CSE 221 Draft 2

Allison Turner (A59009879)
Alex Yen (A59002185)
Rukshani Athapathu (A59009507)
aturner@ucsd.edu
alyen@ucsd.edu
dathapathu@ucsd.edu

1 INTRODUCTION

The goal of this project is to understand the performance of the underlying hardware and operating system of 'seed-e60-119.ucsd.edu' server that reside in San Diego Supercomputer Center. This is a decade old server, 'ProLiant DL380 G6' to be exact, manufactured in 2009.

2 MACHINE DESCRIPTION

We report the following machine specifications and our methods for acquiring our machine specifications.

Processor: we recorded the following CPU and cache information through the *lscpu* command:

Processor Model: **Intel(R) Xeon(R) CPU, E5520**
Cycle frequency: **2.27 GHz (0.44ns)**
Cache sizes:
L1i: **32K**
L1d: **32K**
L2: **256K**
L3: **8192K**

DRAM: to record the type of DRAM, the clock speed, and the capcity, we used the *lshw* command with the flags *-short -C memory*:

Type: **DIMM DDR3**
Clock: **1333 MHz (0.8ns)**
Capacity: **24453452 kB (24 GB)**

We calculated the **memory bus bandwidth** with the following information. DDR2, DDR3, and DDR4 are 64 bits wide. Our machine has 12 memory channels. We also assume that our data rate is 1 bit per cycle (we are not sure if this could be found on our machine). As a result, our machine's memory bus bandwidth is calculated as follows:

Memory Bus Bandwidth: 1333MHz * 12 channels * 64 bits / 8 bits = **128 GB/s**

We determine our **I/O bus type and bandwidth** through the following command: *sudo dmidecode | grep "PCI"*. Our machine only tells us "PCI Express," so we assume that our version over PCI Express is **PCIe 1.0**. We found that we have 4 I/O bus slots, so our total bandwidth is as calculated:

Bandwidth = 250MB/s * 4 = **1GB/s (or 8 Gbps)**

For permanent storage, we first determine that our machine uses three hard drives through the *lsblk* command. We also determine the model specifically through the *lshw* command with *-class disk* flag. For each of the hard drives, we use the *hdparm* command with *-t /dev/[disk name]* flags, where [disk name] is replaced with each hard drive name (which is *sda*, *sdb*, and *sdc* in our scenario) – this is to determine the disk latency. Finally, we calculate latency through the disk RPM, which is calculated by $1/(RPM/60s)$.

First Disk (sda):
Model: **SCSI**
Capacity: **465GB**
RPM: **15,000**
Transfer Rates: **157.80 MB/s**
Latencies: **2ms**

Second Disk:
Model: **SCSI**
Capacity: **1.8TB**
RPM: **15,000**
Transfer Rates: **384.89 MB/s**
Latencies: **2ms**

Third Disk:
Model: **SCSI**
Capacity: **2.8TB**
RPM: **15,000**
Transfer Rates: **525.95 MB/s**
Latencies: **2ms**

We calculate the network card bandwidth through the *ethtool* command.

Network card bandwidth: **10,000MB/s**

Lastly, we determine the operating system and version through *cat /etc/os-release*

Operating System: **CentOS Linux 8**

3 METHODOLOGY

We wrote most of the code for our tests in C, in order to balance the desire for low-level testing with a need to avoid over-optimization of code. GCC version we use to compile our code is 8.5.0 20210514 (Red Hat 8.5.0-4) (GCC). We did not employ any optimizations for the compilation.

We use 'taskset' command to bond our processes to a single cpu on the system to get accurate measurements. We also set our process priority to -20 to give our process the highest priority.

To measure the CPU operations we use cycle counts. This choice was most important for capturing CPU operations with very short compute times. Therefore to get accurate timing measurements it is vital that we use a low overhead mechanism. The machine under test has an Intel(R) Xeon(R) CPU and has support for RDTSCP assembly instruction. We referred Intel's white paper [4] to develop our CPU benchmarking tests with RDTSCP assembly instructions.

We also checked to make sure that the CPUs of the machine under test has "constant_tsc" flag so that the the cycle counter updates at a fixed frequency independent of the operating frequency of the CPU.

3.1 Measuring Cycle Time

We compute the cycle time by counting the number of cycles in a known time interval as mentioned in [5]. Seed server reports the following frequencies.

- model name : Intel(R) Xeon(R) CPU E5520 @ 2.27GHz
- cpu MHz : 1654.168

Cycle counts for a 10 second known time interval is as follows.

```
Cycle Count: 22667474081
Cycle Count: 22667467275
Cycle Count: 22667477505
Cycle Count: 22667468036
Cycle Count: 22667487880
Cycle Count: 22667474574
Cycle Count: 22667464493
Cycle Count: 22667461916
Cycle Count: 22667476230
Cycle Count: 22667479232
Samples: 10
Mean: 22667473122
sd: 7796.8504
```

This indicates that the CPU frequency is 2.27GHz and not 1654.168MHz. Therefore, we can conclude that one cycle takes 0.44ns.

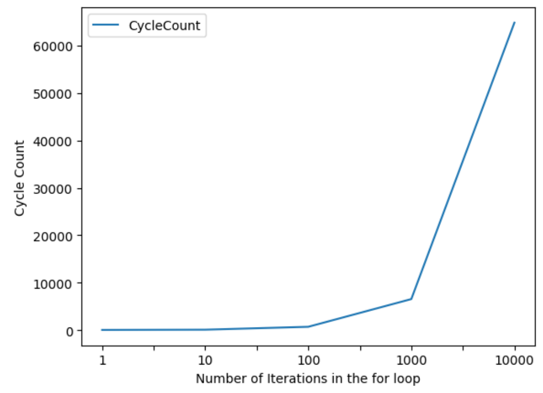

Figure 1: Loop overhead for different numbers of iterations

4 CPU, SCHEDULING AND OS SERVICES

For all CPU operations we took ten samples each running thousand iterations and computed the mean and the standard deviation. The standard deviation is measured across these ten trials.

4.1 Measurement Overhead

We predicted the timing overhead to be 4 cycles for each read so altogether 8 cycles. The actual measurement revealed it to be 44 cycles. The standard deviation among the ten samples is 3.439961.

After measuring the timing overhead we predicted the looping overhead for just one item to be 264 cycles ((44 * 5) + 44). That turned out to be wrong again. 'for' loop with one iteration only took 45 cycles same amount as the timing overhead. Based on that result then we predicted the iterations with 10, 100, 1000, 10000 to take roughly 45, 50, 70, 100 cycles. We found ourselves to be far off once again. It took 45, 96, 705, 6554 and 64855 cycle counts for 1, 10, 100, 1000, 10000 iterations. Standard deviation turned out to be 2.750207, 4.376706, 11.499275, 158.636552, and 610.22372 for 1, 10, 100, 1000, 10000 iterations respectively among the ten trials. Figure 1 shows the actual cycle counts with different numbers of iterations.

4.2 Procedure Call Overhead

Our estimation for a procedure with no arguments was 45 cycles based on the experience from the for loop with just one iteration. Then we predicted that each argument would take 10 cycles each. That is, a procedure with 1-7 integer argument would take 65, 75, 85, 95, 105, 115, 125 cycles respectively. We were wrong again. Cycle counts did not increase steadily with parameter count. There is roughly a 10 cycles difference between no parameters and 7 parameters. These

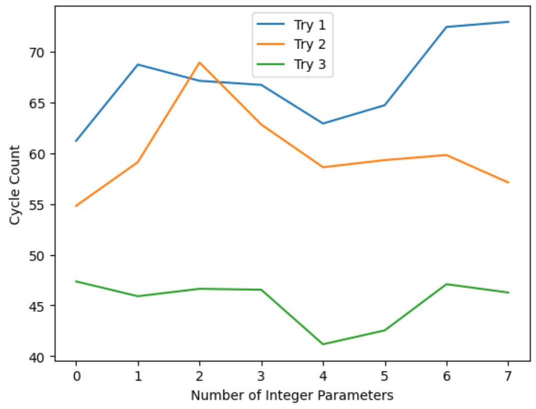

Figure 2: Procedure overhead for different numbers of integer parameters

numbers are from the ten trials we did each with a thousand iterations. The standard deviation among the ten trails for each function varied between 0-15. We repeated this three times to make sure we are getting consistent results. But unfortunately, we do not see a clear pattern between cycle count and the number of parameters for procedure overhead. Figure 2 shows the actual results of these three tries.

4.3 System Call Overhead

To measure system calls, we wrote functions to time, in clock cycles a sample of system calls via their C library functions. The system calls that we measure in this manner thus far include:

- fork - average 158,322 clock cycles
- exit - average 1.94363E+16 clock cycles
- kill - average 21,468 clock cycles
- open file in read mode - average 14,871 clock cycles
- open file in write mode - average 525,008 clock cycles
- close file - average 4,743 clock cycles
- sleep(0) - average 134,018 clock cycles
- getpid - average 3,903 clock cycles

Overall, it seems that "cheap" system calls, such as kill or close, take between 3,000 and 5,000 cycles, while average calls take a few hundred thousand cycles, and an outlier like exit takes trillions.

We added kill to this list as a comparison point to exit. Exit has a surprisingly high duration every time we measured it, no matter how we refactored our code. We hypothesized that perhaps the self-directed cleanup procedures of exit were taking up so much time, and that ending a process with SIGKILL would be much faster. The measurements from our kill test confirm this, with its average being a tiny fraction of exit's.

The sleep test is the only measurement that takes a parameter. We thought it would be interesting to have the ability to compare overhead time for different requested sleep times, and to also account for how accurately the requested duration was fulfilled. At the moment, we have only obtained an average for sleep(0), but will be iterating on the depth of analysis.

4.4 Task Creation Time

We can compare the time required to create a process vs a thread by comparing the cycle-time measurements of fork and pthread_create. pthread_create took an average of 131,758 clock cycles, ranging from 107,848 cycles to 154,719 cycles. fork took an average of 158,322 clock cycles, ranging from 135,204 cycles to 286,928 cycles. We can see, as expected, that threads are faster to create, although the margin by which it is faster is relatively smaller than expected.

4.5 Context Switch Time

To measure the context switch time between processes, we use blocking pipes. Our test is such that the parent process writing from the pipe and the child process reading from the pipe. We record the start time right after we write to the pipe in the parent process and record the end time just before the child reads from the pipe. Our initial results show that we have an average of 207,570 cycles for context switching with a standard deviation of 27,940 cycles. We would like some feedback with regard to this section as we are not entirely sure whether the way we have conducted this test is correct.

We also use pipes to measure thread context switches. One thread is blocked as it tries to read from the other, and so there is a context switch into the writing thread and a context switch back to the reading thread. This test still needs some tweaking, as output from rusage shows 6-8 voluntary context switches, even when running with a nice value of 0. In its current state, this test shows context switch times ranging between 81,558 clock cycles and 188,379 clock cycles, with an average of 103,520 clock cycles. The wide variation in measurements for this test confirms that we need to improve it significantly. One known issue is that the time range measured includes the read and write pipe operations between the two threads, since the read and write pipe operations are how we trigger a context switch. Barring a technical rework of the code, we can factor in a measured overhead for pipe operations, as Bendersky does in [1]

5 MEMORY

5.1 RAM Access Time

To measure the latency of our caches and memory, we create a linked list of about two million nodes and first populate the caches and memory by unrolling each next node access.

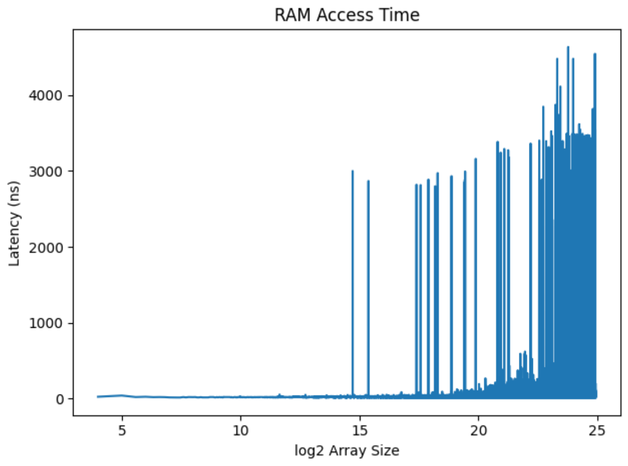

Figure 3: Raw Accessing Caches and Main Memory

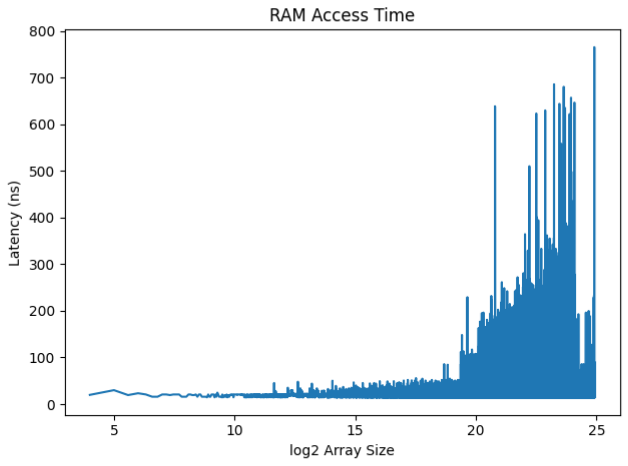

Figure 4: Accessing Caches and Main Memory (Filtered)

Then, to access the cache and memory according to the most recently access memory, we traverse the linked list in reverse order. We note that something is still wrong with our implementation – in Figure 3, we observe entries and access times that have very high cycles which are not reflective of the cache nor memory access times; we are unsure of why these are present and will fix this issue for the final submission. We thus try to filter out some of these high cycle entries by only recording access times that were less than 2000 cycles, though this also does not seems to give clean results, although the access times are a bit clearer through this in Figure 4.

We double check our work based from our machine's cache sizes, and we do not think the numbers line up. On our graph, we should see a change in latency at around 2^{23}

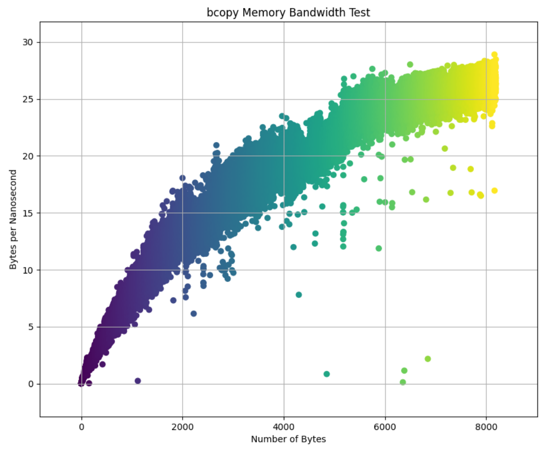

Figure 5: Testing Memory Bandwidth with bcopy

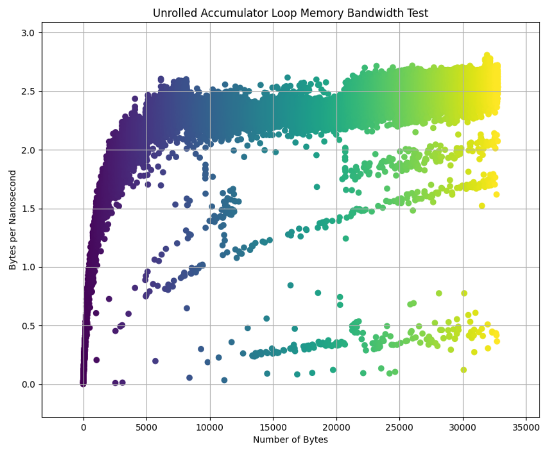

Figure 6: Testing Memory Bandwidth with an Unrolled Accumulator Loop

bytes, though I believe our results are showing around 2^{19} bytes instead. We have double checked what the memory latency should be, and we believe it is around 120ns.

5.2 RAM Bandwidth

After referencing [3] and [2], we implemented memory bandwidth tests using bcopy and loop unrolling. For the bcopy test, we allocated two page-aligned chunks of memory, "warmed" the cache by zeroing out both regions, and then bcopy-ed the contents of one allocated memory region to the other. For the loop-unrolling test, we defined macros that would increment through a region of memory and add each element. Each test was executed several times, on a range of memory region sizes from 1 byte to more than two pages' worth of bytes. We compiled with no optimizations, using the '-O0' flag, and ran the executable on a single core with high user-level priority, using 'sudo nice -n -20 taskset –cpu-list 1 ./memory_measurement'

bcopy and loop unrolling yielded very different ranges of bandwidth rate, however both produced very strong trendlines: bcopy shows a power-series-type trend with a bandwidth rate ceiling around 27 bytes per ns, and loop unrolling shows a logarithmic-type trend with a ceiling around 2.75 bytes per ns. Loop unrolling also shows some smaller trendlines underneath the upper-bound of the logarithmic trendline; these could perhaps show test iterations subject to cache evictions, page faults, or similar.

Since our results differ so much between our two tests, we still have some refinement to perform on our code. Loop unrolling has likely had its timings falsely inflated, since the method for iterating through an arbitrarily large region of memory involves checks and management variables that could be using up cycle time instead of memory operations. [3] and [2] permitted the slight inflation of the cycle measurements from addition operations, but conditionals and other variables are perhaps more weighty in consideration.

5.3 Page Fault Service Time

'getconf PAGESIZE' command reveals the page size of the seed server to be of 4096 bytes. We have created a file with the size of 4096 bytes and used mmap system call to map that file into memory. 'mmap' system call triggers a page fault and we verified it with 'ps -o min_flt,maj_flt' command. Our measurements reveal the page fault service time to be 1328278ns (0.00133 seconds).

REFERENCES

[1] Eli Bendersky. 2018. Measuring context switching and memory overheads for Linux threads. (September 2018). Retrieved February 1 2022 from https://eli.thegreenplace.net/2018/measuring-context-switching-and-memory-overheads-for-linux-threads/

[2] Aaron B. Brown and Margo I. Seltzer. 1997. Operating System Benchmarking in the Wake of <i>Lmbench</i>: A Case Study of the Performance of NetBSD on the Intel X86 Architecture. *SIGMETRICS Perform. Eval. Rev.* 25, 1 (jun 1997), 214–224. https://doi.org/10.1145/258623.258690

[3] Larry McVoy and Carl Staelin. 1996. lmbench: Portable Tools for Performance Analysis. In *USENIX 1996 Annual Technical Conference (USENIX ATC 96)*. USENIX Association, San Diego, CA. https://www.usenix.org/conference/usenix-1996-annual-technical-conference/lmbench-portable-tools-performance-analysis

[4] Gabriele Paoloni. 2010. How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf. (2010).

[5] Geoff Volker. [n. d.]. Measuring Time. https://cseweb.ucsd.edu/classes/wi22/cse221-a/timing.html. ([n. d.]).