SDC

STORAGE DEVELOPER CONFERENCE

SNIA SANTA CLARA, 2017

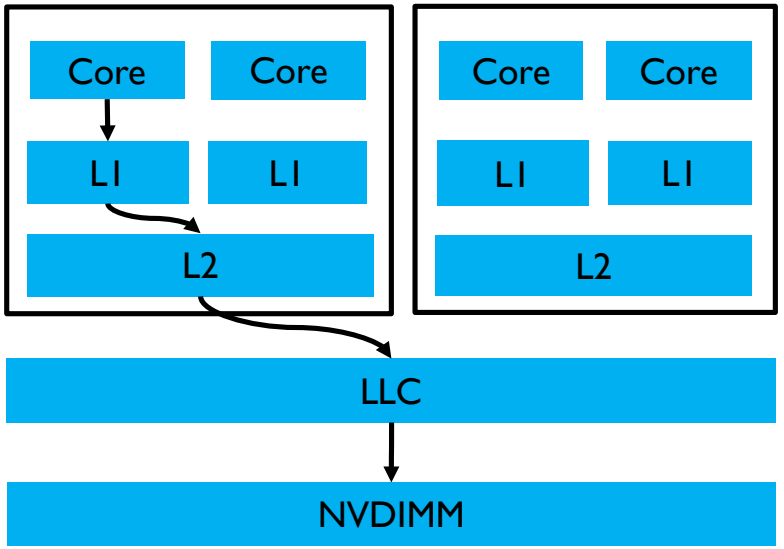
Solving the Challenges of Crash Recoverability in Persistent Memory Programming

Zhiqiang Ma

Intel Corporation

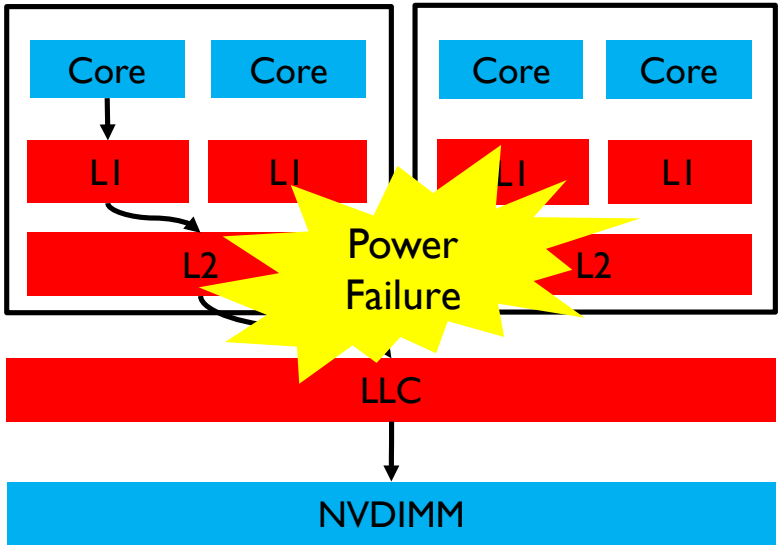
Persistent Memory

- Survive power or system failures
- Byte addressable
- Load/store instructions



Store != Persist

- Stored data is not persistent until it reaches the persistent memory domain
- Persistence order and store order can be different
- Volatile cache does not survive power loss



Crash Recoverability

- A power or system failure can leave a data structure in corrupted or unrecoverable state
- Crash recoverability at data structure level is crucial to higher level data consistency

Address Book Example

writeaddressbook.c

```
struct address {
        char name[64];
        char address[64];
        int valid;
};
int main()
{
        struct address *head = NULL;
        int fd;

        fd = open("addressbook.pmem", O_CREAT|O_RDWR, 0666);
        posix_fallocate(fd, 0, sizeof(struct address));
        head = (struct address *)mmap(NULL, sizeof(struct address),
                PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
        close(fd);
        strcpy(head->name, "Clark Kent");
        strcpy(head->address, "344 Clinton St, Metropolis, DC 95308");
        head->valid = 1;

}
```

readaddressbook.c

```
struct address {
        char name[64];
        char address[64];
        int valid;
};
int main()
{
        struct address *head = NULL;
        int fd;

        fd = open("addressbook.pmem", O_RDWR);
        head = (struct address *)mmap(NULL, sizeof(struct address),
                PROT_READ|PROT_WRITE, MAP_PRIVATE|
                MAP_NORESERVE, fd, 0);
        close(fd);
        if (head->valid == 1)
                printf("%s %s\n", head->name, head->address);
        ……
}
```


What Could Be in The Memory After The Power Failure?

Clark Kent\0
344 Clinton St, Metropolis, DC 95308\0
1

Clark Kent\0
344 Clinton St, Metropolis, DC 95308\0
0

Clark Kent\0
\0
1

\0
\0
1

\0
344 Clinton St, Metropolis, DC 95308\0
1

.....

Flushing Cached Stores to Memory

Instruction	Function
CLFLUSH	Flush cache line
CLFLUSHOPT	Flush cache line optimized
CLWB	Cache line write back

- A store fence is usually needed after CLFLUSHOPT or CLWB instruction(s) to enforce correct ordering
- Compilers provide intrinsic functions, _mm_clflushopt(), _mm_sfence(), etc.
- NVML provides high level APIs, pmem_flush(), pmem_persist(), etc.

Flushing Cached Stores Example

```
struct {
        int x;
        ……
        int y;        //x and y are in different cache lines
} z;        // z is in persistent memory
z.x = 0;
_mm_clflushopt(&z.x);
_mm_sfence();
……
z.y = 1;    // z.x is already in persistent memory domain
```


But ……

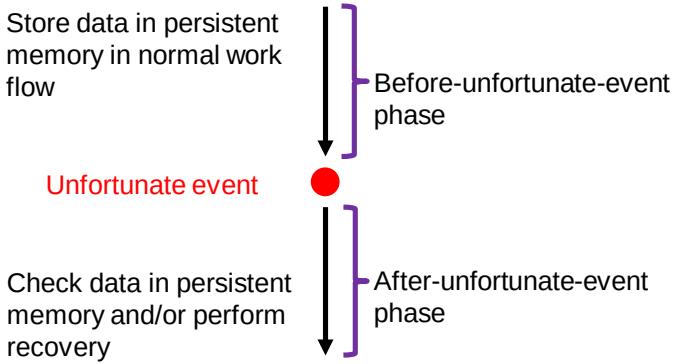
- When a cache line should be flushed?
  - Missing cache line flushes may leave data structures corrupted or unrecoverable
  - Excessive cache line flushes hurt performance
- In what order should cache lines be flushed (or what is the persistence order)?
  - Is it correct to flush “head->address” and “head->name” after “head->valid” is set?
- In what order should data be stored?
  - Is it correct to set “head->valid” before storing “head->address” and/or “head->name”?

Intel® Persistence Inspector

- Dynamic tool for finding missing/redundant cache flushes, missing store fences etc.
- Issues do not need to actually occur
- No source changes required
- Pinpoint issues to exact source locations with stack traces

Unfortunate Events and Persistent Memory Applications

- An unfortunate event, for example, a power failure, divides a persistent memory application into 2 phases
- Normal completion is a special unfortunate event



Two Phases of Address Book Example

Before-unfortunate-event: writeaddressbook

```
struct address {
        char name[64];
        char address[64];
        int valid;
};
int main()
{
        struct address *head = NULL;
        int fd;

        fd = open("addressbook.pmem", O_CREAT|O_RDWR, 0666);
        posix_fallocate(fd, 0, sizeof(struct address));
        head = (struct address *)mmap(NULL, sizeof(struct address),
                PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
        close(fd);
        strcpy(head->name, "Clark Kent");
        strcpy(head->address, "344 Clinton St, Metropolis, DC 95308");
        head->valid = 1;
}
```

After-unfortunate-event: readaddressbook

```
struct address {
        char name[64];
        char address[64];
        int valid;
};
int main()
{
        struct address *head = NULL;
        int fd;

        fd = open("addressbook.pmem", O_RDWR);
        head = (struct address *)mmap(NULL, sizeof(struct address),
                PROT_READ|PROT_WRITE, MAP_PRIVATE|
                MAP_NORESERVE, fd, 0);
        close(fd);
        if (head->valid == 1)
                printf("%s %s\n", head->name, head->address);
        ……
}
```


Why Two Phases?

- Persistence matters only if the application stops upon an unfortunate event
- Persistence matters only if the data is used after the application restarts after an unfortunate event
- How data is used after an unfortunate event defines how (the order in which) data should get persistent before the unfortunate event

Using Intel® Persistence Inspector (1/3)

- Step 1: Analyze before-unfortunate-event phase
  - Watch and record persistent memory events

```
$ pmeminsp check-before-unfortunate-event -pmem-file \
addressbook.pmem -- writeaddressbook
```


Using Intel® Persistence Inspector (2/3)

- Step 2: Analyze after-unfortunate-event phase
  - Find persistence orders

```
$ pmeminsp check-after-unfortunate-event -pmem-file \
addressbook.pmem – readaddressbook
```


Using Intel® Persistence Inspector (3/3)

- Step 3: Report
  - Check and report potential persistence order violations

$ pmeminsp report – writeaddressbook readaddressbook

```
#=============================================================================
# Diagnostic # 1: Missing cache flush
#-------------------
 The first memory store
   of size 8 at address 0x7F062D148050 (offset 0x50 in /samples/addressbook.pmem)
   in /samples/addressbook/writeaddressbook!main at writeaddressbook.c:45 - 0x71E
   in /lib/x86_64-linux-gnu/libc.so.6!__libc_start_main at <unknown_file>:<unknown_line> - 0x21F43
 is not flushed before the second memory store
   of size 4 at address 0x7F062D148080 (offset 0x80 in /samples/addressbook/addressbook.pmem)
   in /samples/addressbook/writeaddressbook!main at writeaddressbook.c:46 - 0x73F
   in /lib/x86_64-linux-gnu/libc.so.6!__libc_start_main at <unknown_file>:<unknown_line> - 0x21F43
```

First store location and call stack

Second store location and call stack

Address Book Example (Corrected)

writeaddressbook.c

```
struct address {
        char name[64];
        char address[64];
        int valid;
};
int main()
{
        struct address *head = NULL;
        int fd;

        fd = open("addressbook.pmem", O_CREAT|O_RDWR, 0666);
        posix_fallocate(fd, 0, sizeof(struct address));
        head = (struct address *)mmap(NULL, sizeof(struct address),
                PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
        close(fd);
        strcpy(head->name, "Clark Kent");
        strcpy(head->address, "344 Clinton St, Metropolis, DC 95308");
        _mm_clflushopt(&(head->name));
        _mm_clflushopt(&(head->address));
        _mm_sfence();
        head->valid = 1;
        _mm_clflushopt(&(head->valid));
        _mm_sfence();
}
```

readaddressbook.c

```
struct address {
        char name[64];
        char address[64];
        int valid;
};
int main()
{
        struct address *head = NULL;
        int fd;

        fd = open("addressbook.pmem", O_RDWR);
        head = (struct address *)mmap(NULL, sizeof(struct address),
                PROT_READ|PROT_WRITE, MAP_PRIVATE|
                MAP_NORESERVE, fd, 0);
        close(fd);
        if (head->valid == 1)
                printf("%s %s\n", head->name, head->address);
        ……
}
```


Takeaways

- Persistent memory presents new challenges to software developers
- To maintain data recoverability and consistency, cached data updates need to be flushed to persistent memory domain in correct orders
- Intel® Persistence Inspector tool helps developers solve the challenges

Questions?

zhiqiang.ma@intel.com

Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

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.

Optimization Notice

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.

Notice revision #20110804