Advanced OS - Lab 1

Dinesh Reddy Chinta; dc47444

February 17, 2023

1 Abstract

In this lab we understand System Memory modules through experiments using Peformonce Monitoring Units. We interface with the OS using "mmap" system call and explore the behaviour of the OS by running various configurations of mmap and running workloads with controlled memory access patterns. We gather essential statistics like L1-data cache, L1-data TLB, utime, stime etc to study the behaviour.

2 Setup and Specifications

2.1 Host Hardware - Software Specifications

I used a server provided by Cloudlab infrastructure of machine type rs620, having Xeon E5-2660 processors. The Host runs Ubuntu Ubuntu 20.04 LTS Operating system with a Linux 5.4.0-100 kernel version.

All experiments except one have been performed on this machine. It's PMU is equipped with hardware counters that support most of our Dcache and TLB measurements except prefetching. I havn't reported any prefetching measurements.

For the last experiment, which require kernel modifications to "mm/vmscan.c", I used the VM build in last lab.

Listing 1: Memory

```
# getconf -a | grep CACHE
...
LEVEL1_DCACHE_SIZE                      32768
LEVEL1_DCACHE_ASSOC                     8
LEVEL1_DCACHE_LINESIZE                  64

# sudo lshw -class memory
...
*-memory
     ...
   size: 128GiB
...
# sudo x86info -c
...
 Data TLB: 4KB pages, 4-way associative, 64
      entries
 Data TLB: 4K pages, 4-way associative, 512
      entries.
...
```

Listing 2: Software

```
# lsb_release -a
...
Description:    Ubuntu 20.04 LTS
...
# uname -r
5.4.0-100-generic
```

Listing 3: Guest Config

```
# cat /etc/os-release
NAME=Buildroot
VERSION=2022.11-1026-g5ec326fb97
ID=buildroot
VERSION_ID=2023.02-git
PRETTY_NAME="Buildroot 2023.02-git"

# uname -a
Linux buildroot 6.1.7 #3 SMP PREEMPT_DYNAMIC Thu Jan 26 21:20:51
CST 2023 x86_64 GNU/Linux
```

2.2 PMU capabilities

Listing 4: PMU capabilities

```
sudo perf list | grep "Hardware"
L1-dcache-load-misses     [Hardware cache event]
L1-dcache-loads           [Hardware cache event]
L1-dcache-prefetch-misses [Hardware cache event]
L1-dcache-store-misses    [Hardware cache event]
L1-dcache-stores          [Hardware cache event]
...
dTLB-load-misses          [Hardware cache event]
dTLB-loads                [Hardware cache event]
dTLB-store-misses         [Hardware cache event]
dTLB-stores               [Hardware cache event]
...
```

2.3 Checking exit status for system calls

I checked for the exit status for all system calls in my code. Please feel free to look into the code (and request for permissions) in my git repository.

3 Memory Map

Each line in the file represents a single memory mapping, and it contains the following information:

- Start and end addresses: The start and end addresses of the memory mapping.
- Permissions: The read, write, and execute permissions for the mapping.
- Offset: The offset into the file or device that the mapping originates from.
- Device: The device major and minor number that the mapping originates from.
- Inode: The inode number of the file or device that the mapping originates from.
- Pathname: The name of the file or device that the mapping originates from, if available.

The base address: 0x55b251de1000 The start address of libc: 0x7fbee36ed000

They are significantly different because libc is a shared library, where as the base address corresponds to the virtual address of the text segment that is private to the process.

Listing 5: mmap output

```
55b251de1000-55b251de2000 r--p 00000000 08:01 680183                    /users/dineshc/adv_os-lab1/bin/
        test_my_perf_event
55b251de2000-55b251de5000 r-xp 00001000 08:01 680183                    /users/dineshc/adv_os-lab1/bin/
        test_my_perf_event
55b251de5000-55b251de7000 r--p 00004000 08:01 680183                    /users/dineshc/adv_os-lab1/bin/
        test_my_perf_event
55b251de7000-55b251de8000 r--p 00005000 08:01 680183                    /users/dineshc/adv_os-lab1/bin/
        test_my_perf_event
55b251de8000-55b251de9000 rw-p 00006000 08:01 680183                    /users/dineshc/adv_os-lab1/bin/
        test_my_perf_event
55b2531b7000-55b2531d8000 rw-p 00000000 00:00 0                         [heap]
7fbee36ed000-7fbee370f000 r--p 00000000 08:01 39718                     /usr/lib/x86_64-linux-gnu/libc
        -2.31.so
7fbee370f000-7fbee3887000 r-xp 00022000 08:01 39718                     /usr/lib/x86_64-linux-gnu/libc
        -2.31.so
7fbee3887000-7fbee38d5000 r--p 0019a000 08:01 39718                     /usr/lib/x86_64-linux-gnu/libc
        -2.31.so
7fbee38d5000-7fbee38d9000 r--p 001e7000 08:01 39718                     /usr/lib/x86_64-linux-gnu/libc
        -2.31.so
7fbee38d9000-7fbee38db000 rw-p 001eb000 08:01 39718                     /usr/lib/x86_64-linux-gnu/libc
        -2.31.so
7fbee38db000-7fbee38e1000 rw-p 00000000 00:00 0
7fbee38e9000-7fbee38ea000 r--p 00000000 08:01 39702                     /usr/lib/x86_64-linux-gnu/ld-2.31.
        so
7fbee38ea000-7fbee390d000 r-xp 00001000 08:01 39702                     /usr/lib/x86_64-linux-gnu/ld-2.31.
        so
7fbee390d000-7fbee3915000 r--p 00024000 08:01 39702                     /usr/lib/x86_64-linux-gnu/ld-2.31.
        so
7fbee3916000-7fbee3917000 r--p 0002c000 08:01 39702                     /usr/lib/x86_64-linux-gnu/ld-2.31.
        so
7fbee3917000-7fbee3918000 rw-p 0002d000 08:01 39702                     /usr/lib/x86_64-linux-gnu/ld-2.31.
        so
7fbee3918000-7fbee3919000 rw-p 00000000 00:00 0
7ffdc2404000-7ffdc2425000 rw-p 00000000 00:00 0                         [stack]
7ffdc2555000-7ffdc2558000 r--p 00000000 00:00 0                         [vvar]
7ffdc2558000-7ffdc2559000 r-xp 00000000 00:00 0                         [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                 [vsyscall]
```

4 Getrusage

The struct rusage object contains a number of fields that provide detailed information about the resource usage of the process.

- ru_utime: The amount of CPU time used by the process in user mode.
- ru_stime: The amount of CPU time used by the process in system mode.
- ru_maxrss: The maximum resident set size (in kilobytes) used by the process. This is the maximum amount of physical memory that the process has used at any point during its execution.
- ru_ixrss, ru_idrss, ru_isrss: The amount of shared memory, unshared data, and unshared stack memory used by the process, respectively.
- ru_minflt, ru_majflt: The number of minor and major page faults.
- ru_nswap: The number of times the process has been swapped out to disk.
- ru_inblock, ru_oublock: The number of input and output operations performed on block devices.
- ru_nsignals: The number of signals received by the process. ru_nvcsw, ru_nivcsw: The number of voluntary and involuntary context switches performed by the process.

Listing 6: getrusage output

```
User CPU time: 18.042895 seconds
System CPU time: 0.363977 seconds
Maximum resident set size: 1050104 kB
Page reclaims (soft page faults): 262250
Page faults (hard page faults): 0
Block input operations: 0
Block output operations: 0
Voluntary context switches: 0
Involuntary context switches: 75
Signals received: 0
IPC messages sent: 0
IPC messages received: 0
Number of times the process was swapped out of memory: 0
Block I/O operations: 0
Integral shared memory size: 0 kB * clock ticks
Integral unshared data size: 0 kB * clock ticks
Integral unshared stack size: 0 kB * clock ticks
```

5 perf_event_open

5.1 syscall

There is no opcode called syscall in the objdump output.

Instead, in the objdump output, we see that syscall is linked to glibc library routines. I believe that, inside the glibc compile binaries, there would an interrupt instruction, “int $0x80”, to trap into the kernel and execute the corresponding system-call in the kernel mode.

In our code, as shown in Listing 7, I am calling the libc provided syscall function to implement perf_event_open function.

In Listing 8, You can notice the assembly instructions corresponding to loading registers with the system call paramters and then calling syscall function. Inside syscall, we can notice a call to the linked libc syscall function.

Listing 7: perf_event_open function

```
long long my_perf_event_open(struct perf_event_attr *hw_event, pid_t pid, int cpu, int group_fd,
     unsigned long flags)
{
  int ret;
  ret = syscall(__NR_perf_event_open, hw_event, pid, cpu, group_fd, flags);

  return ret;
}
```

Listing 8: diassembed pref_event_open

```
# objdump -d
...
0000000000001330 <syscall@plt>:
    1330:       f3 0f 1e fa             endbr64
    1334:       f2 ff 25 35 5c 00 00    bnd jmpq *0x5c35(%rip)        # 6f70 <syscall@GLIBC_2.2.5>
    133b:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
...
00000000000021ee <my_perf_event_open>:
    21ee:       f3 0f 1e fa             endbr64
    21f2:       55                      push   %rbp
    21f3:       48 89 e5                mov    %rsp,%rbp
    21f6:       48 83 ec 30             sub    \$0x30,%rsp
    21fa:       48 89 7d e8             mov    %rdi,-0x18(%rbp)
    ...
    2207:       4c 89 45 d0             mov    %r8,-0x30(%rbp)
    220b:       48 8b 7d d0             mov    -0x30(%rbp),%rdi
    ...
    221c:       49 89 f9                mov    %rdi,%r9
    221f:       41 89 f0                mov    %esi,%r8d
    2222:       48 89 c6                mov    %rax,%rsi
    2225:       bf 2a 01 00 00          mov    \$0x12a,%edi
    222a:       b8 00 00 00 00          mov    \$0x0,%eax
    222f:       e8 fc f0 ff ff          callq  1330 <syscall@plt>
    2234:       89 45 fc                mov    %eax,-0x4(%rbp)
    2237:       8b 45 fc                mov    -0x4(%rbp),%eax
    223a:       48 98                   cltq
    223c:       c9                      leaveq
    223d:       c3                      retq
```

Listing 9: using perf_event infrastructure

```
#include <linux/perf_event.h>

...

// structure to represent a PMU counter
struct perf_event_attr pe_l1_rd_acc = {
    .type = PERF_TYPE_HW_CACHE,
    .config = (PERF_COUNT_HW_CACHE_L1D
               | (PERF_COUNT_HW_CACHE_OP_READ << 8)
               | (PERF_COUNT_HW_CACHE_RESULT_ACCESS << 16)) , // totoal number of access
    .size = sizeof(struct perf_event_attr),
    .disabled = 1,
    .exclude_kernel = 1,
    .exclude_hv = 1
};

...

// open the counter
fd_l1_rd_acc = my_perf_event_open(&pe_l1_rd_acc, 0, -1, -1, 0);

...

// control the counter
ioctl(fd_l1_rd_acc, PERF_EVENT_IOC_RESET, 0);

...

// read the measurements
read(fd, &count, sizeof(long long));
```

5.2 Memory Access Pattern

The do_mem_access function accesses the memory in the buffer pointer at byte granularity by selecting one contiguous 512B memory blocks at once, section each contiguous block is random or sequential based on the user option we provide. Once it selects one 512B block, its behavior is the same for both random and sequential options. For each of the 512B blocks, it repeats the accesses 16 times to capture locality. Each time it iterates, it sequentially accesses the 512B memory with the first byte of a cache line with a store operation and the subsequent bytes as load operations.

5.3 Random Access Generation

It is a linear feedback shift register (LFSR) generator that produces a sequence of pseudorandom numbers. The algorithm works by performing a series of bitwise operations (shifts and XORs) on a set of four variables x, y, z, and w, which are updated on each call to the simplerand function.

The XOR operations and bit shifts used in the simplerand function are designed to produce a sequence of values that exhibit good statistical properties and are well-distributed over a large range of values. In particular, the shift operations help to spread the influence of each input bit across multiple output bits, while the XOR operations help to combine the effects of multiple input bits.

5.4 strace

Run your program under strace. Enter the output for arch$_{prctlinyourreportandexplainwhatthissystemcalldoes.Put}$

Listing 10: strace arch_prctl & /etc/ld.so.preload

```
...
execve("bin/test_proc", ...
brk(NULL)                              = 0x55d908170000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffdaeba7f10) = -1 EINVAL (Invalid argument)
access("/etc/ld.so.preload", R_OK)...
openat(AT_FDCWD, "/etc/ld.so.cache", ...
...
mmap(...
close(3)
arch_prctl(ARCH_SET_FS, 0x7f1f33a4b540)...
mprotect(...
...
```

The arch_prctl(ARCH_SET_FS, 0x7f1f33a4b540) system call is setting the thread-local storage (TLS) base address for the current process.

The ARCH_SET_FS argument specifies that we want to set the value of the FS register, which is a general-purpose register used for storing the TLS base address. The second argument, 0x7f1f33a4b540, is the new value for the x86 FS register.

By setting the TLS base address in this way, the process can use the FS register to access thread-local data. This is a mechanism that allows each thread in a process to have its own copy of data that is stored in a specific location in memory. The TLS base address provides a reference to the start of the thread-local storage area, and the process can use this to access its thread-local data without having to worry about interfering with other threads.

The "/etc/ld.so.preload" is part of an advanced Linux functionality to override the default libraries with custom libraries by writing an entry into the file.

6 Isolating the experiments

Isolating the CPU is essential to study the behaviour of our program. There are two ways to isolate a CPU.

- taskset all current process to run other cores.
- while booting the kernle, isolate a cpu using isolcpu option and run onnly our task on the cpu.

I used the first option because the first option required controlling kernel boot options.

I also flushed the L1D cache and L1D TLB as showin in Listing 11.

6.1 flusing Data and TLB cache

Listing 11: L1D cache flush

```
void flush_l1_data_cache(void) {
    // Allocate a buffer that is larger than the L1 data cache size
    const int cache_size = 32768;  // a 32KB L1 data cache
    char* buffer = (char*) malloc(cache_size * 2);
    if (buffer == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }

    memset(buffer, 0, cache_size * 2);
    // Touch every page of the buffer to ensure it is loaded into the cache
    for (int i = 0; i < cache_size * 2; i += 4096) {
        if (buffer[i] != 0) {
            perror("touch");
            exit(EXIT_FAILURE);
        }
    }

    // Write to the buffer to force the cache to evict its contents
    for (int i = 0; i < cache_size; i++) {
        buffer[i] = i % 256;
    }

    // Free the buffer to release the memory
    free(buffer);
```

Device	Operation	Hits	Misses	Total	Miss-Rate%
L1D	READ	16645660050	257725814	16903385864	1.52
L1D	WRITE	6414354060	51390573	6465744633	0.79
DTLB	READ	16906513698	98368	16906612066	0.00
DTLB	WRITE	6461392399	2185452	6463577851	0.03

Table 1: PMU reading for Anonymous Memory Mapped buffer.

```
}

void flush_l1_dtlb(void){
    asm volatile ("wbinvd");
}
```

6.2 Isolating CPU

We can lock the process to CPU0 with sched_setaffinity() system call; to flush the cache, I sequentially accessed the 32KB of memory; to flush data TLB, I used "wbinvd" assembly instruction.

sched_setaffinity(0, sizeof(cpu_set_t), cpuset) == -1)

6.3 L1D and L1DTLB misses

It is interesting to note the number of loads and stores from the Table 1. Based on my calculations, the number of observed stores are much greater than expected number of 2 billion. However, the number of loads are close to the expected 16 billion. I expected both the loads and store to be lower than the actual number as the Miss Status Hold Register (MSHR) can coalace multiple independant load/store instructions.

7 plots

Each plot has 9 experiments.

- ano: Anonymous Memory Mapping
- ano_shr: Anonymous Shared Memory Mapping
- fs: File backed Shared Mapping
- fsrand: File backed shared mapping doing random access
- fspop: File backed shared mapping with populate flag set
- fsmset: File backed shared mapping with Msync and Memset
- comp_ano: Anonyous Memory mapping in the precense of a competing process.
- comp_fs: File backed Memory mapping in the presense of a competing process.
- comp_fspop: File backed memory mapping wiht population flag set in the presense of a competing process.

8 Analysis, Comparision & Discussions

8.1 Anonymous vs File Backed

In this comparision, I considered Anonymous MMAP and File-backed shared MMAP.

From the plots, we can notice that L1D cache miss rate increased for READs but it decreased for WRITES. This is interesting! In case of File backed memory, the number of context switches has slightly increased. Although we tried to reduce the interferance of other processes, we still didn't

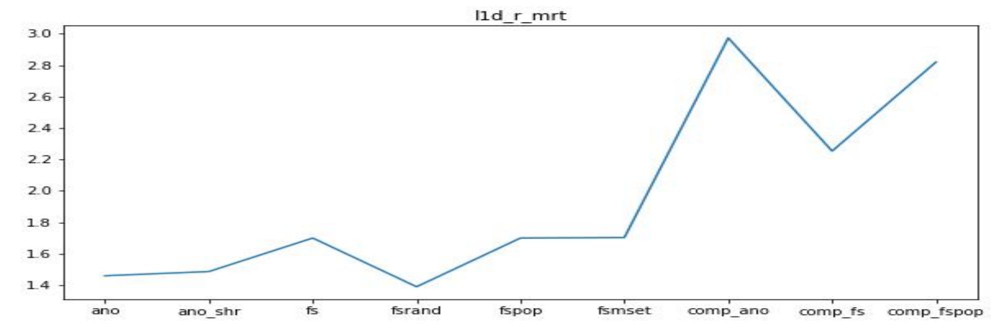

Figure 1: L1 Data Cache Read

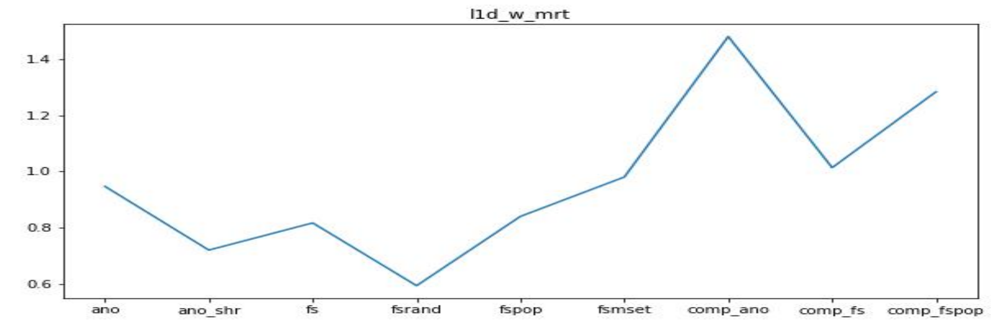

Figure 2: L1 Data Cache Write

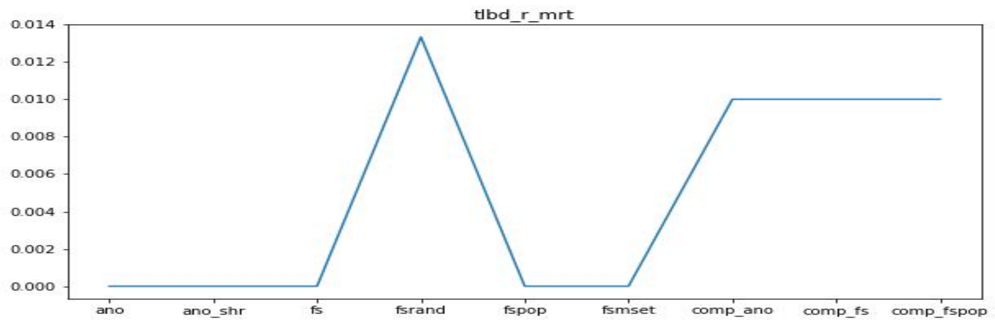

Figure 3: L1 TLB Data Cache Read

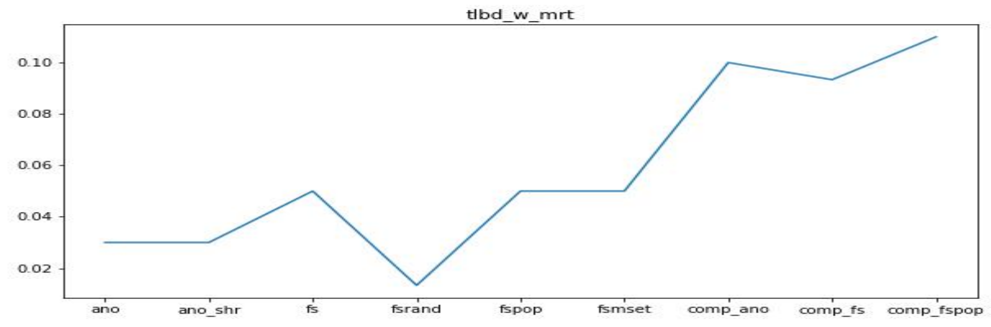

Figure 4: L1 TLB Data Cache Write

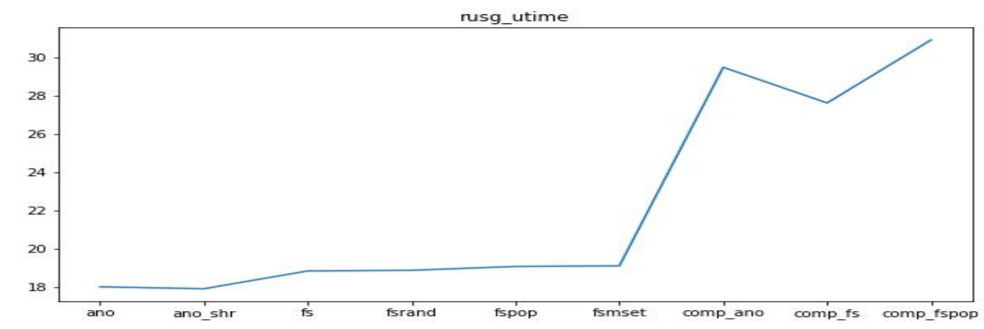

Figure 5: User Time

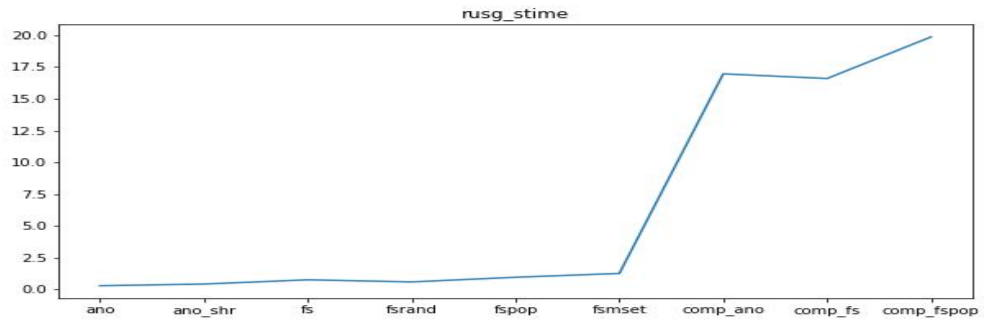

Figure 6: System Time

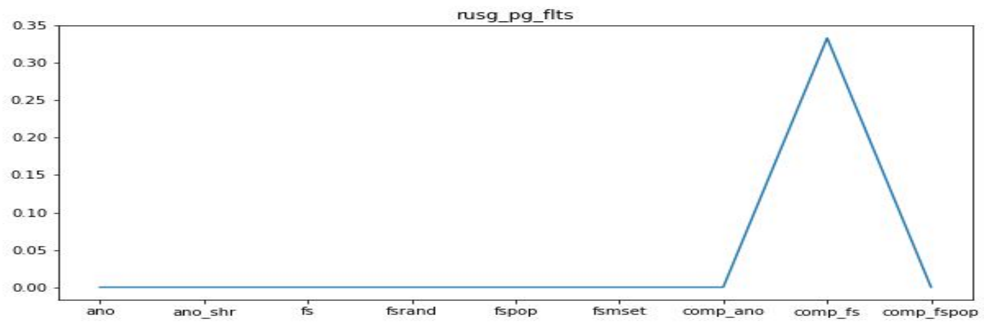

Figure 7: Page Faults

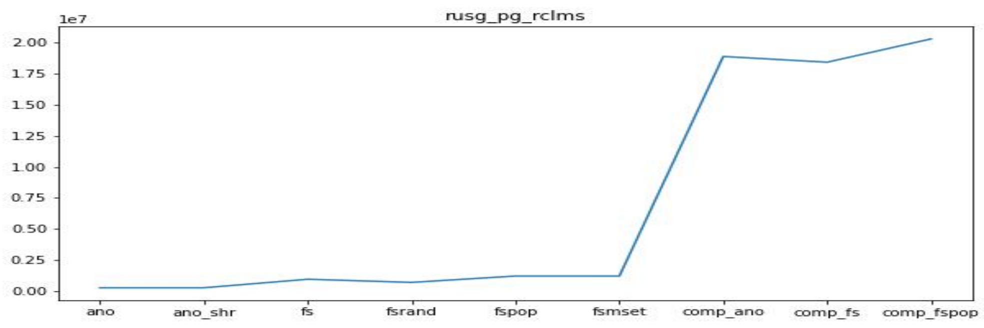

Figure 8: Page Reclaims

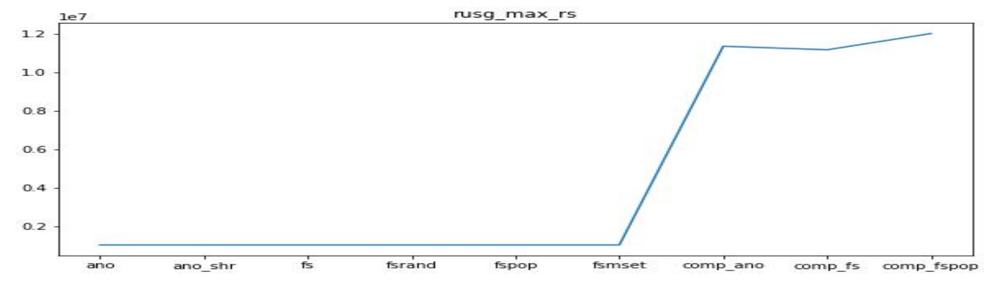

Figure 9: Max Resident Memory Size

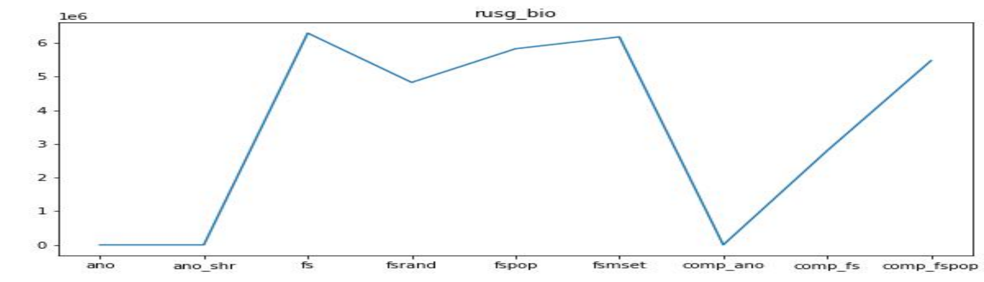

Figure 10: Block Input-Output

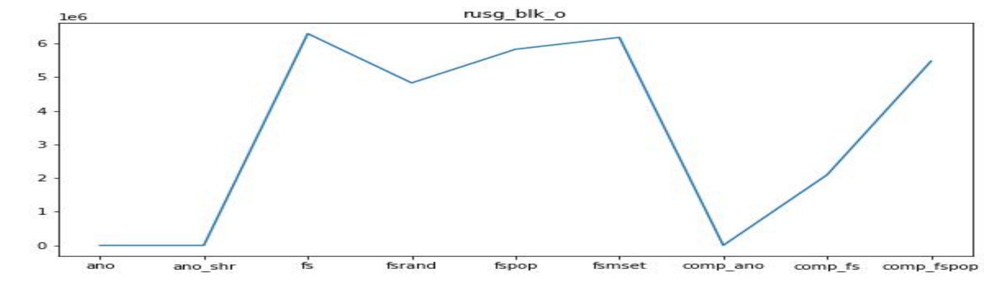

Figure 11: Block Output Operations

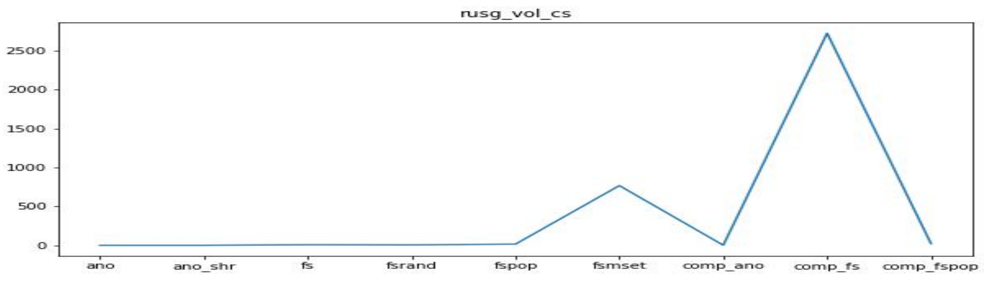

Figure 12: Volentary Context Switches

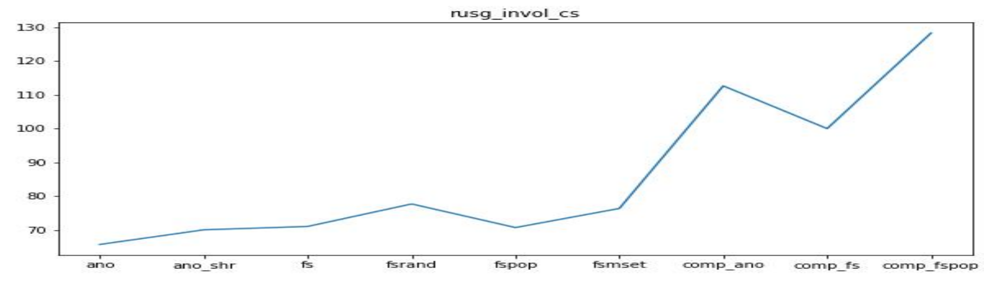

Figure 13: Involentary Context Switches

address the interferance of kernel mode working set polluting the cache. Hence, the number of read misses increased.

For stores, we have 1/16 of them to be compulsory misses. This can't be decreased. In case of file-backed mmap, we write a page to disk; and while doing this, we flush the contents of the cache. I suspect that this reduces the cache contention and hence increases the effective useful residant size in the L1D cache.

The TLB misses were small enough that they did not change.

User time: There are slight increaments in User time. I think this is because of the increased number of L1D cache and other cache misses (L2, LLC).

Block Operations: Becuse of the file backed memory, the number of block operations increased significantly. We can also see this impacting the page reclaims proportionately.

This incresed block operations slightly increased system time owing to the extra file operations. We should note that it did not increase the volantery context switches. This is because the process actually writes to block cache instead of the device itself. Hence the hard-disk latency is hidden.

The number of involantary context switches increased slightly. We can attribute this to the increase in the overall execution time (utime + stime) – the larger the total time, the larger the context switches.

8.2 Sequential vs Random

We run the random access with File Backed Memory with private pages and none of the other flags set.

For Data Cache, we notice an interesting result. Both the data read and write misses decrease by a significant amount. This has an explonation in the block operations measurement. The total number of block operations reduced by over 20% from the file-shared experiemnt (Figure 10). This means, the total working set of the application is also 20% smaller; and there was a 25% reuse in the memory. I believe that this improved reuse has contributed to the reduce in the L1D miss rates.

Apart from reduced I/O and resultantly reduced L1D miss rates, every other measurement is similar to the file shared one except the involentary context switches. I am unable to explain why the involentary context switches increased; we may need further experements to understand this.

8.3 Private vs Shared

I performed the experiment between anonymous-private and anonymous-shared memory mapping. I did not expect the L1D write access to decrease. Indeed, I expected to be exactly same (or similar) to anonymous-private case. Further experiments need to be done to explain the scenario.

8.4 Impact of Populate

I expected the number of page reclaims to become zero after setting the flag. However, I convinced myself that even though we populated the file, the file doesn't have any data – it's inode table has negligible entries. Hence, the populate flag did not have any affect.

I suspect that, if another thread has created the file-backed shared memory, written some data to it, and memory synced the file to hard-disk then the current thread would have read the blocks into the memory, reducing the number of soft faults. However, this is just a speculation and needs further experiments and investigation.

Most of the other metrics remained similar to file-shared experiment. This is consistant with our expectations.

8.5 Impact of Memset

This is a very tricky experiment. It is intuitive to expect the memset to increase the number of input block operations, the input block operations remained zero. I suspect that the OS cached the blocks and hence we did not notice any input I/O traffic.

While we don't have any I/O traffic, the volantary context switches increased by an order of magnitude. I again impute this to the increase in contention at the block device. Though we don't need to I/O we might have to context switch for the data to be put into the page cache.

I suspect that, if we have made the process sleep for some time after doing msync and started our experiment, we would have noted a clear difference in the input block operations atleast.

9 Compete for memory

9.1 setup

I did two different experiments, with different setups – one on Cloudlab and the second on Lab0 VM, to complete this section. By running on Cloudlab, I have got satisfactory and accurate measurements for L1 caches, memory and I/O because of the fairly isolated environment. Hence, I used the same setup to acquire as many meaningful experiments as possible. I tried to boot up my kernel on Clould lab by following this wiki. However, I was mostly unsuccessful. Hence, I decided to get use my Lab0 VM for observing affect on soft faults.

9.2 fflush

The fflush function is used to flush the output buffer associated with a stream. When you write to a stream, such as stdout or stderr, the output is first written to an internal buffer rather than directly to the output device. The buffer is flushed automatically under certain conditions, such as when the buffer is full or when you call a function that reads from the stream, such as fgets or fread. However, sometimes you may want to flush the buffer explicitly to ensure that the output is written to the output device immediately.

In the example code that I showed you, fflush is called after printf to ensure that the message is written to the console immediately. Without fflush, the message may remain in the output buffer and not be written to the console until later, which can be confusing if the program crashes or terminates unexpectedly.

Regarding your second question, the behavior of fflush on stderr is implementation-dependent. According to the C standard, calling fflush on stderr has undefined behavior. However, many implementations, including most POSIX systems, treat stderr as an unbuffered stream, which means that output to stderr is not buffered and is written immediately to the output device. In such implementations, calling fflush on stderr has no effect.

So, in general, you don't need to call fflush after writing to stderr. However, if you want to ensure that the output is written immediately, you can use fflush as a workaround, but note that it may not work on all systems.

9.3 Anonymous mmap in the presense of a Competing Memory Process

All metrics related to memory have shown degradation. It is interesting to learn that getrusage report statistics of the entire process, not just the thread we are interested in. There is no easy way to hack this with multi-threading. Exec-ing a new process would require significant changes to my code. However, the results are consistant.

The PMU metrics show that the cache and TLB hit rates decreased due to coherance between the main and the competing thread where the competing thread constantlly pollute the caches.

Most of the getrusage metrics are skewed by the competing thread. The system time has increased several fold because of the competing process's memory requests being serviced by the system. The resident size increased almost to the available size due to the competing process.

Volentary Context Switches (Figure 12.) is an important metric to analyze here. For unpopulated mmap, we can notice a significant increase in the context switches. This could be because, the system doesn't have enough memory to host the requested pages. Hence, the thread yeilds.

The Block I/O operations reduce in half for File Shared Case. I was not able to understand this behaviour.

9.4 PAGEREF_ACTIVATE

I haven't been able to get measurement for this section. My VM constantly killed the process with OOM – increasing the memory size did not help.

10 Time and Effort spent

I have spent more than 50 hours working on the lab. Getting meaningful measurement was extremely challenging. The last part was also challenging as my VM kept crasing on OOM.