ParSplice Keyspace Locality

1 INTRODUCTION

[DESCRIBE PROBLEM WE ARE TRYING TO SOLVE - CUT/PASTE FROM PDSW PAPER]

To solve this problem, we integrate Mantle into ParSplice [1]. Mantle is a dynamic policy engine that decomposes complicated services into manageable pieces, where developers write policies for "when" they want data moved, "where" they want data moved, and "how much" of the data to move. The framework then executes these policies whenever a decision needs to be made. This abstraction helps developers unfamiliar with the domain quickly reason about, develop, and deploy new policies that control temporal and spatial locality. It has already proven to be a critical control plane for improving file system metadata load balancing [2] and in this work we show its usefulness in cache management.

We show how Mantle:

- decomposes cache management into independent policies, making the problem more manageable and facilitating rapid development. We quickly implement the "how much" policy and focus on the "when" policy for the majority of this paper.
- has useful primitives that, while designed for file systems, are crucial for effective cache management. This finding shows how the Mantle engine generalizes to a different domain and code-base.
- can be used to quickly deploy a variety of cache management strategies, ranging from basic algorithms and heuristics to statistical models and machine learning. This mixing and matching may not outperform a hard-coded solution but can more adeptly change to different workloads and hardware.

We demonstrate these contributions by changing the input to our molecular dynamics simulation, which changes the keyspace access patterns.

2 PARSPLICE KEYSPACE ANALYSIS

2.1 Structured Access Regimes

Figures 4, 5 and 6.

- change immediately, different sizes
- monotonically increasing
- random access to a single key
- early keys accessed more

Conclusion: unique patterns of a real HPC application

2.2 Cache Size Analysis

Figure 7.

Conclusion: workload phases (high request rate to a small number of keys and then low request rate to a large number keys) need a dynamic load balancing policy.

2.3 Overfitting Policies

Figures 8 and 9.

Side Idea: we need to figure out what ParSplice memory is sensitive to: max usage or usage over time.

Conclusion: dynamic policies absorb the cost of a high read request rate for a 2.5 hour run, but it is infeasible to do this for every combination of system setup, job lengths, parsplice parameters.

Discussion. Why don't we just an LRU cache:

- we can do better (workload is structured and has locality)
- finding the size of the cache is hard
- hotspots dissipate too quickly

3 MANTLE ENGINE: METHODOLOGY

3.1 Experimental Setup

3.2 Mantle: Dynamic Load Balancing Policies

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

3.3 Integrating Mantle into ParSplice

- providing environment of metrics
- identifying where policies are made

4 MANTLE BRAINS: TOOLS WE PLUG IN TO DETECT "WHEN"

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

4.1 System Specific Knowledge

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

4.1.1 Request Rate.

4.1.2 Belady's Min.

4.2 Domain Specific Knowledge

e.g., ParSplice key access locality.

4.2.1 Regime Detection. At each time step, we find the lowest ID and compare against the local minima, which is the smallest ID we have seen thus far. If we move left to right,

,
2017. 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

' '

```
local function when()

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

Figure 1: ParSplice cache management policy.

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

Figure 2: ParSplice cache management policy.

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

Figure 3: CephFS file system metadta load balancer.

the local minima never changes because the local minima will start small. If we move from right to left, the local minima changes at each access regime. For points z, y, and x, if the local minimum is the same we are in a regime. Processing y, we set the local minima to be $min(y, m_l)$, where m_l is the local minima of the previous time step of z.

This raises false positives for regime changes

4.2.2 Trajectory Length.

4.3 Failed, Overcomplicated Brains

These techniques proliferated more, less transparent knobs

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

4.4 Cloud Techniques: Elastic Search

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

4.5 Future work

How Much: cache policy from past, regime detection

5 RELATION TO FILE SYSTEMS

The code snippets in Figures 2 and 3 are the policies used in ParSplice and CephFS, respectively. ParSplice uses policies to manage its caches and CephFS uses policies to control load balancing, but they both can be expressed with the Mantle API. From a high-level the ParSplice policy trims the cache if the cache reaches a certain size *and* if it has already absorbed the initial burstiness of the workload; the CephFS policy migrates load if the metadata load is higher than the average load *and* the current load has been overloaded for more than two iterations.

Condition for "Overloaded" (Fig. 2:Line 2; Fig. 3:Line 2) - these lines detect whether the node is overloaded using the "load" calculated in the load callback; while the load calculations and thresholds are different, the actual logic is exactly the same. Recall that this decision is made locally because there is no global scheduler or centralized intelligence.

State Persisted Across Decisions (Fig. 2:Lines 5,6; Fig 3:Lines 3,4,10) - these lines use the Mantle API to write/read state from previous decisions. For ParSplice, we save a boolean that indicates whether we have absorbed the workload's initial burstiness. For CephFS, we save the number of consecutive instances that the server has been overloaded. We also clear the count (Line 10) if the server is no longer overloaded. The underlying implementation saves the values to local disk.

Condition that Switches Policy (Fig. 2:Line 3; Fig. 3:Line 5) - these lines switch the policies using information from previous decisions. ParSplice trims its cache once it eclipses the "absorb" threshold while CephFS allows balancing when overloaded for more than two iterations. The persistent state is essential for both of these policy-switching conditions.

- Lustre Trace
- LinkedIn Trace
- Nathan's Trace

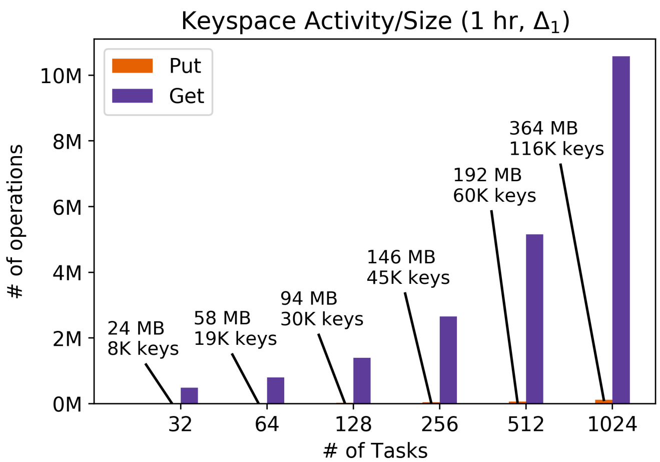


Figure 4

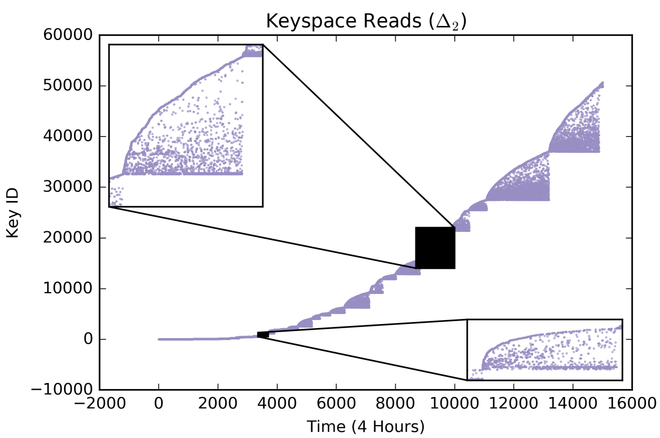


Figure 5

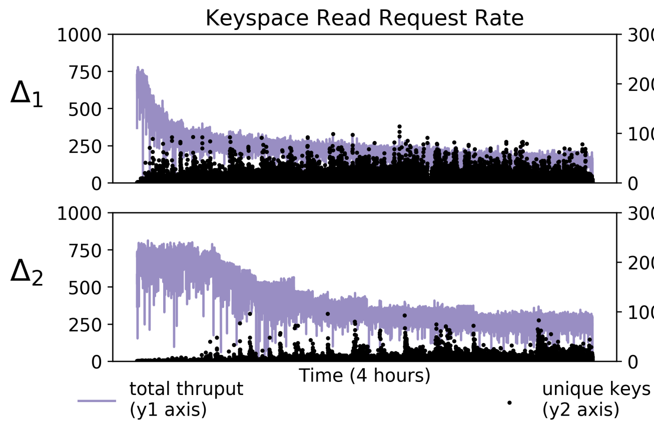


Figure 6

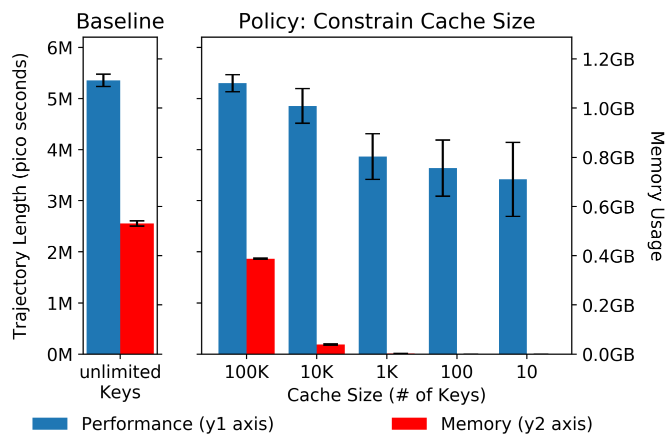


Figure 7

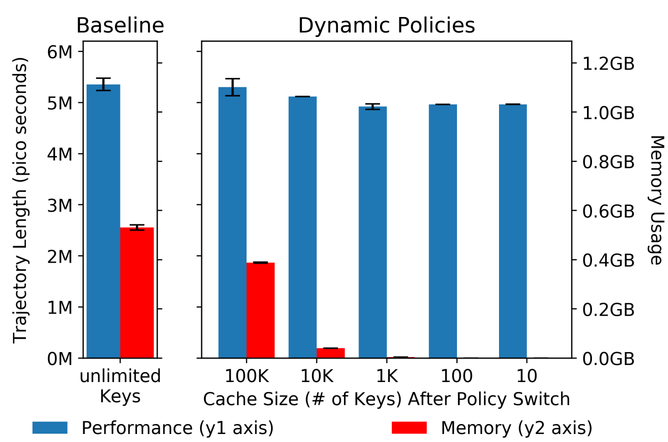


Figure 8

5.1 Using File System Balancers for ParSplice

5.2 Using ParSplice Balancers for File Systems

5.3 Visualizing File System Traces like ParSplice Keyspace Traces

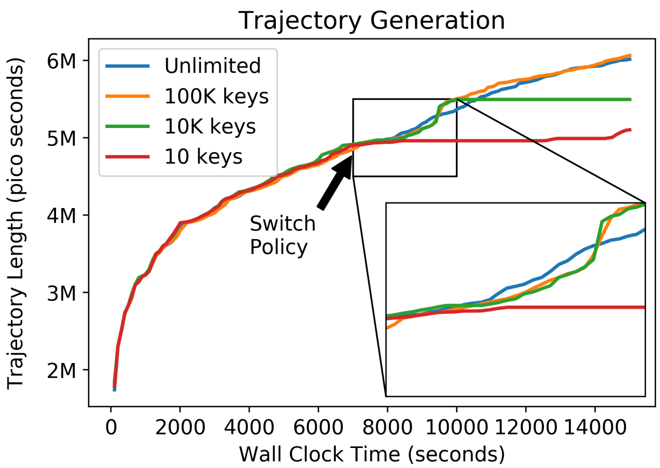
6 CONCLUSION

REFERENCES

[1] Danny Perez, Ekin D Cubuk, Amos Waterland, Efthimios Kaxiras, and Arthur F Voter. Long-Time Dynamics Through Parallel Trajectory Splicing. *Journal of chemical theory and computation* (????).

[2] Michael A. Sevilla, Noah Watkins, Carlos Maltzahn, Ike Nassi, Scott A. Brandt, Sage A. Weil, Greg Farnum, and Sam Fineberg. 2015. Mantle: A Programmable Metadata Load Balancer for the Ceph File System. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*.

, ,



Trajectory Generation
Unlimited
100K keys
10K keys
10 keys
Switch Policy
Trajectory Length (pico seconds)
6M
5M
4M
3M
2M
0
2000
4000
6000
8000
10000
12000
14000
Wall Clock Time (seconds)

Figure 9