PERFORMANCE GUIDELINES FOR DCPM

Speaker: Szymon Romik (Intel Data Center Group)

<szymon.romik@intel.com>

May, 2019

Agenda

- Different instructions to flush a cache-line

- ECC block size impact on performance of memcpy

- Cache misses and how to avoid them

- Achieving failure atomicity at a low cost

- Memory Interleaving and data location

- How to correctly measure performance

Performance Guidelines

Cache flush instructions - clflush

```c
struct my_data {
    char cacheline_A[64];
    char cacheline_B[64];
    int persistent;
};
struct my_data *data = mmap(…);

memset(data->cacheline_A, 0xC, 64);
memset(data->cacheline_B, 0xD, 64);
__mm_clflush(data->cacheline_A);
__mm_clflush(data->cacheline_B);

data->persistent = true;
__mm_clflush(&data->persistent);
```

- Invalidates the cache-line
- Stalls the CPU, serializes code execution
- Commits data to WPQ (Write Pending Queue), minimal required persistence domain (ADR)

Cache flush instructions - clflushopt

```c
struct my_data {
    char cacheline_A[64];
    char cacheline_B[64];
    int persistent;
};
struct my_data *data = mmap(…);

memset(data->cacheline_A, 0xC, 64);
memset(data->cacheline_B, 0xD, 64);
__mm_clflushopt(data->cacheline_A);
__mm_clflushopt(data->cacheline_B);
__mm_sfence();

data->persistent = true;
__mm_clflushopt(&data->persistent);
__mm_sfence();
```

- Invalidates the cache-line
- Does not serialize execution
- Requires fencing to ensure ordering and persistence

Cache flush instructions – clwb (cache-line write back)

```
struct my_data {
    char cacheline_A[64];
    char cacheline_B[64];
    int persistent;
};
struct my_data *data = mmap(…);

memset(data->cacheline_A, 0xC, 64);
memset(data->cacheline_B, 0xD, 64);
__mm_clwb(data->cacheline_A);
__mm_clwb(data->cacheline_B);
__mm_sfence();

data->persistent = true;
__mm_clwb(&data->persistent);
__mm_sfence();
```

- Might not invalidate the cache-line
- Does not serialize execution
- Requires fencing to ensure ordering and durability

Identical to clflushopt on Cascade Lake platforms!
Works with Ice Lake

(intel)

Cache flush instructions – libpmem

```c
struct my_data {
    char cacheline_A[64];
    char cacheline_B[64];
    int persistent;
};
struct my_data *data = mmap(…);

pmem_memset(data->cacheline_A, 0xC, 64, PMEM_F_MEM_NODRAIN);
pmem_memset(data->cacheline_B, 0xD, 64, PMEM_F_MEM_NODRAIN);
pmem_drain();

data->persistent = true;
pmem_persist(&data->persistent, sizeof(data->persistent));
```

- Detects CPU features and uses the most efficient instructions available
- Automatically takes care of alignment and size of the buffer
- Has flags to enable better control over the memset/memcpy process
- Requires fencing to ensure ordering and durability

- If possible, applications should avoid flushing clean cache-lines.
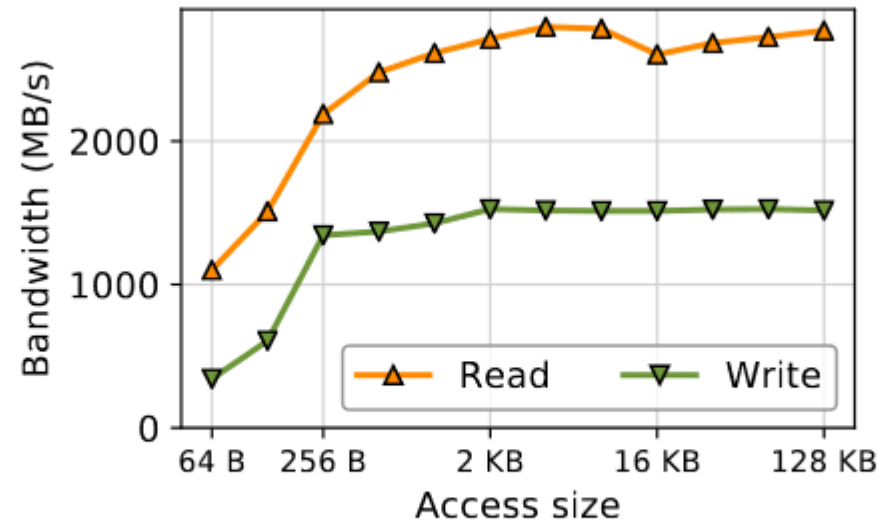
ECC block size and memory throughput

Figure 2: **Optane DC Random Access Bandwidth** Bandwidth for small accesses rises quickly but begins to taper off at 256 B. This data is for one thread accessing one DIMM.
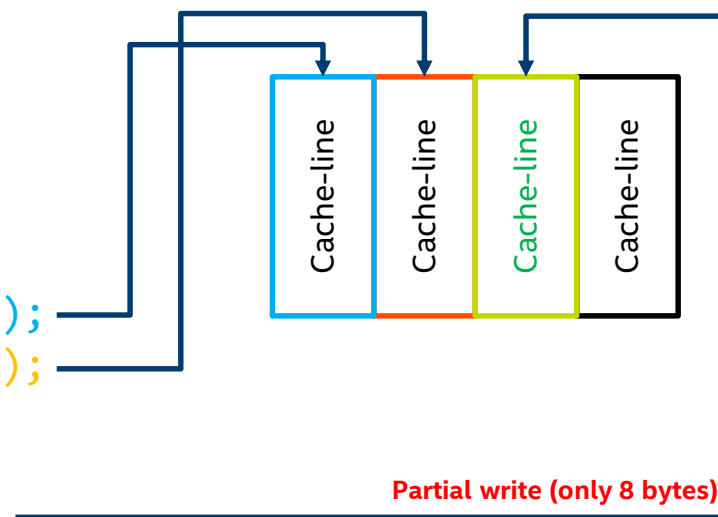
- Intel® Optane™ DC Persistent Memory achieves best throughput at 256 bytes.

Basic Performance Measurements of the Intel Optane DC Persistent Memory Module
https://arxiv.org/abs/1903.05714

ECC block size and memory throughput

```
struct my_data {
    char cacheline_A[64];
    char cacheline_B[64];
    int persistent;
};

pmem_memset(data->cacheline_A, …);
pmem_memset(data->cacheline_B, …);

data->persistent = true;
pmem_persist(&data->persistent,
    sizeof(data->persistent));
```

Cache-line Cache-line Cache-line Cache-line

Partial write (only 8 bytes)

Size of the ECC block is hardware dependent,
It may be different in other NVDIMMs and future
generations of Intel® Optane™ DC Persistent Memory.

- The CPU transfers data in cache-lines, but DIMMs must write full 256 byte blocks
- To utilize the entire available bandwidth, applications must read and write data in 256 blocks

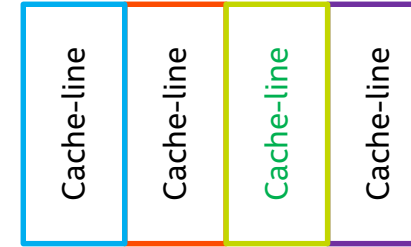- This example utilizes only about ~50% of the available throughput

Taking advantage of non-temporal stores

- Non-temporal stores bypass the CPU cache

  - No flushing is necessary!

- They are weakly-ordered

  - They require a fence to become visible to other threads and persistent

- They can take advantage of write-combining

  - This helps ensure that entire ECC blocks are written without back-filling

- libc does *not* guarantee the use of non-temporals for any variants of memcpy. Applications must explicitly use them.

  - Possible through libpmem

Avoiding cache misses and taking advantage of write-combining

```
struct my_data {
    char cacheline_A[64];
    char cacheline_B[64];
    char cacheline_C[64];
    char cacheline_D[64];
};

struct my_data *data = mmap(…);

for (size_t i = 0; i < 256; ++i) *((char *)data + i) = 0xC; /* BAD example, uses temporal stores */
-----------------
pmem_memset(data, 0xC, 256, PMEM_F_MEM_NONTEMPORAL); /* GOOD example, uses NT stores */
```


- The DIMMs must write entire ECC blocks, writing less than 256 bytes means that the remaining data will need to be fetched from its location.
- Similarly, but at the CPU level, reading/writing less than entire cache-lines can cause cache-misses to fetch the remaining data
- Applications can leverage non-temporal stores (and write-combining) to avoid the extra unnecessary cache-miss

Faster failure atomicity – BAD example

```
struct my_data {
    char cacheline_A[64];
    char cacheline_B[64];
    int persistent;
};
struct my_data *data = mmap(…);

void init() {
    pmem_memset(data->cacheline_A, 0xC, 64, PMEM_F_MEM_NODRAIN);  /* NO WRITE-COMBINING */
    pmem_memset(data->cacheline_B, 0xD, 64, PMEM_F_MEM_NODRAIN);
    pmem_drain();                                    /* FENCE */

    data->persistent = true;                         /* CACHE MISS */
    pmem_persist(&data->persistent, sizeof(data->persistent));    /* CLFLUSH + FENCE */
}
void recover() {
    if (data->persistent) {
    printf("my_data: %s %s", data->cacheline_A, data->cacheline_B);
    }
}
```

Faster failure atomicity – GOOD example

```
struct my_data {
    struct cachelines {
        char A[64];
        char B[64];
    } clines;
    uint64_t checksum;
    char padding[56]; /* structure is padded to avoid a cache-miss on partial cache-line write */
};
struct my_data *data = mmap(…);
void init() {
    struct my_data stack_data;    /* data prepared in DRAM on stack */
    memset(stack_data.clines.A, 0xC, 64); memset(stack_data.clines.B, 0xD, 64);
    stack_data.checksum = checksum_calc(&stack_data.clines); /* checksum is used for durability */
    pmem_memcpy(data, &stack_data, /* NT to avoid flushing and to take advantage of write-combining */
        sizeof(stack_data), PMEM_F_MEM_NONTEMPORAL);
}
void recover() {
    if (checksum_calc(&data.clines) == data->checksum)
    printf("my_data: %s %s", data->clines.A, data->clines.B);
}
```

0 Cache-misses, 1 fence and more efficient NVDIMM utilization

# Faster failure atomicity within the same cache-line

```
struct my_data {
    uint8_t A[32];
    uint8_t B[24];
    uint64_t persistent;
}; /* sizeof(struct my_data) == 64 */

struct my_data *data = mmap(…);

void init() {
    pmem_memset(data->B, 0xC, sizeof(data->B), PMEM_F_MEM_NODRAIN); /* can store data in any order */
    pmem_memset(data->A, 0xD, sizeof(data->A), PMEM_F_MEM_NODRAIN);
    uint64_t persistent = true;
    /* persistent flag must be stored last using either NT stores or with CLWB */
    pmem_memcpy(&data->persistent, &persistent, sizeof(persistent), PMEM_F_MEM_NONTEMPORAL);
}

void recover() {
    if (data->persistent) /* flag is guaranteed to be stored at the same or later time as rest of CL */
        printf("my_data: %s %s", data->A, data->B);
}
```

Memory interleaving
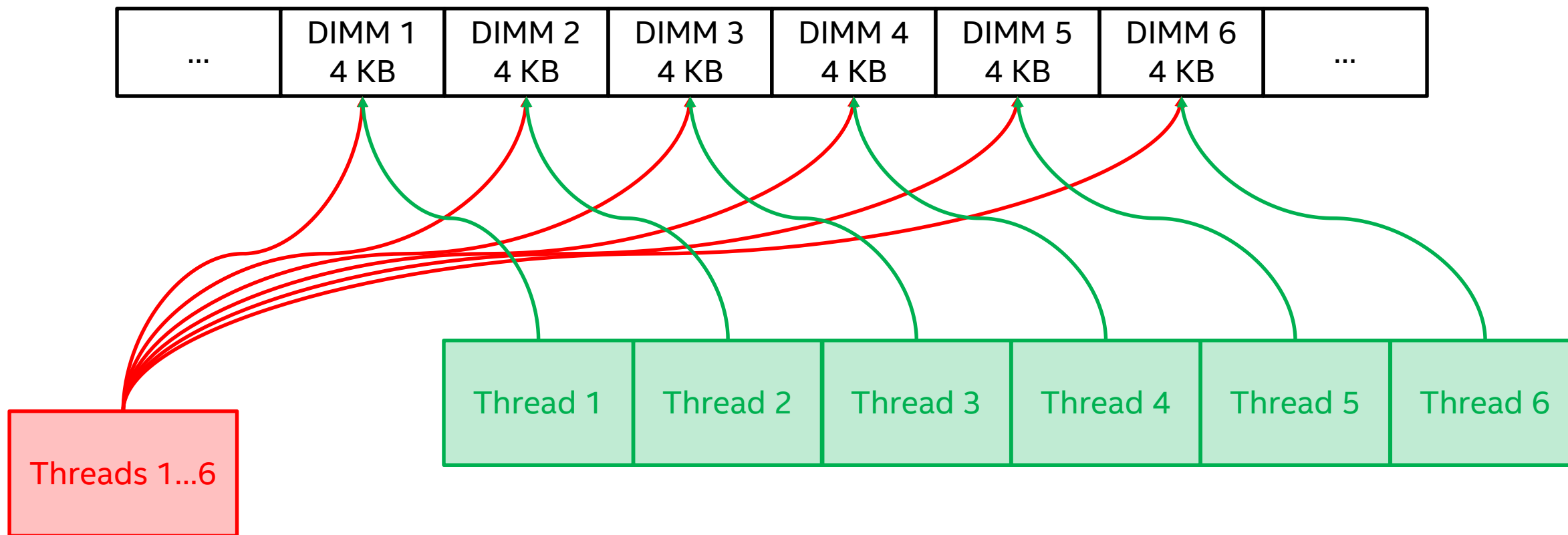
When using interleaving, remember to spread the traffic across all the DIMMs, otherwise performance will suffer greatly.

Measuring performance

- When using App-Direct, make sure to use DAX mount option.

  - Forgetting it might silently make filesystem fallback to using page cache

- Avoid benchmarking in shared or unstable (for example, overheating) environments.

- For reproducible results, disable hyper-threading and set a fixed clock-rate for the CPU.

- Make sure that all memory pages are allocated and faulted before starting time measurements. Use MAP_POPULATE mmap flag or manually prefault all pages.

  - Pages are allocated at the moment of first use. This can take a non-trivial amount of time.

- Remember to set appropriate NUMA node to execute the application at.

  - The benchmark should be running on the same NUMA node on which the work is performed.

- When benchmarking PMDK, make sure to use release libraries and that the prefault configuration option is enabled (PMEMOBJ_CONF="prefault.at_create=1;")

Summary

- Using Persistent Memory efficiently takes a bit of conscious effort and knowledge of the underlying hardware.

- PMDK helps with that by providing primitives which make it easy to fully utilize the hardware, as well as having an already optimized codebase.