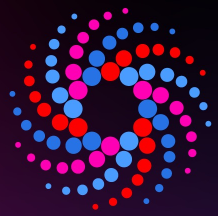


SC22
Dallas, TX | hpc accelerates.

Tutorial on the DAOS API - Exercises

Mohamad Chaarawi, AXG, Intel

Notices and Disclaimers

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration.

No product or component can be absolutely secure.

Tests document performance of components on a particular test, in specific systems. Differences in hardware, software, or configuration will affect actual performance. For more complete information about performance and benchmark results, visit http://www.intel.com/benchmarks .

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit http://www.intel.com/benchmarks .

Intel Advanced Vector Extensions (Intel AVX) provides higher throughput to certain processor operations. Due to varying processor power characteristics, utilizing AVX instructions may cause a) some parts to operate at less than the rated frequency and b) some parts with Intel® Turbo Boost Technology 2.0 to not achieve any or maximum turbo frequencies. Performance varies depending on hardware, software, and system configuration and you can learn more at http://www.intel.com/go/turbo.

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Cost reduction scenarios described are intended as examples of how a given Intel-based product, in the specified circumstances and configurations, may affect future costs and provide cost savings. Circumstances will vary. Intel does not guarantee any costs or cost reduction.

Intel does not control or audit third-party benchmark data or the web sites referenced in this document. You should visit the referenced web site and confirm whether referenced data are accurate.

Creating a Container

- kv1.c
  - Program to insert some keys in KV object
  - Run with ./a.out pool_name
- Will fail with:

```
container open failed: DER_NONEXIST (-1005) aborting
```

- Modify to create the container “cont1”
- Answer in kv1_answer.c

- `int daos_cont_create_with_label (daos_handle_t poh, const char *label, daos_prop_t *cont_prop, uuid_t *uuid,  daos_event_t *ev);`
- `int daos_cont_destroy(daos_handle_t poh, const char *cont, int force, daos_event_t *ev);`

Recap Program Flow

```c
#include <daos.h>
int main(int argc, char **argv)
{
        daos_handle_t   poh, coh;

        daos_init();
        daos_pool_connect("mypool", NULL, DAOS_PC_RW, &poh, NULL, NULL);
        daos_cont_create_with_label(poh, "mycont", NULL, NULL, NULL);
        daos_cont_open(poh, "mycont", DAOS_COO_RW, &coh, NULL, NULL);

        /** do things */

        daos_cont_close(coh, NULL);
        daos_pool_disconnect(poh, NULL);
        daos_fini();
        return 0;
}
```

Reading from KV

- kv2.c
  - Program to insert and read some keys with variable size buffer values.
  - Run with ./a.out pool_name
- Will only insert the keys.
- Modify to query the size of the key values and get each value.
- Answer in kv2_answer.c

KV put/get example

```
/** init, connect, cont_open */

oid.hi = 0;
oid.lo = 1;
daos_obj_generate_oid(coh, &oid, DAOS_OF_KV_FLAT, 0, 0, 0);
daos_kv_open(coh, oid, DAOS_OO_RW, &kv, NULL);


/** set val buffer and size */
daos_kv_put(kv, DAOS_TX_NONE, 0, "key1", val_len1, val_buf1, NULL);
daos_kv_put(kv, DAOS_TX_NONE, 0, "key2", val_len2, val_buf2, NULL);

/** to fetch, can query the size first if not known */
daos_kv_get(kv, DAOS_TX_NONE, 0, "key1", &size, NULL, NULL);
get_buf = malloc (size);
daos_kv_get(kv, DAOS_TX_NONE, 0, "key2", &size, get_buf, NULL);
daos_kv_close(kv, NULL);

/** free buffer, cont_close, disconnect, finalize */
```

Enumerating Keys from KV

- kv3.c
  - Program to insert and enumerate 20 keys.
  - Run with ./a.out pool_name
- Will only insert the keys
- Modify to enumerate all keys in list_key()
- Answer in kv3_answer.c

KV list example

```
/** enumerate keys in the KV */
daos_anchor_t   anchor = {0};
d_sg_list_t     sgl;
d_iov_t         sg_iov;

/** size of buffer to hold as many keys in memory */
buf = malloc(ENUM_DESC_BUF);
d_iov_set(&sg_iov, buf, ENUM_DESC_BUF);
sgl.sg_nr               = 1;
sgl.sg_nr_out           = 0;
sgl.sg_iovs             = &sg_iov;
```

```
daos_key_desc_t kds[ENUM_DESC_NR];

while (!daos_anchor_is_eof(&anchor)) {
  /** how many keys to attempt to fetch in one call */
  uint32_t        nr = ENUM_DESC_NR;


  memset(buf, 0, ENUM_DESC_BUF);
  daos_kv_list(kv, DAOS_TX_NONE, &nr, kds, &sgl,
               &anchor, NULL);

  if (nr == 0)
    continue;
  /** buf now container nr keys */
  /* kds arrays has length of each key */
}
```

Accessing Array of Integers

- array1.c
  - Program to create an integer array and write/read using the Array API.
  - Run with ./a.out pool_name
- Will fail as array is not created
- Modify and add the TODO items
  - Create the array
  - Set the Array IOD
  - Write the populated buffer of 100 integers
  - Read those integers
- Answer in array1_answer.c

DAOS Array example

```
/** create array - if array exists just open it */
daos_array_create(coh, oid, DAOS_TX_NONE, 1, 1048576, &array, NULL);


daos_array_iod_t iod;
d_sg_list_t      sgl;
daos_range_t     rg;
d_iov_t          iov;


/** set array location */
iod.arr_nr = 1; /** number of ranges / array iovec */
rg.rg_len = BUFLEN; /** length */
rg.rg_idx = rank * BUFLEN; /** offset */
iod.arr_rgs = &rg;


/** set memory location, each rank writing BUFLEN */
sgl.sg_nr = 1;
d_iov_set(&iov, buf, BUFLEN);
sgl.sg_iovs = &iov;


daos_array_write(array, DAOS_TX_NONE, &iod, &sgl, NULL);
daos_array_read(array, DAOS_TX_NONE, &iod, &sgl, NULL);
daos_array_close(array, NULL);
```

Multi-Level KV with Array and Single Value

- mkv1.c
  - Program to create a multi-level KV with 1 dkey that has 2 akeys:
    - 1 SV akey
    - 1 Array value of 100 integers
  - Run with ./a.out pool_name
- Modify and add the TODO items
  - Update 1 dkey with 2 akeys: set IODs for the update operation
  - Fetch each akey individually: set IOD for each update operation
- Answer in mkv1_answer.c

DAOS Object Update Example

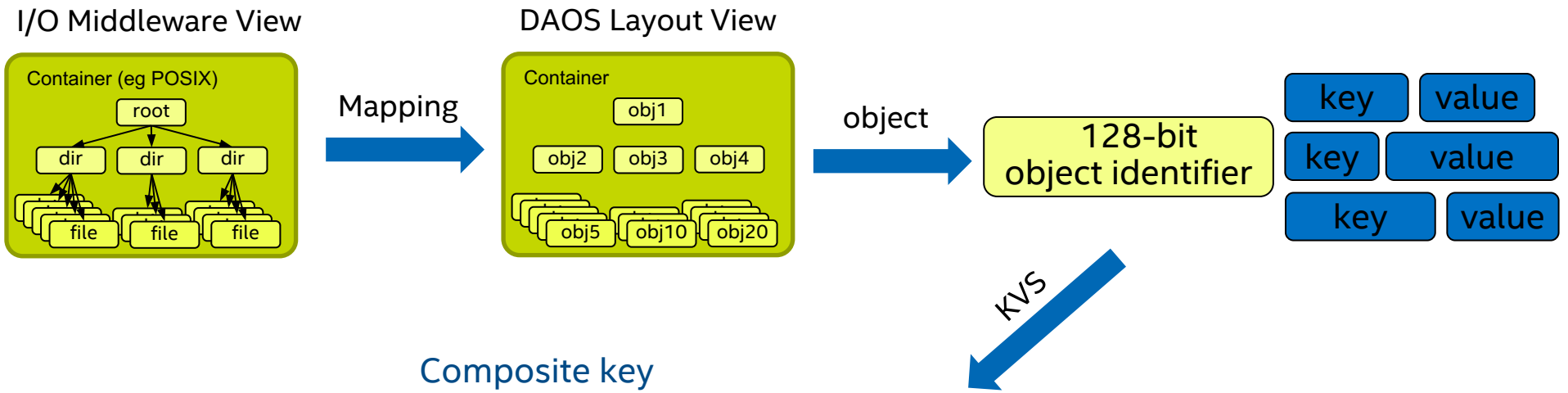
```
daos_obj_open(coh, oid, DAOS_OO_RW, &oh, NULL);
d_iov_set(&dkey, "dkey1", strlen("dkey1"));

d_iov_set(&sg_iov, buf, BUFLEN);
sgl[0].sg_nr = 1;
sgl[0].sg_iovs = &sg_iov;
sgl[1].sg_nr = 1;
sgl[1].sg_iovs = &sg_iov;

d_iov_set(&iod[0].iod_name, "akey1", strlen("akey1"));
d_iov_set(&iod[1].iod_name, "akey2", strlen("akey2"));

iod[0].iod_nr = 1;
iod[0].iod_size = BUFLEN;
iod[0].iod_recxs = NULL;
iod[0].iod_type = DAOS_IOD_SINGLE;

iod[1].iod_nr = 1;
iod[1].iod_size = 1;
recx.rx_nr = BUFLEN;
recx.rx_idx = 0;
iod[1].iod_recxs = &recx;
iod[1].iod_type = DAOS_IOD_ARRAY;

daos_obj_update(oh, DAOS_TX_NONE, 0, &dkey, 2, &iod, &sgl, NULL);
```

DAOS Object Fetch Example

```
daos_obj_open(coh, oid, DAOS_OO_RW, &oh, NULL);
d_iov_set(&dkey, "dkey1", strlen("dkey1"));

d_iov_set(&sg_iov, buf, BUFLEN);
sgl[0].sg_nr = 1;
sgl[0].sg_iovs = &sg_iov;
sgl[1].sg_nr = 1;
sgl[1].sg_iovs = &sg_iov;

d_iov_set(&iod[0].iod_name, "akey1", strlen("akey1"));
d_iov_set(&iod[1].iod_name, "akey2", strlen("akey2"));

iod[0].iod_nr = 1;
iod[0].iod_size = BUFLEN; /** if size is not known, use DAOS_REC_ANY and NULL sgl */
iod[0].iod_recxs = NULL;
iod[0].iod_type = DAOS_IOD_SINGLE;

iod[1].iod_nr = 1;
iod[1].iod_size = 1; /** if size is not known, use DAOS_REC_ANY and NULL sgl */
recx.rx_nr = BUFLEN;
recx.rx_idx = 0;
iod[1].iod_recxs = &recx;
iod[1].iod_type = DAOS_IOD_ARRAY;

daos_obj_fetch(oh, DAOS_TX_NONE, 0, &dkey, 2, &iod, &sgl, NULL, NULL);
```


intel

DAOS Data Model: Objects



Distribution key
Variable length
Integer, string, data structure, …
Hashed/distributed across engines

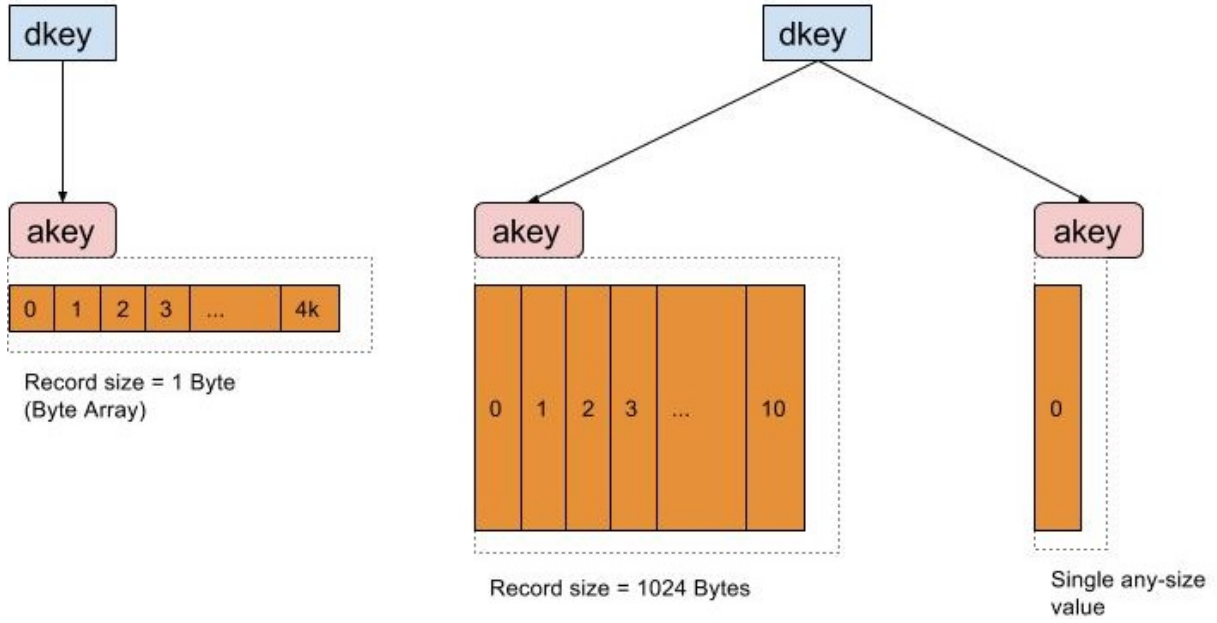
Attribute key
Variable length
Integer, string, data structure, …
Co-located on same engines
All akeys can be retrieved in a single RPC

Index
64-bit Integer
Address array of value
Support operation over range
of indices (= extent)

Record
Variable length value
From 1 byte to GB's

DAOS Multi-Level KV Object

- 2 level keys:
  - Distribution Key - Dkey (collocate all entries under it), holds multiple akeys
  - Attribute Key - Akey (lower level to address records)
  - Both are opaque (support any size / type)
- Value types (under akey):
  - Single value: one blob (traditional value in KV store)
  - Array value:
    - 1 record size per akey
    - Array of records that can be updates via different extents / iovec



- Intentionally very flexible, rich API; but at the expense of higher complexity for the regular user.