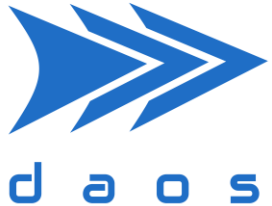
Profiling and identifying bottlenecks in DAOS

DUG'23, 2023-11-13

Adrian Jackson, Nicolau Manubens

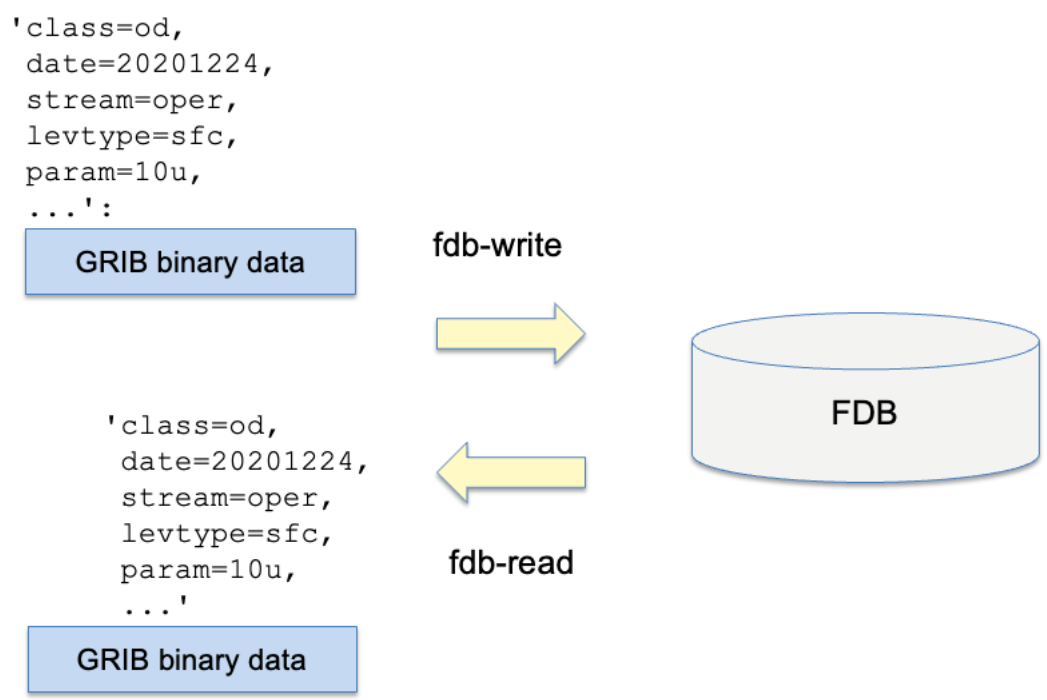
a.jackson@epcc.ed.ac.uk
nicolau.manubens@ecmwf.int



ECMWF epcc

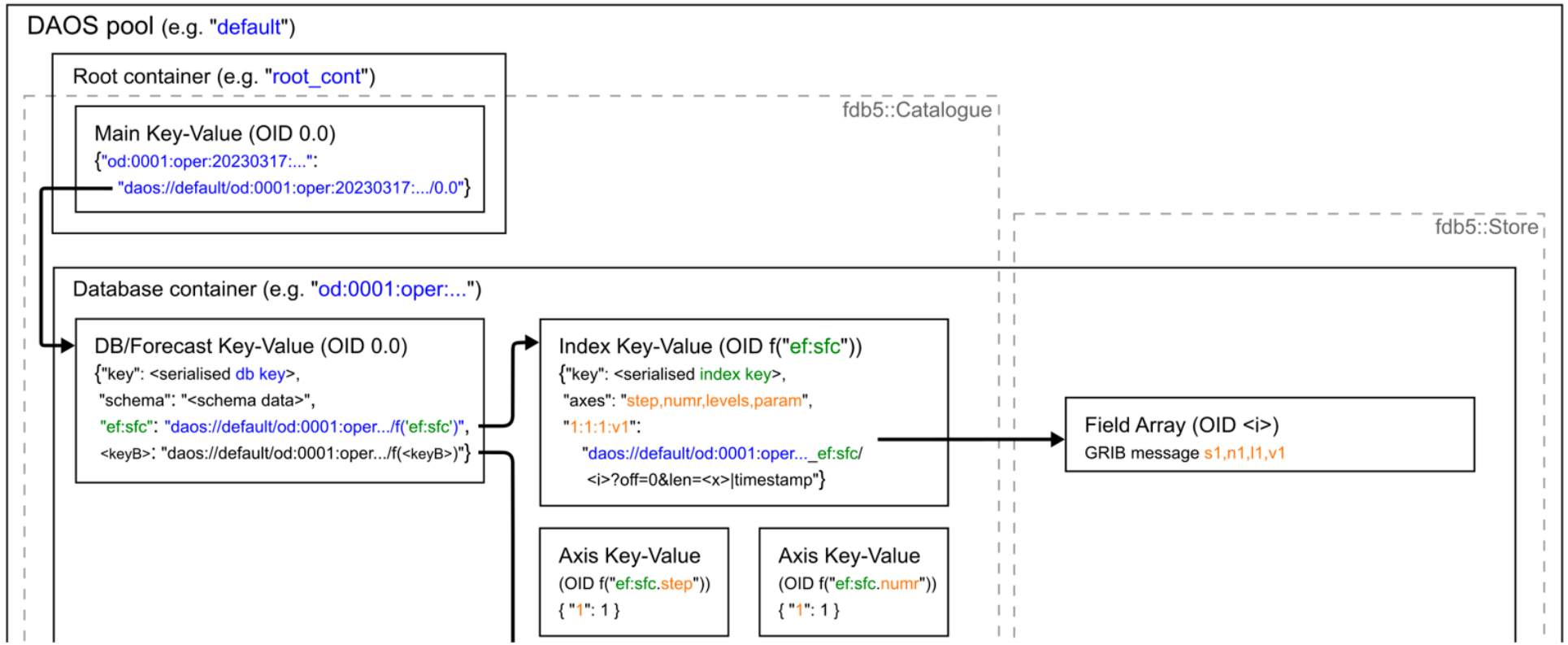
Use case

- ECMWF's FDB
  - library for weather field storage and indexing
  - domain-specific object store
  - C++
- Currently runs on Lustre operationally
  - clever use of files and directories to minimise IO ops, maximise bandwidth and throughput
  - all transparent to the user. A simple, domain-specific API is exposed to the user



Use case

- Now expanded to operate on DAOS
  - native use of DAOS via C API Key-Values and Arrays



Benchmark

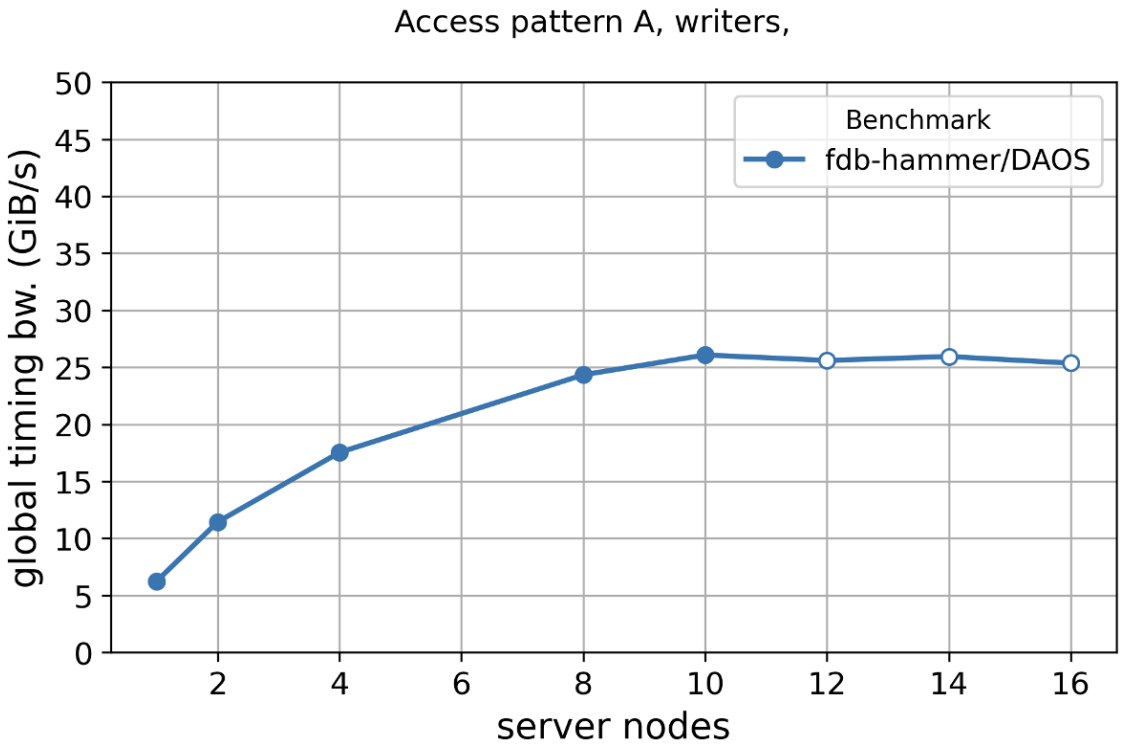
- Employed the **fdb-hammer** benchmark
  - <C> client nodes run <N> processes in **parallel** which **write** a sequence of <F> fields of 1 MiB
  - then **parallel read**
- No synchronisation
  - to better mimic real operational I/O
  - no MPI
  - no sharing of pool and container handles across processes
  - per-process static pool and container cache to avoid reopening
- Benchmark runs on a system with Optane DCPMM, without NVMe
- Bandwidth measurements for each run
  - measured wall-clock time from start of first parallel IO to end of last parallel IO and divided total transferred data by that time

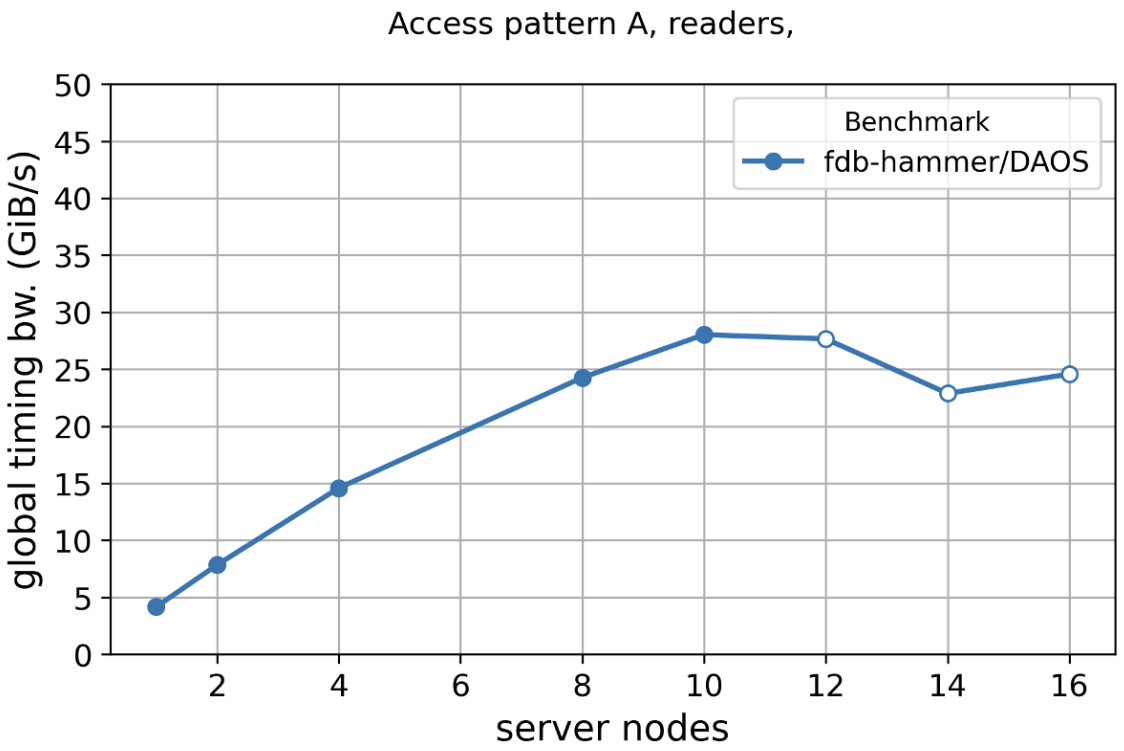
Profiling

- Instrumented all DAOS API calls in FDB to identify bottlenecks

```
void DaosArray::create() {

    [...]
    const daos_handle_t& coh = cont_.getOpenHandle();
    [...]

    fdb5::StatsTimer st{"daos_array_create", timer, ...};
    DAOS_CALL(
        daos_array_create(
            coh, oid_.asDaosObjIdT(), DAOS_TX_NONE,
            fdb5::DaosSession().objectCreateCellSize(),
            fdb5::DaosSession().objectCreateChunkSize(),
            &oh_, NULL
        )
    );
    st.stop();

    open_ = true;

}
```

Initial performance





Avoid Key-Value contention

- For a specific benchmark run configured with contention across processes on indexing Key-Values:
  - 20 GiB/s write
  - 13 GiB/s read
- Tweaking the benchmark configuration to have all processes operate on a separate Key-Values:
  - 35 GiB/s write
  - 68 GiB/s read
- This may not be trivial or possible for all applications. FDB allows some adjustment, which made this easy

Avoid RPCs where possible

- If non-critical objects are checked frequently, you may be able to cache some of them in DRAM
- Use `daos_array_open_with_attr` to avoid `daos_array_create` calls
  - Only supported for `DAOS_OT_ARRAY_BYTE`, not for `DAOS_OT_ARRAY`
  - Warning: the cell size and chunk size attributes need to be provided consistently on any future `daos_array_open_with_attr` to avoid data corruption
- `daos_array_get_size` calls can consume a lot of time
  - we avoided it by storing array size in our indexing Key-Values
  - alternative: use `DAOS_OT_ARRAY_BYTE`, over-allocate the read buffer, and read without querying the size. The actual read size (`short_read`) will be returned
- `daos_cont_alloc_oids` is expensive, call it just once per writer process

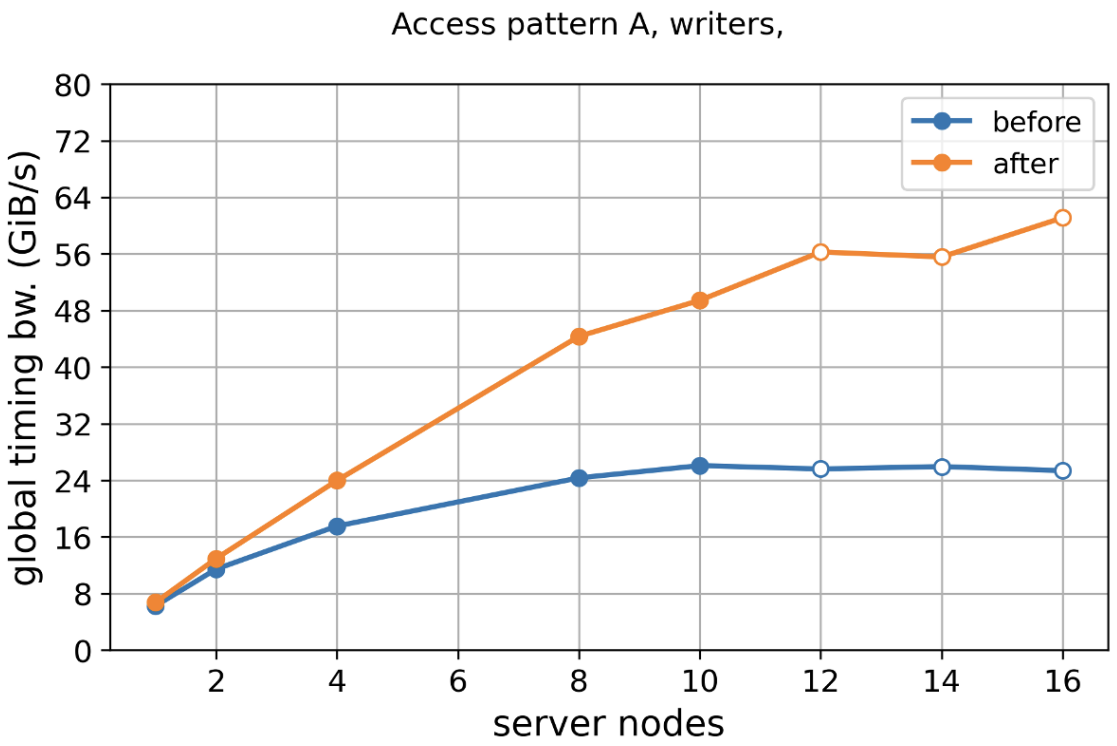
Avoid using too many containers

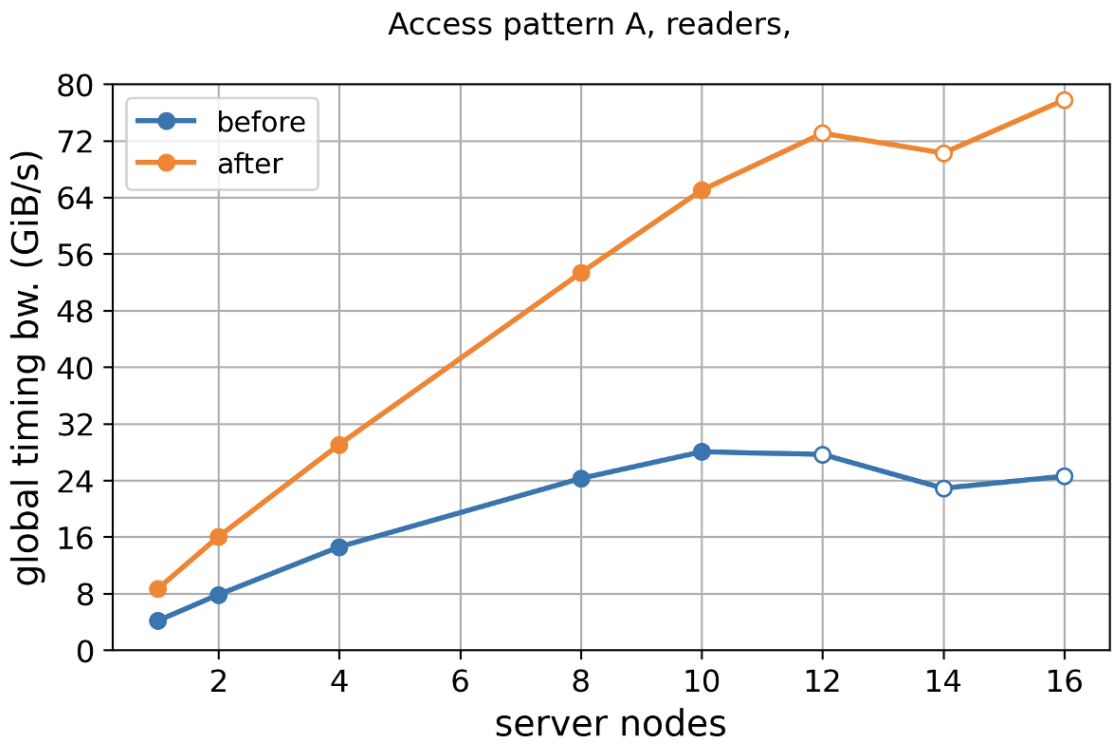
- Creating several containers (starting at ~300) in a DAOS pool makes it slow
- If not sharing handles, opening a same container from all processes is expensive
  - this happens even if only a few containers exist in the DAOS pool
  - e.g. out of 20 seconds taken by a process to write 2000 fields, 1.5 seconds were spent just to open one container
  - we observed this starting at ~200 parallel processes
- Opening more than one container per process is very expensive
  - e.g. out of 30 seconds taken by a process to read 2000 fields, 6 seconds were spent just to open two containers

Avoid using too many containers (2)

- We minimised use of containers as much as posible
- With longer benchmark runs the container opening overheads become negligible
- Container performance can vary depending on the DAOS version
  - container opening became slower in v2.4 compared to v.2.2.0

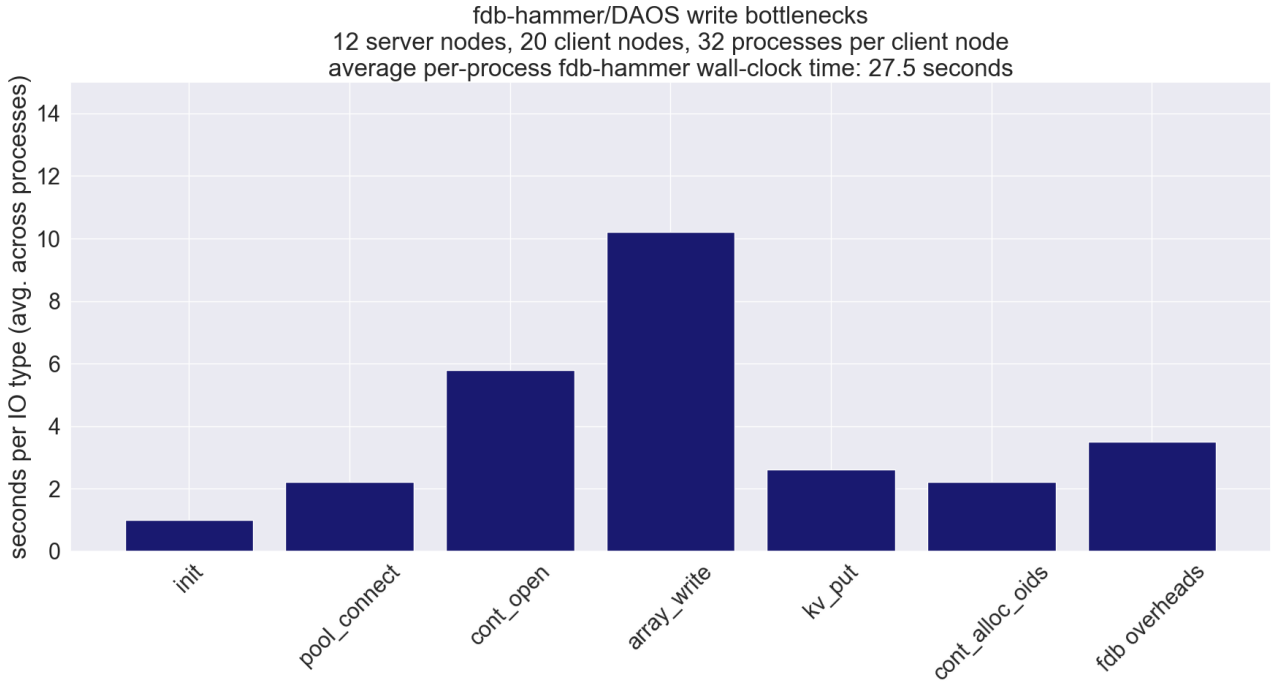
Final performance

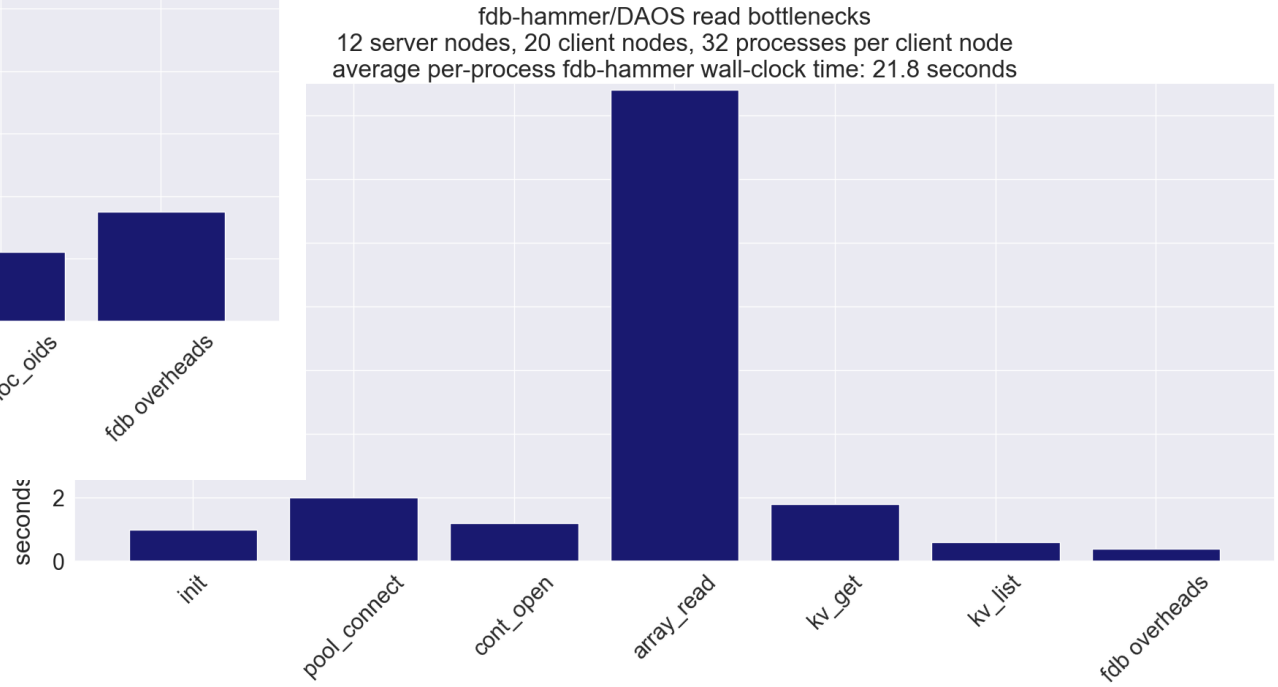




Profiling after all optimizations

- Most of time is spent in array write and read, which is a good sign. Connection overheads can be ignored





Other observations

- `daos_key_value_list` is expensive
- `daos_array_open_with_attrs`, `daos_kv_open` and `daos_array_generate_oid` are very cheap (no RPC)
- normal `daos_array_open` is expensive
- `daos_cont_alloc_oids` is expensive
- `daos_kv_put` and `get` are generally cheap. The shorter the strings stored as values the better
- `daos_obj_close`, `daos_cont_close` and `daos_pool_disconnect` are cheap
- `daos_array_read` behaves strangely
  - when performed after a `daos_array_get_size`, it is faster than a corresponding `daos_array_write` (as it should be)
  - when performed without a prior `daos_array_get_size`, it performs worse than the write. It looks as if a `get_size` were being performed internally if not performed manually beforehand.
  - this makes the read calls slower than write calls
  - to be investigated

Other observations

- Single engine (and rail) can result in worse write performance and better read performance
  - On a dual network system

- Pinning is important in dual-rail configurations