CS 380L Advanced Operating Systems Lab1

Soujanya Ponnapalli
Lab Date: 09/28/2017
Due Date: 09/28/2017

Abstract

This document contains learnings, findings and details for:
1. Measuring a program's behavior using Perf and other profiling tools
2. Using mmap for memory-mapped file IO

Hardware/Software Details

Note that the following was the Hardware/Software set up on which the experiments described later were run. I used **sudo lshw -C memory**, **lscpu**, **cpuid** and **sudo dmidecode -t cache -t memory** commands to get the information about the cache, TLB and memory details of the Host. Most of the details of CPU, like TLB size, entries and associativity I used **cpuid (pipe) grep -i tlb**

1. Operating Systems: Host: Ubuntu 16.04.3, Guest: Ubuntu 16.04.3
2. Host Kernel version: 4.4.0.96-generic
3. Guest Kernel version: 4.13.1
4. Host: RAM : 16GB, Disk: 500GB, Architecture: x86_64
5. Vendor ID: GenuineIntel
6. Virtualization: VT-x
7. CPU op-modes: 32-bit, 64-bit, CPUs: 8, On-line CPUs list: 0-7, Threads per core: 2
8. CPU family: 6, Model:58
9. Model name: Intel(R) Xeon(R) CPU E3-1270 V2 @ 3.50GHz 8
10. Cache: L1d : 32K, L1i : 32K, L2 : 256K, L3 : 8192K
11. dTLB is 4-way associative and iTLB is 8-way. My hardware has Level-2 cache as well and the details are as follows:
    (a) 0x63: data TLB: 1G pages, 4-way, 4 entries
    (b) 0x03: data TLB: 4K pages, 4-way, 64 entries
    (c) 0x76: instruction TLB: 2M/4M pages, fully, 8 entries
    (d) 0xb5: instruction TLB: 4K, 8-way, 64 entries
    (e) 0xc1: L2 TLB: 4K/2M pages, 8-way, 1024 entries
    (f) L1 TLB/cache information: 2M/4M pages& L1 TLB
    (g) L1 TLB/cache information: 4K pages & L1 TLB
    (h) L2 TLB/cache information: 2M/4M pages & L2 TLB
    (i) L2 TLB/cache information: 4K pages & L2 TLB

1. Memory Map and Getrusage

Following are the interesting details of the Task1 of reading the contents of /proc/self/maps file and Task2 of understanding the output of getrusage

1. /proc/self/map is a file containing the currently mapped memory regions and their access permissions
2. The fields in the file are address, permissions, offset, device(major number: minor number), inode and pathname.
   (a) The address field is the address space in the process that the mapping occupies.
   (b) The perms field is a set of permissions: r = read, w = write, x = execute, s = shared, **p = private (copy on write)**
   (c) There are additional helpful pseudo-paths like **[stack], [stack:thread-id, [heap], [vdso]**
   (d) Thus the most **interesting** things found in the output are the p:copy-on-write permissions to the executable, the different address representations for the user programs as that of the libc program and the pathname giving information about mmap.
   (e) **If the pathname field is blank, this is an anonymous mapping as obtained via the mmap function.**
   (f) There are different address spaces ranges reserved for different processes for **isolation purpose** like range around 00400000-011f7000 (User programs), 35b1800000-35b1fb2000 (libc), f2c6ff8c000-7fffb2d49000 (stack and vdso). Thus there is an obvious difference in the base address fields for the a.out and libc.
   (g) base address of executable: 00400000
   (h) getrusage is called and the fields **utime, stime, maxrss, minflt, majflt, inblock, oublock, nvcsw (voluntary context switches), nivcsw (involuntary context switches)** are recorded.

PERF_EVENT_OPEN and Memory Access Behavior

Using perf event open to monitor the program. I measured the values running the program on **Host**.

1. Configured kernel to support perf and installed perf tools myself locally.
2. Is there a syscall in your program?
   (a) Indeed there is a syscall routine in our program; a syscall opcode is present in the object dump of the program.
   (b) **objdump -d (pipe) grep syscall** gives me the following output:
   4013f0: e8 eb f6 ff ff callq 400ae0 [syscall@plt]
3. level 1 data cache accesses and misses, and data TLB misses are monitored using perf.
4. Read, Write and Prefetch events are opened; but prefetch isn't supported by the architecture on which my experiments were done.

5. **Brief description of the behavior of do_mem_access function:**
    (a) Takes a 1GB mmapped char* pointer p and the size allocated as arguments
    (b) Starting from the address sent via the pointer, constructs a working set base pointer (initially zero) either by generating a random base or by incrementing the previous base pointer by 512
    (c) Sequentially traversing the next 512 cache lines; after every 7 reads, one write is followed onto the current address pointed by the pointer.
    (d) This is done power(2,20) times.
6. **Why is simplerand() called irrespective of the value of opt_random_access?**
    (a) The computation of simplerand() is pretty straight forward. Once the algorithm is known, it is pretty easy to compute the output of the next iteration.
    (b) If we only called simplerand() in cases when the opt_random_access is set, it no longer can be called as a random value generating function.
7. **Changes to the do_mem_access code!**
    (a) The current do_mem_access randomizes just the base of the working set but the reads and writes are sequential to the following 512 cache lines
    (b) **Randomized the addresses to read from, the address to write to and the working set base address**.
8. **Msync**
    (a) flushes changes made to the in-core copy of a file, mapped into memory using mmap back to the filesystem.
    (b) There is no guarantee that changes are written back before munmap is called if msync is not used (start address and the length are updated)
9. **Running with strace**
    (a) Ouput of arch_prctl : **arch_prctl(ARCH_SET_FS, 0x7fbdd0229700) = 0**
    (b) Ran the a.out executable file using the command **strace -e access ./a.out** and found **access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)**.
    (c) **access** is the system calls that involves calls /etc/ld.so.preload, which contains a list of ELF shared objects to be loaded before the program.
    (d) /etc/ld.so.preload has a system-wide effect, causing the specified libraries to be preloaded for all programs that are executed on the system, which is usually undesirable.
10. **fflush to stdout and stderr**
    (a) forces a write of all user-space buffered data for the given output or update stream
    (b) Generally writes to stderr are not buffered and hence we might not need to fflush to stderr
    (c) But previous linux versions did have implementations of fflush for stderr too. Though the stderr is written out before stdout, I guess fflush to stderr forces write to the stream.

11. **Page Replacement Policy**
    (a) Using PAGEREF_ACTIVATE case is changed to fall through the PAGE_REFRECLAIM case and we observe that there is a significant decrement in the performance, increase in the page_faults.

Reproducibility of Results

1. Used **sched_setaffinity** to set the program (both the parent and the child) to run on a single core.
2. Used the system(command) where command is **system("sudo echo 3 ¿ /proc/sys/vm/drop_caches")** to flush the level-1 cache before enabling the perf event counters.
3. The Following observations in the standard deviation and mean of the results in multiple iterations, it can be concluded that the results were reproducible with good accuracy.

Results

For results, analysis and conclusions, please follow the following link:
https://github.com/SoujanyaPonnapalli/AdvancedOperatingSystems.git