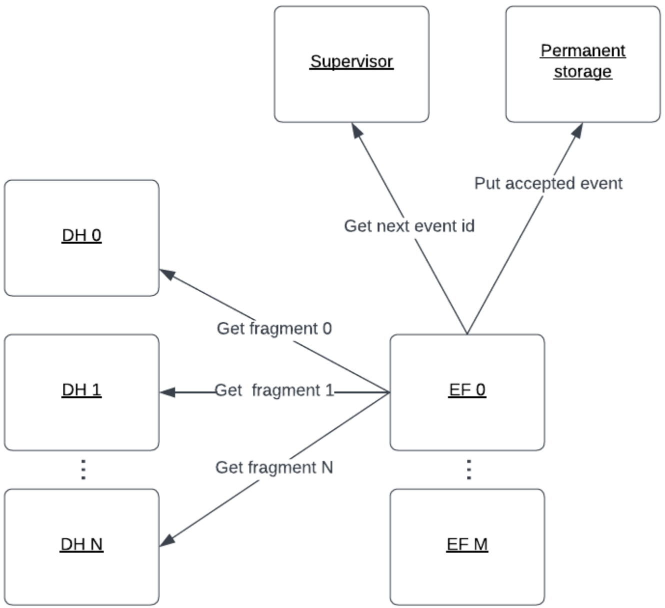
Study of high-throughput distributed caching system based of Intel DAOS for ATLAS Phase-II Dataflow

Mateusz Kojro

Openlab Technical Workshop 2023

ATLAS Dataflow - overview



Dataflow is the TDAQ sub-system that manages the movement of data from the detector readout system to the processing farm that analyses and selects interesting physics events

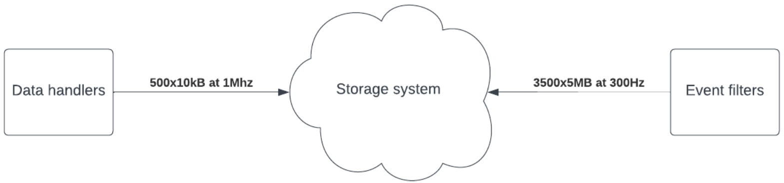
Intel DAOS - Overview

“DAOS is an open-source software-defined object store, providing high bandwidth, low latency and high IOPS storage for HPC “

https://github.com/daos-stack/daos

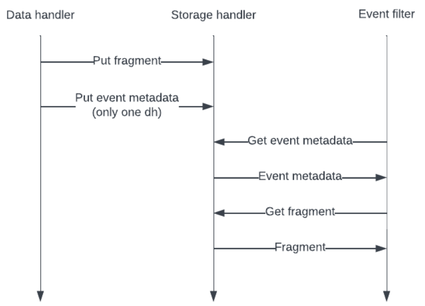
- Optimized specificly for the new drive technologies (NVMe, SCM)
- Provides key-value interface, non blocking I/O operations, end-to-end data integrity and transactional access
- Allows for massively distributed deployments
- Test cluster hosted at openlab
- How can it be used as a caching solution for dataflow?

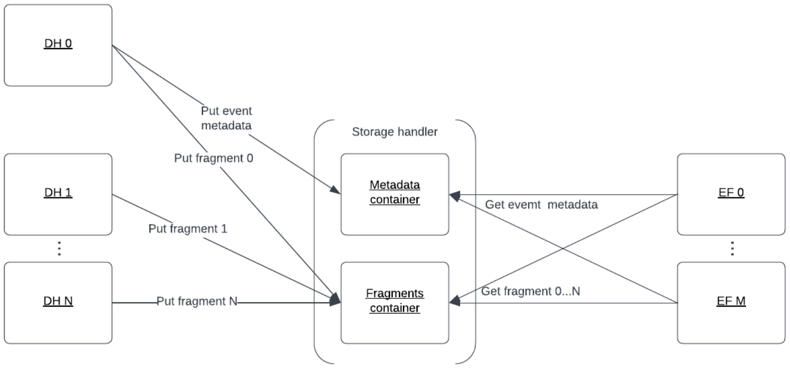
Phase-II Dataflow in numbers



- ~500 nodes producing 10 kB fragments (on average)
- Data comes in at 1Mhz (synchronized)
- Very high overall throughput: 5 TB/s
- ~3500 consumer machines

Proposed design





To investigate the performance characteristics of the proposed solution, existing emulators have been extended. Storage Handler implementation making use of DAOS capabilities has been implemented.

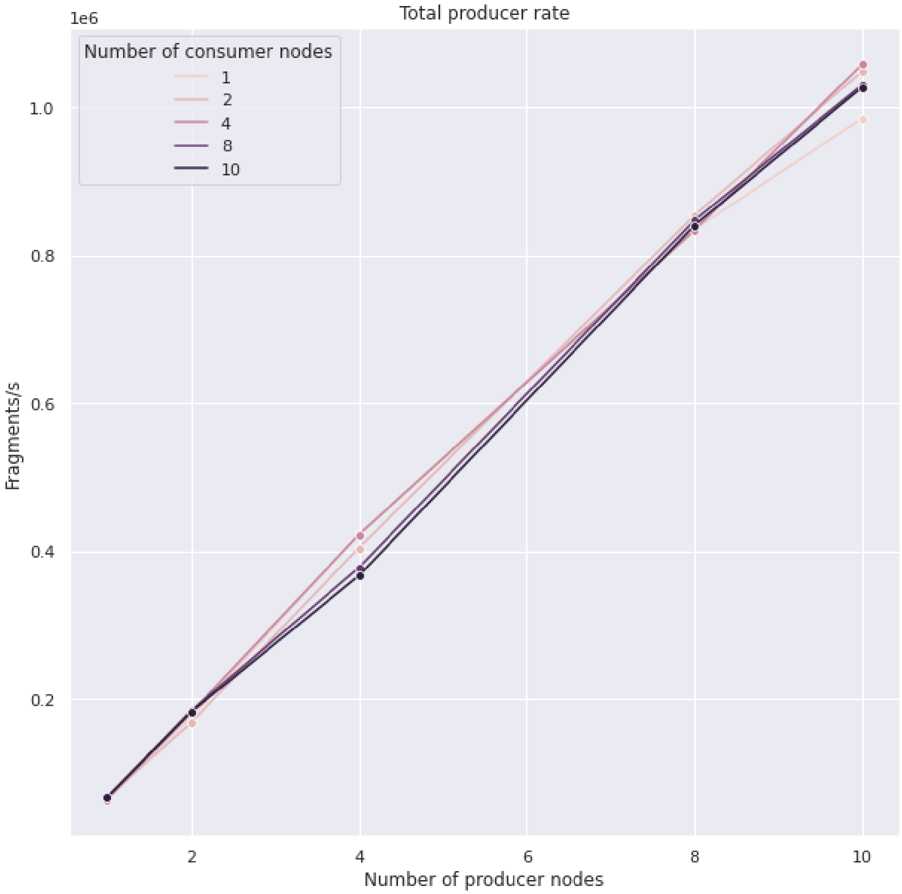
Proposed design – tradeoffs

Advantages	Disadvantages

• Separation between tightly constrained real-time system and the processing farm
• Previously event filter farm needed to be sized to accommodate for the peak data production, big enough buffer allows for sizing it for the average
• Persistent storage of metadata
	• Persistent storage solution requires:
	  ○ High throughput
	  ○ High IOPS
	• Big upfront cost
	• Big maintenance cost – drive wear (can be limited by the use of persistent memory devices)

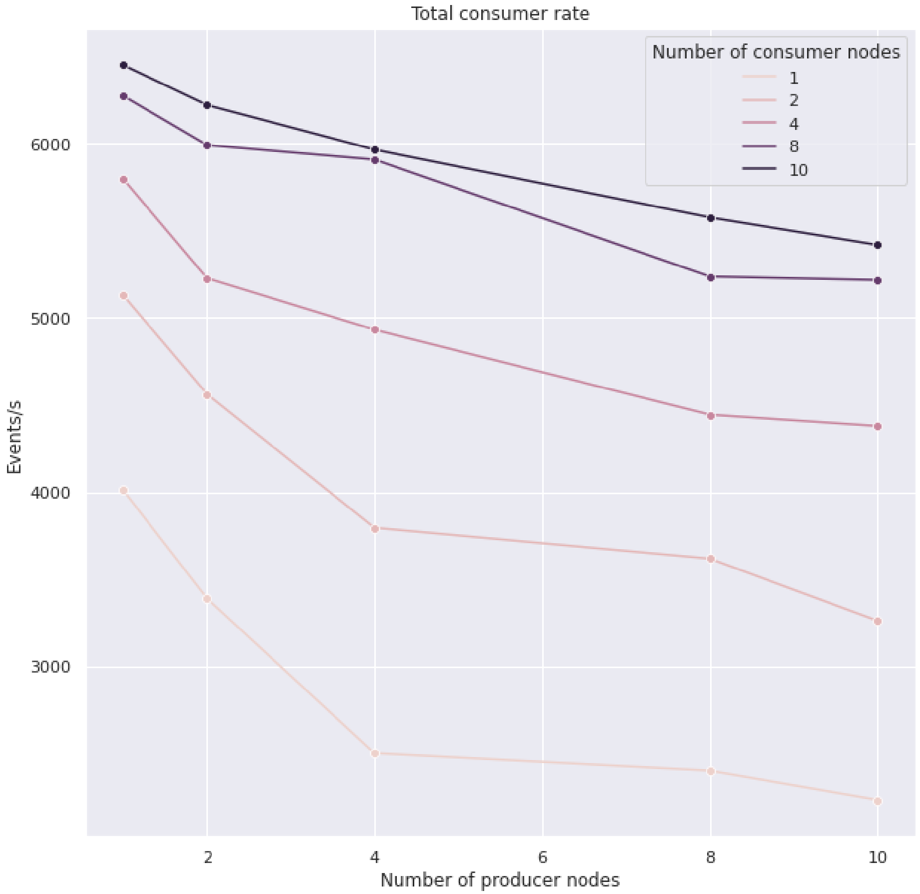
Results – data handlers

- As expected the number of consumer nodes does not impact the performance of the dh machines
- As expected given big enough storage backend the rate of pushing fragments scales linearly with the number of producers
- Currently the biggest problem to overcome is the rate at which we are able to access the storage (this is true for both producer and consumers of the buffer)
- Limited to 1 thread per machine



Results – event filters

- The rate of consumption is much smaller than the production rate:
    - Additional calls needed for the transfer of event metadata
    - Additional synchronization required for the event building
    - Ongoing collaboration with DAOS team to optimize it
- Decrease of consumption rate is expected due to the need to collect more fragments



Summary

- Dataflow emulators have been extended and modified to make use of DAOS capabilities
- Suite of different benchmarks has been performed
- The work on optimising the system is still ongoing

Thank you for your attention!

Benchmarking system

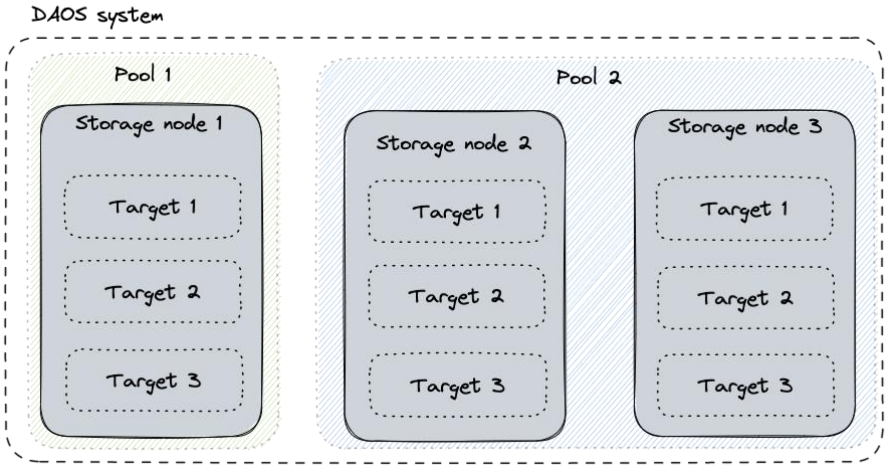
All the benchmarks have been run on the Intel Endevour cluster

- DAOS Cluster composed of 7 x Intel Cascade Lake nodes (2 DAOS engines per node)
- Each node equiped with
  - 2 x Intel Xeon Platinum 8260L @ 2.40 GHz
  - 196GB of RAM
  - 2 x ConnectX-6-HDR addapters (100Gbps)
  - 12 x 1.5 TB of Optane 100 PMem
  - 64 TB of NVMe drives

Intel DAOS - Basic concepts

- Pool – unit of provisioning, isolation and collection of targets
- Container – namespace for objects
- Object – stores user data, and has a unique object id

- Object id – unique location in the container (can be used to specify the way in wich associated data should be spread out)
- Distribution key (dkey) – all entries with the same dkey are located on the same target
- Attribute key (akey) – index into an array located at dkey



ATLAS → DAOS mapping

RUN NUMBER / run id	EVENT NUMBER / event id	MODULE ID / subdetector id
32 bits	64 bits (~10^4)	16 bits (~10^2)

Name	Container	Object id	dkey	akey
Strategy 4	Container per run	event_id	module_id	None
Strategy 5	Container per run	module_id	event_id	None
Strategy 6	Container per run	event_id + module_id	None	None

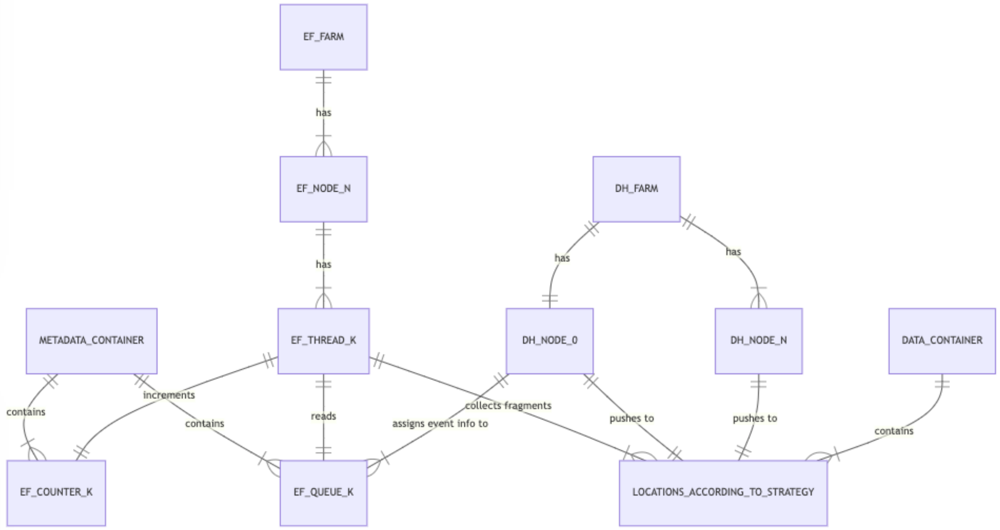
Producer → consumer notification

Queue producer algorithm

1. Choose *EF_THREAD_K* that should process given event (either uniformly or by taking into account states of *EF_QUEUES* and *EF_COUNTERS)*
2. Push event metadata at offset one bigger than last pushed

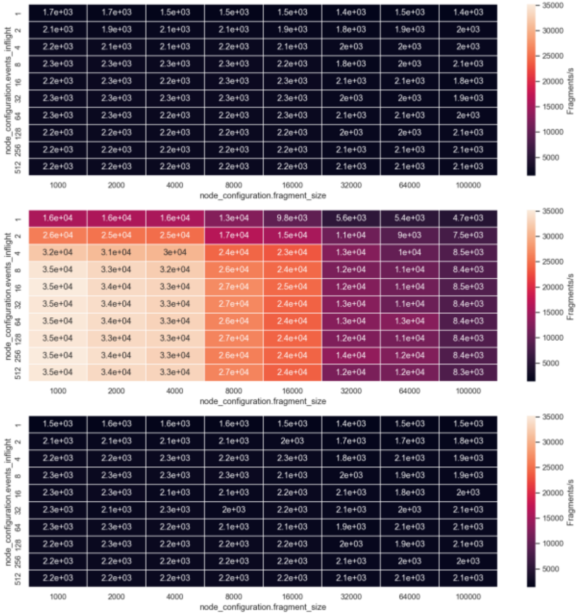
Queue consumer algorithm

1. *EF_THREAD_K* queries associated *EF_QUEUE_K* at offset *O*
2. if element at offset *O* exists process it and increment *EF_COUNTER_K* by one
3. if element at offset *O* does not exist sleep for XXXX (this is XXXHz)

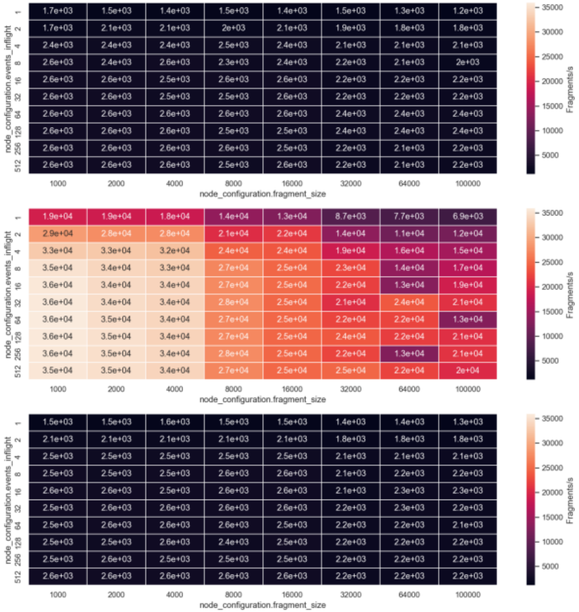


Results – number of nodes comparison

29 procs

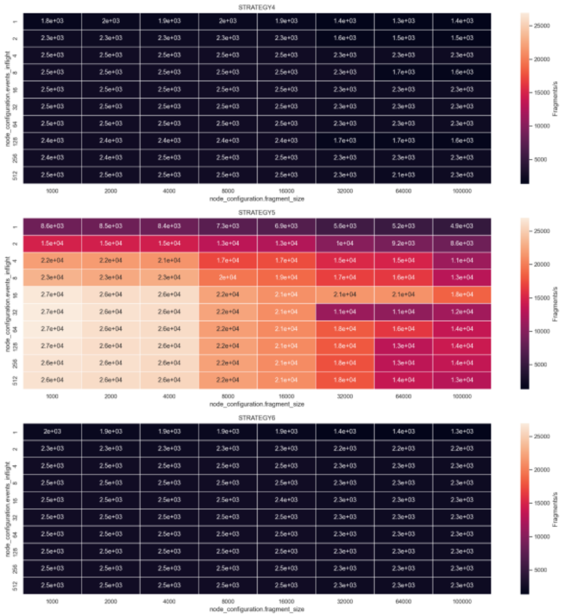


1 proc



Results – metadata transfer

w/ metadata transfer



w/o metadata transfer