Our analysis shows that ParSplice accesses keys in a structured and predictable way. The following 4 observations shape the policies we design later in the paper.

1) 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 modest the memory usage scales linearly with the number of tasks, which 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 at 8GB for one 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.

2) An active but small keyspace: The bars in Figure 3 show $50-100\times$ as many reads (`get()`) as writes (`put()`). Tasks read the same key for extended periods because the trajectory gets stuck in so-called superbasins composed of tightly connected sets of states. 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.

3) Initial conditions influence key activity: 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 access patterns for different growth rates have 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. The Δ_1 growth rate adds atoms every 100K picoseconds while the Δ_2 growth rate adds atoms every 1 million picoseconds. So Δ_2 has a smaller

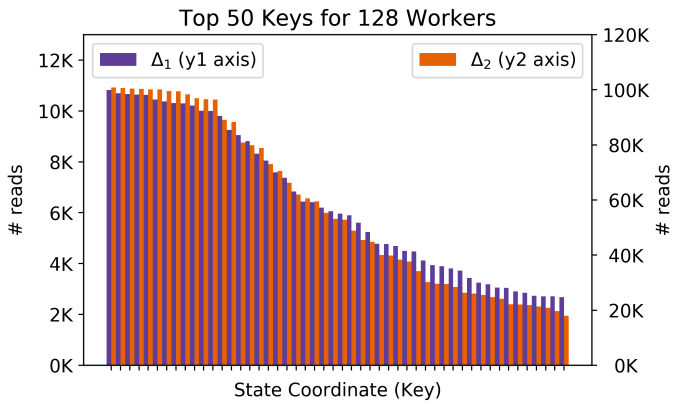


Fig. 5: Over time, tasks start to access a larger set of keys resulting in some keys being more popular than others. Despite different growth rates (Δ), the spatial locality of key accesses is similar between the two runs. (*e.g.*, some keys are still read 5 times as many times others).

growth rate resulting in hotter keys and a smaller keyspace. Values smaller than Δ_2's growth rate or a temperature of 400 degrees result in very little database activity because state transitions take very long. Similarly, growth rate values larger than Δ_1 lead to far out-of-equilibrium growth, while temperatures in excess of 800 degrees result in the melting of the nanoparticle, which is not relevant in this application.

This figure demonstrates that small changes to Δ can have a strong effect on the timing and frequency with which new minima are discovered and referenced. Trends also exist for temperature and number of workers but are omitted here for space. This finding suggests that we need a flexible policy language and engine to explore these trade-offs.

4) 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 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). 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. Figure 5 also 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).

III. Methodology

To explore software-defined cache management, we use the data management language and policy engine presented in [6]. The prototype in that paper, Mantle, was built on CephFS and lets administrators control file system metadata load balancing policies. We now refer to Mantle as a policy engine that

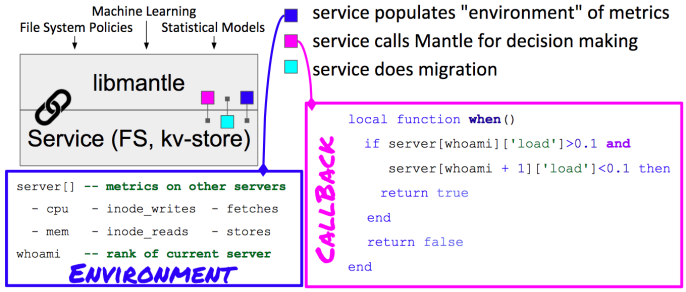


Fig. 6: Extracting Mantle as library.

Metrics	Data Structure	Description
Cluster	{server → {metric → val}}	resource util. for servers
Time Series	[(ts, val), ..., (ts, val)]	accesses by timestamp (ts)
	Storage System	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 storage system to the policy engine using Mantle.

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". The "when" policy 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.*, $2 \times \texttt{cpu} + 3 \times \texttt{memory usage}$).

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 to 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 storage system service. Administrators write policies in Lua from whatever domain they choose (*e.g.*, statistics, machine learning, storage system) and the policies are embedded into the runtime by Mantle. 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++. When the storage system makes decisions it executes the administrator-defined policies for when/where/how much and returns a decision. To do this, the storage system 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) Environment of Metrics: storage systems expose **cluster** metrics for describing resource utilizations 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 storage system passes a dictionary to Mantle. 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. 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 storage system; Mantle in CephFS leverages metrics from other servers collected using CephFS's heartbeats. For example, a policy written for an MPI-based storage system can access the CPU utilization of the first rank in a communication group using: `servers[0]['cpu']`.

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

```
d = timeseries()        -- d(ata) from storage system
for i=1,d:size() do     -- iterate over timeseries
  ts, value = d:get(i) -- index into timeseries
  if value == 'mydirectory' then
    count = count + 1
  end
end
```

2) Policies Written as Callbacks: the "callback" box in Figure 6 shows an example policy for "when()", where the current server migrates work if it is has load and if its neighbor does not have load; `whoami` is the current server, its neighbor is `whoami+1`, and the load threshold is 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 §IV 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 into all caches in the system and 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. We only do cache management once every second to avoid maintaining the cache for every request. We expected to have to increase this polling interval to accommodate more complex policies but even our most complicated policy in Section §V had a negligible effect on performance when executed every second (within the standard deviation for multiple runs when compared against a policy that returns immediately). This may be because the worker is not overloaded and the bottleneck is somewhere else in the system. 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 Storage System Architecture Knowledge

Using the Mantle policy engine, we test a variety of cache management algorithms on the worker using the keyspace analysis in Section §II-C. 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. The policy should avoid reducing the trajectory length and be fast enough to run as often as we want to detect key access patterns. First we size the cache according to our system specific knowledge, *i.e.* the hardware and software of the storage hierarchy.

We implement a basic LRU cache using a "when" policy of: `server[whoami]['cachesize']>n` and a "how much" policy of `servers[whoami]['cachesize']-n`. 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 ($y1$ axis) and utilization is measured with memory footprint of just the cache ($y2$ axis). The middle graph labeled "Fixed Cache Size" shares the y axes and shows the trade-off of using a basic LRU-style cache of different sizes, where the penalty for a cache miss is retrieving the data from the persistent database. 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 fall through to the rest of the storage hierarchy and the excessive traffic

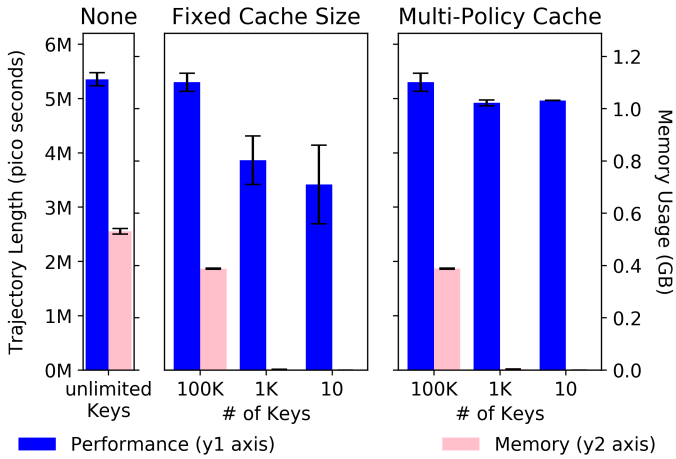


Fig. 7: Policy performance/utilization shows the trade-offs of different sized caches (x axis). "None" is ParSplice unmodified, "Fixed Sized Cache" evicts keys using LRU, and "Multi-Policy Cache" switches to fixed sized cache after absorbing the workload's initial burstiness. This parameter sweep identifies the "Multi-Policy Cache" of 1K keys as the best solution but this only works for this system setup and initial configurations.

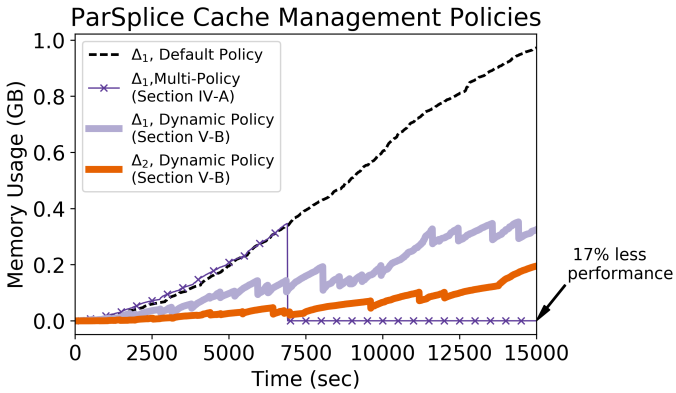


Fig. 8: Memory utilization for "No Cache Management" (unlimited cache growth), "Multi-Policy" (absorbs initial burstiness of workload), and "Dynamic Policy" (sizes cache according to key access patterns). The dynamic policies saves the most memory without sacrificing performance.

triggers a LevelDB compaction on the persistent database. To avoid these compactions, which temporarily block operations, we design a multi-policy cache 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 Section §VI in Figure 11a. The plot on the right side of Figure 7 shows the performance/utilization trade-off of the multi-policy cache, where the cache sizes for the n key limit policy are along the x axis. The performance and memory utilization for a 100K key cache size is the same as the 100K bar in the "Fixed Cache Size" 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 policy switches to the 10 key limit policy, which achieves 94% of the performance while only using 40KB of memory.

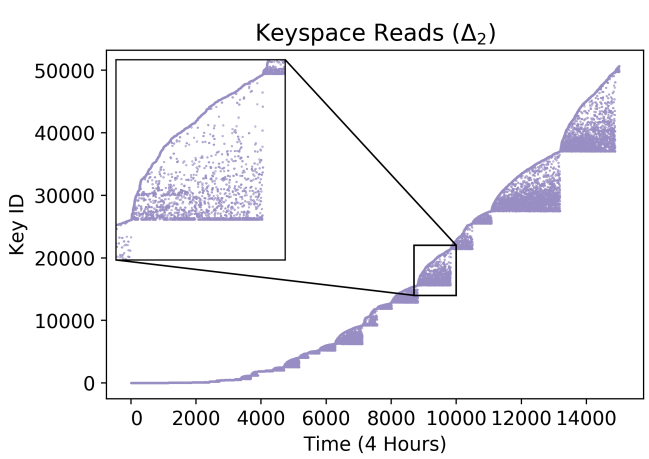
Caveats: The results in Figure 7 are slightly deceiving for two reasons: (1) segments take longer to generate later in the run and (2) the memory footprint is the value at the end of 2.5 hours. For (1), the trajectory length vs. wall-clock time curves down over time; as the nanoparticle grows it takes 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 it reaches the 100K key switch threshold at 0.4GB and then reduces to the final value after switching policies. The memory usage over time for this policy is shown by the "Δ_1, Multi-Policy" curve in Figure 8 but in Figure 7 we plot the final value. 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. To improve the policy even more, we need a way to identify what thresholds to use for different system setups (*e.g.*, different ParSplice parameters, number of worker tasks, and job lengths).

V. Cache Management Using Application-Specific Knowledge

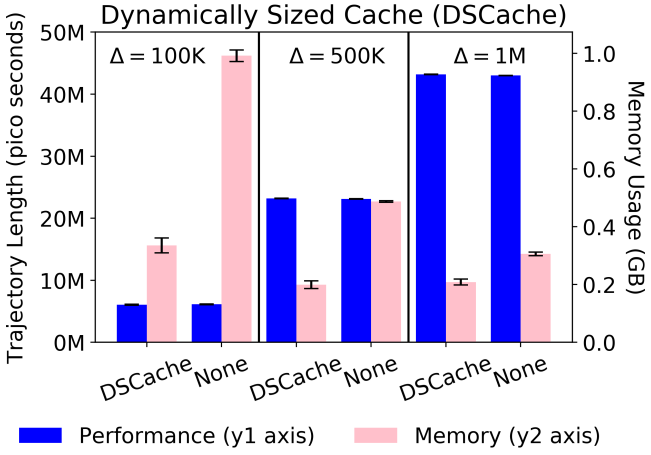
Feeding application-specific knowledge about ParSplice into a policy leads to a more accurate cache management strategy. The goal of the following section is not to find an optimal solution, as this can be done with parameter sweeps for thresholds; rather, we try to find techniques that work for a range of inputs and system setups.

Figure 9a shows which keys (y axis) are accessed by tasks over time (x axis). The groups of accesses to a subset of keys occurs because the system is stuck in so-called superbasins, *i.e.* coarse regions of space from which it is difficult to escape, but within which it is easy to move. Systems stuck in superbasins will explore the same set of minima for a long time, leading to the same keys being accessed repeatedly. In Figure 9a, superbasins are typically not re-visited because the simulation only adds atoms; we can never revisit a superbasin that contains less atoms than the simulation currently contains. This is why keys are never re-accessed after a given amount of time.

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



(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 and the results are in Figure 9b.



(b) The performance/utilization for the dynamically sized cache (DSCache) policy. With negligible performance degradation, DSCache adjusts to different initial configurations (Δs) and saves 3× as much memory in the best case.

Fig. 9: Different cache management policies tested over the Mantle policy engine.

```
d = timeseries()
ts, id = d:get(d:size())
fan  = {start=nil, finish=ts, top=0, bot=id}
fans = {}
for i=d:size(),1,-1 do      -- iterate backwards
  ts, id = d:get(i)
  if id < fan['bot'] then  -- found a new fan!
    fan['start']  = ts
    fans[#fans+1] = fan
    fan = {start=nil, finish=ts, top=0, bot=id}
  end

  if id > fan['top'] then  -- track top of fan
    fan['top'] = id
  end
end
fan['start'] = 0
fans[#fans+1] = fan

if #fans < 2 then -- do not evict current fan
  return false
else
  WRstate(fans[#fans-1]['top']-fans[1]['bot'])
  return true
end
```

Fig. 10: The dynamically sized cache policy iterates backwards over timestamp-key pairs and detects when accesses move on to a new subset of keys (*i.e.* "fans"). The performance and total memory usage is in Figure 9b and the memory usage over time is in Figure 8.

the cache. The zoomed portion of Figure 9a 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. Despite these patterns, the following characteristics of superbasins make them hard to detect:

- 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

A. Failed Strategies

To detect the access patterns in Figure 9a, 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, 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 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 samples per core.
- Edge 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 O(n) algorithm in Figure 10 works best. The algorithm detects

groups of key accesses, 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 23). The algorithm is O(n), where n is the number events, but the benefit is that the 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 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 9a 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.

The performance and memory utilization is shown by the "DSCache" bars in Figure 9b. 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 over time is shown by the "Dynamic Policy" curves in Figure 8, where the behavior resembles the key access patterns in Figure 9a[3]. We also show a Δ_2 growth rate to demonstrate the dynamic policy's ability to adjust to a different set of initial conditions.

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 application and storage system. In this section, we compare and contrast the policies examined for file system metadata load balancing in [6] with the ones we designed and evaluated above for cache management in ParSplice.

A. Using Load Balancing Policies for Cache Management

From a high-level the cache management policy we designed in Figure 11a 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 CephFS metadata load balancing policy in Figure 11b, which was presented in [6]. 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:

Condition for "Overloaded" (Fig. 11a: Line 2; Fig. 11b: 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. 11a: Lines 4,6; Fig 11b: 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 9) if the server is no longer overloaded.

Multi-Policy Strategy (Fig. 11a: Line 6; Fig. 11b: 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 suggest that the heuristics and techniques in other load balancers can be used for cache management. The result supports the notion that concepts and problems that architects grapple with are transcendent across domains and the solutions they design can be re-used in different code bases.

B. Using Cache Management Policies for Load Balancing

The cache management policies we developed earlier can be used by load balancing policies to effectively spread load across a cluster. For example, distributed file systems that load balance file system metadata across a dedicated metadata cluster could use the caching policies to determine what metadata to move and when to move it. To demonstrate this idea, we analyze a 3-day Lustre file system metadata trace, collected at LANL. The trace is anonymized so all file names are replaced with a unique identifier and we do not know which applications are running. We visualize a 1 hour window of the trace in Figure 12, where the dots are the file system metadata reads in a 1 hour window. The x axis is time and the y axis is the file ID, listed in the order that file IDs appear in the trace. The groups of accesses look similar to the ParSplice key accesses in Figure 9a.

Although other access pattern detection algorithms are possible, we use the one designed for cache management in Section §V-B with slight modifications based on our knowledge of file systems[4]. The vertical lines in Figure 12 are the groups of accesses identified by the algorithm; it successfully detects the largest group of key accesses that starts at time 1000 seconds and ends at time 2200 seconds. File systems

[3]The memory usage is not *exactly* the same because these are two different runs; Figure 9a has key activity tracing turned on, which reduces performance.

[4]We filtered out requests for key IDs less than 2000, as these are most likely path traversal requests to higher parts of the namespace.

```
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 [3], reimplemented in Lua in [6]. This policy has many similarities to the ParSplice cache management policy.

Fig. 11: ParSplice's cache management policy has the same components as CephFS's load balancing policy.

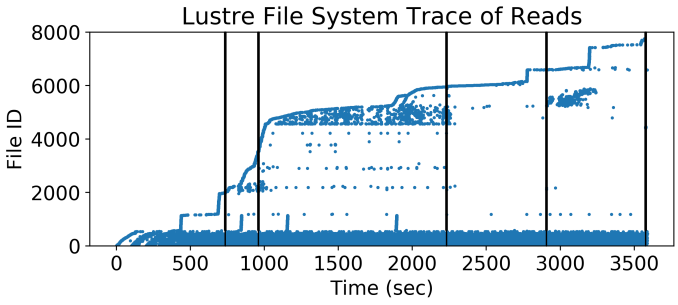


Fig. 12: File system metadata reads for a Lustre trace collected at LANL. The vertical lines are the results of running the same pattern detection algorithm used for cache management in Section §V over this Lustre trace. A file system that load balances metadata across a cluster of servers could use the same pattern detection to make migration decisions, such as avoiding migration when the workload is accessing the same subset of keys or keeping groups of accesses local to a server.

that load balance file system metadata across a cluster would want to keep metadata in that group of key accesses on the same server for locality and would want to avoid migrating metadata to a different server until the group of key accesses completes.

Before we showed how policies designed for load balancing heavily influence our cache management in a different application and storage system. But in this section we show how an *unmodified* cache management policy can be used in a load balancing strategy. This generalization may reduce the work that needs to be done for load balancing as ideas may have already been explored in other domains and could work "out-of-the-box".

C. Other Use Cases

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[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 key access traces.

More generally, the similarities between load balancing and cache management 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: when to move clients, where to move clients, how much of the reservation to move. We could use Mantle to implement something like the reservation algorithms based on utilization and period in Fahrrad [9] to achieve better guarantees without sacrificing performance.
- Scheduling: when to yield computation cycles to another process, how much of a resource to allocate. We could use Mantle to implement the fairness/priority models used in the Mesos [10] "how many" policies.
- Batching: how many operations to group together, when to send large batches of updates. We could use Mantle to implement pathname leases from IndexFS [1] or the capabilities from CephFS[8].
- Prefetching: how much to prefetch, how to select data. We could use Mantle to implement forward/backward/stride detection algorithms for prefetching in RAID arrays or something more complicated, like the time series algorithms for adaptive I/O prefetching from [11].

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 [12]. 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

[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/
[8]http://docs.ceph.com/docs/master/cephfs/capabilities/

Malacology project [13] which seeks to provide fundamental services from within the storage system (*e.g.*, consensus) to the application. Our plan is to use MDHIM [14] 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 [4] takes the replication approach to the extreme by copying all directory state to all nodes. CephFS [3], [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+ [15] 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 [16], big data systems systems [17], and databases [18]. 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 [16].

VIII. Conclusion

Data management encompasses a wide range of techniques that vary by application and storage system. 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.

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 applications and storage systems. The value of such a collection eases the burden of policy development and paves the way for automated solutions such as (1) adaptable policies that 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 by examining the collection of existing policies. Ultimately, we hope that this automation enables control of policies by machines instead of administrators.

Acknowledgment

We thank the CCGrid reviewers for their suggestions. This work was supported by the US DOE, Office of Science, Advanced Scientific Computing Research (ASCR) under award number DE-SC0015234, the DOE grant number DE-SC0016074, and the Center for Research in Open Source Software. Danny Perez is supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation's exascale computing imperative.

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] 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.

[4] 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.

[5] 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.

[6] 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.

[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] A. Povzner, D. Sawyer, and S. Brandt, "Horizon: efficient deadline-driven disk I/O management for distributed storage systems," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10.

[10] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-grained Resource Sharing in the Data Center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11.

[11] N. Tran and D. A. Reed, "ARIMA Time Series Modeling and Forecasting for Adaptive I/O Prefetching."

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.*

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