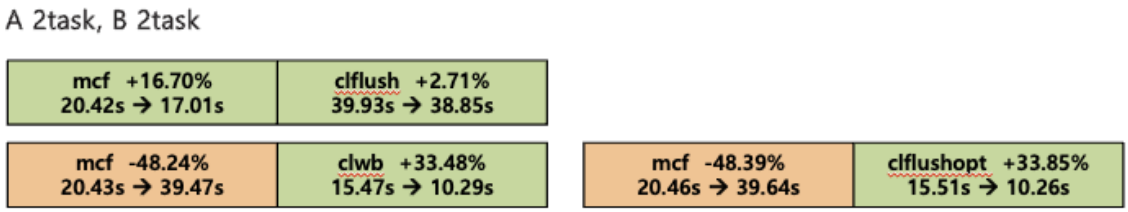
2022/06/18 ~ 2022/06/20

CLFLUSH vs. CLFLUSHOPT & CLWB

- 아래의 실험 결과와 같이 기존 PM DAX sequential write 실험에서 flush instruction으로 어떤 instruction을 사용하는가에 따라 성능 경향성이 다르게 나오는 것을 확인할 수 있다. 이때, clflushopt와 clwb의 결과가 거의 동일한 것을 통해 cache invalidation이 clflush와 나머지 flush instruction(clflushopt, clwb) 사이의 차이를 만들어내는 원인이 아님을 알 수 있다.



- 이에 따라 하단과 같이 clflush 및 clflushopt의 차이에 관련된 자료(Intel의 Software Developer's Manual, Intel의 Optimization Reference Manual 등)를 찾아보았고, clflush와는 다르게 clflushopt는 다른 writes, clflush, clflushopt instruction들과의 ordering을 보장하지 않고 flush을 진행한다는 차이점이 존재함을 알 수 있었다.

> Executions of the CLFLUSH instruction are ordered with respect to each other and with respect to writes, locked read-modify-write instructions, fence instructions, and executions of CLFLUSHOPT to the same cache line[1]. They are not ordered with respect to executions of CLFLUSHOPT to different cache lines. For updated memory order details of CLFLUSH and other memory traffic, please refer to the CLFLUSH reference pages in Chapter 3 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*, and the "Memory Ordering" section in Chapter 8 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

> Executions of the CLFLUSHOPT instruction are ordered with respect to fence instructions and to locked read-modify-write instructions; they are also ordered with respect to the following accesses to the cache line being invalidated: older writes and older executions of CLFLUSH. They are not ordered with respect to writes, executions of CLFLUSH that access other cache lines, or executions of CLFLUSHOPT regardless of cache line; to enforce CLFLUSHOPT ordering with any write, CLFLUSH, or CLFLUSHOPT operation, software can insert an SFENCE instruction between CLFLUSHOPT and that operation.
>
> 출처 : https://www.felixcloutier.com/x86/clflushopt

- 이에 따라 clflush와 clflushopt의 성능 경향성 차이가 ordering을 지키지 않는 것에서 비롯됨을 증명하기 위해 하단의 Intel Optimization Reference Manual 내용을 기반으로 clflushopt의 전후에 sfence instruction을 넣어서 실험을 진행하였다. (단, clflushopt 간에도 ordering이 보장이 안되기 때문에 이를 고려하여 하단의 예시와 다르게 clflushopt가 실행될 때마다 전후 및 직후에 sfence를 넣어주었으며, 이렇게 하면 서로 간의 ordering이 보장되는 clflush와 동일한 ordering 보장 효과를 발휘할 수 있다)

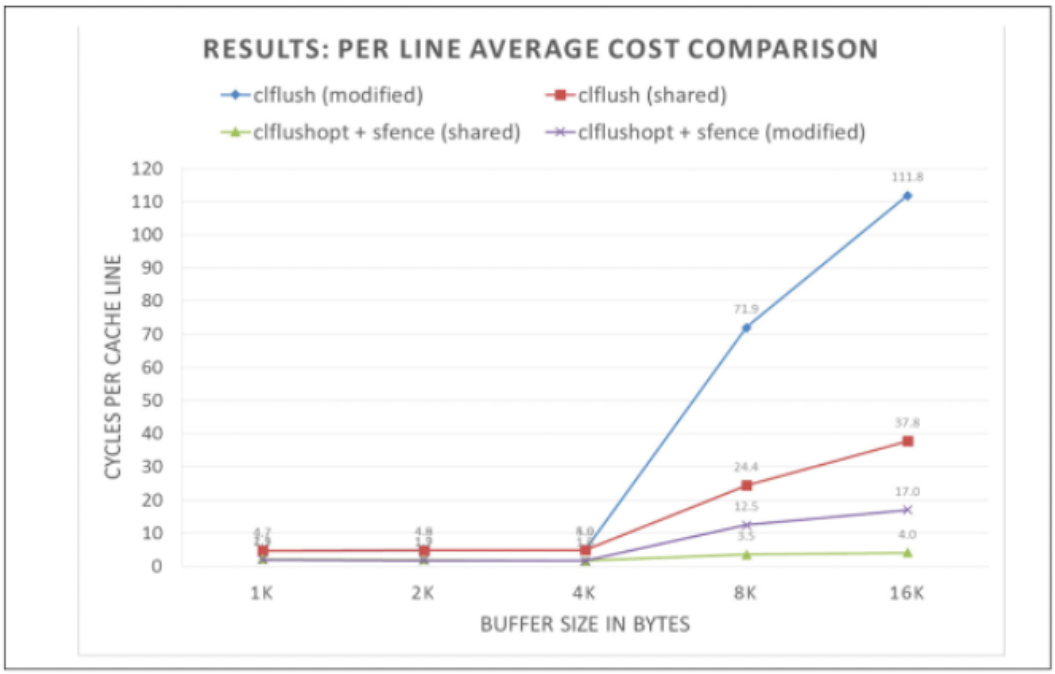


Figure 9-1. CLFLUSHOPT versus CLFLUSH In SkyLake Microarchitecture

User/Source Coding Rule 13. *If CLFLUSHOPT is available, use CLFLUSHOPT over CLFLUSH and use SFENCE to guard CLFLUSHOPT to ensure write order is globally observed. If CLUSHOPT is not available, consider flushing large buffers with CLFLUSH in smaller chunks of less than 4KB.*

Example 9-2 gives equivalent assembly sequences of flushing cache lines using CLFLUSH or CLFLUSHOPT. The corresponding sequence in C are:

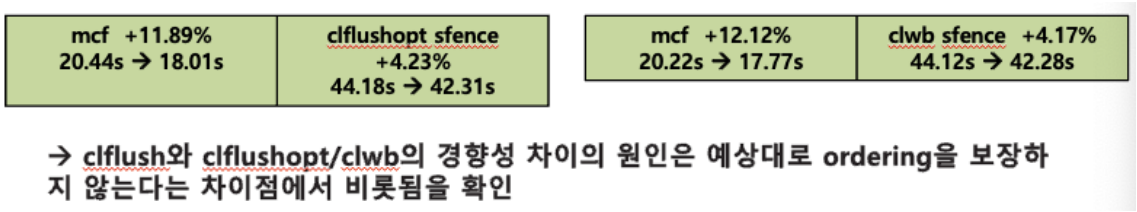
CLFLUSH:

```
For (i = 0; i < iSizeOfBufferToFlush; i += CACHE_LINE_SIZE) _mm_clflush( &pBufferToFlush[ i ] );
```

CLFLUSHOPT:

```
_mm_sfence();
For (i = 0; i < iSizeOfBufferToFlush; i += CACHE_LINE_SIZE) _mm_clflushopt( &pBufferToFlush[ i ] );
_mm_sfence();
```

- 실험 결과는 다음과 같으며, 동일하게 clwb도 clwb instruction 직전 및 직후에 sfence를 넣어서 clwb 간의 ordering을 보장해주었다. 실험 결과를 통해 알 수 있듯이, sfence를 통해 instruction 간 ordering을 보장해줌으로써 clflushopt 및 clwb의 성능 경향성이 clflush와 거의 유사하게 바뀐 것을 확인할 수 있다.



cf) lfence vs. sfence vs. mfence

- lfence (load fence)
  - Performs a serializing operation on all load-from-memory instructions that were issued prior the LFENCE instruction. This serializing operation guarantees that every load instruction that precedes in program order the LFENCE instruction is globally visible before any load instruction that follows the LFENCE instruction is globally visible. The LFENCE instruction is ordered with respect to load instructions, other LFENCE instructions, any MFENCE instructions, and any serializing instructions (such as the CPUID instruction). It is not ordered with respect to store instructions or the SFENCE instruction.
  - `lfence` is documented to order loads prior to the `lfence` with respect to loads after, but this guarantee is already provided for normal loads without any fence at all: that is, Intel already guarantees that "loads aren't reordered with other loads". As a practical matter, this leaves the purpose of `lfence` in user-mode code as an out-of-order execution barrier, useful perhaps for carefully timing certain operations.
- sfence (store fence)
  - Performs a serializing operation on all store-to-memory instructions that were issued prior the SFENCE instruction. This serializing operation guarantees that every store instruction that precedes in program order the SFENCE instruction is globally visible before any store instruction that follows the SFENCE instruction is globally visible. The SFENCE instruction is ordered with respect store instructions, other SFENCE instructions, any MFENCE instructions, and any serializing instructions (such as the CPUID instruction). It is not ordered with respect to load instructions or the LFENCE instruction.
  - `sfence` is documented to order stores before and after in the same way that `lfence` does for loads, but just like loads the store order is already

guaranteed in most cases by Intel. The primary interesting case where it doesn't is the so-called non-temporal stores such as `movntdq`, `movnti`, `maskmovq` and a few other instructions. These instructions don't play by the normal memory ordering rules, so you can put an `sfence` between these stores and any other stores where you want to enforce the relative order. `mfence` works for this purpose too, but `sfence` is faster.

- mfence (memory fence)
  - Performs a serializing operation on all load-from-memory and store-to-memory instructions that were issued prior the MFENCE instruction. This serializing operation guarantees that every load and store instruction that precedes in program order the MFENCE instruction is globally visible before any load or store instruction that follows the MFENCE instruction is globally visible. The MFENCE instruction is ordered with respect to all load and store instructions, other MFENCE instructions, any SFENCE and LFENCE instructions, and any serializing instructions (such as the CPUID instruction).
  - Unlike the other two, `mfence` actually does something: it serves as a full memory barrier, ensuring that all of the previous loads and stores will have completed before any of the subsequent loads or stores begin execution. This answer is too short to explain the concept of a memory barrier fully, but an example would be Dekker's algorithm, where each thread wanting to enter a critical section stores to a location and then checks to see if the other thread has stored something to its location.

랩 미팅 내용 정리

- Intel PM product workload 조사 및 flush instruction 종류/빈도 분석
- LLC 공유 시 PM I/O의 noisy 정도 실험
  - LLC를 더럽히기 위해 더 많은 PM DAX write task들을 배치하게 되면 PM write 과정에서의 경합이 발생해서 PM DAX write task의 stall이 길어질거고, 그러면 자연스레 sibling core에 있는 CPU task의 성능이 좋아지는데 이러면 해석 어려운 것 아닌가?
    → PM DAX write task가 많아진다는 것은 bus mastering으로 인해 write task가 bus를 잡고 있는 시간이 늘어나는거고, 이는 결국 cache를 많이 사용하는 CPU task 입장에서는 bus를 못쓰게 되는 시간이 늘어나는 것이나 다름 없음. 즉 cache 민감한 CPU task는 성능이 매우 안좋아질 것임

- dram(ramdisk)로의 sequential write 실험
    - 만약 PM이 매우 빨라서 ordering을 안지키고 막 flush를 내리는 애들은 stall 시간이 거의 없으니까 sibling core의 integer 워크로드가 성능 저하를 엄청 당하는 거라면, PM보다 더 빠른 dram으로의 I/O는 훨씬 더 큰 폭으로 성능이 악화되는 것이 아닐까?