CS380L: Advanced Operating Systems Lab #1

Zeyuan Hu [1], iamzeyuanhu@utexas.edu

EID:zh4378 Spring 2019

1 Environment

Unless otherwise noted, we use a Linux server for all the experiments. The server has 4 Intel(R) Xeon(R) CPU E3-1220 v5 @ 3.00GHz processors and 16GB of memory, and runs Ubuntu 16.04.2 LTS (kernel version 4.11.0). The CPU has 32KB of L1 data cache per core (8-way set associative) (found through `getconf -a | grep CACHE`). In addition, it has two-level TLBs. The first level (data TLB) has 64 entries (4-way set associative), and the second level has 1536 entries for both instructions and data (6-way set associative) (found through `cpuid | grep -i tlb`).

2 Memory map

`/proc/[pid]/maps` file contains process `[pid]`'s mapped memory regions and their access permissions [1]. We use the following code to read content of `/proc/self/maps` file [2]:

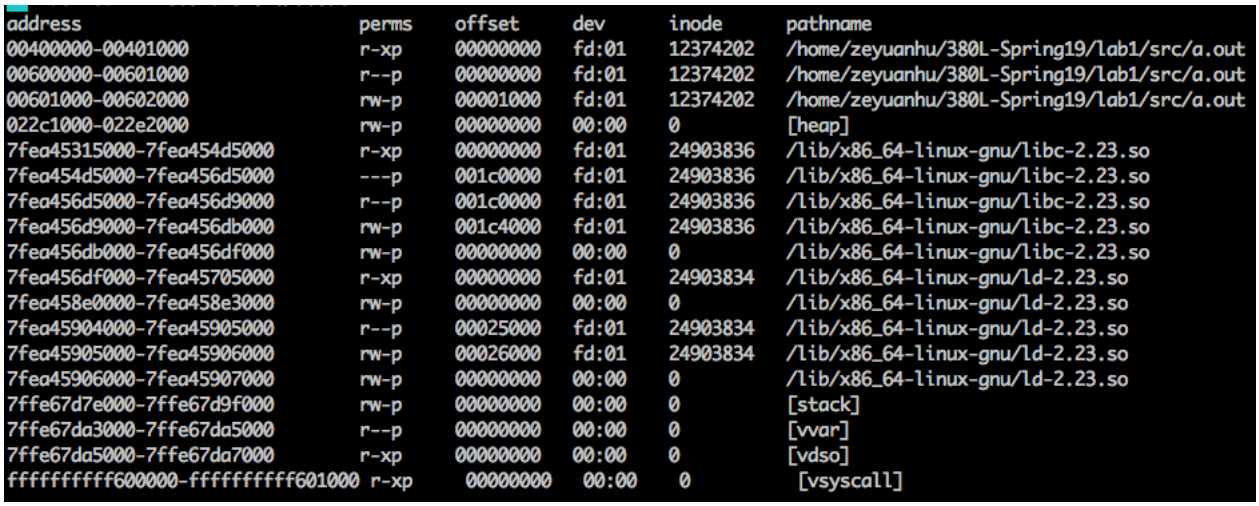
```
sprintf(filepath, "/proc/%u/maps", (unsigned)getpid());
FILE *f = fopen(filepath, "r");

printf("%-32s %-8s %-10s %-8s %-10s %s\n", "address", "perms", "offset", "dev", "
    inode", "pathname");
while (fgets(line, sizeof(line), f) != NULL) {
     sscanf(line, "%s%s%s%s%s%s", address, perms, offset, dev, inode, pathname);
     printf("%-32s %-8s %-10s %-8s %-10s %s\n", address, perms, offset, dev, inode,
          pathname);
}
fclose(f);
```

In the file, each line corresponds to a mapped memory region. There are six columns of each line, which represent six properties of the mapped memory region: `address`, `perms`, `offset`, `dev`, `inode`, and `pathname`. The result of running `memory_map.c` is below:

The `address` field gives range of virtual memory address of the mapped memory region. Access permission of each memory region is indicated by `perms` field. There are four bits in the field: `rwx`

[1] 30 hours spent on this lab.

[2] see `memory_map.c` for complete code



address	perms	offset	dev	inode	pathname
00400000-00401000	r-xp	00000000	fd:01	12374202	/home/zeyuanhu/380L-Spring19/lab1/src/a.out
00600000-00601000	r--p	00000000	fd:01	12374202	/home/zeyuanhu/380L-Spring19/lab1/src/a.out
00601000-00602000	rw-p	00001000	fd:01	12374202	/home/zeyuanhu/380L-Spring19/lab1/src/a.out
022c1000-022e2000	rw-p	00000000	00:00	0	[heap]
7fea45315000-7fea454d5000	r-xp	00000000	fd:01	24903836	/lib/x86_64-linux-gnu/libc-2.23.so
7fea454d5000-7fea456d5000	---p	001c0000	fd:01	24903836	/lib/x86_64-linux-gnu/libc-2.23.so
7fea456d5000-7fea456d9000	r--p	001c0000	fd:01	24903836	/lib/x86_64-linux-gnu/libc-2.23.so
7fea456d9000-7fea456db000	rw-p	001c4000	fd:01	24903836	/lib/x86_64-linux-gnu/libc-2.23.so
7fea456db000-7fea456df000	rw-p	00000000	00:00	0	/lib/x86_64-linux-gnu/libc-2.23.so
7fea456df000-7fea45705000	r-xp	00000000	fd:01	24903834	/lib/x86_64-linux-gnu/ld-2.23.so
7fea458e0000-7fea458e3000	rw-p	00000000	00:00	0	/lib/x86_64-linux-gnu/ld-2.23.so
7fea45904000-7fea45905000	r--p	00025000	fd:01	24903834	/lib/x86_64-linux-gnu/ld-2.23.so
7fea45905000-7fea45906000	rw-p	00026000	fd:01	24903834	/lib/x86_64-linux-gnu/ld-2.23.so
7fea45906000-7fea45907000	rw-p	00000000	00:00	0	/lib/x86_64-linux-gnu/ld-2.23.so
7ffe67d7e000-7ffe67d9f000	rw-p	00000000	00:00	0	[stack]
7ffe67da3000-7ffe67da5000	r--p	00000000	00:00	0	[vvar]
7ffe67da5000-7ffe67da7000	r-xp	00000000	00:00	0	[vdso]
ffffffffff600000-ffffffffff601000	r-xp	00000000	00:00	0	[vsyscall]

Figure 1: Output of memory_map.c

represents read, write, and executable respectively; the last bit (`p` or `s`) represents whether the region is private or shared. `offset` field represents the offset in the mapped file. `dev` field indicates the device (represented with format of `major:minor`) that the mapped file resides . There are two kinds of value in this column for our case: `fd:01` and `00:00`. The former one is the device id (in hex) of `/` (checked with `mountpoint -d /`) and the latter one represents no device associated with the file. `inode` field represents the inode number of the file on the device. `0` means no file is associated with the mapped memory region. `pathname` field gives the absolute path to the file associated with the mapped memory region. It can be some special values like `[heap]`, `[stack]`, `[vdso]`, etc.

To locate the start of the text section of the executable, we invoke `objdump -h` on the binary and get `0000000000400600`. Output of `/proc/self/maps` shows that the start address of `libc` is `7fea45315000`. The reason for these two addresses are different is `libc` is dynamic loaded library, which is loaded during the runtime of executable, which is not compiled and linked as part of executable. The code segment contains the executable instruction, not the dynamic loaded library.

One interesting thing happens between runs of the executable: the content of `/proc/self/maps` is different. Addresses of all mapped memory regions are different except for the regions mapped to the executable and `[vsyscall]`. The root cause behind this phenomenon is Address Space Layout Randomization (ASLR) [2] for programs in user space. This feature is enabled by default and can be seen via the content of `/proc/sys/kernel/randomize_va_space` file. In our case, the value is 2, which means the positions of stack itself, virtual dynamic shared object (VDSO) page, shared memory regions, and data segments are randomized [3].

3 Getrusage

To get resource usage of the current process, we use `getrusage` [4]. The result is stored in `rusage` struct. Not all fields of the struct are completed: unmaintained fields are set to zero by the kernel. Those fields exist for compatibility with other systems purpose. The following code instantiates `rusage` struct and print all the maintained fields [3]:

```
struct rusage usage;
if (getrusage(RUSAGE_SELF, &usage) != 0) {
    perror("getrusage");
    return 0;
}

// user CPU time used
printf("utime = %ld.%06ld s\n", usage.ru_utime.tv_sec,
usage.ru_utime.tv_usec);
// system CPU time used
printf("stime = %ld.%06ld s\n", usage.ru_stime.tv_sec,
usage.ru_stime.tv_usec);
// maximum resident set size
printf("maxrss = %ld KB\n", usage.ru_maxrss);
// page reclaims (soft page faults)
printf("minflt = %ld\n", usage.ru_minflt);
// page faults (hard page faults)
printf("majflt = %ld\n", usage.ru_majflt);
// block input operations
printf("inblock = %ld\n", usage.ru_inblock);
// block output operations
printf("oublock = %ld\n", usage.ru_oublock);
// voluntary context switches
printf("nvcsw = %ld\n", usage.ru_nvcsw);
// involuntary context switches
printf("nivcsw = %ld\n", usage.ru_nivcsw);
```

man page of `getrusage` explains the meaning of each field [4] in details. `utime` and `stime` are about CPU time usage; `minflt` and `majflt` are related to page faults; `maxrss` represents the maximum size of working set; `inblock` and `oublock` are about file system I/O.

[3] complete code can be seen in `getrusage.c`

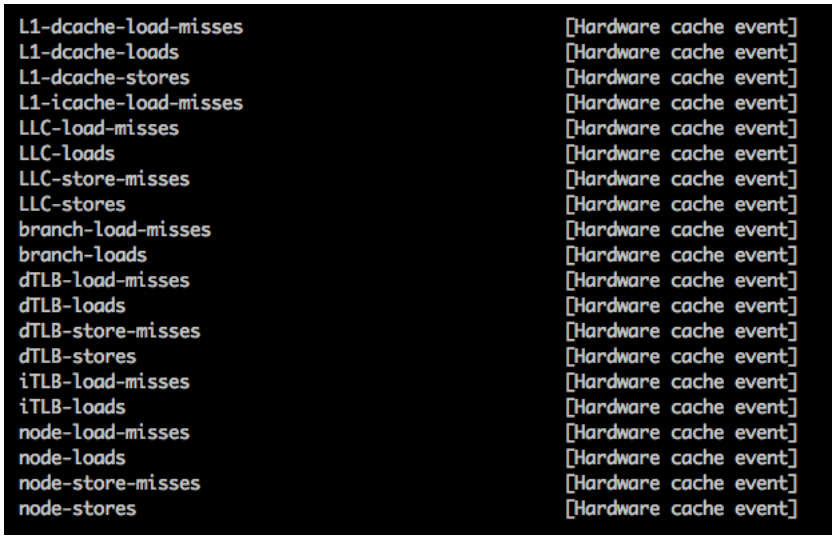


Figure 2: Part of output of `perf list` related to cache

4 perf_event_open

We first check the support of `perf_event_open` interface by checking the existence of `/proc/sys/kernel/perf_event_paranoid`, which is true in our case (value is set to 2). In Linux, `perf_event_open` interface [5] is used to setup performance monitoring. Specifically, it provides an interface that allows user to access various events (i.e., events counted by performance counters [6]). `perf list` gives available events on current machine. Counters related to cache in our machine is shown in Figure 2. We are interested in counters related to L1 data cache and data TLB. As shown in Figure 2, we have counters for number of times the L1 cache was accessed for data (`L1-dcache-loads`), the number of those access that resulted in a cache miss (`L1-dcache-load-misses`), and write access of L1 data cache (`L1-dcache-stores`). Similarly, for data TLB, we have read access (`dTLB-loads`), read miss (`dTLB-load-misses`), write access (`dTLB-stores`), and write miss (`dTLB-store-misses`).

According to man page [5], there is no glibc wrapper for `perf_event_open` system call. However, we can wrap it on our own as the following:

```
1 static long perf_event_open(struct perf_event_attr *hw_event, pid_t pid, int cpu,
      int group_fd, unsigned long flags) {
2   return syscall(__NR_perf_event_open, hw_event, pid, cpu, group_fd, flags);
3 }
```

Using the `syscall` does not mean there is a syscall opcode in the program. Figure 3 shows `perf_event_open` section of `objdump -d`. Using `syscall` function will have `perf_event_open` system call number compiled in the program. In this case, the system call number of `perf_event_open` is 298 (can be checked in `arch/x86/entry/syscalls/syscall_64.tbl` of the Linux kernel source tree),

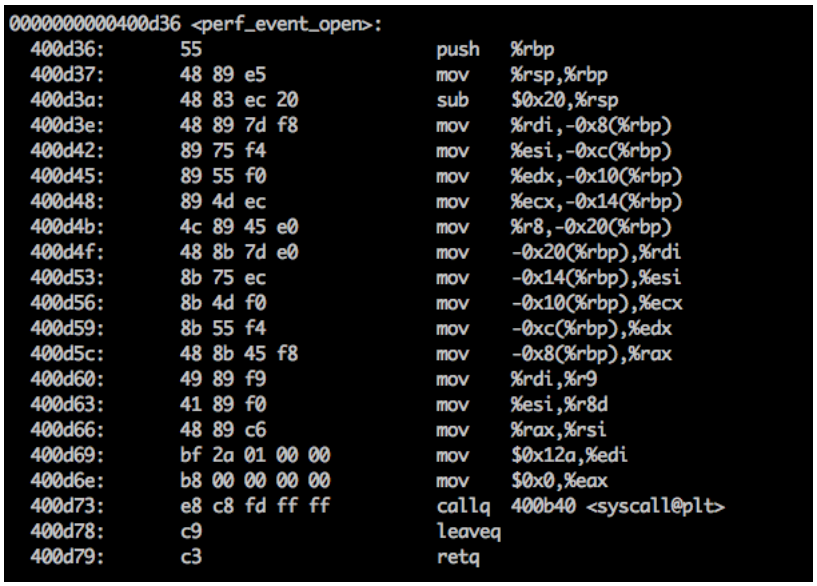


Figure 3: `perf_event_open` section of `objdump -d` output

which is `$0x12a` in hex.

We want to monitor the following events: read, write, read miss for level 1 data cache and read miss, write miss for data TLB. A call to `perf_event_open` gives a file descriptor that corresponds to one event being measured. We can use `ioctl` interface to control the counters and read file descriptors to obtain counter vales. The following code highlights the usage of `perf_event_open` interface to measure all five events for trivial `printf` [4].

```
1  int hw_cache_perf_event_open(int group_fd, int cache_id, int cache_op_id,
2  int cache_op_result_id) {
3    struct perf_event_attr pe;
4    memset(&pe, 0, sizeof(struct perf_event_attr));
5    pe.type = PERF_TYPE_HW_CACHE;
6    pe.size = sizeof(struct perf_event_attr);
7    pe.config = cache_id | (cache_op_id << 8) | (cache_op_result_id << 16);
8    pe.disabled = 0;
9    if (group_fd == -1) {
10     pe.disabled = 1;
11   }
12   pe.exclude_kernel = 1;
13   pe.exclude_hv = 1;
14   int fd = perf_event_open(&pe, 0, cpu_id, group_fd, 0);
15   if (fd == -1) {
```

[4] complete code can be seen in `perf_event_open_usage.c`

```
    perror("perf_event_open");
    exit(EXIT_FAILURE);
  }
  return fd;
}

int main(int argc, char **argv) {

  int l1_read_access_fd = hw_cache_perf_event_open(
      -1, PERF_COUNT_HW_CACHE_L1D, PERF_COUNT_HW_CACHE_OP_READ,
      PERF_COUNT_HW_CACHE_RESULT_ACCESS);
  int leader_fd = l1_read_access_fd;

  int l1_read_miss_fd = hw_cache_perf_event_open(
      leader_fd, PERF_COUNT_HW_CACHE_L1D, PERF_COUNT_HW_CACHE_OP_READ,
      PERF_COUNT_HW_CACHE_RESULT_MISS);
  int l1_write_access_fd = hw_cache_perf_event_open(
      leader_fd, PERF_COUNT_HW_CACHE_L1D, PERF_COUNT_HW_CACHE_OP_WRITE,
      PERF_COUNT_HW_CACHE_RESULT_ACCESS);
  int tlb_read_miss_fd = hw_cache_perf_event_open(
      leader_fd, PERF_COUNT_HW_CACHE_DTLB, PERF_COUNT_HW_CACHE_OP_READ,
      PERF_COUNT_HW_CACHE_RESULT_MISS);
  int tlb_write_miss_fd = hw_cache_perf_event_open(
      leader_fd, PERF_COUNT_HW_CACHE_DTLB, PERF_COUNT_HW_CACHE_OP_WRITE,
      PERF_COUNT_HW_CACHE_RESULT_MISS);

  ioctl(leader_fd, PERF_EVENT_IOC_RESET, PERF_IOC_FLAG_GROUP);
  ioctl(leader_fd, PERF_EVENT_IOC_ENABLE, PERF_IOC_FLAG_GROUP);

  // Do the work that we want to analyze
  printf("Do some work that we want to measure here\n");

  ioctl(leader_fd, PERF_EVENT_IOC_DISABLE, PERF_IOC_FLAG_GROUP);

  uint64_t l1_read_miss = 0;
  uint64_t l1_read_access = 0;
```

```
  uint64_t l1_write_access = 0;
  uint64_t tlb_read_miss = 0;
  uint64_t tlb_write_miss = 0;

  read(l1_read_access_fd, &l1_read_access, sizeof(uint64_t));
  read(l1_read_miss_fd, &l1_read_miss, sizeof(uint64_t));
  read(l1_write_access_fd, &l1_write_access, sizeof(uint64_t));
  read(tlb_read_miss_fd, &tlb_read_miss, sizeof(uint64_t));
  read(tlb_write_miss_fd, &tlb_write_miss, sizeof(uint64_t));

  close(l1_read_access_fd);
  close(l1_read_miss_fd);
  close(l1_write_access_fd);
  close(tlb_read_miss_fd);
  close(tlb_write_miss_fd);

  printf("[Performance counters]\n");
  printf("Data L1 read access: %" PRIu64 "\n", l1_read_access);
  printf("Data L1 write access: %" PRIu64 "\n", l1_write_access);
  printf("Data L1 read miss: %" PRIu64 "\n", l1_read_miss);
  printf("Data L1 read miss rate: %.5f\n",
         (double)l1_read_miss / l1_read_access);
  printf("Data TLB read miss: %" PRIu64 "\n", tlb_read_miss);
  printf("Data TLB write miss: %" PRIu64 "\n", tlb_write_miss);

  fflush(stdout);

  return 0;
}
```

File descriptors returned from calling `perf_event_open` can be grouped together so that we can measure corresponding events simultaneously. There are five events we want to measure and as shown in the code above, we group them together and measure them at the same time. However, measuring all five events at the same time may not possible for some machine as CPU only has limited amount of machine specific registers (MSRs) for low-level performance counting. In our

environment, this number is 8 [5]. For the following experiment, we also want to lock our process onto a single processor. To achieve it, we use the following code:

```
cpu_set_t set;
CPU_ZERO(&set);
CPU_SET(cpu_id, &set);
if (sched_setaffinity(0, sizeof(cpu_set_t), &set) == -1) {
       perror("sched_setaffinity");
       exit(EXIT_FAILURE);
}
```

Here, all counters we setup are associated with CPU specified by `cpu_id`. We use `sched_setaffinity` system call to set the CPU affinity of current process and ensure that it runs on CPU of `cpu_id` only.

5 Measuring memory access behavior

`mmap` system call maps files or devices into memory. The created mapping can be file-backed or anonymous. File-backed mapping maps an area of the process's virtual memory to files (i.e., an area of the process's virtual memory is mapped to file-backed memory); reading those areas of memory causes the file read. Anonymous mapping is the opposite of file-backed mapping (i.e., not backed by file; an area of the process's virtual memory is mapped to anonymous memory). In this section, we experiment with both types of mappings. Since file-backed mapping can be shared or private, we have three memory mappings setup in our experiment: anonymous, file-based (private) and file-based (share). In this experiment, we use `MAP_POPULATE` flag. `MAP_POPULATE` populates page tables for a mapping. For file-backed mapping, this means read-ahead on file, which reduce blocking on page faults later. In addition, we use `memset` to initialize the mapped region and `msync` to flush the change made to the file-backed memory back to file system whenever possible.

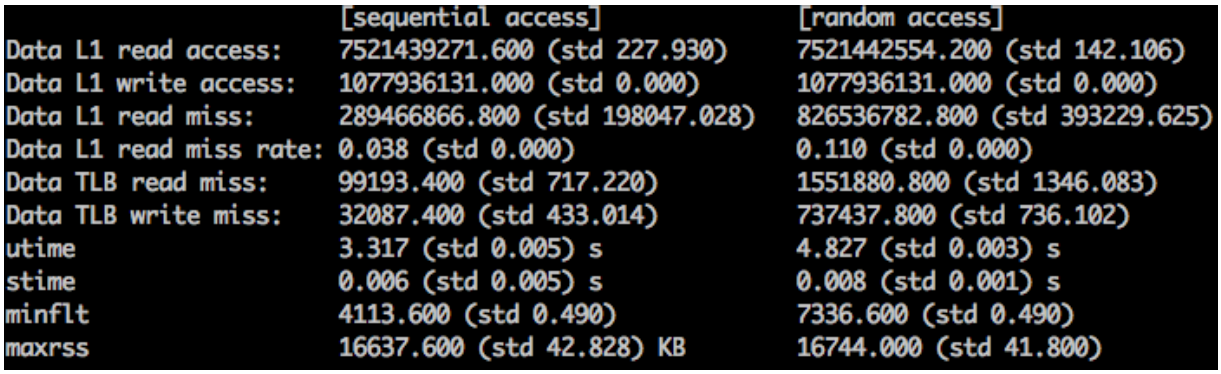
We study the following code, which is used to access the mapped region:

```
#define CACHE_LINE_SIZE 64
int opt_random_access;
void do_mem_access(char *p, int size) {
  int outer, locality, i;
  int ws_base = 0;
  int max_base = size / CACHE_LINE_SIZE - 512;
  for (outer = 0; outer < (1 << 20); outer++) {
```

[5] check via `cpuid|grep 'number of counters per logical processor'`

```
    long r = simplerand() % max_base;
    if (opt_random_access) {
      ws_base = r;
    } else {
      ws_base += 512;
      if (ws_base >= max_base) {
        ws_base = 0;
      }
    }
    for (locality = 0; locality < 16; locality++) {
      volatile char *a;
      char c;
      for (i = 0; i < 512; i++) {
        a = p + ws_base + i * CACHE_LINE_SIZE;
        if (i % 8 == 0) {
          *a = 1;
        } else {
          c = *a;
        }
      }
    }
  }
}
```

`simplerand()` is a function generates random number. A global variable `opt_random_access` is used to specify the memory access to be sequential or random. Specifically, `db_mem_access()` routine will access a working set consists of 512 consecutive cache lines. Among those 512 cache lines, $\frac{1}{8}$ will be write access and $\frac{7}{8}$ will be read access. The same working set is accessed 16 times. The whole routine will work on 2^{20} working sets. Working sets are selected either sequentially or in random (specified by `opt_random_access`). In sequential access case, offset between working sets is 512 (e.g., since there are 512 cache lines of a set). In random access case, the start(i.e., base) of working set is selected randomly based on the number generated by `simplerand()`. `simplerand()` is called no matter the sequential memory access or random access because we want to keep the experiment results difference only due to the memory access pattern difference. In other words, we want to have both memory access patterns have the same function call and context switch overheads (caused by calling `simplerand()`).



	[sequential access]	[random access]
Data L1 read access:	7521439271.600 (std 227.930)	7521442554.200 (std 142.106)
Data L1 write access:	1077936131.000 (std 0.000)	1077936131.000 (std 0.000)
Data L1 read miss:	289466866.800 (std 198047.028)	826536782.800 (std 393229.625)
Data L1 read miss rate:	0.038 (std 0.000)	0.110 (std 0.000)
Data TLB read miss:	99193.400 (std 717.220)	1551880.800 (std 1346.083)
Data TLB write miss:	32087.400 (std 433.014)	737437.800 (std 736.102)
utime	3.317 (std 0.005) s	4.827 (std 0.003) s
stime	0.006 (std 0.005) s	0.008 (std 0.001) s
minflt	4113.600 (std 0.490)	7336.600 (std 0.490)
maxrss	16637.600 (std 42.828) KB	16744.000 (std 41.800)

Figure 4: The statistics of sequential and random memory accesses with anonymous mapped memory regions

We study `db_mem_access()` behavior on a VM with 2GB of memory and 4 virtual CPU cores. The VM runs Ubuntu 18.10 with kernel version 4.19.6. `-cpu host` is added as part of QEMU option to ensure our VM can access performance counters. We allocate 1GB buffer via `mmap` as input for `do_mem_access`. To ensure the repeatable result, we flush the level 1 data cache by accessing a 16MB buffer before calling `do_mem_access`. We invoke `getrusage` just before and after `do_mem_access` to collect resource usage statistics of initialization code and `do_mem_access`. We run the program 5 times and calculate mean and stand deviation of statistics [6]. We also submit the raw results along with this report.

We first compare sequential access and random access (indicated by `opt_random_access`). We use anonymous mapped memory regions. Figure 4 shows statistics for both cases [7]. As shown in the figure, the ratio between read and write access is 7:1 for data L1 cache for both cases, which is as expected. The read miss rate in random access case is 3x higher than sequential access. For data TLB, read miss and write miss of random access is 15.6X and 23x higher than sequential access respectively. Random access runs 1.46x slower than sequential access. From those statistics, we can see that there is no significant difference in terms of runtime and miss rate of data L1 cache between random access and sequential access (e.g., miss rate for random access is around 0.1). In the case of sequential access, the number of data TLB read miss is about 99193, which is 396MB for 4KB page size. 396MB is smaller than the size of our buffer. If we take a look at the code, we notice `max_base` in `do_mem_access` restricts the maximum base address of the working set. If we take a look at how `max_base` is initialized and the line we initialize `a`, we can see that the working set is essentially the first $\frac{1}{64}$ of the given buffer, which is 16MB, which matches with the value of `maxrss` we observed and is much lower than 1GB. Realizing this fact also explains why TLB miss count is lower than the total number of data pages [8].

[6] We compile our program with `-O2` to prevent any unwanted memory access from the generated executable

[7] `majflt`, `inblock`, `oublock` are omitted because they are all 0

[8] If we want to access all the allocated buffer (1GB), we should replace line 21 (`a` initialization) with `a = p + ( ws_base + i)* CACHE_LINE_SIZE;`

	[private, seq]	[private,rand]	[share, seq]	[share, rand]
maxrss (init)	17540.800 (std 17.046) KB	17545.600 (std 18.693) KB	17550.400 (std 23.269) KB	17516.800 (std 15.052) KB
inblock (init)	0.000 (std 0.000)	0.000 (std 0.000)	0.000 (std 0.000)	0.000 (std 0.000)
oublock (init)	0.000 (std 0.000)	0.000 (std 0.000)	0.000 (std 0.000)	0.000 (std 0.000)
maxrss (fun)	16658.400 (std 45.789) KB	16816.000 (std 52.642) KB	16630.400 (std 45.438) KB	16839.200 (std 19.823) KB
inblock (fun)	0.000 (std 0.000)	0.000 (std 0.000)	0.000 (std 0.000)	0.000 (std 0.000)
oublock (fun)	0.000 (std 0.000)	0.000 (std 0.000)	6566.400 (std 13132.800)	6566.400 (std 13132.800)

Figure 5: The statistics of sequential and random memory accesses with file-backed mapping with `MAP_PRIVATE` and `MAP_SHARED`. `fun` is a shorthand for `do_mem_access`.

	[private, pop]	[share, pop]	[share, memset]
maxrss (init)	1066060.000 (std 106.883) KB	1064109.600 (std 274.140) KB	1066129.600 (std 31.455) KB
inblock (init)	509718.400 (std 224734.764)	80596.800 (std 161193.600)	0.000 (std 0.000)
oublock (init)	0.000 (std 0.000)	0.000 (std 0.000)	2064320.000 (std 0.000)
maxrss (fun)	0.000 (std 0.000)	2334.400 (std 259.200)	320.800 (std 34.260)
inblock (fun)	99.200 (std 198.400)	0.000 (std 0.000)	0.000 (std 0.000)
oublock (fun)	0.000 (std 0.000)	6566.400 (std 13132.800)	32832.000 (std 0.000)

Figure 6: The statistics of file-backed mapping for `MAP_POPULATE` and memset after buffer allocation. `fun` is a shorthand for `do_mem_access`.

In the next step, we experiment with file-based memory mappings. In this part, we only concern `maxrss`, `inblock`, and `oublock` because the statistics related to data L1 cache and data TLB are generated with the same setup as anonymous mappings. In addition, statistics related to memory cache is not affected by the file-backed mapping. The statistics we have is shown in Figure 5. Compare to previous result, we notice there are large amount of `oublock` during `do_mem_access` for file-backed mapping with `MAP_SHARED`. This is intuitive as the changes to the file have to be saved in shared mode. We notice that there is `inblock` is zero, which is due to the internal cache in Linux kernel for reading file. `oublock` for both random access and sequential access under shared mode are exactly the same. This may due to the fact that access pattern is not significant for the shared mode as synchronization for internal file cache is needed no matter what memory access pattern we may encounter.

In this experiment, we further examine `MAP_POPULATE` flag and `memset` the mapped memory for initialization. Figure 6 shows the statistics. Note that all the columns are from the sequential memory access. With `MAP_POPULATE` flag, the mapped memory will be populate page tables at the very beginning and perform read-ahead on the file. Thus we see the `maxrss` will match the allocated buffer size, which is around 1GB during the initialization. If the mapped memory is initialized with `memset` and then perform synchronization to the file system via `msync`, then there will be large amount of `oublock` during the initialization.

We also using `strace` to trace our program [9]. We find a system call `arch_prctl(ARCH_SET_FS , 0x7f83a006d540)` in the printout. The system call means setting FS register to `0x7f83a006d540` [7]. FS register is a segment register with no specific processor-defined purpose and is given purpose by the OS. In Linux, FS register is for thread local storage. We can also see the system call `access`

[9] we trace our program as `strace ./memory_access 0 0 0 0 0 0`, where `memory_access` is the executable from `memory_access.c`

("/etc/ld.so.preload", R_OK), which tries to access /etc/ld.so.preload file. This file contains ELF shared objects, which are loaded before the execution of the program [8]. From the printout, we can see that this file is accessed before libc.so is loaded.

6 Measuring memory access behavior with background activity

References

[1] "proc(5) - linux man page." http://man7.org/linux/man-pages/man5/proc.5.html.

[2] "Address space layout randomization (aslr)." https://en.wikipedia.org/wiki/Address_space_layout_randomization, 2018.

[3] "Linux and aslr: kernel/randomize_va_space." https://linux-audit.com/linux-aslr-and-kernelrandomize_va_space-setting, 2016.

[4] "getrusage(2) - linux man page." http://man7.org/linux/man-pages/man2/getrusage.2.html.

[5] "perf_event_open(2) - linux man page." http://man7.org/linux/man-pages/man2/perf_event_open.2.html.

[6] "Hardware performance counter." https://en.wikipedia.org/wiki/Hardware_performance_counter, 2018.

[7] "arch_prctl(2) - linux man page." http://man7.org/linux/man-pages/man2/arch_prctl.2.html.

[8] "ld.so(8) - linux man page." http://man7.org/linux/man-pages/man8/ld.so.8.html.