2022/03/29 ~ 2022/04/07

매일 당일 진행한 연구 내용을 연구일지에 항상 기록하는 것을 목표로 하였으나, 삽질한 시간 대비 명확한 결과물이 나오지 않은 날이 있기도 하고, 일정 기간 동안 연구한 내용을 한 번에 모아서 연구일지를 작성하는 편이 가독성이 좋고 연구일지 작성에 효율적이라고 판단되어, 앞으로는 기간 단위로 연구일지를 작성하려고 한다.

1. SPEC cpu 벤치마크 integer workload 실험

- 기존에는 mcf(정확히는 rate mode인 mcf_r)에 대해서만 실험을 진행하였기 때문에 해당 실험 결과를 integer workload의 경향성이라고 단정짓기에는 근거가 빈약했었음
  → 이에 따라 mcf 이외에도 perlbench, omnetpp, xalancbmk 워크로드들을 rate mode로 실험을 진행하여 아래의 표와 같은 결과를 얻음
- A 2tasks & B 2tasks 실험 : 결과를 측정할 task를 core 2번에, sibling core인 34번 core 및 4, 6번 core에 나머지 task들을 배치

AVX +7.79% 74.06fps → 79.83fps	perlbench -9.40% 39.99s → 43.75s		AVX +6.35% 73.38fps → 78.04fps	mcf -10.76% 19.99s → 22.14s
clflush +9.79% 53.40s → 48.17s	perlbench +5.87% 39.71s → 37.38s		clflush +9.74% 53.58s → 48.36s	mcf +8.79% 20.47s → 18.67s
msync +7.63% 43.53s → 40.21s	perlbench -3.45% 39.72s → 41.09s		msync +6.43% 43.55s → 40.75s	mcf -8.48% 20.40s → 22.13s
AVX +13.69% 73.51fps → 83.57fps	omnetpp -12.43% 117.72s → 132.35s		AVX +1.16% 74.13fps → 74.99fps	xalancbmk -11.69% 124.42s → 138.96s
clflush +10.10% 53.65s → 48.23s	omnetpp +1.45% 109.97s → 108.37s		clflush +10.09% 53.60s → 48.19s	xalancbmk +0.71% 121.88s → 121.02s
msync +9.01% 43.62s → 39.69s	omnetpp -15.58% 109.27s → 126.29s		msync +5.28% 43.52s → 41.22s	xalancbmk -6.00% 120.58s → 127.81s

- A 6tasks & B 6tasks 실험 : 결과를 측정할 task를 core 2번에, sibling core인 34번 core 및 4, 6, 8, 10, 12, 14, 16, 18, 20, 22번 core에 나머지 task들을 배치

AVX +7.80% 73.37fps → 79.09fps	perlbench -10.10% 40.01s → 44.05s		AVX +6.62% 72.96fps → 77.79fps	mcf_r -11.41% 20.59s → 22.94s
clflush +13.40% 56.28s → 48.74s	perlbench +4.53% 39.72s → 37.92s		clflush +13.20% 56.83s → 49.33s	mcf_r +4.44% 20.03s → 19.14s
msync +19.46% 54.47s → 43.87s	perlbench -6.85% 39.73s → 42.45s		msync +13.98% 56.81s → 48.87s	mcf_r -17.70% 20.11s → 23.67s
AVX +14.47% 72.75fps → 83.28fps	omnetpp -11.90% 127.27s → 142.42s		AVX +12.67% 71.64fps → 80.72fps	xalancbmk -17.75% 118.00s → 138.95s
clflush +12.33% 56.36s → 49.41s	omnetpp -5.73% 126.10s → 133.33s		clflush +9.91% 55.78s → 50.25s	xalancbmk -8.00% 118.54s → 128.02s
msync +20.98% 55.35s → 43.74s	omnetpp -19.01% 127.56s → 151.81s		msync +19.31% 52.73s → 42.55s	xalancbmk -13.41% 117.39s → 133.13s

- 결과 분석
  - 전체적인 경향성이 integer workload의 종류와 상관없이 일관성 있게 나오는 결과를 확인 가능
    → integer workload의 경향성으로 생각해도 무방
    - AVX와 integer workload는 sibling core에 배치할 경우 overall 성능에서 별다른 이점을 얻지 못함
    - PM DAX로의 IO의 경우 clflush로 sync를 진행할 경우에는 overall 성능에서 큰 이점을 얻는 반면, msync로 sync를 진행할 경우에는 overall 성능에서 별다른 이점을 얻지 못함
      → clflush와 msync의 차이점 분석 필요!
  - A 6tasks & B 6tasks 실험 결과 중 sibling core에 clflush로 sync를 진행하는 PM DAX로의 I/O task가 있는 결과들에서 omnetpp 및 xalancbmk의 실험결과가 perlbench 및 mcf의 실험결과와 반대 성향을 보여줌
    → omnetpp 및 xalancbmk가 perlbench 및 mcf와 어떤 차이가 있는지 분석 필요!

2. clflush 워크로드와 msync 워크로드의 차이점 분석 - 가설1 : interrupt 관리

- clflush는 아키텍처 instruction으로 직접 cache line flush를 진행하기 때문에 스케줄러에 의해 task가 계속 running state에 머물게 됨
- msync는 syscall이므로 flush 과정에서 interruptable 상태로 전환됨
  - 과거 linux 3에서는 core 0이 global interrupt 관리를 담당했는데, 현재는 각 core 별로 담당하는 것으로 역할 담당이 분산된 것으로 파악됨
  - 이에 따라 혹시 msync가 flush를 하면서 interrupt 상태로 빠지게 되면, 해당 core가 idle 상태가 되는 것이 아닌 interrupt 관리를 담당하게 되어서 sibling core 입장

에서 충분히 자원을 사용하지 못하고 방해를 받을 수도 있음
→ 이 가설이 맞는지 간접적으로 확인하기 위해 proc stat을 통해 interrupt 수 및 context switch 수의 변화를 core 별로 파악해봐야 함!

- Core 별 interrupt 정보는 /proc/interrupts에 저장되며, 이 값은 부팅 시 메모리에 올라와서 shut down 될 때까지 값이 누적됨
  → 즉, 재부팅 시에만 /proc/interrupts에 저장된 core 별 interrupt 정보가 초기화 됨
- A 2tasks & B 2tasks 실험 중, clflush와 msync가 각각 sibling core에 위치할 때의 결과가 극단적으로 차이 나는 omnetpp 실험에 대해 10초 간격으로 /proc/interrupts의 값을 프로파일링하여 의미 있는 interrupt 값들만 변화량을 아래의 표와 같은 형태로 정리함

	omnetpp & clflush				omnetpp & msync			
Core 2 & Core 34	omnet & omnet	omnet & clflush	clflush & omnet	clflush & clflush	omnet & omnet	omnet & msync	msync & omnet	msync & msync
NMI	1 1	1 1	1 2	2 1	1 1	1 1	1 1	1 1
LOC	2700 2703	2702 2503	2503 2700	2504 2504	2700 2699	2698 2503	2503 2699	2443 2503
PMI	1 1	1 1	1 2	2 1	1 1	1 1	1 1	1 1
IWI	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0
RES	0 0	0 0	0 0	0 0	1 0	1 0	0 0	0 0
CAL	1 0	1 1	1 0	1 1	1 0	1 1	1 0	2 2
TLB	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0
MCP	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0

NMI : Non-maskable interrupts
LOC : Local timer interrupts
PMI : Performance monitoring interrupts
IWI : IRQ work interrupts
RES : Rescheduling interrupts
CAL : Function call interrupts
TLB : TLB shootdowns
MCP : Machine check polls

- 결과 분석
  - 약 17%의 성능 차이에 비하면 10초간 프로파일링한 interrupt 수의 변화량의 차이는 영향이 없는 것으로 판단될 정도로 작음
    → Interrupt 관리 가설은 틀린 것으로 판단되며, ftrace를 이용해서 function call stack을 확인해서 clflush와 msync의 차이점을 분석해야 함 (omnetpp의 경우

user-level 프로파일링을 해야하므로 ftrace로 큰 차이를 발견할 수 없겠지만, clflush와 msync는 kernel-level 프로파일링 tool인 ftrace로 충분히 유의미한 결과를 발견할 수 있을 것으로 생각됨)

3. ftrace 프로파일링 및 결과 분석

- A 2tasks & B 2tasks 실험 중 omnetpp의 clflush, msync 실험에 대해 10초 동안 ftrace로 function trace 기록을 프로파일링 함
  → core 2와 core 34의 기록만 확인하면 되므로 /sys/kernel/tracing/trace_stat에 생성되는 function2 파일과 function34 파일만 확인하면 됨 (ftrace를 실행시킬 때마다 /sys/kernel/tracing/trace_stat 디렉토리의 파일들은 초기화되므로 필요한 function 파일은 다른 디렉토리로 복사해둬야함)
  - 참고) ftrace 프로파일링 하는 방법
    아래의 3개의 command를 순서대로 실행하여 ftrace를 실행시킬 준비를 함

    ```
    echo 0 > /sys/kernel/debug/tracing/tracing_on
    ```

    ```
    echo 1 > /sys/kernel/debug/tracing/tracing_on
    ```

    ```
    echo 1 > /sys/kernel/debug/tracing/options/sleep-time
    ```

    아래의 command 중 위 command는 ftrace 프로파일링 시작, 아래 command는 ftrace 프로파일링 중지 command임

    ```
    echo 1 > /sys/kernel/debug/tracing/function_profile_enabled
    ```

    ```
    echo 0 > /sys/kernel/debug/tracing/function_profile_enabled
    ```

- 프로파일링을 진행하면 다음과 같이 각 kernel function 별로 얼마 만큼의 시간이 소모되었는지 확인할 수 있으며, 내림차순으로 정렬되어 있기 때문에 위쪽 함수부터 call path를 타고 들어가는 방식으로 분석 가능함 (아래의 결과는 omnetpp가 core2번, msync가 core 34번에서 실행되는 실험에서의 msync의 ftrace 결과인 function34 파일임)

Function	Hit	Time	Avg	s^2
handle_mm_fault	178861	5661945 us	31.655 us	9.458 us
__handle_mm_fault	178861	5505083 us	30.778 us	9.175 us
do_wp_page	178541	5413691 us	30.321 us	7.470 us
ext4_dax_fault	178542	5316507 us	29.777 us	7.284 us
ext4_dax_huge_fault	178542	5292944 us	29.645 us	7.264 us
dax_iomap_fault	178542	4362680 us	24.435 us	3.714 us
dax_iomap_pte_fault	178542	4340608 us	24.311 us	3.704 us
schedule	6	3999807 us	666634.5 us	206693913162 us
__x64_sys_msync	178542	3689567 us	20.664 us	3.333 us
vfs_fsync_range	178542	3490412 us	19.549 us	3.154 us
ext4_sync_file	178542	3467984 us	19.423 us	3.132 us
dax_fault_iter	178542	3438631 us	19.259 us	2.870 us
vmf_insert_mixed_mkwrite	178542	3039375 us	17.023 us	2.521 us
__vm_insert_mixed	178542	3017019 us	16.898 us	2.501 us
track_pfn_insert	178542	2794245 us	15.650 us	2.295 us
lookup_memtype	178542	2742173 us	15.358 us	2.236 us
pat_pagerange_is_ram	178542	2540192 us	14.227 us	2.075 us
walk_system_ram_range	178542	2514923 us	14.085 us	2.059 us
find_next_iomem_res	178542	2489264 us	13.942 us	2.041 us
file_write_and_wait_range	178542	1677265 us	9.394 us	1.260 us
filemap_fdatawrite_wbc	178542	1491657 us	8.354 us	1.098 us
do_writepages	178542	1278242 us	7.159 us	0.941 us
blkdev_issue_flush	178542	1240830 us	6.949 us	1.223 us
ext4_dax_writepages	178542	1225382 us	6.863 us	0.899 us
next_resource.part.0	18747001	1135867 us	0.060 us	0.006 us
dax_writeback_mapping_range	178542	1078268 us	6.039 us	0.764 us
submit_bio_wait	178542	948383.3 us	5.311 us	0.978 us
submit_bio	178542	756593.9 us	4.237 us	0.820 us
submit_bio_noacct	178542	734315.8 us	4.112 us	0.804 us
__submit_bio	178542	706077.5 us	3.954 us	0.796 us
dax_entry_mkclean	178542	549288.1 us	3.076 us	0.379 us
iomap_iter	357085	538065.8 us	1.506 us	1.867 us
ext4_iomap_begin	178542	468338.8 us	2.623 us	0.332 us
pmem_submit_bio	178542	404260.3 us	2.264 us	0.606 us
__ext4_journal_start_sb	181044	343892.8 us	1.899 us	0.234 us

- 위의 ftrace 결과를 바탕으로 아래와 같은 call path를 얻어낼 수 있음
  - **SYSCALL_DEFINE3**(mm/msync.c) → **vfs_fsync_range**(fs/sync.c)
    → **ext4_file_operations**(fs/ext4/file.c, vfs_fsync_range 함수 내부의 fsync에서 cscope ctrl+\+g를 통해 fsync operation이 정의된 위치를 찾으면 현재 위치가 나옴)
    → **ext4_sync_file**(fs/ext4/fsync.c) →
    file_write_and_wait_range(mm/filemap.c)
    → **__filemap_fdatawrite_range**(mm/filemap.c)
    → **filemap_fdatawrite_wbc**(mm/filemap.c) → **do_writepages**(mm/page-writeback.c)
    → **ext4_dax_aops**(fs/ext4/inode.c, cscope ctrl+\+g를 통해 writepages operation이 정의된 위치를 파악해보면 ext4의 경우 fs/ext4/inode.c 파일에 해당 operation이 정의되어 있으며, 현재 우리는 DAX mode를 사용하고 있으므로 ext4_dax_aops에 있는 writepages operation을 참조하면 됨. 추가적으로 만약 writepages operation이 정의되지 않았다면, do_writepages에서 mapping→a_ops→writepages가 아니라 generic_writepages가 실행되었을 것)
    → **ext4_dax_writepages**(fs/ext4/inode.c) →

dax_writeback_mapping_range(fs/dax.c)
→ **dax_writeback_one**(fs/dax.c) → **dax_flush**(drivers/dax/super.c)
→ **arch_wb_cache_pmem**(arch/x86/lib/usercopy_64.c)
→ **clean_cache_range**(arch/x86/lib/usercopy_64.c) →
clwb(arch/x86/lib/usercopy_64.c)

- ftrace 결과를 통해 얻어낼 수 있는 결과
  - data의 flush는 최종적으로 x86 아키텍처 의존적인 clwb instruction에 의해서 수행됨
    → CLWB와 clflush의 차이점을 알아보고, clflush 대신 CLWB를 사용하여 flush를 진행하는 워크로드 작성 후 실험을 진행하여 msync로 인한 integer workload의 성능 저하가 CLWB 때문인지를 확인
  - ftrace 결과에 __ext4_journal_start_sb와 같은 저널링 관련 함수들이 존재하며, Hit 횟수 및 Time 값을 통해 실제로 metadata의 저널링이 진행되는 것을 유추할 수 있음
    → msync에서 metadata의 경우 jbd2를 통해 저널링을 진행함을 알 수 있음. 따라서 msync의 성능이 느려지더라도 jbd2 및 metadata 저널링 메모리를 실험을 돌리는 NUMA node가 아닌, 다른 NUMA node(NUMA node 1)에 배치해서 저널링 요소를 배제한 실험 필요