CSE 221 Project Report
Group 8

Yudong Wu, A53216872 yuw466@eng.ucsd.edu
Chengcheng Xiang, A53219708 c4xiang@ucsd.edu
Yihong He, A53218157 yih251@eng.ucsd.edu

February 27, 2017

1 Introduction

The goal of this project is to measure the performance characteristics of a HP's workstation in Opera group when running ubuntu 16.04.4. There is 3 people in our group: Chengcheng, Yihong and Yudong. We finished this system measurement project by C, a typical system implementation languages. Our code is compiled by gcc 5.4.0, with optimization flag **-O0**.

2 Machine Description

Our experiment machine is an HP Pavilion Elite HPE workstation, and the system running on this machine is Ubuntu 16.04.1. The specs of this machine and the operating system will be stated below.

2.1 CPU

The processor on this machine is an Intel(R) Core(TM) i7 CPU 860 @ 2.80GHz with 4 cores. This CPU supports constant tsc feature, which makes cycle counting much more trivial. When running this project, we disabled 3 of the 4 cores, and made only a single core available for running our measurement benchmark. The specs summery for this core is listed below:

- The frequency of this CPU should be 2.80GHz (read from model infomation). After a simple measurement, we found out that the frequency of the clock of this core is 2.793291 GHz, and the clock period of this core should be

$$\frac{1.000000}{2.793291GHz} = 0.358001ns$$

- This core has 64K on core L1 cache, which 32K of them are data cache, and the rest 32K are instruction cache
- This core has 25k on core unify L2 cache
- The L3 cache is 8M, and it is shared by all cores. Since we only enable single core when running, we can seem it as a private 8M L3 cache for this core.
- For the precision purpose, we turned off multi-threading and turbo boosting feature on this core

2.2 Memory

This machine has totally 8G main memory, with 4 bank of 1333 MHz DDR3.

2.3 Buses

The memory bus is with 64 bits width and 1333 MHz clock. The IO bus is PCI bus with 32 bits width and 33MHz clock.

2.4 Disk

The machine has a 750GB ATA disk.

2.5 NIC

The NIC is RTL8111/8168/8411 PCI Express Gigabit Ethernet Controller with 1Gbit/s capacity.

2.6 Operating System

When running this project, the operating system running on the machine is normal Ubuntu 16.04.1. The kernel is Linux 4.4.0-57-generic x86_64.

3 CPU Operations

This section mainly reports our estimation and measurement result about the cpu operations and os service like system calls and context creation and switch.

3.1 Overhead Measurement

When performing measurement, we are using the following code to measure the overhead of a piece of code:

```
1  #define START_COUNT(cycles_high, cycles_low)       \
2      asm volatile ("CPUID\n\t" "RDTSC\n\t" \
3                    "mov %%edx, %0\n\t" \
4                    "mov %%eax, %1\n\t": "=r" (cycles_high), "=r" (cycles_low)::\
5                    "%rax", "%rbx", "%rcx", "%rdx")
6
7  #define STOP_COUNT(cycles_high1, cycles_low1)      \
8      asm volatile ("RDTSCP\n\t" \
9                    "mov %%edx, %0\n\t" "mov %%eax, %1\n\t" \
10                   "CPUID\n\t": "=r" (cycles_high1), "=r" (cycles_low1):: \
11                   "%rax", "%rbx", "%rcx", "%rdx")
```

In the source code, we use RDTSC and RDTSCP to read out the timestamp counter register on the cpu. CPUID instruction serves as a barrier to ensure only cycles runs between RDTSC and RDTSCP instructions will be counted. So when we want to measure the overhead of some functions (code pieces), we can just put START_COUNT before the target code pieces and STOP_COUNT after, then we can easily get the cycles ticks during executing the code pieces we want to measure.

3.1.1 Methodology

When measuring the overhead of the measurement code we mentioned above, we use the following code snipes to perform the overhead measurement:

```
1        START_COUNT(high, low);
2        STOP_COUNT(high1, low1);
```

We just measure nothing between two count marcos, so that this measurement will response the overhead of the execution of thos two marcos.

When performing the measurement, we will run the measurement code in a 10000 round loop, then we take the arithmatic mean of the cycles responsed as the final result.

3.1.2 Estimation and Results

The estimation and the measurement result will be list in the following table:

Table 1: **Measurement Overhead: Estimation and Experiment Results**

Hardware Overhead Estimation	**Software Overhead Estimation**	**Total Overhead Estimation**	**Expr. Results**
6 (inst) or 44 (cycles)	0	6 (instructions)	102

When estimation, the first evidence for us to perform the estimation is the code we write and generate, following is the acutall assembly code after compilation:

```
1     4005f8: 0f a2                cpuid
2     4005fa: 0f 31                rdtsc
3     4005fc: 89 d7                mov    %edx,%edi
4     4005fe: 89 c6                mov    %eax,%esi
5     400600: 89 7d d0             mov    %edi,-0x30(%rbp)
6     400603: 89 75 d4             mov    %esi,-0x2c(%rbp)
7     400606: 0f 01 f9             rdtscp
8     400609: 89 d7                mov    %edx,%edi
9     40060b: 89 c6                mov    %eax,%esi
10    40060d: 0f a2                cpuid
11    40060f: 89 7d d8             mov    %edi,-0x28(%rbp)
12    400612: 89 75 dc             mov    %esi,-0x24(%rbp)
```

We can found out: there are 4 instructions between RDTSC and RDTSCP instruction: two of them are inline aseembly we write in code, and the rest two are compile-time generated code caused by register saving, so the totally overhead for the measurement should be 4 instructions + 2 instructions (RDTSC and RDTSCP themselves).

The second issue for estimating the hardware overhead is that, how many cycles each instruction will take, or in other words, what is the cpu's CPI (cycles per instruction). We haven't found out the most related information, so we performed the following estimation: for those 6 instructions, we assume that each cycle the cpu can commit one instruction, and the pipeline stages is around 20; as those 6 instructions don't have any data dependencies that can not be solved by data forwarding, the pipeline will not stall,and for those 6 instructions, from first instruction enter the pipeline to the last instruction commit, the cycles consumed should be:

$$20 + 4 + 20 = 44 cycles$$

After the experiments, the result shows that we underestimated the overhead by around 50%, but as we don't know much about the internal of the Intel CPU, so that this kind of mistakes is somewhat reasonable.

3.2 Loop Overhead

When performing estimation, it is reasonable that using repeated experiments to reduce the impact of variance, so before all other overhead measurement, we need to determin the overhead of the loop first.

3.2.1 Methodology

When measuring the loop overhead, we simply measures the cycles diff before and after a simple do nothing loop. In order to reduce the impact of variance, we also run this test 10000 times, and then take the arithmatic means of 10000 samples. We also subtract the time measurement overhead (102 cycles as **Section 3.1.2**) from the loop overhead, so the final loop overhead should be:

$$average(cycles) - 102$$

We also perform our measurement upon loops rounds from 0 to 50, in order to see the overhead of different loop scale.

3.2.2 Estimation and results

Our Estimation and the measurement results are shown in the **Figure 1**:

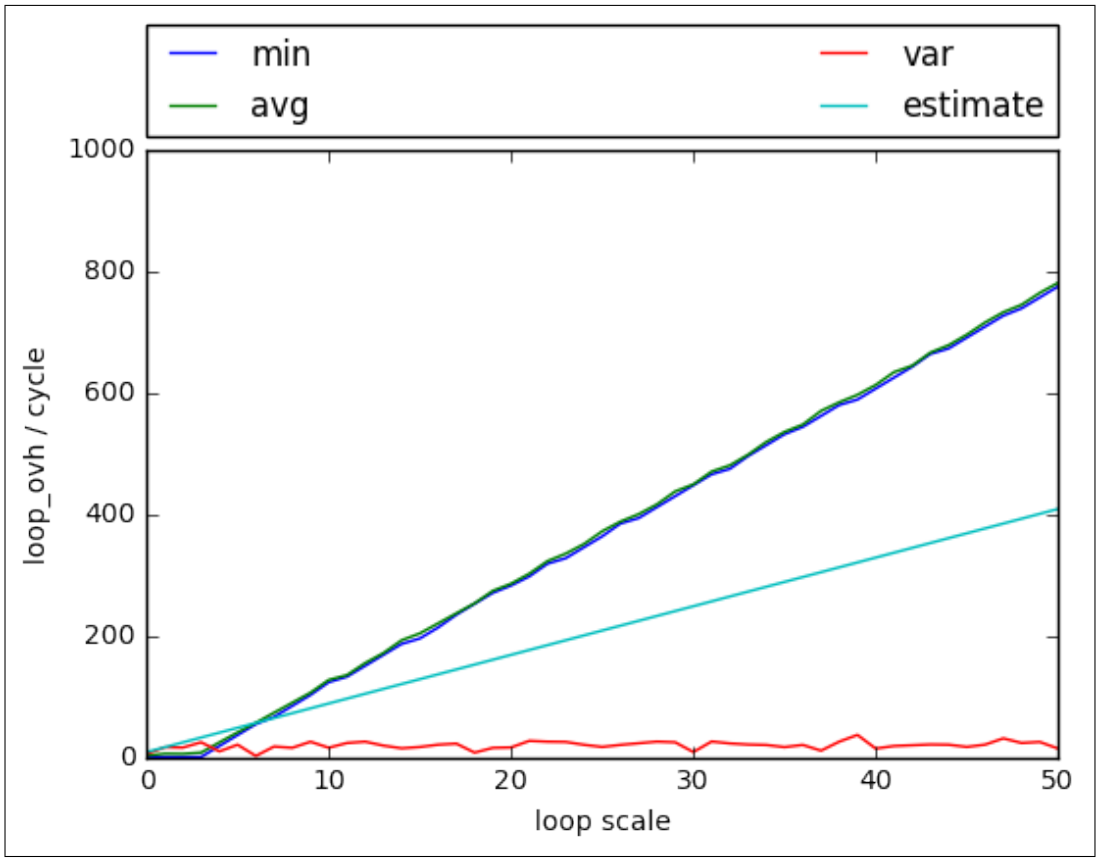

Figure 1: Loop Overhead: Estimation and Experiment Results

As shown in the graph, when estimating, we estimate the overhead increasing by 8 cycles when loop scale increase by 1. We made this estimate based on the instructions of the loop after compile:

```
1    4006a1: c7 45 c4 00 00 00 00   movl   $0x0,-0x3c(%rbp)
2    4006a8: eb 04                  jmp    4006ae <main+0xf8>
```

```
3      4006aa: 83 45 c4 01            addl   $0x1,-0x3c(%rbp)
4      4006ae: 8b 45 c4               mov    -0x3c(%rbp),%eax
5      4006b1: 3b 45 c0               cmp    -0x40(%rbp),%eax
6      4006b4: 7c f4                  jl     4006aa <main+0xf4>
```

As shown in above, there will be 4 instructions for each round of the loop, as the **Section 3.1.2** shows that, each instruction will taken more than one cycle tick in average. We finally made a assumption that each instructions will take up 2 cycles on average.

The result shows that we still underestimated the increament ratio of the overhead, and in fact, each each loop round increase will increase the overhead by 18. The red line is the variance of our data, which shows our result is quite stable.

3.3 Procedure Call

3.3.1 Methodology

Since procedure call is very quick (our rough measurement shows it takes less than 10 cycles), we measure procedure calls in loops rather than individually to reduce the variance of the measurement's own overhead. The code is as following:

```
1 for (i = 0; i < outerloop; i++) {
2     START_COUNT();
3     for (j=0; j < innerloop; j++){
4         procedure_0();
5     }
6     STOP_COUNT();
7 }
```

When calculating the time consumed by procedure_0, we first calculate the arithmatic mean of the duration cycles collected by START_COUNT and STOP_COUNT, then subtract the overhead of the time measurement and the innerloop from it, and finally divide the result by the innerloop times. We set outerloop and innerloop to be 10000 and 50 separately.

3.3.2 Estimation and results

We estimate the procedure call with no parameters will take 4 cycles and expect for two additional cycles for every one more parameter.

Figure 2 shows our Estimation and the measurement results. The comparison indicates that procedure call is far more quicker than we expected and the overhead increases linearly when the parameter number increases. The variance line justifies that we have successfully eliminated most variance.

3.4 System Call

3.4.1 Methodology

When measuring the system calls, basically we want to measure the cycle ticks between the begin and the return points of the system call. This means that we require the kernel return the control to the process immediately after the system call service routine finishes, so that what we can measure is some 'light weight' system call, like **getpid()** or **getcwd()**. However, **getpid()** will be cached by the kernel, so that only the first call of the getpid() can trap into the kernel space, so that in the end, we choose **getcwd()** system call to be the system call we measured.

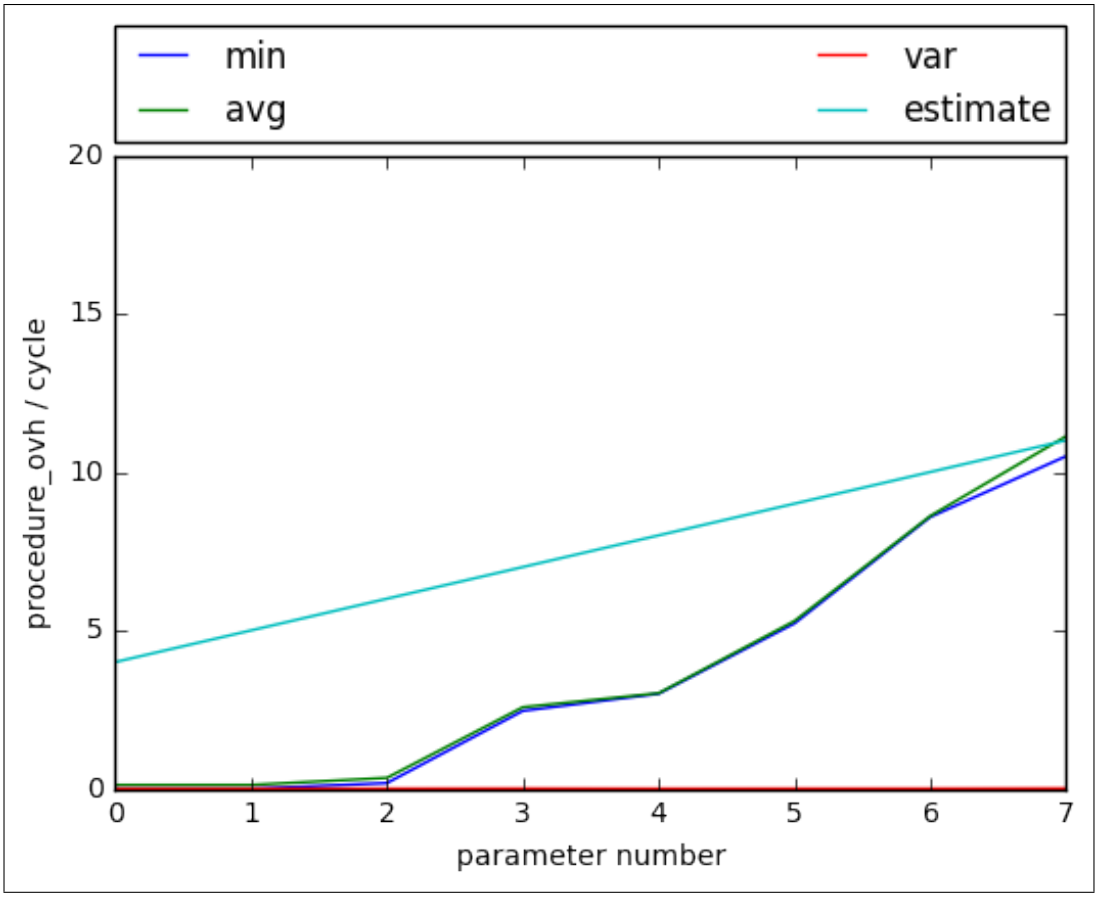

Figure 2: Procedure Overhead: Estimation and Experiment Results

For specific implementation, the measurement process will first start counting cycles, then invoke **getcwd()** system call, finally stop counting. The cycles counted should be the overhead of the **getcwd()** system call.

3.4.2 Estimation and results

The estimation and the measurement results are shown in **Table 2**

Table 2: **System Call Overhead: Estimation and Experiment Results**

Hardware Overhead Estimation	Software Overhead Estimation	Total Overhead Estimation	Expr. Results	Standard Deviation
1000 inst	0	1000 inst	1250	31

For estimation, system calls in a 64-bit system will first save a bunch of system call parameters into registers, then trap into the kernel by invoke **int** instruction or **syscall** instruction, then the kernel will serve the system call, and finally turn back to user mode with similar cost with trap into the memory. As there is a lot of assistance routine when traping into the system, then we believe 1000 cycle shoule be a reasonable estimation.

The results shows that our estimation is quite accurate.

For the questions about comparison between system call and procedure call, system call is definitely much heavy weight than procedure call, as a procedure call with 7 parameters only cost 11 cycles, but the system call will cost more than 1000 cycles. This is caused by a lot of checking and setting operations when traping into kernel mode and return to user mode.

3.5 Process and Kernel Thread Creation

3.5.1 Methodology

When measuring Task, or process creation time, we need to ensure serveral things. The first one when the measurement process fork a child process after fork syscall, the os will schedule either the measurement process or the child of the measurement process to be executed. To ensure that, we run our measurement process with the round robin real time schedualer and set the process the highest priority. The cmd is as follows:

```
1    sudo chrt -rr 99 ./measurement
```

However, even we make our process the highest priority, we are still not sure which process (parent, or child) will be executed first. So we need to ensure to handle both cases so that we can get the cycles between the point just before fork call and the point fork return.

In order to get the results we want, we implement the following measurement sequence:

1. We initialize a pipe for child and parent process to comunicate
2. We start counting cycles before we call fork
3. Both at the parent and the child process start point after fork return, we stop counting cycles
4. Child process passing the end counting to the parent via the pipe we initialized at **1**
5. We calculate both cycles spend for parent fork return and child fork return, then choose the smaller one (as the smaller one indicates that this process executes before the other one)

For the kernel thread creation, we use a similar method. The measurement sequence is listed at following:

1. We initialize a global variable for child process to store the end clock cycle count
2. We start counting cycles before we call pthread_create
3. Both at the main and the peer thread start point after pthread_create finished, we stop counting cycles
4. peer thread passing the end counting to the main thread via the global variable we initialized at **1**
5. We calculate both cycles spend for main thread pthread_create return and pthread_create start execution, then choose the smaller one (as the smaller one indicates that this thread executes before the other one)

We repeate each measurement for 10000 times to get the accurate cycles.

3.5.2 Estimation and results

The estimation and the measurement result will be list in the **Table 3**:

Table 3: **Process/Thread Creation Time: Estimation and Experiment Results**

Process or Kthread	**HW Overhead Estimation**	**SW Overhead Estimation**	**Total Overhead Estimation**	**Expr. Results** (AVG)	**Standard Deviation**
Process	NA	NA	20000 (7.16μs)	134507	3062
Kernel Thread	NA	NA	10000 (3.58μs)	20635	1546

When we make estimation, it is hard for us to do the hard ware and software estimation, as there are a lot of issues hard to estimate. For example, for fork(), the kernel will create a new process control block for the new process, and then set up a bunch of member's value inside the pcb; after that, it will at least initialize the child process's address space, by creating a bunch of page tables; all these operations might cost more then 20000 cycles. Compared with process creation, thread creation is much more light weight, as thread doesn't need to construt a new address space. Finnally we made a pridiction that, process creation will cost 20000 cycles and thread will cost half of the process cost, 10000 cycles; or in other word, 7.16μs for process and 3.58μs for thread.

Our experimental results show that the process creation and thread creation takes 134507 and 20635 cycles correspondingly, which means creating a process takes much longer time than we have estimated and creating a thread is much more lightweight. The standard deviations for our experiments are 3062 and 1546, which are relatively small to the average results.

3.6 Process and Kernel Thread Context Switch

3.6.1 Methodology

For the overhead measurement of the context switch, the basic idea to measure the overhead is to count the cycles spend on switch from one process (thread) to another process (thread). To make our measurement accurate, we need to ensure the same thing we indicates in **section 3.5.1**, that the measuring processes (threads) are the only two processes (threads) that to be scheduled. In the real system, this is impossible, but we can estimate this situation by increasing the priority of the process (threads).

For specific how to count the cycles of a context switch, we need to force a context switch to happen in one process (thread) the another (thread) process can stop the counting afterwards. To achieve this goal, we implement the following measure sequence by using blocking reading pipe. To be more precisely, at parent process, we first start a pipe for communication, then fork new process; after that, parent process will first start counting cycles, then write to the communiting pipe, then call wait() to wait the child process; at the same time, child process will read from the pipe, then stop counting the cycles, and finally send the stop cycles back to the parent process via the pipe then exit; after that, parent will wake up and get the value and count the tick.

This method uses read and wait to synchronize. Either child or parent process executing first, the final result will be the period start from parent writing to the pipe, then context switch to the child, and finally the the child read from pipe.

The method for thread is similar, the differences are:

1. We use pthread_join() instread of wait()
2. We use global variable, instead of pipe, to pass the final tick to the main thread;

3.6.2 Estimation and results

Table 4: **Process/Thread Context Switch Time: Estimation and Experiment Results**

Process or Kthread	**HW Overhead Estimation**	**SW Overhead Estimation**	**Total Overhead Estimation**	**Expr. Results** (AVG)	**Standard Deviation**
Process	NA	NA	20000	79097	650
Kernel Thread	NA	NA	10000	2562	538

For the estimation of the overhead of a context switch, we are not pretty sure the precise overhead of the context switch, so we just make a guess that the overhead should be more then 10000 cycles for thread, and the overhead for the process should be at least twice as the thread's.

The final results shows that we under estimate the overhead of the process context switch, and it should be around 80000 cycles; but amazingly for the thread one, it only cost around 2500 cycles for context switch. We might perform some more measurement about that in the future.

For the question about difference between process context switch and kernel context switch, as the results show that, the overhead of thread context switch around 30 times less then process context switch. The main difference between thread and process context switch is that, when thread context switching, kernel need neither to construct the virtual memory space to the thread switch to, nor to flush the tlb, which means that thread context switch is much less expensive than process context switch.

4 Memory

This section we will mainly discuss our measurement about memory performance on the target machine. More specificly, we will report measurements about access latency on each level of memory hirachy, the memory bandwidth, and the overhead of page fault handling.

4.1 Memory Latency

4.1.1 Methodology

When measure access latency on each level of memory hirachy, basically we followed the method suggest by lmbench[2].

We create a linklist which only contains a pointer to the next element:

```
1    struct Linklist {
2        struct Linklist * next;
3    };
4
5    typedef struct Linklist Linklist;
```

The total size of the linklist will increase from 1 KB to 256 MB, by the factor of 2. When initializing, a stride will be provided by the tester. The number of the elements in the linklist is always divisible by the stride, and when initializing, the linklist will be devided as many chunks contains stride elements, and each elements in one chunk will point to a random element in the next chunk, and the last chunk in the linklist element array will point to the first chunk, which make this linklist an infinite cyclical linklist. This structure will eliminate the impact of cache prefetch, as the stride and random elements can easily make the next element in the linklist out of the prefetch range of each level of cache.

When performing measurement, in order to get rid of all other unnecessary memory access, we have to use at least **O1** optimization; however, **O1** optimization will also optimized the code accesing the linklist. In our final measurement code, the travel of the linklist if write in inline assembly which can avoid compiler optimizations:

```
1    START_COUNT(high, low);
2    while (step --> 0) {
3#define INST "movq  (%%rax), %%rax\n\t"
4        asm volatile ("mov %0, %%rax\n\t" \
5                      HUNDRED(INST) \
6                      "mov %%rax, %1"
7                      : "=r" (iter)
8                      : "r" (iter)
9                      : "%rax");
10   }
11   STOP_COUNT(high1, low1);
```

where iter is the head of the linklist. We will measure 100 times of accesing memory on each measurement, and totally perform 1000 (stored in **step**) measurements.

4.1.2 Estimation and Results

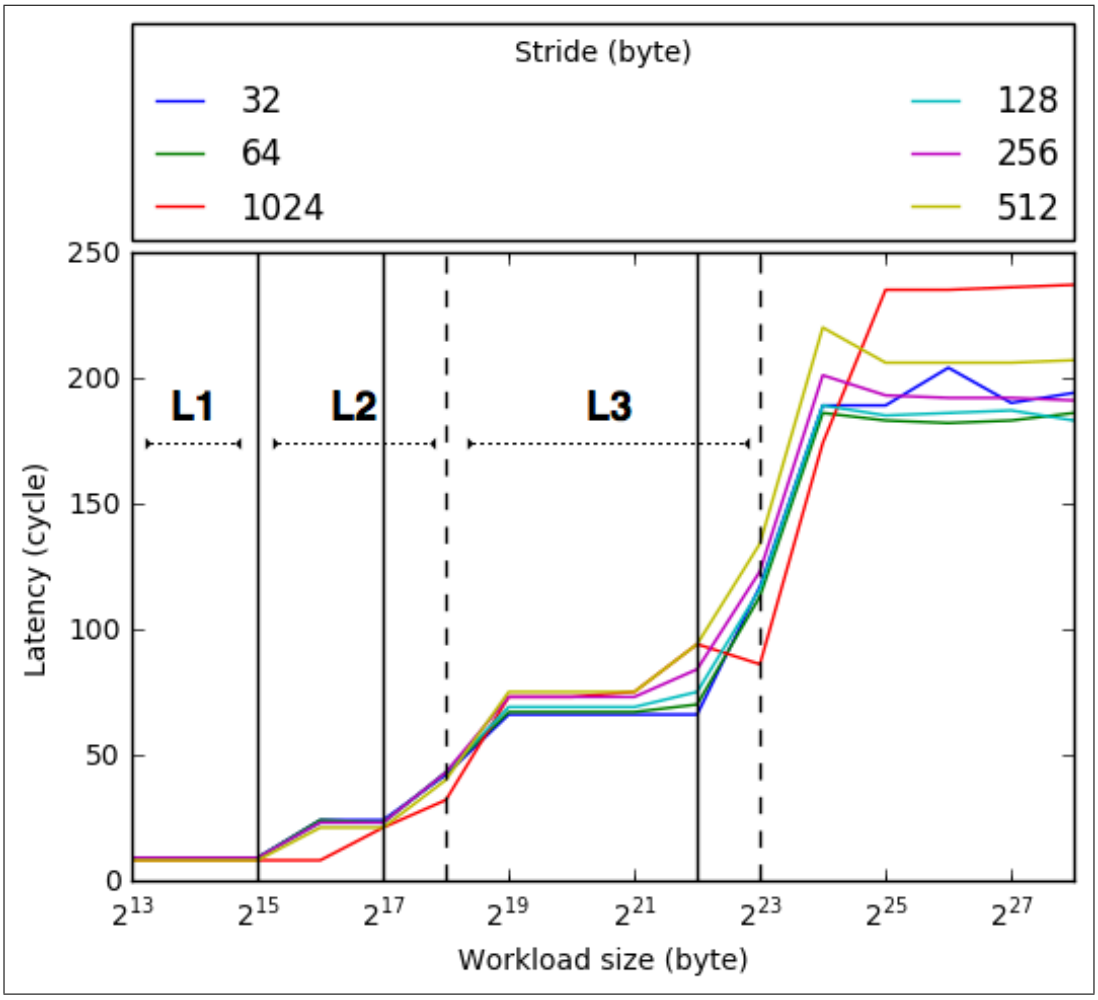

Figure 3: Memory Hirachy Latency: Estimation and Experiment Results

Table 5: **Memory Hirachy Latency: Estimation and Experiment Results**

Memory Hiracht	**Latency Estimation**	**Expr. Results** (AVG)
L1	5 cycles	9 cycles
L2	20 cycles	20 cycles
L3	60 cycles	70 cycles
L4	150 cycles	190 cycles

For estimation, we believe the latency for l1 cache references should be pretty low, let's say, within 5 cycles; the latency for l2 cache should be higher, let's say, 20 cycles; the latency for l3 cache should be much higher than l2, let's say, 60 cycles; and the latency for main memory should be the highest and much higher than l3 cache, let's say, 150 cycles.

The result is shown in **Figure 3** and **Table 5**. In **Figure 5**, the real lines separate the latency turning point for different level of memory hirachy, and the dotted lines separate the real size of different level of memory hirachy in the target machine. As we can see in the figure, the dotted line and the real line for l1 cache is on the same position, but for l2 cache and l3 cache, the latency turning point is always a little bit smaller than the actual size point. For why that happened, we guess that the following reason contribute it: as both l2 and l3 cache are unify instruction and data cache, so that some instructions stored in the l2 cache and l3 cache will accelarate that data access latency turning point to happen.

Table 5 shows the latency esitimation and measurement results. We under estimate the access latency

of l1 cache and memory.

4.2 Memory Bandwidth

4.2.1 Methodology

When measuring memory bandwidth, we used **memcpy()** function, to copy a large chunks of data (same size as l3 cache) from main memory to l3 cache when measuring read bandwith, or to copy a large chunks of data (same size as l3 cahce) from l3 cache to main memory when measuring write bandwidth. When calculating the results, we made an assumption that, only around a half of the overhead of **memcpy()** is caused by main memory reading or writing, as **memcpy()** itself will perform a lot of protection and validation operations. We believe that it is reasonable to assume that around a half of the overhead is caused by desired memory accesing.

4.2.2 Estimation and Results

Table 6: **Memory Bandwidth: Estimation and Experiment Results**

Read or Write	**Bandwidth Estimation**	**Expr. Results** (AVG)	**Standard Deviation**
Read	10600 MB/s	9356.70657557 MB/s	336.209601362
Write	10600 MB/s	8364.93322737 MB/s	500.236423781

For estimation, Frank Denneman stated on his web blog [1] that, the bandwidth for both reading and writing of a DDR3-1333 memory is 10600 MB/s, we take this as our estimation.

The measurement result is shown in **Table 6**. The results shows that memory reading bandwidth should be **9356.71 MB/s**, and writing bandwidth should be **8364.93 MB/s**. Measurement result is slightly below the estimation value, we think the reason for that is:

First, although we estimate that, the memcpy will copy all data from main memory to l3 cache only for reading measurement, and from l3 cache to main memory only for writing measurement, there is possibility that l3 cache cached address not in the target range for measurement right before memcpy, so that l3 cache replacement needed when memcpy data, which will cause additional overhead;

Second, we might under estimated the software overhead (we estimate as half of total overhead when doing measurement), which will cause the measurement result a little bit smaller than the actual value. However, as we are not sure how **memcpy()** implement, it is impossible for us to accurately estimate the software overhead of **memcpy()**.

The writing bandwidth is smaller than reading bandwidth, as **Table 6** shown. This is reasonable, as writing do be slower than reading for DRAM.

4.3 Page Fault Handling

4.3.1 Methodology

When measuring page fault handling overhead, we will first create a large file (40 MB) on disk, then using **mmap()** system call, to map the file into the main memory. Then, we will reading data in that file, as **mmap()** only mapping that file into destination memory chunks, not reading file contents into main memory, the reading operation will cause a page fault. The operating system will handle that page fault, and that's what we want to measure.

There is still one problem needs to be addressed out, the file system optimization. The file system will cache the contents of the file and also perform file prefetch from the disk, so that, if we failed to bypass

those optimization, we could not accurately measure the page fault handling overhead. To bypass those optimization, the measurement program will access the file with a stride applied with a random offset on page.

4.3.2 Estimation and Results

Table 7: **Pagefault Handling Overhead: Estimation and Experiment Results**

HW Overhead Estimation	**SW Overhead Estimation**	**Total Overhead Estimation**	**Expr. Results** (AVG)	**Standard Deviation**
111600 cycles	10000 cycles	121600 cycles	136875.058065 cycles	13433.5243954

For estimation, the I/O speed of a 7200 RPM HDD is at the scale of 100 MB/s [3], so that transfering a 4K page will take:

$$\frac{4KB}{100MB/s} = \frac{1}{25600}s$$

$\frac{1}{25600}s$ equals to around $111600cycles$.

The software overhead mainly comes from page table maintains and protection mechanism, and we assume this overhead is around 10000 cycles. Then, the total estimation overhead for a pagefault handling is 121600 cycles.

The result is shown in **Table 7**. We underestimate the overhead of the pagefault handling by around 15000 cycles. This might be caused by two reasons: first, we might underestimate software overhead caused by Operating System's handling logic; second, the I/O speed of the HDD might not reach its theorical speed.

For the question about speed compare between memory bandwidth and page fault page transfer bandwidth on byte scale, on average, the memory can transfer a byte in

$$\frac{1Byte}{9356.70657557MB/s} = 0.101924ns$$

and according to **Section 4.2.2**, the page fault mechanism can transfer a byte in

$$\frac{1Byte}{\frac{4KB}{136875cycles} * \frac{1}{0.358001ns/cycle}} = 11.963229ns$$

.

Thus the page fault handling is more than 100 times more expensive than a page hit memory access.

5 Networking

6 File System

7 Interesting Founding Outs

References

[1] Frank Denneman. *Memory Deep Dive: Memory Subsystem Bandwidth*. 2015. URL: http://frankdenneman.nl/2015/02/19/memory-deep-dive-memory-subsystem-bandwidth/ (visited on 02/26/2017).

[2] Larry W McVoy, Carl Staelin, et al. "lmbench: Portable Tools for Performance Analysis." In: *USENIX annual technical conference*. San Diego, CA, USA. 1996, pp. 279–294.

[3] Wikipedia. *Hard disk drive — Wikipedia, The Free Encyclopedia*. [Online; accessed 27-February-2017]. 2017. URL: https://en.wikipedia.org/w/index.php?title=Hard_disk_drive&oldid=765835114.