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

March 20, 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** by default.

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 SATA hard drive disk. The disk model is HDS721075CLA332 with 32MB cache and 7200RPM spindle speed. The external data transfer rate is 300Mbps.

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:

```
#define START_COUNT(cycles_high, cycles_low)       \
    asm volatile ("CPUID\n\t" "RDTSC\n\t" \
                  "mov %%edx, %0\n\t" \
                  "mov %%eax, %1\n\t": "=r" (cycles_high), "=r" (cycles_low)::\
                  "%rax", "%rbx", "%rcx", "%rdx")

#define STOP_COUNT(cycles_high1, cycles_low1)      \
    asm volatile ("RDTSCP\n\t" \
                  "mov %%edx, %0\n\t" "mov %%eax, %1\n\t" \
                  "CPUID\n\t": "=r" (cycles_high1), "=r" (cycles_low1):: \
                  "%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 macros, so that this measurement will response the overhead of the execution of those two macros.

When performing the measurement, we will run the measurement code in a 10000 round loop, then we take the arithmetic mean of the cycles responsded 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	Standard Deviation
6 (inst) or 44 (cycles)	0	6 (instructions) (15.7520 ns)	102 (36.5161 ns)	3.8

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 assembly 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 determine 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 arithmetic 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**:

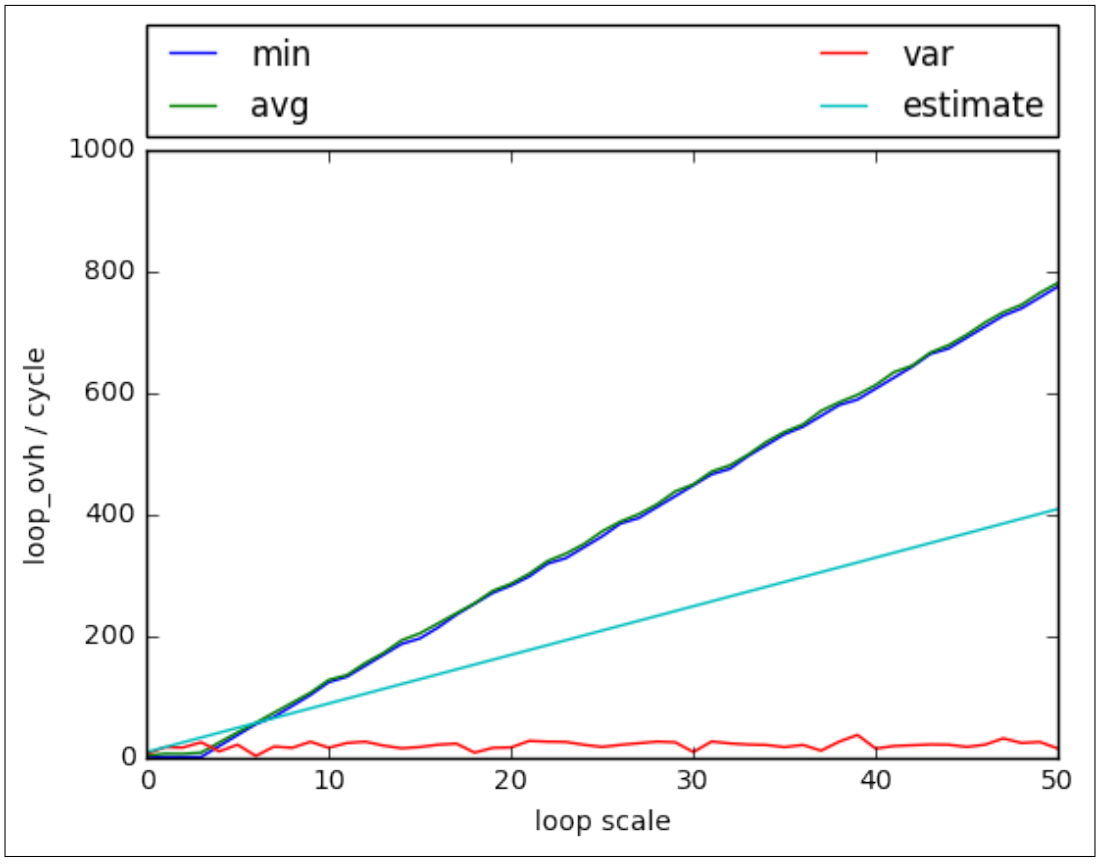

Figure 1: Loop Overhead: Estimation and Experiment Results

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 incremental 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 arithmetic 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. There is a sudden jump on overhead between 2 and 3 parameters calls, and we speculate it is caused by the different alignments when the callees push 2 and 3 parameters to stack to save registers.

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 libc, so that only the first call of the getpid() can trap into the kernel space. The **getcwd()** system call will always trap into kernel, so that it is a choice to measure system call overhead.

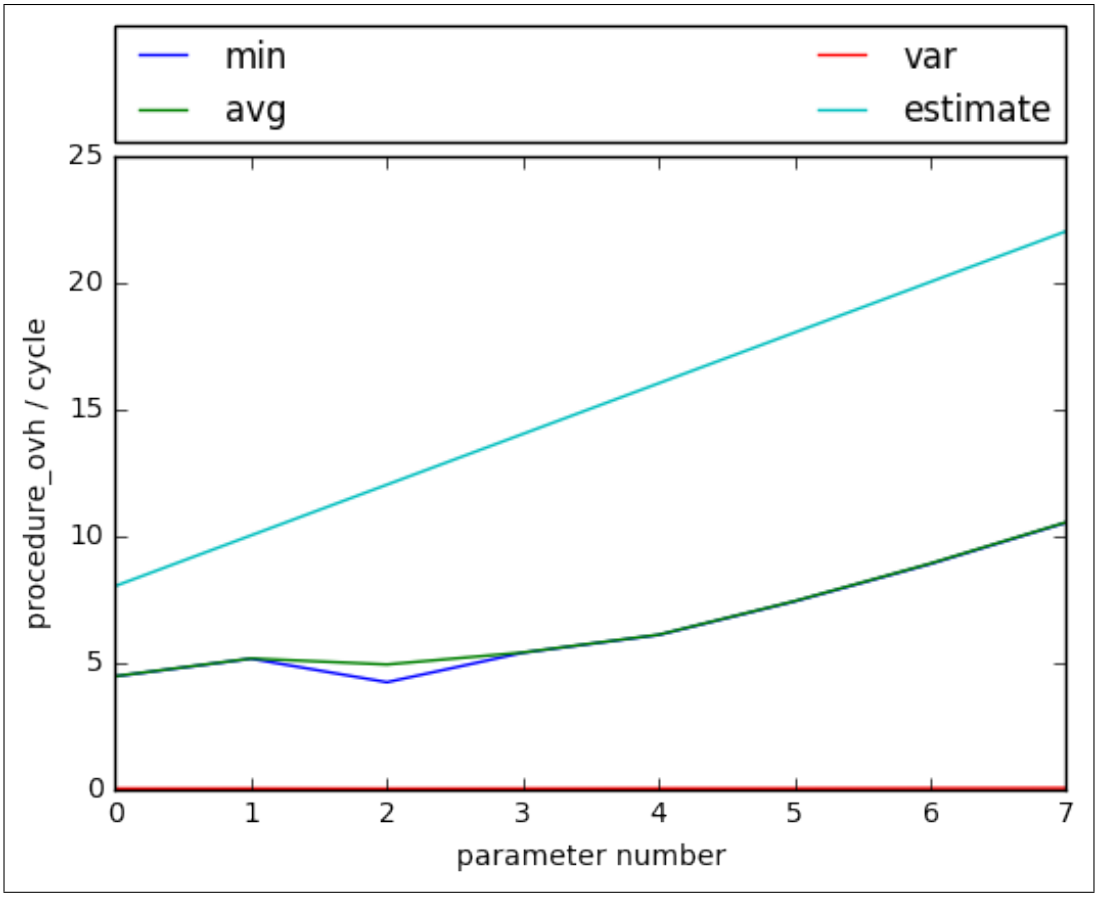

Figure 2: Procedure Overhead: Estimation and Experiment Results

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.

On the other hand, as system call cache is made by libc layer, another way to bypass this cache is to directly use linux kernel abis, instead of libc wrapping functions. There are two way to invoke kernel system call abis, first, by interrupt gate, and second, by syscall instruction. For example, when invoke **getpid()** system call, we can use:

```
1    #define __NR_getpid 20
2    asm volatile
3    (
4        "int \$0x80"
5        : "=a" (pid)
6        : "0"(__NR_getpid)
7        : "cc", "edi", "esi", "memory"
8    );
```

or

```
1    #define __NR_getpid 39
2    asm volatile
3    (
4        "syscall"
5        : "=a" (pid)
6        : "0"(__NR_getpid)
7        : "cc", "rcx", "r11", "memory"
8    );
```

For the second method (bypass glibc layer), we measure and compare the overhead of **getpid()** and **getcwd()**.

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**

system call type	H/W Overhead Estimation	Total Overhead Estimation	Expr. Results	Standard Deviation
getcwd() libc	1000 cycles	1000 cycles (358.001 ns)	502.9880 cycles (180.0702 ns)	10.3023
getcwd() int	1000 cycles	1000 cycles (358.001 ns)	2554.4114 cycles (914.4818 ns)	142.9443
getcwd() syscall	1000 cycles	1000 cycles (358.001 ns)	493.6172 cycles (176.7154 ns)	1.5946
getpid() int	1000 cycles	1000 cycles (358.001 ns)	550.3574 cycles (197.0285 ns)	22.3033
getpid() syscall	1000 cycles	1000 cycles (358.001 ns)	152.9067 cycles (54.7408 ns)	0.7112

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 should be a reasonable estimation.

The results shows that for getcwd(), in a typically 64 bit linux kernel, the overhead should be around 500 cycles or 180 ns, and for getpid(), the overhead should be around 152 cycles or 55 ns. This is much more lower than what we estimate, and the result for it is that: in 64 bit, system call is optimized to running in a pretty fast way. This can be proved by the comparison between same system call invoked by interrupt gate and syscall instruction: syscall instruction overhead is much lower than interrupt gate.

For getcwd() system call, the overhead for glibc version and invoking by syscall instruction is similar, which proves that in 64 linux glibc, syscalls are invoked by syscall instruction. Furthermore, compared with getcwd(), results show that getpid() is much more light weight.

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 several 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 scheduler 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 communicate
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 repeat 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 (48.154μs)	3062
Kernel Thread	NA	NA	10000 (3.58μs)	20635 (7.387μs)	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 construct a new address space. Finally we made a prediction 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 communicating 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 difference is we use **pthread_cond_wait()** to synchronize threads.

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 (7.16μs)	79097 (28.317μs)	650
Kernel Thread	NA	NA	10000 (3.58μs)	2562 (0.917μs)	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 specifically, we will report measurements about access latency on each level of memory hierarchy, the memory bandwidth, and the overhead of page fault handling.

4.1 Memory Latency

4.1.1 Methodology

When measuring access latency on each level of memory hierarchy, 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 decided 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 accessing 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 accessing memory on each measurement, and totally perform 1000 (stored in **step**) measurements.

4.1.2 Estimation and Results

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

Memory Hiracht	**Latency Estimation**	**Expr. Results** (AVG)
L1	5 cycles (1.79 ns)	9 cycles (3.22 ns)
L2	20 cycles (7.16 ns)	20 cycles (7.16 ns)
L3	60 cycles (21.48 ns)	70 cycles (25.06 ns)
Memory	150 cycles (53.70 ns)	190 cycles (68.02 ns)

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 3**, the real lines separate the latency turning point for different level of memory hierarchy, and the dotted lines separate the real size of different level of memory hierarchy 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.

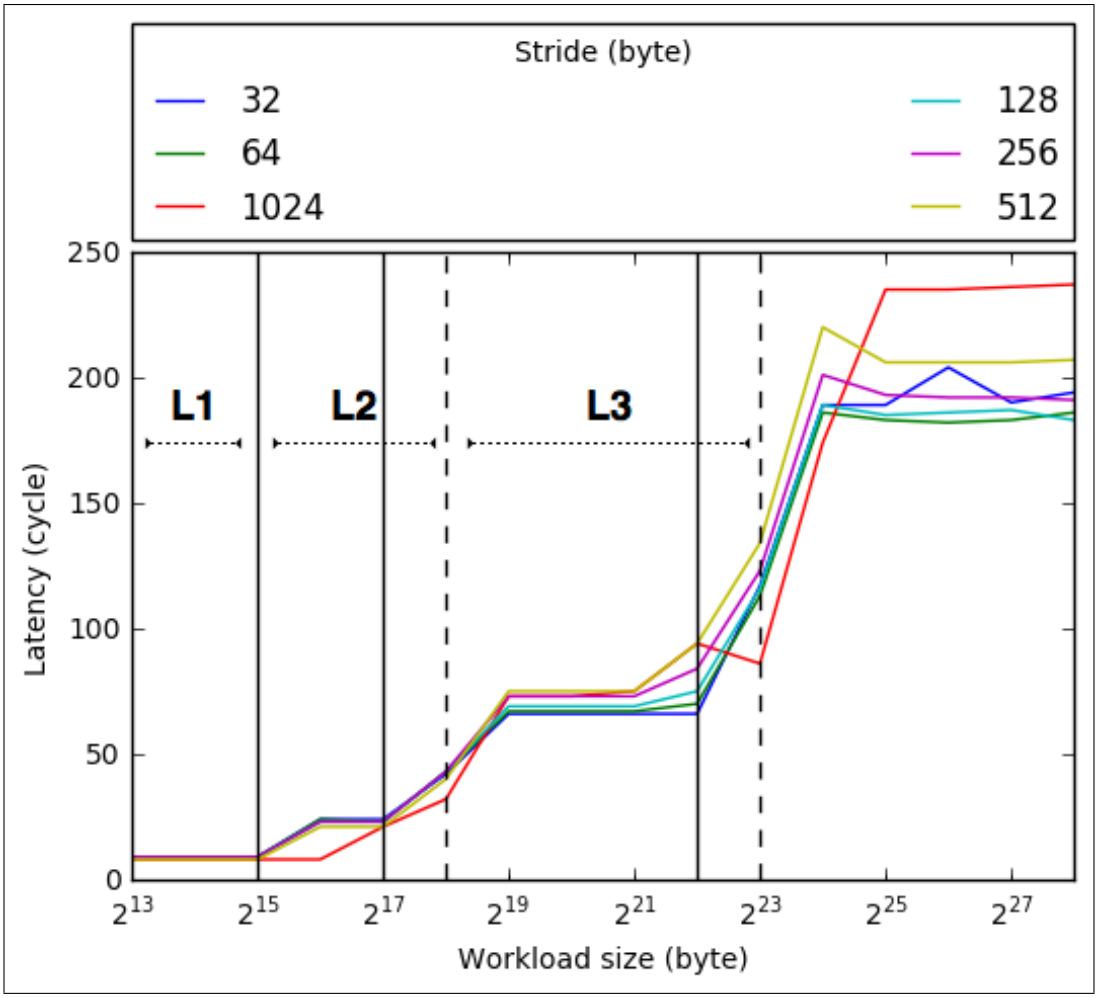

Figure 3: Memory Hirachy Latency: Estimation and Experiment Results

Table 5 shows the latency estimation 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 use the method metioned in [2], which is using integer reading and writing which miss at l3 cache to measure the reading and writing bandwidth for main memory. However, there are several technical details not mentioned in [2] we need to point out here.

First, although we are reading and writing 4 Byte integers when measuring code, when calculating memory bandwidth, we need to use 64 Byte as the size for each memory access, as when transfering data back and forth between main memory and l3 cache, data is always transferred with a l3 cache line size. The l3 cache line size can be retrieve by command:

```
1 cat /sys/devices/system/cpu/cpu0/cache/index3/coherency_line_size
```

At the same time, in order to get ride of cache line prefetch, we need to accesing data with a stride at least as large as the l3 cache line size. In our experiments, we use 256 byte (or 64 integers) as a accesing stride.

Furthermore, by using command:

```
sudo lshw -C memory
```

we can retrive the writing policies for each level cache. In our experiment computer, l1 and l2 cache are both write-through cache, while l3 cache is a internal write-back unified cache. Although it does not mention the write miss policy, [3] indicates that write back policy usually pairs with write allocation policy. Due to these fact, there is some details in the implementation:

1. In order to get ride of caching effect of l3 cache, we first fill l3 cache with data that will never be read and write during measurement process.
2. When measuring reading bandwidth, we need to fill l3 cache with clean cache lines, orderwise when we reading the measurement data from main memory to l3 cache, there will be a high dirty block writing back overhead when evict cache lines.
3. When measuring writing bandwidth, we want all measurement data to experience a write miss. However, l3 cache itself is write allocation, thus even a write miss will not write back to main memory. In this case, the way we solve this problem is filling all l3 cache line with dirty cache data before we start measurement. As a result, each time we write a measurement writing data, as l3 cache is a write allocation cache with full of dirty data, so that each time l3 cache will evict a dirty line as each time we are accessing different l3 cache set, which means that each time we write, we will finally write a dirty cache line back to main memory, and thus we can measure the memory writing bandwidth.

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	7131.15239595 MB/s	394.490122927
Write	10600 MB/s	6709.49876645 MB/s	1191.12060254

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 **7131.15239595 MB/s**, and writing bandwidth should be **6709.49876645 MB/s**. Measurement result is below the estimation value, we think the reason for that is:

First, in this experiments, under our accessing pattern, it is hard for main memory to achieve its theorical optimized reading and writing speed, as the bank row cache inside the DRAM will miss, thus an entire row and column retrive process is needed each time we accessing;

Second, the software overhead is unavoidable, this will also decrease our memory performance.

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

We used to create a 150 MB file in /tmp directory for mapping, however, it turns out that the overhead for handling pagefault after mapping in this file is significantly smaller than a HDD's overhead, and there is no seek penalty exist, and obviously this file is actually stored in memory. In the second round, we changed the directory for the tmp mapping file to the current directory of the benchmark binary, the expanded the file size to 16 GB, after that, we can observe seek penalty now.

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**
28000000 cycles	10000 cycles	28010000 cycles (10.03 ms)	22672005.8858 cycles (8.12 ms)	7299383.2799

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

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

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

In checkpoint 2 we made a mistake: we ignored that in a HDD environment, seek and rotation is the most significant overhead for page fault handling, when taking in consider about seek and rotation, the time consumption is of the order of 10 ms. So that, the hardware overhead should be:

$$\frac{10ms}{0.350001ns/cycle} = 27932882cycles$$

So the totally hardware overhead should be around $28000000cycles$

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 28010000 cycles.

The result is shown in **Table 7**. We underestimate the overhead of the pagefault handling by around 2 ms. As there is a high variance for the performance of the disk, it is not strange that this difference exist. It will not experience the largest disk access overhead for each access request.

The result is shown in **Table 7**. The speed is a little bit faster than what we estimate, this should be caused by each time a page fault happened, the disk seeking and rotation overhead is not always as high as 10 ms.

For the question about speed compare between memory bandwidth and page fault page transfer bandwidth on byte scale, according to **Section 4.2.2**, on average the memory can transfer a byte in

$$\frac{1Byte}{7131.15239595MB/s} = 0.133734ns$$

and the page fault mechanism can transfer a byte in

$$\frac{1Byte}{\frac{4KB}{8.12ms}} = 1982.421875ns$$

.

Thus the page fault handling is around 15000X more expensive than a page hit memory access.

5 Networking

This section we will mainly discuss our measurement about network performance on the target machine. More specifically, we will report measurements about round trip time, peak bandwidth and connection overhead for the tcp protocol.

5.1 Machine Description

In our experiments of this part, the client is running on the original machine and the server is running on the new machine. Below is the machine description of the server.

5.1.1 CPU

The processor on this machine is an AMD Athlon(tm) II X4 630 Processor @ 2.80GHz with one core.

5.1.2 Memory

This machine has totally 4G main memory.

5.1.3 NIC

The NIC is Broadcom Corporation NetLink BCM57788 Gigabit Ethernet PCIe with 1Gbit/s capacity.

5.1.4 Operating System

When running this project, the operating system running on the machine is normal Ubuntu 16.04.1. The kernel is Linux 4.4.0-51-generic x86_64.

5.2 Round Trip Time

5.2.1 Methodology

When measuring round trip time, basically we measure the round trip time by calculating time interval between the send time and the ack time for each packet. We first start a tcp server and a tcp client. And then, the client connects the server and sends one byte data to the server every 1 millisecond. We

use tcpdump to capture tcp traffics on the client side and calculate the round trip time according to the tcpdump result. We recognize the send and the ack packet by checking whether two consecutive packets match the flag patterns [P.] and [.] respectively. Below is an example of the send and ack pattern for remote interface:

```
1 23:10:17.953051 IP 0xcc.ucsd.edu.39660 > dyn54.sysnet.ucsd.edu.9014: Flags [P.],
    seq 1:2, ack 1, win 229, options [nop,nop,TS val 890347228 ecr 200781447],
    length 1
2 23:10:17.953387 IP dyn54.sysnet.ucsd.edu.9014 > 0xcc.ucsd.edu.39660: Flags [.],
    ack 2, win 227, options [nop,nop,TS val 200781447 ecr 890347228], length 0
```

In order to make the round trip time more accurate, on the server side, we mark each socket quick ack, which means that the tcp server immediately reply an ack after receiving each packet. In our experiment, the tcp client sends 10,000 times 1 byte data to the server.

5.2.2 Estimation and Results

Table 8: **Round Trip Time: Estimation and Experiment Results**

Remote or Local	**RTT Estimation**	**Expr. Results** (AVG)	**Standard Deviation**
Remote	0.395ms	0.338 ms	0.099
Local	0.025ms	0.028ms	0.095

We take average of 100 ping results as our estimation. For both the remote interface and the local interface, the estimations are close to the experiment result with tolerable errors.

5.3 Peak Bandwidth

5.3.1 Methodology

When measuring peak bandwidth, basically we measure the peak bandwidth by calculating maximum number of bytes newly acked by the server every second. We first start a tcp server and a tcp client. And then, the client connects the server and sends 5GB data to the server. We use tcpdump to capture tcp ack traffics on the client side and calculate peak bandwidth according to the tcpdump result.

5.3.2 Estimation and Results

Table 9: **Peak Bandwidth: Estimation and Experiment Results**

Remote or Local	**Peak Bandwidth Estimation**	**Expr. Results** (AVG)
Remote	125MB/s	112MB/s
Local	2.6GB/s	2.7GB/s

For remote estimation, its remote interface is ethernet and has 1Gbit/s capacity according to machine description. And in the tcp protocol, its maximum protocol bandwidth is

$$\frac{maximum\_window\_size}{estimated\_round\_time\_trip} = \frac{65536Bytes}{0.000395seconds} = 166MB/s$$

So the remote estimation for the remote interface is 125MB/s.

For local estimation, its local interface is loopback and its bandwidth should be equal to the memory bandwidth which is 10GB/s. And in the tcp protocol, its maximum protocol bandwidth is

$$\frac{maximum\_window\_size}{estimated\_round\_time\_trip} = \frac{65536Bytes}{0.000025seconds} = 2.6GB/s$$

So the local estimation for the remote interface is 2.6GB/s.

For remote result, the experiment is a little less than the estimation. The reason should be that there are transmission overhead in network.

For local result, the experiment is a little larger than the estimation. The reason should be that the round trip time is so small that even a tiny error may cause the huge change in the estimated bandwidth value.

5.4 Connection Overhead

5.4.1 Methodology

When measuring connection overhead, basically we measure the connection overhead by calculating time interval between the sending the first setup packet and sending the last setup packet for each connection and time interval between sending the first teardown packet and sending the last teardown packet for each connection. We first start a tcp server and a tcp client. And then, the client connects the server and then closes the connection after 1 millisecond. We use tcpdump to capture tcp traffics on the client side and calculate the connection overhead according to the tcpdump result. We recognize setup packets by checking whether three consecutive packets match the flag patterns [S], [S.], [.] respectively and teardown packets by checking whether three consecutive packets match the flag patterns [F.], [F.], [.]. Below are examples of the setup and teardown patterns for remote interface:

```
1 21:56:37.340417 IP 0xcc.ucsd.edu.55148 > dyn54.sysnet.ucsd.edu.9014: Flags [S],
    seq 1531103784, win 29200, options [mss 1460,sackOK,TS val 867642075 ecr 0,nop
    ,wscale 7], length 0
2 21:56:37.340766 IP dyn54.sysnet.ucsd.edu.9014 > 0xcc.ucsd.edu.55148: Flags [S.],
    seq 1736035707, ack 1531103785, win 28960, options [mss 1460,sackOK,TS val
    178076294 ecr 867642075,nop,wscale 7], length 0
3 21:56:37.340805 IP 0xcc.ucsd.edu.55148 > dyn54.sysnet.ucsd.edu.9014: Flags [.],
    ack 1, win 229, options [nop,nop,TS val 867642075 ecr 178076294], length 0
```

```
1 21:56:37.341915 IP 0xcc.ucsd.edu.55148 > dyn54.sysnet.ucsd.edu.9014: Flags [F.],
    seq 1, ack 1, win 229, options [nop,nop,TS val 867642076 ecr 178076294],
    length 0
2 21:56:37.342255 IP dyn54.sysnet.ucsd.edu.9014 > 0xcc.ucsd.edu.55148: Flags [F.],
    seq 1, ack 2, win 227, options [nop,nop,TS val 178076294 ecr 867642076],
    length 0
3 21:56:37.342287 IP 0xcc.ucsd.edu.55148 > dyn54.sysnet.ucsd.edu.9014: Flags [.],
    ack 2, win 229, options [nop,nop,TS val 867642076 ecr 178076294], length 0
```

5.4.2 Estimation and Results

Table 10: **Setup Overhead: Estimation and Experiment Results**

Remote or Local	**Setup Overhead Estimation**	**Expr. Setup Results** (AVG)	**Standard Deviation**
Remote	0.405ms	0.353ms	0.031
Local	0.035ms	0.029ms	0.004

Table 11: **Teardown Overhead: Estimation and Experiment Results**

Remote or Local	**Teardown Overhead Estimation**	**Expr. Teardown Results** (AVG)	**Standard Deviation**
Remote	0.415ms	0.361ms	0.043
Local	0.045ms	0.042ms	0.012

For setup overhead estimation, there is a three-way handshake process between the client and the server. So the setup overhead is equal to RTT + time of processing the setup packet on the server

side + time of sending the third setup packet on the client side. In our estimation, remote RTT is equal to 0.395ms and local RTT is equal to 0.025ms, processing time and sending time should be some microseconds. So the remote setup estimation is 0.405ms and the local setup estimation is 0.035ms.

For teardown overhead estimation, in our case, the teardown is three-way since only the client actively close the connection. So the teardown overhead is equal to RTT + time of processing the teardown packet on the server side + time of sending the third teardown packet on the client side. In our estimation, remote RTT is equal to 0.395ms and local RTT is equal to 0.025ms, processing time and sending time should be some microseconds but it needs to go through some user-level logics on the server side. So the remote teardown estimation is 0.415ms and the local teardown estimation is 0.045ms.

The difference between estimation and result is small and acceptable.

6 File System

This section presents our measurement methodology and result on file system performance.

6.1 Size of File Cache

6.1.1 Methodology

File cache is adopted by the OS to improve the file access performance on disk. To measure the file cache size in our system, we generate files of different sizes and measure the average sequence read time of a block. The expectation is when the read file is small, the whole file could be cached in memory so the reading time would be as short as memory access; but when the file is large, the file cache is not enough to cache the whole file, so some read would cause cache miss and thus leads to disk access, which is much longer than memory access. Based on this, our methodology determines the file cache size by observing on which file size, the access time increases largely.

More specifically, we created files of sizes from 128MB to 8GB increased by a step of 128MB. We choose the upper bound of file size since the total physical memory of the tested machine is 8GB. To bypass the cache in the C library, we use the **open()** system call directly instead of using the **fopen()** function call in C library. To Warm up the cache and to avoid outliers, each file is read for 11 times and the average of last 10 times is took as the result.

6.1.2 Estimation and Results

Since the tested machine's total physical memory is 8GB, the linux kernel caches files very aggressive, and the tested system is Ubuntu, which usually takes less than 1GB memory, we estimate the file cache size would be between 7GB to 8GB.

Figure 4 shows how the measured read time varies against different file sizes. We can see that when the file size increases to 7296MB, the reading time increases largely, from 0.5us to 38us. So the file cache size should be about 7296MB (7.1GB), which is in the range we estimated.

6.2 File Read Time

6.2.1 Methodology

To measure the file read time without file cache, we use the **open()** system call with the flag **O_DIRECT**, which indicates reading directly from disk instead of cache. For sequential read time, we use **read()** to read a 4K block each time and accumulate the total time of reading from offset 0 to the file end. For

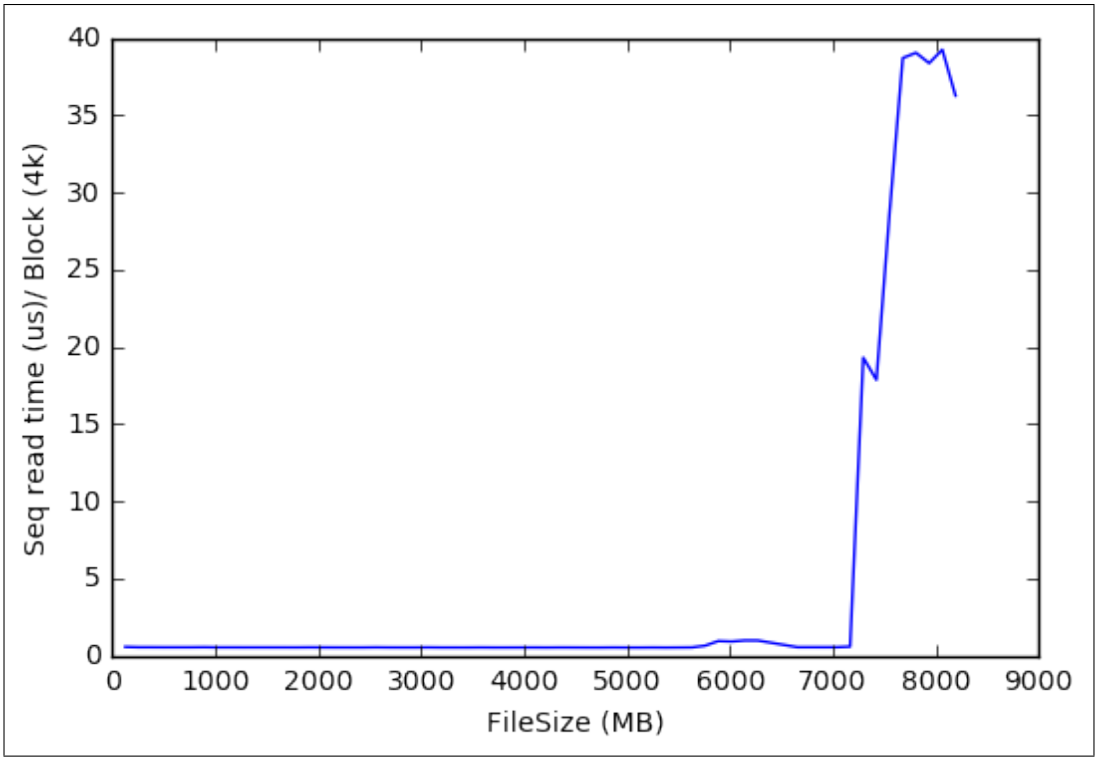

Figure 4: Reading time of different sized files

random read time, we use **(rand()%blocknum)*blocksize** to generate read offset and do the sampled read 1000 times for each file. We generate files of sizes from 4MB to 1GB as reading workload. And both the sequential read and random read test are conducted for 10 times and the average read time is calculated as the result

6.2.2 Estimation and Results

We estimate the sequential read time would only be determined by the disk data transfer rate, without seek time and rotation latency. So the sequential read time of different sized file would be the same. And since the max data transfer rate is 300Mbps (37.5MB/s), we estimate the sequential read time per block would be:

$$\frac{4KB}{37.5 * 1000KB/s} = 107us.$$

For the random read time, we take into account seek time and rotation latency. As the rotation speed is 7200RPM, the average rotation latency should be:

$$\frac{\frac{60s}{7200}}{2} = 4.2ms.$$

And since the usual seek time of a hard disk is about 9ms, the random read time per block should be:

$$107us + 4.2ms + 9ms = 13.3ms.$$

And we estimate the random read time will increase as the file size increase, because large files may locates in many cylinders, which requires many head moving while be random read, while small files can be stored in a single cylinder, which requires no head moving.

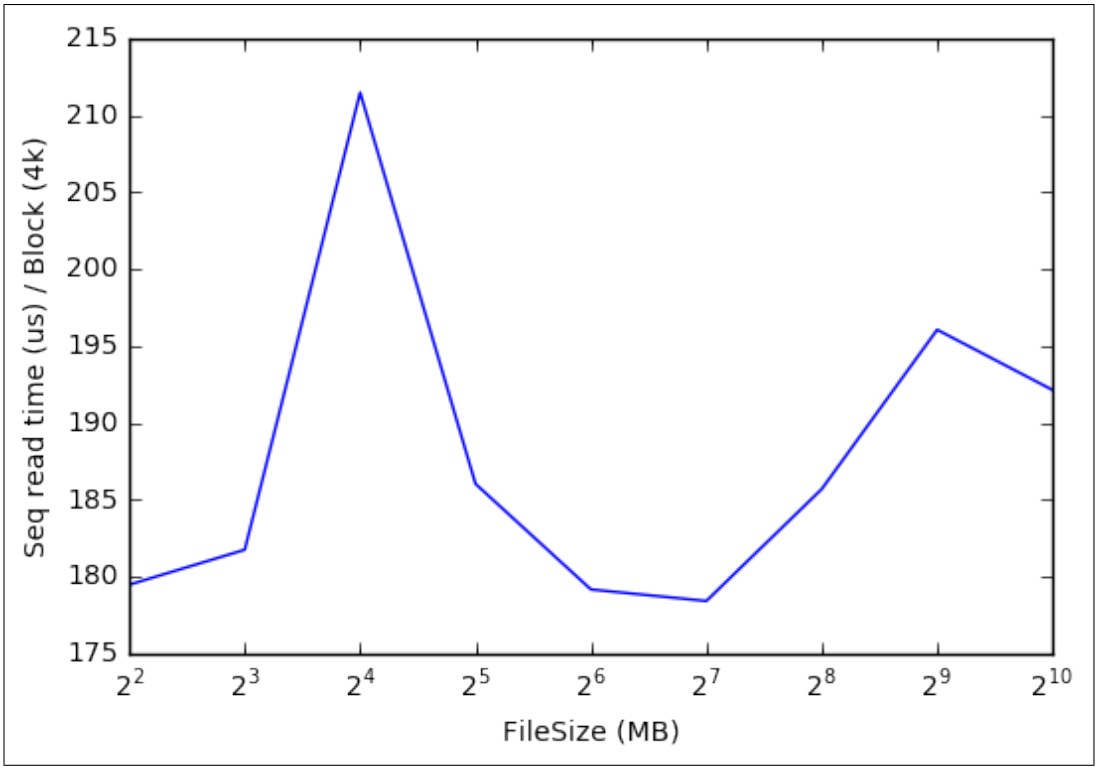

Figure 5: Sequential read time per block (4KB)

Figure 5 shows sequential read time of different file sizes. The total average per-block read time is **187.8us**, which is a little longer than we estimated (107us). In addition, the read time varies when the file size increase. We speculate the reason is that a file is not totally stored in sequential disk sectors. The data blocks may be on different cylinders, especially for large files. So seeking time and rotation latency are also necessary for sequential read. And since sequentially reading different-sized files need different times of seeking and rotation, the per-block read time varies. For large files, although more seeking and rotation may be required, but the large block number can also amortize the latency; therefore, a large file may have a shorter read time than a small file, as **Figure 5** shows.

Figure 6 shows sequential read time of different file sizes. The average per-block read time increases as the file size increases, as we expected. As aforementioned, the reason is that large files span more sectors and cylinders than small files, so more seeking and rotation is needed for reading. And for small files (4MB to 16MB), increasing of file sizes only causes small increasing of read time, compared to that of large files (16MB to 512MB). The reason is that file size increasing of small files only leads to more sectors while file size increasing of large files can cause more cylinders.

6.3 Remote File Read Time

6.3.1 Methodology

To test the read time of remote files, we set up a NFS sever on a different machine, as described in **Section 5.1**, and mount it on our main testing machine. The key challenge is to bypass both the server side and client side file cache. One possible way to address it is use **O_DIRECT** flag when call open(). However, that flag only take effect when the linux kernel support it, and unfortunately our testing system kernel doesn't support that. As a result, we adopts another way: free the file cache before conducting test. That is implemented by calling:

```
1 sudo sh -c 'echo 3>/proc/sys/vm/drop_caches'
```

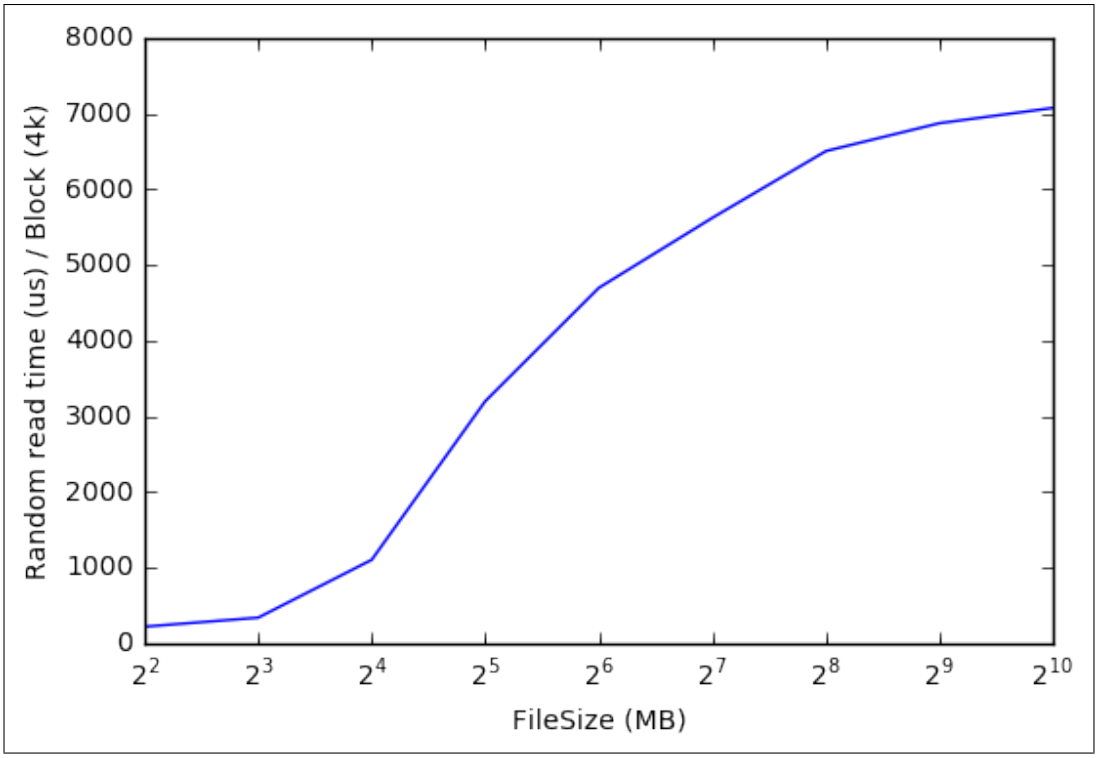

Figure 6: Random read time per block (4KB)

And to avoid NFS transfers more than one block when a data block is read, we set the both read size and write size to 4KB:

```
1 sudo mount dyn54:/export /mnt -o rsize=4096,wsize=4096
```

6.3.2 Estimation and Results

We estimate the network penalty with the round trip time and network bandwidth we246. have measured. **Section 5.1** shows that the measured round trip time is 0.338ms (338us) and the peak network bandwidth is 112MB/s, so the network penalty for a block should be:

$$\frac{4KB}{112 * 1000KB/s} + 338us = 373.7us$$

Figure 7 shows both the remote and local sequential read time of different file sizes. The average per-block read time are 865.3us and 187.8us correspondingly. So the average network penaly is:

$$865.3us - 187.3us = 678us.$$

Compared with our estimated network penalty, the measured one is in the same order of magnitude but almost twice bigger. We speculate the reason is that the NFS implementation introduces additional overhead on read time.

Figure 7 shows both the remote and local random read time of different file sizes. The average per-block read time are 4365.2us and 3961.7us correspondingly. So the average network penaly is:

$$5832.3us - 3961.7us = 1870.5us.$$

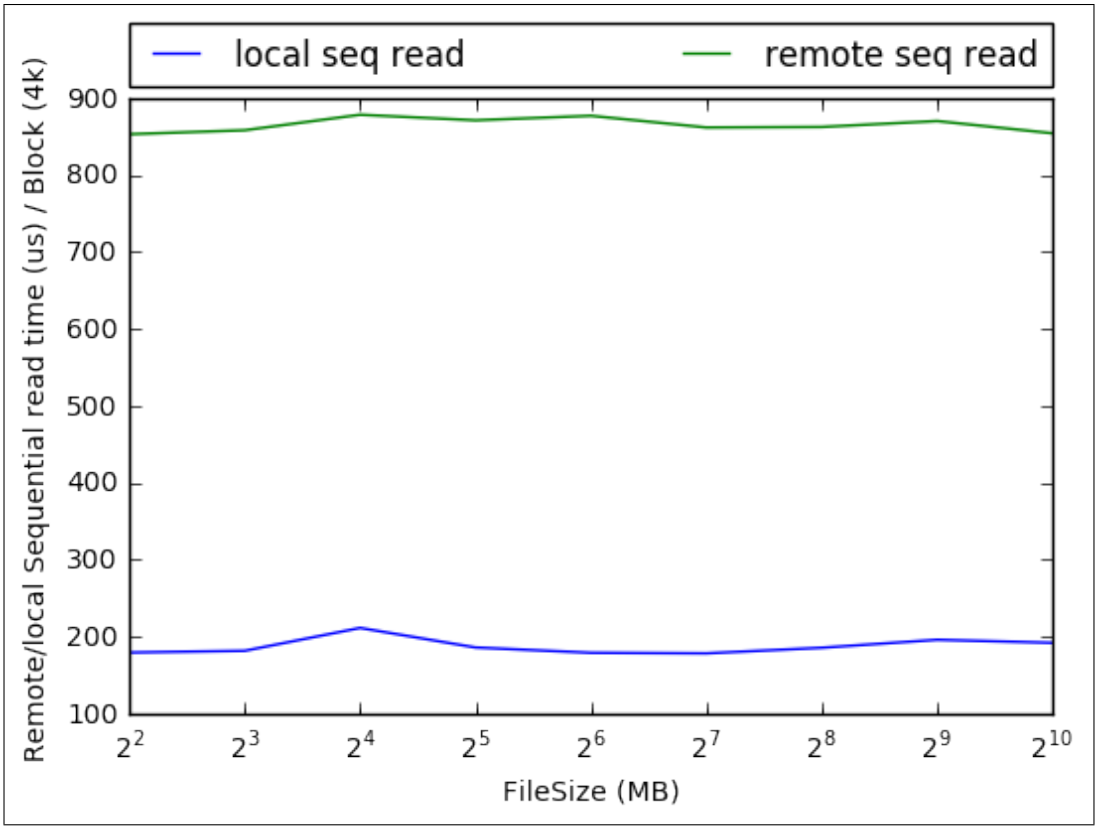

Figure 7: Remote/Local sequential read time per block (4KB)

This penalty is much bigger to the sequential reading one. We speculate the reason is that the remote disk needs more seeking time for random read. The remote disk is 160GB with 8MB cache while local disk is 750GB with 32MB cache. Since the capacity difference, one cylinder of the remote disk should have less sector than the local disk. So the same sized file on the remote disk would span more cylinder than on the local disk, which leads to more seeking time.

6.4 Contention

6.4.1 Methodology

To measure the contention effect of different process reading different files on the same disk, we generate **10** different files with the same size of **128MB** and create **1** to **10** processes to do the reading test. We uses the **open()** call with **O_DIRECT** to bypass the file cache, same as **Section 6.2**. To calculate the time of reading a block, we write test program to sequentially read the file once a block from the beginning to the end and calculate the average time as the result. We repeat the each test for 10 times to eliminate outerliers.

6.4.2 Estimation and Results

Since different processes share the same disk bandwidth when they read the same disk simultaneously, we estimate the reading time increases by times of the process number. As we measured before, the per-block read time of one process is 188us, we estimate contented read time would be:

$$processnumber * 188us$$

Figure 9 show the per-block read time with contention of different process number. We can see that

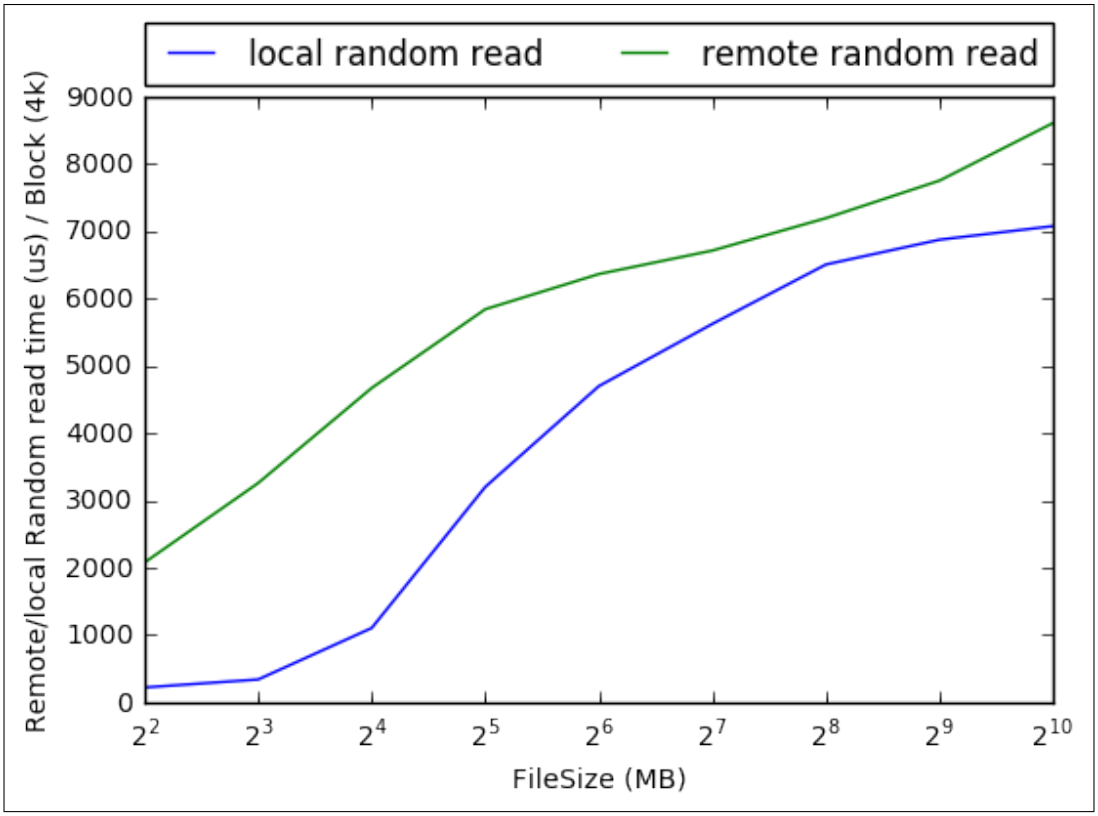

Figure 8: Remote/Local random read time per block (4KB)

the measured read time increase linearly almost as the line:

$$readtime = 174 + process * 240(us)$$

The measured read time is quite close to our estimation but a little bit higer. That is reasonable since we have not taken into account the process switch overhead.

7 Summary

7.1 Code Mapping

We explain in this subsection for all our source code measuring which part of this project.

7.1.1 CPU

- Measurement Overhead: time_ovh.c
- Loop Overhead: loop_ovh.c
- Procedure Call: procedure.c
- System Call: syscall*.c
- Process Creation: process.c
- Thread Creation: thread.c
- Process Context Switch: proc_switch.c
- Thread COntext Switch: thread_switch.c

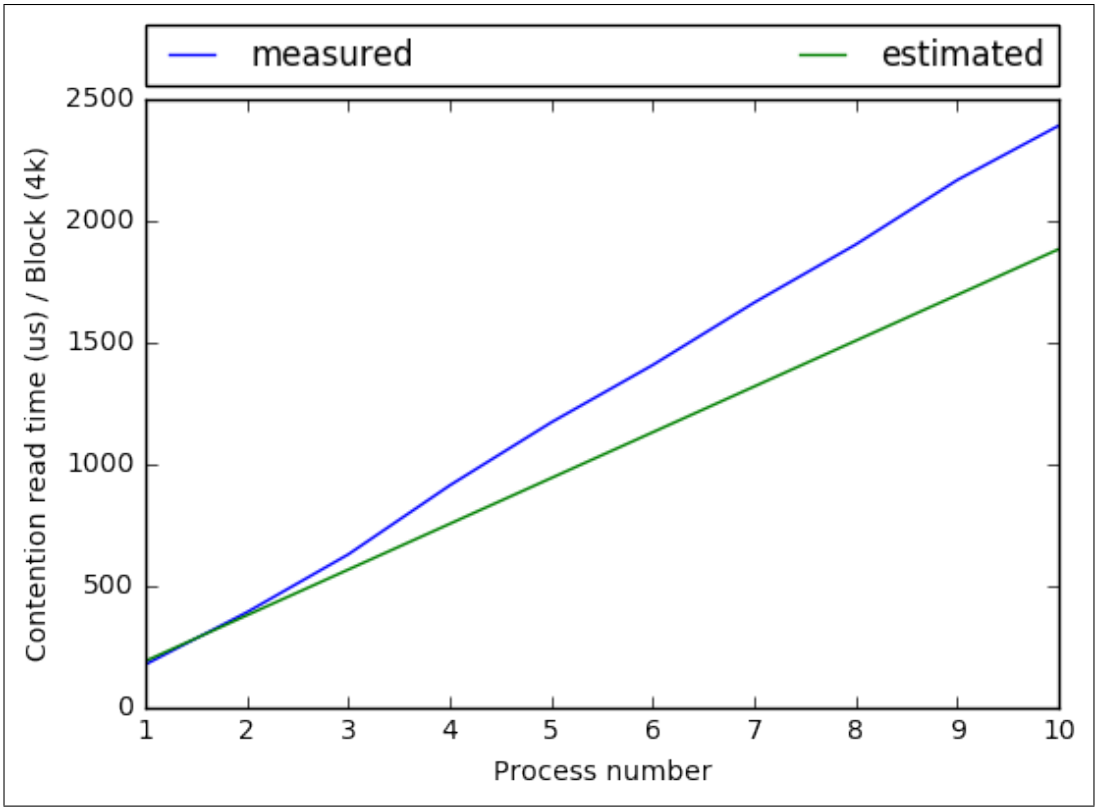

Figure 9: Remote/Local random read time per block (4KB)

7.1.2 Memory

- Memory Hirachy Latency: access_ptr.c
- Memory Bandwidth: bandwidth*.c
- Page Fault Handling: pagefault.c
- Large File Generate: largefile.c

7.1.3 Networking

- Round Trip Time: rtt.c
- Peak Bandwidth: peak.c
- Connection Overhead: conn.c
- Server Side: server.c

7.1.4 File System

- File Cache: filecache.c
- Sequential File Read: seqread.c seqread.sh
- Random File Read: randread.c randread.sh
- Remote Sequential File Read: rseqread.sh
- Remote Random File Read: rrandread.sh
- Contention: contention.c

7.2 Measurement Results Summary

Here we summary all the results we have in **Table 12**

Table 12: **Summary The Results**

Operation Type	Base Hardware Performance	Est. Software Overhead	Predicted Time	Measured Results	Standard Deviation
Measurement	15.7520 ns	0	15.7520 ns	36.5161 ns	1.36
Loop	1.79 ns/round	0	1.79 ns/round	6.3 ns/loop round	N/A
Procedure Call	¡ 22 ns	N/A	¡ 22 ns	¡ 4 ns	0.0607
System Call	358.001 ns	N/A	358.001 ns	54.7408 ns	0.2546
Process Creation	N/A	N/A	7.16μs	48.154μs	1.096
Thread Creation	N/A	N/A	3.58μs	7.387μs	0.553
Process Switch	N/A	N/A	7.16μs	28.317μs	0.233
Thread Switch	N/A	N/A	3.58μs	0.917μs	0.193
L1 Cache Access	1.79 ns	N/A	1.79 ns	3.22 ns	N/A
L2 Cache Access	7.16 ns	N/A	7.17 ns	7.17 ns	N/A
L3 Cache Access	21.48 ns	N/A	21.48 ns	25.06 ns	N/A
Memory Access	53.70 ns	N/A	53.70 ns	68.02 ns	N/A
Mem Read Bandwidth	10600 MB/s	N/A	10600 MB/s	7131.1524 MB/s	394.4901
Mem Write Bandwidth	10600 MB/s	N/A	10600 MB/s	6709.4988 MB/s	1191.1206
Page Fault	10.024 ms	0.004 ms	10.03 ms	8.12 ms	2.6131
RTT Remote	0.395 ms	N/A	0.395 ms	0.338 ms	0.099
RTT Local	0.025 ms	N/A	0.025 ms	0.028 ms	0.095
Local Peak Bandwidth	N/A	N/A	125 MB/s	112 MB/s	N/A
Remote Peak Bandwidth	N/A	N/A	2.6 GB/s	2.7 GB/s	N/A
Remote Conn. Setup	N/A	N/A	0.045 ms	0.353 ms	0.031
Local Conn. Setup	N/A	N/A	0.035 ms	0.028 ms	0.031
Remote Conn. Teardown	N/A	N/A	0.415 ms	0.361 ms	0.043
Local Conn. Teardown	N/A	N/A	0.045 ms	0.042 ms	N/A

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. *Cache (computing) — Wikipedia, The Free Encyclopedia*. [Online; accessed 20-March-2017]. 2017. URL: `https://en.wikipedia.org/w/index.php?title=Cache_(computing)&oldid=770496427`.

[4] 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`.