Profiling Composable HPC Data Services WIP@PDSW, 2019

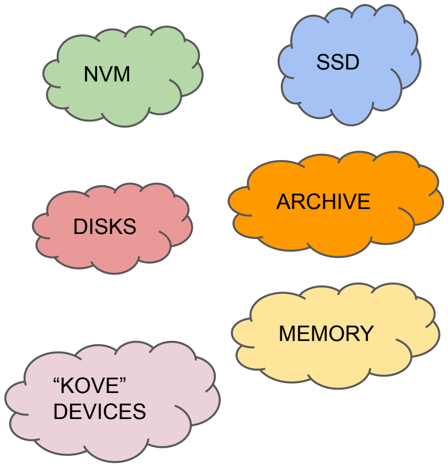
Srinivasan Ramesh
Allen D. Malony

University of Oregon

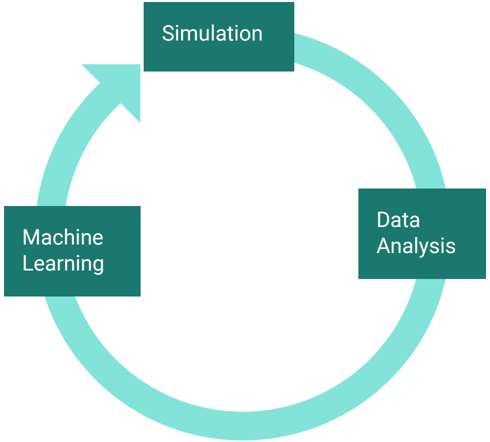
Philip H. Carns
Robert Ross
Shane Snyder
Argonne National Laboratory

Data Services: Managing Heterogeneity and Change

Storage:
Heterogeneous,
Multi-layered



Applications:
Diverse Workflows,
Data-driven



- Difficult to build **custom data services** efficiently:
  - Lots of moving parts
  - Need to dynamically adapt to changing application patterns
- Debugging performance problems is hard:
  - Numerous attempts at debugging microservices: **Dapper@Google, Stardust, X-Trace, etc**
  - We take inspiration from these

Mochi: Composable Data Services

Mobject service: An object store

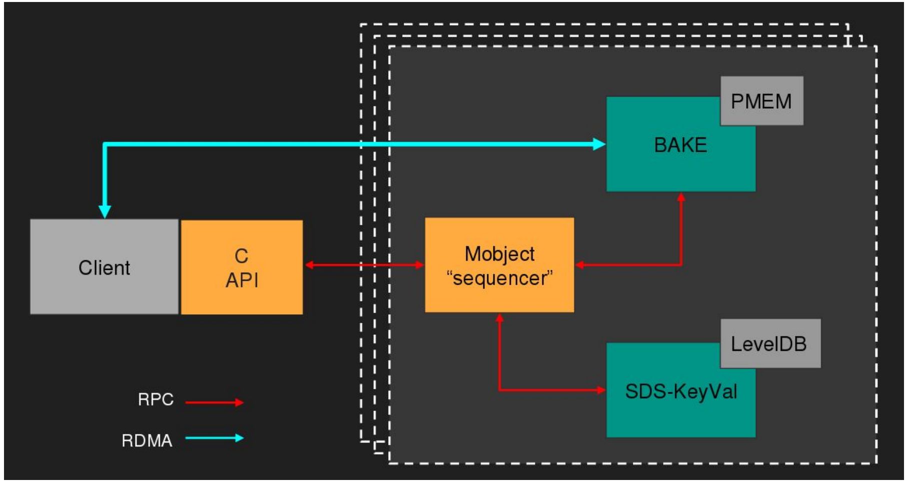
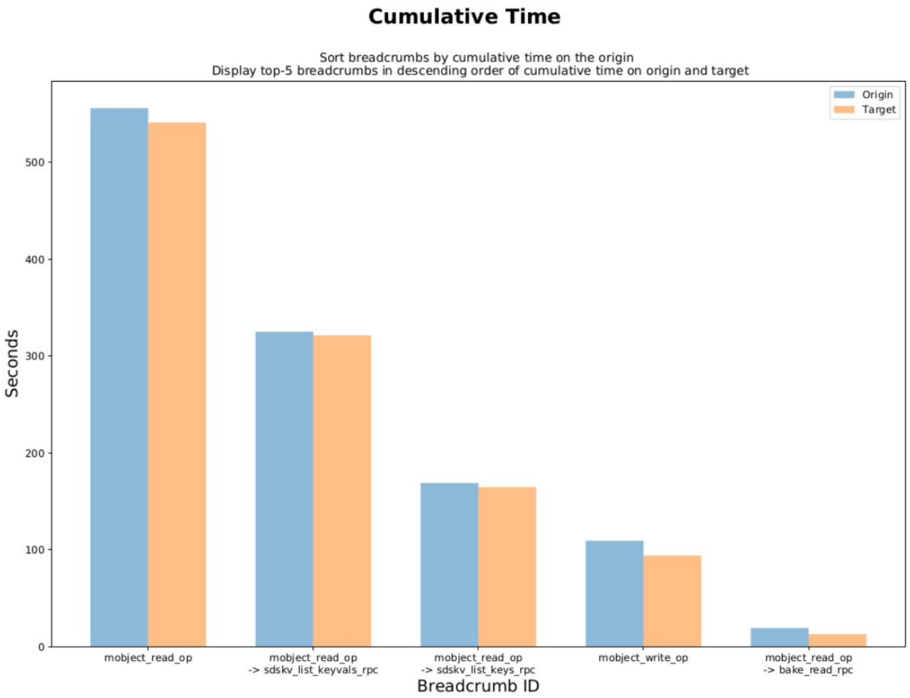


*Image credits: Matthieu Dorier, Argonne National Laboratory

- Mochi data services are built by composing ***microservices:***
  - RPC for control
  - RDMA for data movement
- Mochi’s building blocks:
  - **Mercury**, **Argobots, Margo**
- Performance Analysis in Mochi:
  - Build performance analysis capability directly into Mochi:
    - **Available out-of-the-box!**

Mochi: Performance Analysis

Mobject service: Call path profiling



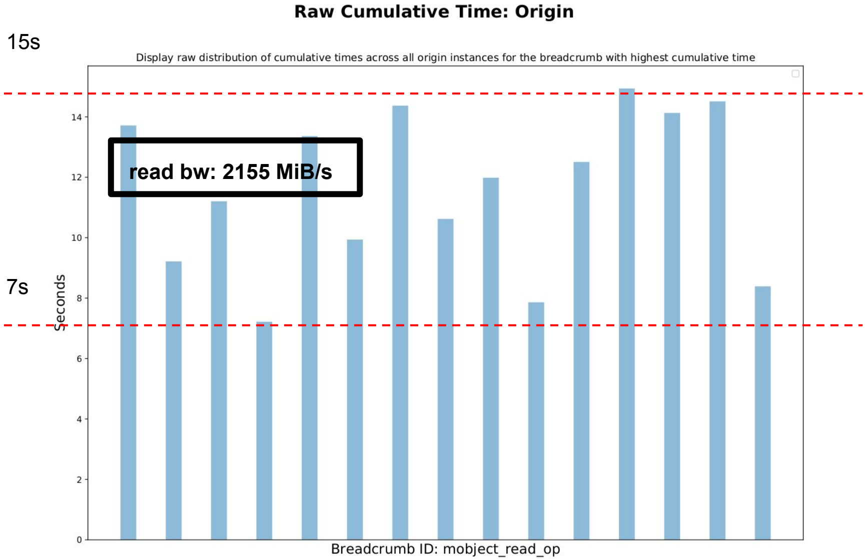
Call path profiling:

- We track the time spent in various ***call paths*** within the service:
  - A->C->D is a different call path from B->C->D
- **Key idea:** Each microservice stores and forwards RPC call path ancestry
- Time, call count, resource-level usage statistics updated at four instrumentation points: Client send/receive, Server send/receive
- What performance questions do we hope to answer?

Call Path Profiling: Detecting Load Imbalance

- Performance question: For a given call path, what is the ***distribution*** of call path times and counts in origin/target entities?

mobject_read_op: Raw distribution of call times across all origin (client) entities


Raw Cumulative Time: Origin

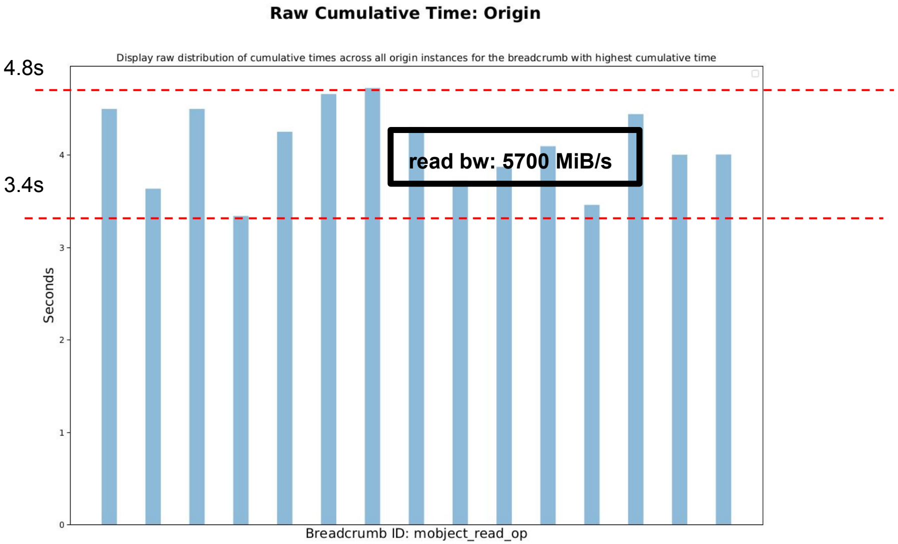
Display raw distribution of cumulative times across all origin instances for the breadcrumb with highest cumulative time

15s

7s

read bw: 2155 MiB/s

Overloaded server: Large variation in response time


Raw Cumulative Time: Origin

Display raw distribution of cumulative times across all origin instances for the breadcrumb with highest cumulative time

4.8s

3.4s

read bw: 5700 MiB/s

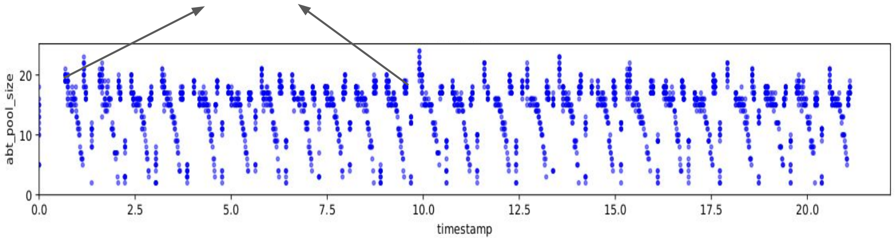
Multi-threaded server: Better read perf. and response time

Tracing: Detecting Resource-Level Inefficiencies

- Margo servers spawn a new Argobot User-Level-Task (ULT) for every incoming RPC request
  - Size of pool of tasks waiting to run is a measure of load and responsiveness of system
- We perform request tracing at the 4 instrumentation points previously described:
  - We collect Argobot pool size info, memory usage along request path
  - This enables correlation of call path behaviour with resource usage on node

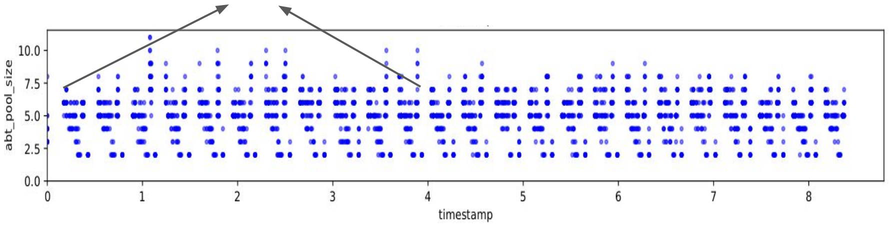
mobject_read_op: Max number of pending Argobot ULT's along request path

20 pending tasks



Overloaded server: Pending tasks are stacking up

7 pending tasks



Multi-threaded server: Reduction in number of pending tasks