System Measurement Project

Charlie Chen, Chicheng Zhang

December 13, 2013

1 Introduction

In this paper, we discuss our method of benchmarking a certain system. Our objective is to accurately measure various operations of the operating system, the various overheads and bottlenecks we experience in our everyday computing. Implementation of these tests will help greatly with our understanding of the system, allowing us to program more efficiently to take advantage of the intricacies of the modern operating system, as well as not falling for what may appear to be quicker code that scales poorly. For this project, we look into characteristics of various parts of hardware, from the CPU, RAM, disk accesses, network, and the file system itself. We analyze the constraints of the hardware and the software alike.

Our benchmarks are programmed using C and compiled using gcc in a Linux kernel. We chose C because we feel it is the most applicable language for this task, being lower level than the runtime languages like Java or Python. We used -O0 compiler optimizations(no optimization) explicitly, because a lot of tests we do will involve using empty functions or loops that the compiler may choose to ignore.

In the Linux kernel, we modified the grub boot settings and set isolcpus to 1 on our dual core processor. What this essentially accomplishes is that it will run most of the kernel threads and all user processes on a single core(except for a few kernel threads), leaving the second core untouched unless explicitly called. By setting the CPU affinity for our test benchmarks(sched_setaffinity function), we are able to then run the benchmark (almost)exclusively on the other processor. We feel this allows us to eliminate a lot of variables that may alter our results, for example, we might get wrong results if we measure the difference of two time stamps read from two different CPUs.

2 Machine Description

Part Name	Details
Machine	Lenovo Thinkpad T60
Processor	Intel Core Duo T2300 @ 1.66GHz - 12 stage pipeline
L1 Cache Size	64 KB instruction cache, 64 KB data cache
L2 Cache Size	2 MB
FSB Speed	667 MHz
RAM	2 x 1 GB DDR2
HDD	250 GB (232 GiB) @ 7200 RPM 512 byte sector size, 15151 KB cache/buffer size
Network Card	1 Gbit/s - 33MHz, 32 bit
Wireless Card	54 Mbit/s - 33MHz, 32 bit
Operating System	Ubuntu 12.10 - Linux version 3.5.0-19
Compiler	GCC version 4.7.2

3 CPU, Scheduling, and OS Services

3.1 Measurement Overhead

In the first benchmark, our goal is to measure the overhead of all proceeding measurements. It takes time to read the register for the processor cycles, and this latency will affect the rest of our readings, so we need to conduct this experiment to ensure we know what kind of overhead to expect when reading this register, then we will need to deduct this overhead from the rest of our experiments.

3.1.1 Methodology

In order to read the time, we use a function called rdtsc(), which is an inline assembly function invoking instruction rdtsc provided by x86, that allows us to read the register that contains the processor cycles. The basis behind this experiment is relatively trivial: we call the rdtsc() function twice, then deduct the readings of the former from the latter to arrive at the difference between the two readings. This will tell us how many cycles have passed between readings. Then with the cycle count, we can convert this to time given our processor's clock speed.

In order to measure the overhead of loops, we wrote a function that called rdtsc() prior to the loop, then rdtsc() immediately after. The loop itself was a simple empty for() loop that iterated from 0 to 1 (1 iteration). Because there were no compiler optimizations, we could ensure that this loop would not simply be ignored. This is then enclosed in another for() loop and repeated 100000 times for accuracy's sake.

3.1.2 Prediction

We predict that on a hardware level, there will not be a large CPU overhead, as this is an assembly level instruction. Because we have no prior knowledge of how rdtsc() works on a hardware level, our guess would be that the reading of the register in our x86 system would be in the neighborhood of 20-30 cycles. On the software side, we hope to see minimal overhead as we have done our best to

ensure exclusivity of the benchmarking process. Our overall estimate for reading the time is 40-50 cycles. However, we have read in the documentation to expect overheads closer to 100-130 cycles, so that probably a more realistic estimate.

For our for() loop estimate, we think that there is a minor overhead, at least 1-2 cycles for incrementing the counter, then another 5 cycles for conditional branch. There is then more overhead from instantiating the for() loop itself and initiating variables, so the overall estimated overhead is 20 cycles.

3.1.3 Results

Note for our results: we have subtracted the rdtsc() overhead, so a median of 10 actually means rdtsc reading of 10 cycles + the overhead of rdtsc().

Type	Hardware Est	Software Est	Total Est	Tested Mean	Tested Median
rdtsc()	20	20	40	102.4	100
for()	10	10	20	22.1	10

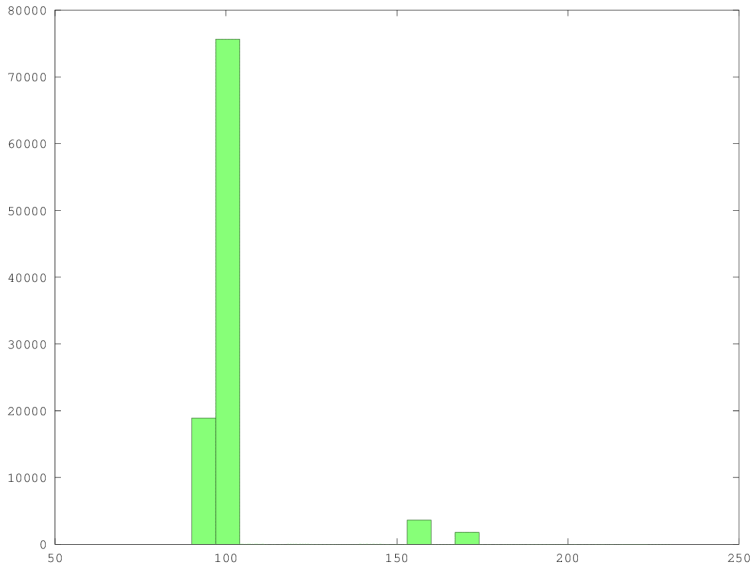

Figure 1: Histogram of rdtsc() overhead
Note: The median will now be deducted from all future results as overhead

3.1.4 Comments

We can see from the histograms that there is a "tail" of non-zero frequency around 160-180 cycles. As a sanity check, we found that they always happens in an initial period, then the majority of the time measurement become stabilizing to 90-100 cycles. We suspect that it is because of the warm-up effect of the cache; when the cache for a specific variable is in L1 cache, the extra overhead of storing to memory becomes negligible.

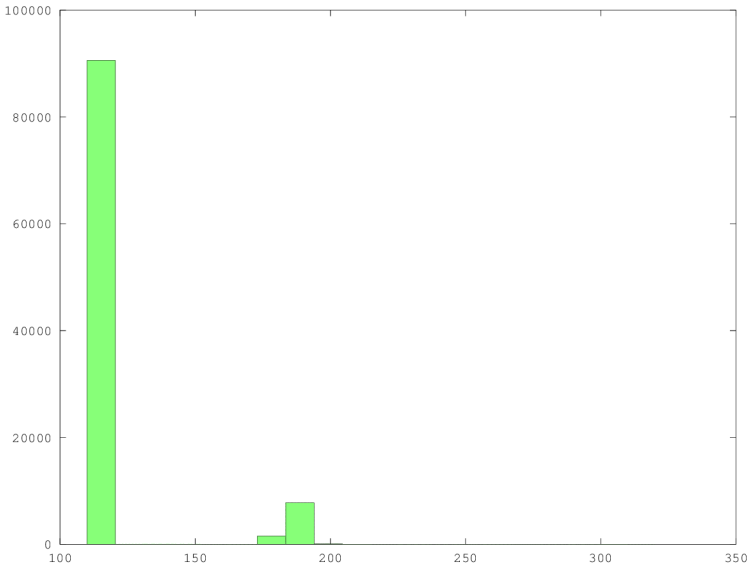

Figure 2: Histogram of for() overhead

Our predictions for how long rdtsc() were quite a bit off from the our actual results, which is not surprising as there were a disparity between our initial guess and that of the article we were referencing.. On the bright side, our actual results are right in line with the documentation, so our most likely accurate methodology is accurate.

Our estimates for the for() loop however, were a lot closer, especially the mean. On the backend, we also ran tests for the for() loop with different iterations to see the overhead of the conditional branch versus updating the hardware register for the iterator.

3.2 Procedure Call Overhead

Procedure calls are another common area of interest because of its frequency in today's programs. Because of the principles of programming, we use procedure calls to add a layer of abstraction to what the program is actually doing and mask the intricacies of its implementation with a procedure call. This simplifies coding as well as makes it easier to read.

3.2.1 Methodology

The implementation for checking procedure call overheads are just as simple as calculating the overhead for loops. Prior to the procedure call, we make a call to our rdtsc() function, then we call the procedure. The procedure itself then contains another rdtsc() call, which represents our "end" timer. Deducting the two we can arrive at our overhead for the procedure call. We can then enclose these calls within a for() loop to ensure that we have a consistent result.

We then repeat procedure calls with variable number of parameter arguments, passing the function between 0-7 arguments to see the effects of number of arguments versus overhead time.

3.2.2 Prediction

We know that for a procedure call, we would need to find the address of the instruction to jump to, then jump accordingly to execute that procedure. There would also need to be storing of return addresses, as well as saving and loading of various variables. Also, depending on the number of parameters the caller is going to pass, it will push the input parameters into the call stack, and push the return parameters into the stack when leaving. We think there is additional overhead to calculate the addresses in the neighborhood of 5 cycles. We think each additional argument will take another 1 cycle per argument, as the arguments are merely integers.

3.2.3 Results

# of args	Hardware Est	Software Est	Total Est	Tested Mean	Tested Median
0	0	5	5	8.03	10
1	0	6	6	7.12	10
2	0	7	7	7.00	10
3	0	8	8	5.01	0
4	0	9	9	1.06	0
5	0	10	10	3.97	0
6	0	11	11	3.08	0
7	0	12	12	4.98	0

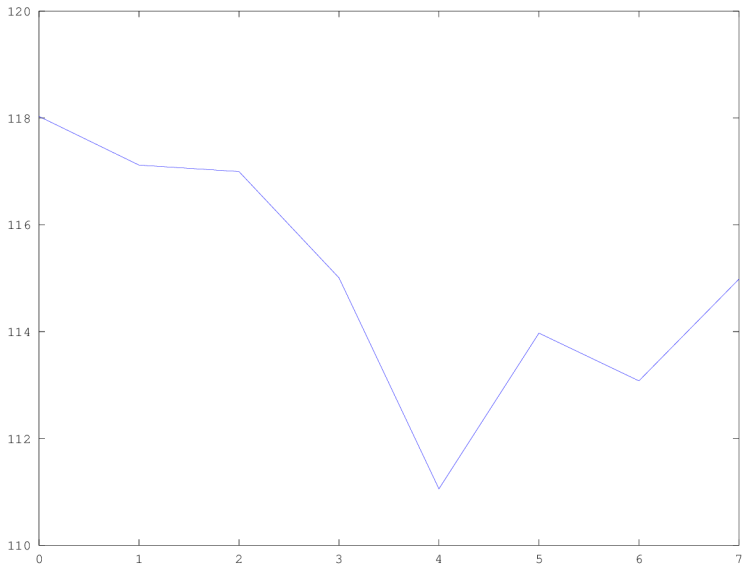

Figure 3: Graph of n arg procedure call overhead

3.2.4 Comments

Our prediction for a fairly small hardware overhead was relatively accurate. However, we did not get a consistent reading for reading in multiple arguments. What we hoped to find was an overhead for increasing the number of arguments, as arguments need to be pushed on the stack in reverse order so with each additional argument, we should at least see a 1 cycle increase.

Our methodology seems to be fairly reliable. If we look at the graph, we can see that towards the end, there is a general trend of increasing cycle count with arguments. We believe the discrepancy for small number of arguments versus the larger number of arguments exist due to buffer/caching effects.

3.3 System Call Overhead

System calls are another set of function calls that happen extremely frequently on the average system. There is a wide range of varying system calls; these calls will force an interrupt and temporarily switch from user mode to kernel mode in order to gain permission execute the system call in a safe and secure setting.

3.3.1 Methodology

This is another relatively simple implementation. Because there are a wide array of differing system calls that handle the operations necessary, we chose one that required the least overhead: getpid(). This basic system call basically returns the process ID when invoked. In order to measure the overhead, we accomplish this in similar fashion as the previous two experiments. We essentially wrap the getpid() function between two rdtsc() calls and find the difference. We then enclose the entire thing inside a for() loop then repeated 10000 times for consistency.

As a bonus, we repeat this methodology with another system call: time(&t). We do this in order to see if there is some sort of correlation between a simple system call and one that is slightly more complicated.

3.3.2 Prediction

Since our machine is of x86 architecture, the system call is realized by a call gate. With necessary segment selector, caller executes LCALL instruction. The CPU provides checks to make sure the entry is valid and the code is operating at sufficient privilege to use the gate. If the check passes, a new CS:EIP is loaded from segment descriptor, and information is pushed into the stack of the new privilege level. Parameters is copied from the old stack to the new stack if needed.

On the software side, we expect significantly more overhead because there needs to be a switch from user mode to kernel mode when invoking a system call, then it would need to switch back into user mode before we make our reading. However, we also think there may be an issue with our approach for getpid() calls as that is a constant value and could be cached by the CPU resulting in significantly shorter call times.

3.3.3 Results

Type	Hardware Est	Software Est	Total Est	Tested Mean	Tested Median
getpid()	10	500	510	13.6	10
time()	20	500	520	288.9	270

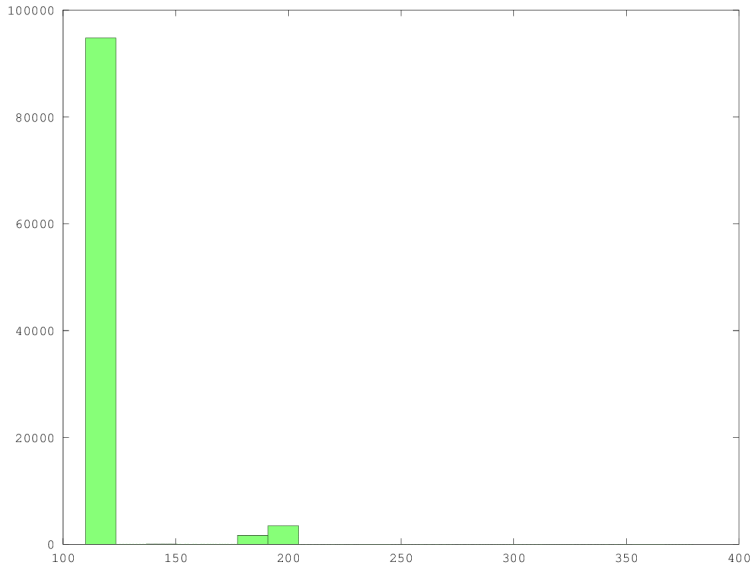

Figure 4: Histogram of getpid() overhead

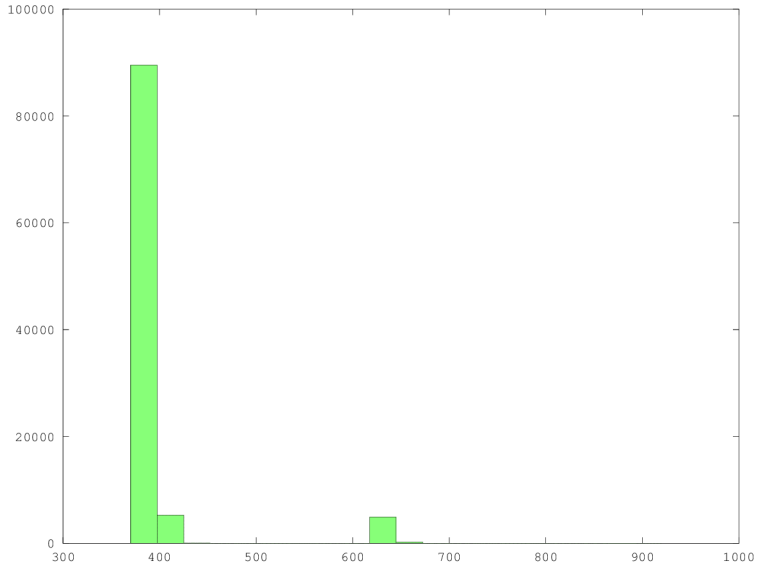

Figure 5: Histogram of time(&t) overhead

3.3.4 Comments

I believe our results show that we were on the right page with regards to a large software overhead. We can see that the system call time() takes significantly longer than our previous procedure calls.

Our methodology for this section could be reworked. Our conjecture is that when the process first executes getpid(), it brings the result into cache, so that when the getpid() executes at a second time, it will load the value from the cache instead of trapped into kernel, thus subsequent access time is significantly reduced. But for time() function, because it is dependent on the access time, it cannot be cached and should be implemented trapping into kernel.

3.4 Task Creation Time

Task creation in this experiment will be treated as the spawning of a new process. It is a way for us to benchmark how long it takes for kernel to create a new thread, properly adjusting its page tables and stack frame within memory. For this particular benchmark, we also want to compare the effects of spawning a user level process versus a kernel level thread.

3.4.1 Methodology

Our methodology for testing task creation time was to execute the system call fork(). From UNIX, we know that fork() spawns a child from its parent, simulating the creation of a user level processes. However, in order to ensure that we do not experience additional overheads or latency issues, we kill off the child immediately after taking a rdtsc() reading.

To simulate the creation of a kernel level thread, we try to build a kernel module .ko file. Then we use insmod command to insert it into kernel, and see its output of print with the use of dmesg. At the kernel module, we create a kernel thread using kthread_create(), the output the time difference using printk();

3.4.2 Prediction

For creating a new process, the fork() creates a new address space for child. If copy-on-write technique is implemented, then the kernel does not copy the memory contents of the child until the child modifies it explicitly. Essentially, because there needs to be a large number of operations, as well as memory allocation (both physical and virtual) as well as creating system page tables for the spawning of a new process, we expect a much larger overhead than any of the previously tested overheads. Creating a user-level thread should take significantly longer than the creation of a kernel level thread because many resources are shared in the kernel-level, whereas in the user-level everything needs to be created new.

3.4.3 Results

Type	Hardware Est	Software Est	Total Est	Tested Mean	Tested Median
User	100	10000	10100	108800	99680
Kernel	50	2000	2050	38409.4	38080

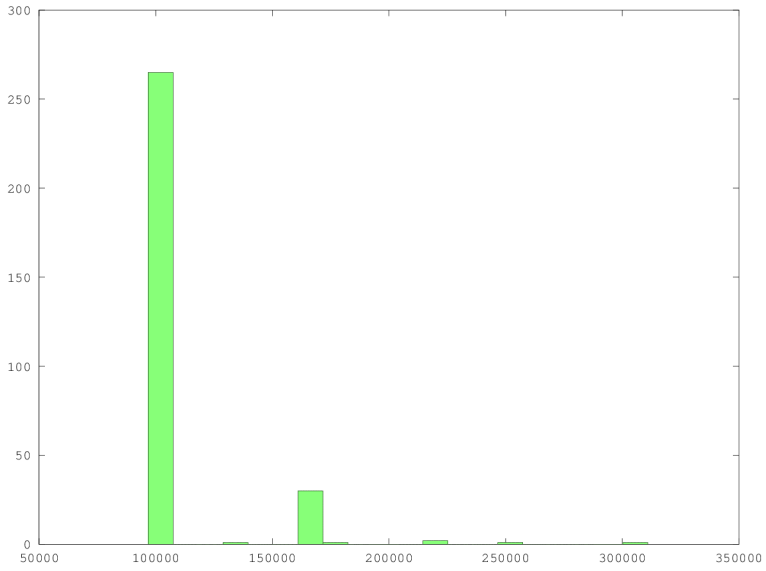

Figure 6: Histogram of user-level thread creation

3.4.4 Comments

Our predicted results while not numerically accurate were on the right track. Creating a new thread takes significantly more than making a function call or even a system call. We were also right in prediction that a kernel level thread takes shorter time to spawn than a user level thread. From our results we see that creating a process has more overhead compared to creating a kernel thread, which matches our predictions.

3.4.5 Sources

http://www.tldp.org/FAQ/Threads-FAQ/

3.5 Context Switch Time

Context switch is an important part of synchronization. When a process tries to yield the control to schduler, or a time slice allocated for a specific process is up, it triggers the scheduler to store PCB, the address space page table, register value, etc of the processing running and put another process which is waiting to execution.

3.5.1 Methodology

We try to use two processes to simulate a context switch. We use the methodology of Li et al.: two processes share an anonymous pipe, one process reading to it and the other writing to it. When the reader performs a blocking read, it will be scheduled out of CPU. Then the writer sends a byte to pipe, and wait for the reader to continue. Then the scheduler will switch reader in, and we take another time stamp so that we get time difference t_1, which is the overhead of context switch, plus the time cost for sending one byte though the pipe. Then we conduct a second set of experiment, which is the time cost of sending one byte through the pipe without causing context switch. It can be measured by the time of one process sending message to itself and receives it, getting a time t_2. $t_1 - t_2$ is the estimate of context switch overhead. To ensure the robustness of our result of t_1, we

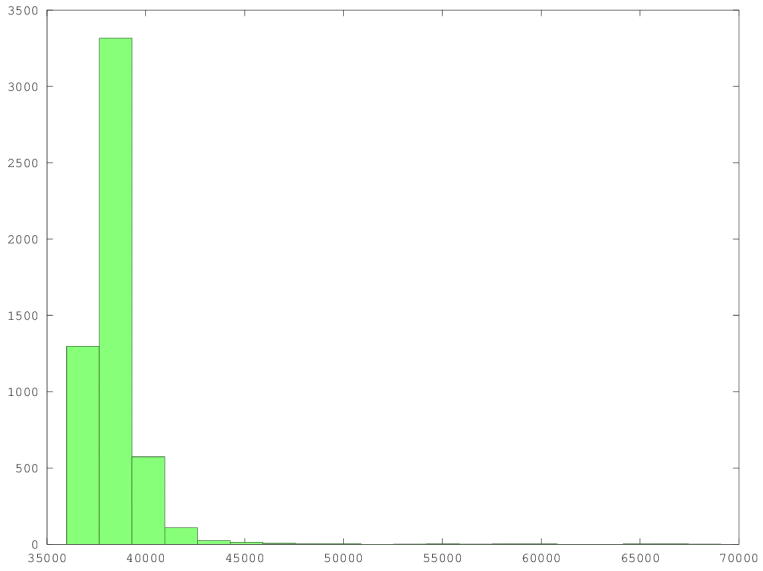

Figure 7: Histogram of kernel-level thread creation

repeat the process 10^7 times in a single run, and divide the total time duration by 10^7. And the nature of blocking read ensures there are $2 * (10^7) \pm 1$ context switches happening.
For switch between two processes, we first apply fork(); for switch between two kernel threads, we use pthread_create() in POSIX thread library, or alternatively, clone() (using options CLONE_VM — CLONE_FILES — CLONE_FS — CLONE_IO — CLONE_SIGHAND — SIGCHLD).

3.5.2 Prediction

When a context switch happens, the PCB for the process scheduled out should be created and saved, stored on a per-process stack in kernel memory. Then it loads the PCB and context of the processing being scheduled into CPU. For selection of the new process, the priority of processes in the waiting queue is considered.

We predict that the context switch itself can be conducted much faster than the creation of threads in the previous experiment. The reason is because we do not need to reallocate memory space which is time consuming as it is on a much higher level. We merely need to work with the CPU registers which are much faster.

It is expected that the switch time for context switch between processes will be much higher than that of between kernel threads, because the latter shares addresss space, file descriptor, etc, does not need TLB flush, essentially the only thing to do is the push the context for the previous thread and change SP, then pop the context for the next thread.

3.5.3 Results

The overhead of reading a bit from a pipe + writing a bit from a pipe = 2847.493

Using *pthread_create*()

$$2 * read + 2 * write + 2 * ctxswc = 14260.959, ctxswc = 4283$$

Using $clone()$:

$$2 * read + 2 * write + 2 * ctxswc = 13808.987, ctxswc = 4057$$

Using $fork()$:

$$2 * read + 2 * write + 2 * ctxswc = 14595.603, ctxswc = 4450$$

3.5.4 Comments

We expect our time overhead of context switch should be less than that of creating a new process. When creating a new process, the PCB needs to be copied from its parent, while for context switch, we only need to load the PCB from kernel memory.

3.5.5 Sources

http://linux.die.net/man/2/fork

4 Memory Operations

4.1 RAM Access Time

In this experiment, our goal is to see the difference in latencies between accessing different levels of memory, from the CPU L1 cache, to the L2 cache, all the way to memory. We know that memory plays a huge part in perceived speed, probably even more important than CPU on some levels for the end user experience.

4.1.1 Methodology

In order to simulate access to different parts of memory, we will be creating an array of different sizes then reading from the array randomly. We know that if an array is small enough, its entirety can fit into lower level cache. As the array size grows, it will not be able to fit in the lower level cache so that it will be stored in a higher level, then eventually into memory. We will be using built-in random number generator to try and simulate random accesses. It is important that we make random reads from the array due to hardware pre-fetching algorithms that may skew our results. We repeat this process 10000 times for each array size, from 2^0 all the way to 2^{27} (that is, from 8 bytes to 512 megabytes of space).

4.1.2 Prediction

We have different predictions for different levels of cache based on our hardware understanding of caches. We predict minimal software overhead once the for loops are "warmed up", that's to say once it begins execution. We ensure that there are no context switches as in our previous experiments. Because we repeat the process for 10000 times we pretty much eliminate any software overhead since all accesses should be on a hardware level.

Based on our resources for our Intel CPU, we can expect a hardware latency of 4 cycles for the L1 cache and 11 cycles for the L2 cache. Factoring in the time it takes for the load/store instruction to resolve, we can expect an additional 1-2 cycles of latency. Finally for software overhead, we can expect minimal impact, between 0-1 cycle once we factor in the 10000 trial average. Because L1 cache is on the CPU, and L2 cache is located slightly further, we can expect an additional 3-4 ns on top of L1 cache latency.

For memory accesses, we need to factor in distance and speed of the FSB. Ours ran at 167 MHz, meaning it would take at least 6 ns for it to resolve. 6 ns at our CPU speed of 1.66 GHz results in about 10 cycle of overhead on top of L1 and L2 for misses, factoring in distance we estimate 150 ns for distance latency. We expect to see the jumps in latency when the arrays reach roughly the size of the different memory hierarchies.

4.1.3 Results

Type	Hardware Est	Software Est	Total Est	Tested Mean	Tested Median
L1 cache	4	1	5	5.523	5.530
L2 cache	11	4	15	13.53	13.92
Main Memory	30	150	180	199.98	196.658

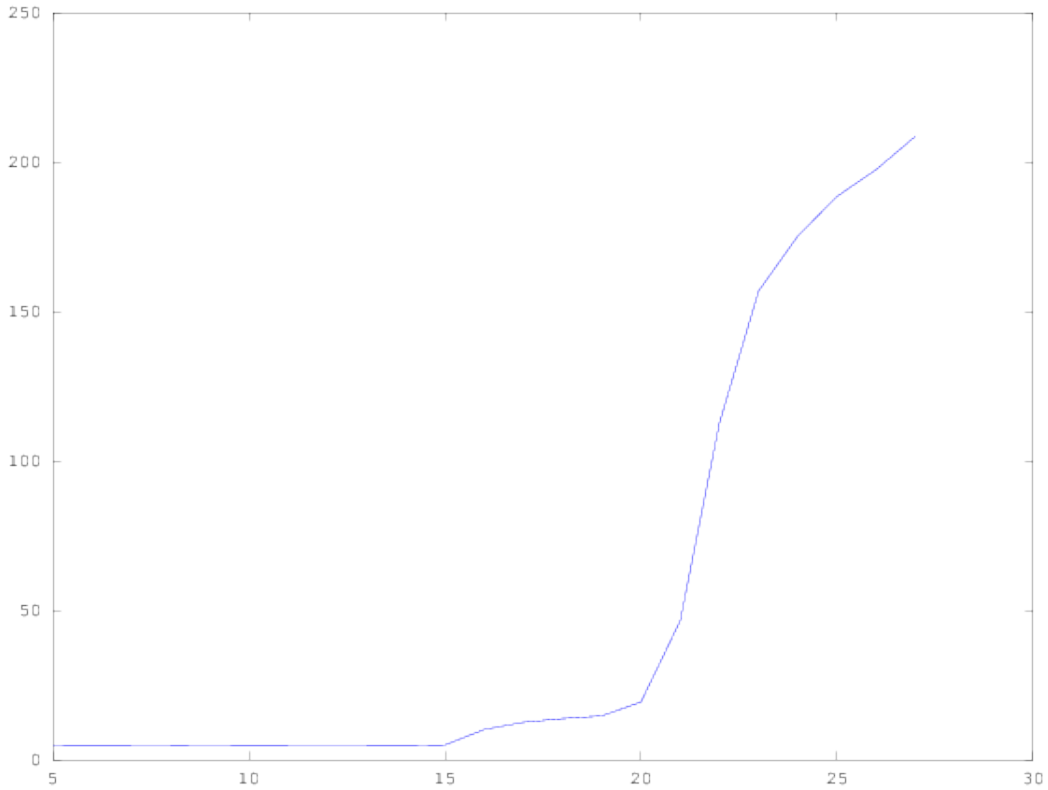

Figure 8: Histogram of memory latency vs range of data

4.1.4 Comments

From our results, we can see two blatant jumps in access time latency from L1 to L2, and from L2 to main memory. We see almost no fluctuation in the L1 cache section compared to L2 and main memory, which are both less clear in terms of having a constant read time. This is to be expected because some of the L2 accesses occurs at the L1 level due to pre-fetching algorithms, and some of the main memory accesses occur at the L1 or L2 level, whereas L1 accesses cannot occur any faster. Our predictions were relatively accurate, with minimal overhead for the L1 cache so it performed almost exactly as Intel specified. L2 cache access saw a hit due to distance (bandwidth was not a factor as 4 bytes of access does not become the bottleneck for the memory controllers with 32-bit width). Memory saw a higher latency hit still.

A first trial for us is: for each iteration generate a random number y within the range ad-hoc, then access the y^{th} element in the array. But experimentally, it may be defeated by prefetching, and does not show any sign of memory hierarchy. Instead we use another approach, use the array like a linked list: initially we build $0 \to 1 \to ... \to k \to 0$ as a circular linked list, then we randomly delete an element and add it back to another random position, repeat it for enough iterations. Then we access the linked list from head to tail. The preprocessing keeps the list circular, but the address of the next element is random enough that is able to defeat prefetcher, on the other hand does not need ad-hoc calling the random number generator.

4.1.5 Sources

http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-cache-latency-bandwidth-paper.pdf

4.2 RAM Bandwidth

RAM bandwidth is an important benchmark to test the throughput of the RAM itself. The CPU is advanced enough to do pre-fetching on multiple levels, so that when we do make reads, even though an array may be stored in RAM, the actual data parts may already be stored on a faster level cache. RAM bandwidth is important to know for a "worst case scenario" sort of situation.

4.2.1 Methodology

In this experiment, we allocated resources to create an array that would fit its entire contents in our available RAM. The size of the array needed to be significantly larger than the size of the largest CPU cache, while being smaller than our available RAM to ensure that nothing gets stored in virtual memory. We chose to create an array of size 512 MB, that is, an array with 2^{27} integers.

We conduct two groups of experiments: first we use memcpy() library function call to move the data from an array to another, to measure the total number of reads+writes within the period. Second we read/write a certain array sequentially, so that it enourages prefetching to succeed, thus increasing the throughput. We repeatedly run 50 trials and collected the statistics for the bandwidth.

4.2.2 Prediction

Unfortunately we are not able to get the exact speed for memory. For the hardware overhead, the FSB speed is 166MHz * 4 = 667MHz (Intel's Quad Pumping). And at each sending, the memory line size of 4B, thus the theoretical maximum bandwidth is 2.67GB/s. And for CPU it is able to handle it since the main frequency is 1.66GHz so that its bandwidth is 6.67GB/s (considering pipelining), assume it can access 32bit data per cycle. But in practice we expect that we are not able to achieve it, because of we do not have a perfect mechanism to control the memory transfer time and monitor its speed in each cycle. (See comments for further discussion). We expect little software overhead, since the operating system rarely have to deal with memory read within address space, except for handling page fault.

Since the test process is not the only process running, it may introduce a little extra overhead for running other process.

4.2.3 Results

Type	Hardware Est	Software Est	Total Est	Tested Mean	Tested Median
Main Memory	2666.7(MB/s)	100(MB/s)	2566.7(MB/s)	2043.4(MB/s)	2045.5(MB/s)

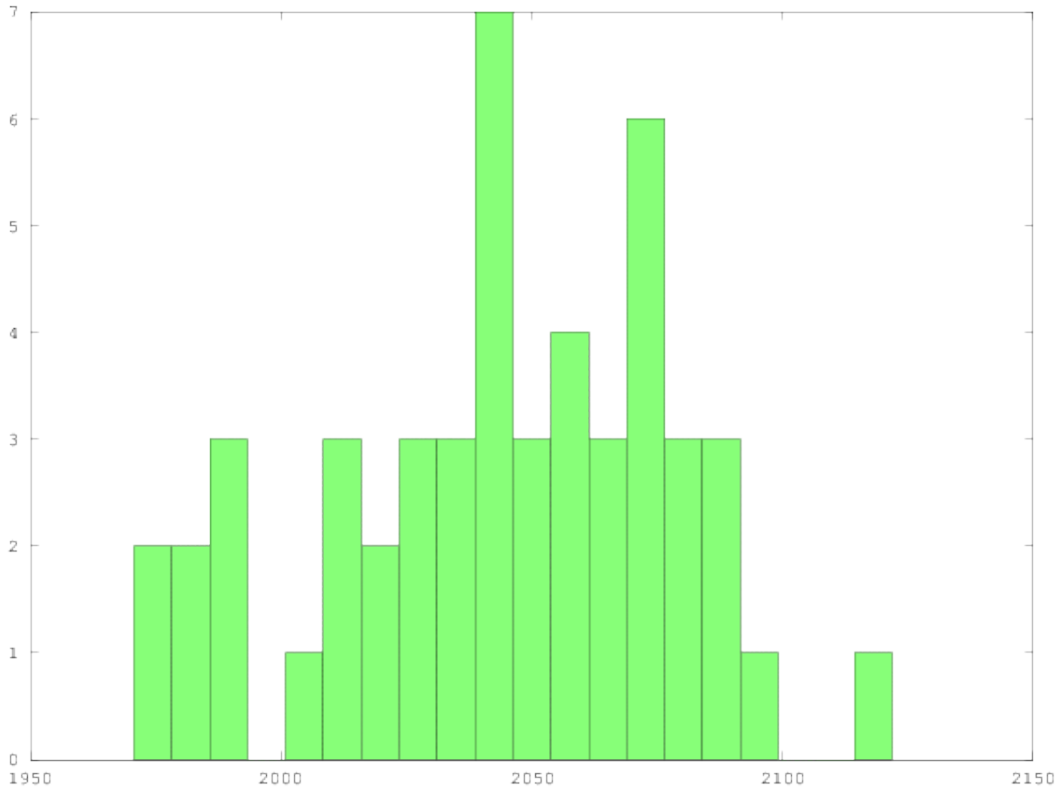

Figure 9: Histogram of memory bandwidth(MB/s)

4.2.4 Comments

The array should not be chosen too large (8MB is better??) that sometimes page fault happens even if accessing sequentially, resulting in super large latency(outlier?) for a single access, which is supposed to be handled in the next section. We are not sure if our approach saturates the memory bandwidth, since when reading a single page, the prefetcher may already have done reading the next page. When we read the second page, the prefetcher may do nothing, because it is already in L1 cache, which highly depends on the algorithm.

But at least our estimate is always a lower bound of memory bandwidth, since in the certain period of time we have observed such amount of memory coming into CPU.

4.2.5 Sources

http://www.crucial.com/support/memory_speeds.aspx
http://ark.intel.com/products/27233/intel-core-duo-processor-t2300-2m-cache-1_66-ghz-667-mhz-fsb
http://www.cyberciti.biz/faq/check-ram-speed-linux/

4.3 Page Fault Service Time

Perhaps the best test to show people the reason why their computer "feels" slow is the page fault experiment. When we experience a page fault, that's when we need to have the virtual machine come into play and help load the proper page into memory.

4.3.1 Methodology

In order to simulate page faults, we essentially create two arrays that may each fit into memory, however cannot both fit into memory. That way, when we access the first array, the first array would be loaded into memory. But now when we need to access the second array, then there would be a page fault and we would need to load the second array from virtual memory and excavate the first array. We then repeat this process to make reads into the second array, then time how long it takes to make a read into array 1 once the memory is filled with information from array 2. We repeat this process 500 times and hope to see a tight clustering.

4.3.2 Prediction

From the hardware page located in our sources, we can see that the average read seek time is 8.9 ms. We know that our clock speed is 1.66 GHz, therefore 8.9 ms is roughly equivalent to $8.9 * 1.66 * 10^6$, or about 14.8 million cycles. We can also calculate average seek time based on rotational speed of the hard drive, which is 7200 RPM. That means 7200 revolutions per minute, or 120 per second, or .12 per millisecond, or about 8 ms per full rotation. So our prediction for hardware is around 14-15 million cycles of page fault time, in which case clearly hardware overhead greatly dominates software overhead. We also know that because the hard drive is a physical rotating disk, and the arm is another mechanical piece, then we will see a wide range of values depending on where the disk head is when we perform the read.

4.3.3 Results

Type	Hardware Est	Software Est	Total Est	Tested Mean	Tested Median
Page Fault	$1.48 * 10^7$	Negligible	$1.48 * 10^7$	$1.50 * 10^7$	$1.52 * 10^7$

4.3.4 Comments

During the implementation step, we actually encountered multiple difficulties with manually invoking a page fault when we want to. We thought it would be very easy to defeat the pre-fetching system, but it turns out the pre-fetcher in our machine is quite good, and is able to cache the data even though we were making almost "random" accesses to the array.

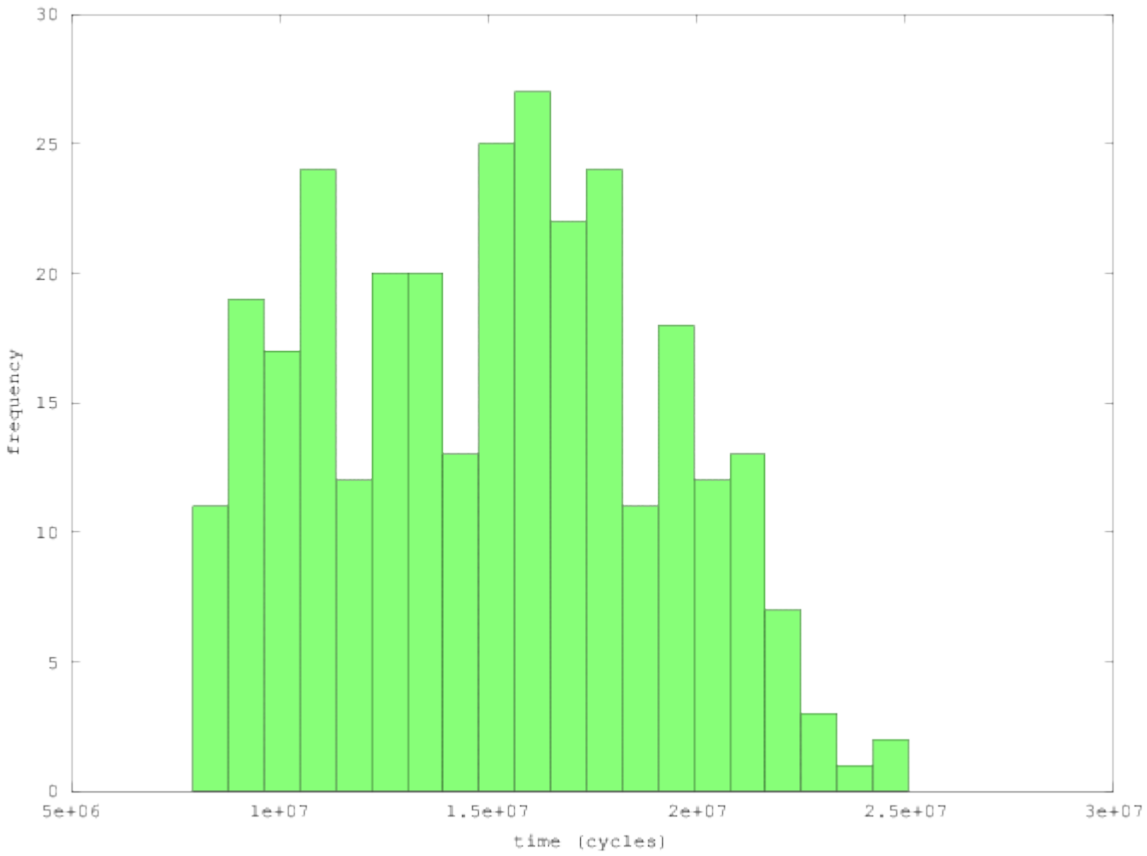

Figure 10: Histogram of page fault cycles

We overcame this by creating two arrays of 1 GB in size. We had to initialize both arrays to overcome clever compiler optimizations, then we made 256 "random" writes to different parts of array 1 in an attempt to flush out any remnants of array 1 from memory. Then we make a single read to array 2. We for the most part that we can expect some large cycle delay, or else we've failed to invoke a page fault, so we looked for values greater than 50000 cycles and filtered out the non-page fault values for a better looking histogram.

We can see that overall, our results were extremely on-par with our prediction, and this is a very good example of good hardware advertisement. However, this also shows why a page fault greatly dominates any other computation since we're working with millions of cycles instead of tens or even hundreds of cycles. When we encounter a page fault to the HDD, we see a very noticeable delay in response as we wait for the disk seek to occur.

4.3.5 Sources

http://support.lenovo.com/en_US/detail.page?LegacyDocID=MIGR-62579
http://www.cyberciti.biz/faq/linux-command-to-see-major-minor-pagefaults/

5 Network

5.1 Round Trip Time

This was an experiment to measure the time it took to make a round trip over our ethernet network using TCP protocol and comparing that to the ICMP ping service. We take the definition in Wikipedia, "round-trip time (RTT) is the length of time it takes for a signal to be sent plus the length of time it takes for an acknowledgment of that signal to be received. This time delay therefore consists of the propagation times between the two points of a signal." In this sense it should not include the time for setting up and tearing down the connection.

5.1.1 Methodology

First we setup an echo service (port 7) provided by system on server side, modifying inetd.conf such that it contains "echo stream tcp nowait root internal" configuration. Then we set up a client which connects to the server, either remotely or loopback. We send a 32 Byte TCP message from the client side to the server side, ensuring it does not incur any fragmentation; then the echo server echo back the same message. In this way the packet has a round trip going from client user process, back to client user process.

Although we know there is a ACK signal from server to client, we cannot determine the exact time of receiving it, instead we measure the time when its blocking read of the echoed packet coming back get unblocked, taking it as the round trip end-time.

For the ICMP round trip time, we use the command $ping 127.0.0.1 - c100 - s32 - U$, which sent 100 packets.

5.1.2 Prediction

The rough process of the round trip is:

(Loopback) user process of client $\rightarrow$ kernel $\rightarrow$ user process of server $\rightarrow$ kernel $\rightarrow$ user process of client.

(Remote) user process of client $\rightarrow$ kernel of client $\rightarrow$ network $\rightarrow$ kernel of server $\rightarrow$ user process of server $\rightarrow$ kernel of server $\rightarrow$ network $\rightarrow$ kernel of client $\rightarrow$ user process of client.

We predict that the average round trip time for TCP protocol should be significantly higher than ICMP. In TCP, we are sending additional data (24 bytes) per packet, as well as overhead in the handshaking.

When comparing local with remote timings, we can expect our local results to be significantly faster than our remote times. Remote packets are limited by the bandwidth of our ethernet adapter, but with local data we are not limited by this. We know to create a pipe within the kernel to copy data of only a few bytes will only take a couple ns (1-2). We should still incur a software overhead due to creating the packet, which we can estimate to be 1ms.

5.1.3 Results

Type	Hardware Est	Software Est	Total Est	Tested Mean	Tested Median(ms)
TCP echo local	0	1	1	0.9041	0.4161
ICMP ping local	0	0	0	0.060	-
TCP echo remote	1	1	2	3.0934	2.8012
ICMP ping remote	1	0	1	1.713	-

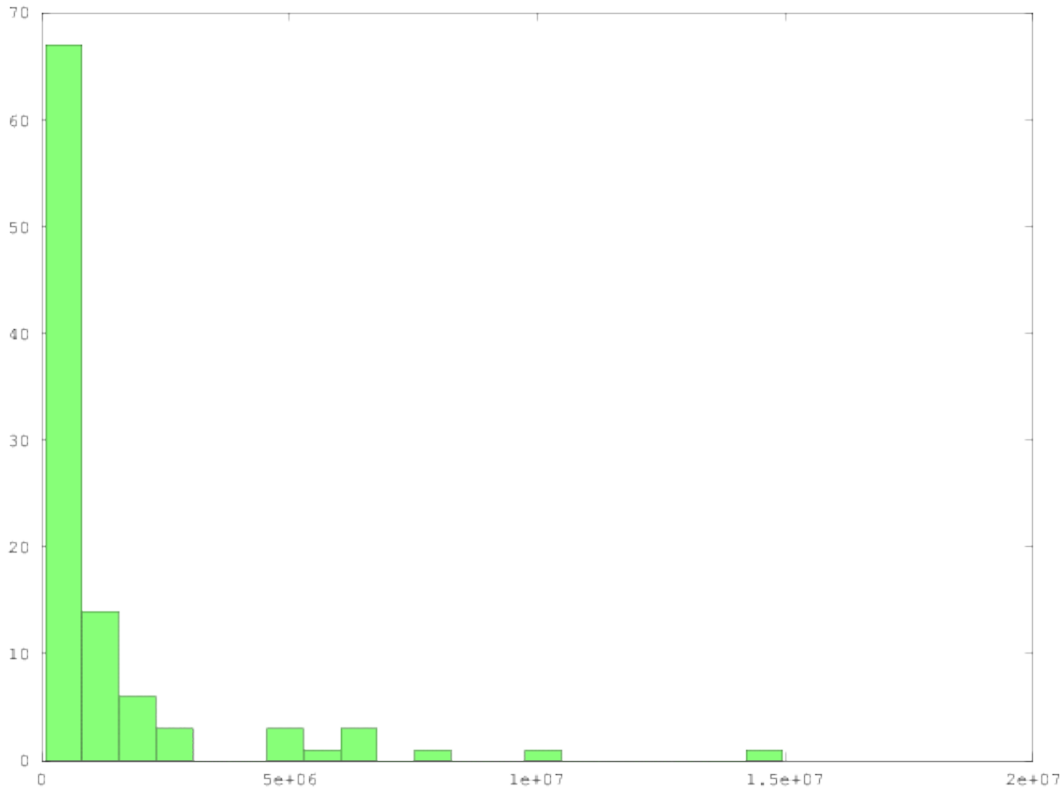

Figure 11: Histogram of local round trip time vs range of data

5.1.4 Comments

We found our methods relatively accurate when it came to measuring round trip time. We included all of the data payload bytes as well as the overheads to ultimately generate a more accurate result regarding our comparison of TCP vs ICMP.

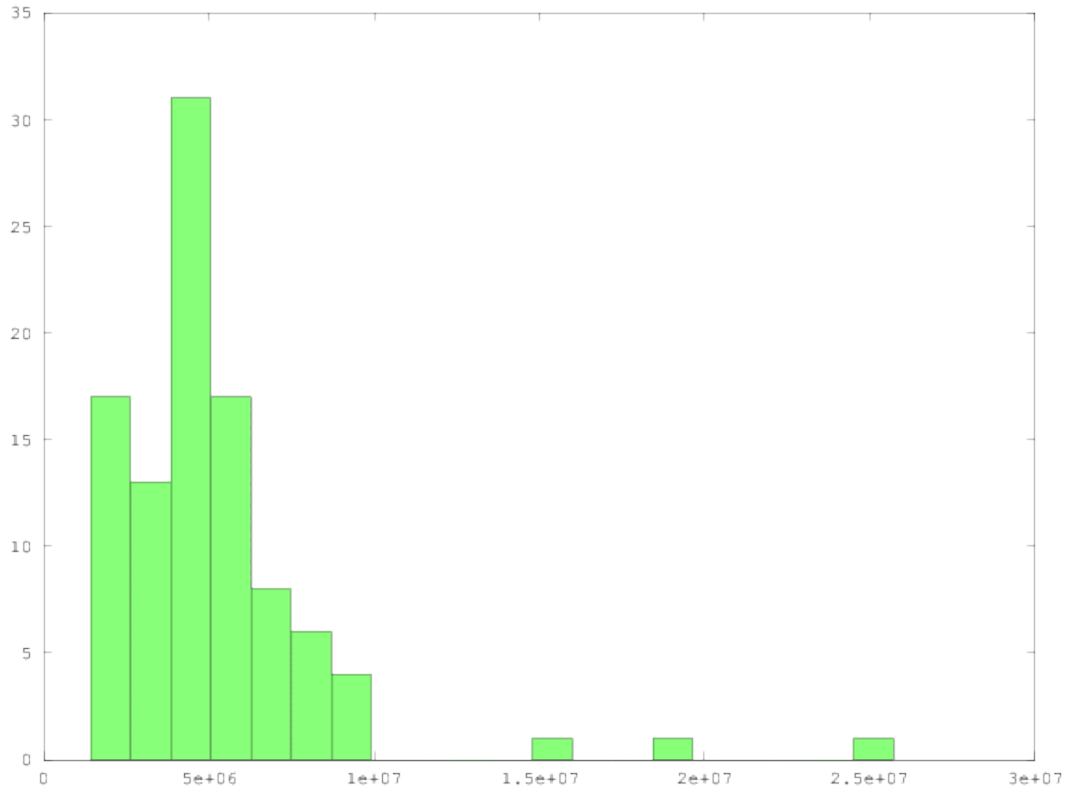

Figure 12: Histogram of remote round trip time vs range of data

5.1.5 Sources

http://en.wikipedia.org/wiki/Echo_Protocol
http://en.wikipedia.org/wiki/Ethernet_frame
http://en.wikipedia.org/wiki/Transmission_Control_Protocol
http://www.networksorcery.com/enp/protocol/icmp.htm

5.2 Peak Bandwidth

This is an extremely pertinent real world test that allows us to determine maximum system bandwidth via sending data over our network link.

5.2.1 Methodology

We create two processes: one for client and the other for server. The server works at TCP, port 30000, in order not to collide with other servers. The client simply sends 32MB data in total sequentially to the server. To maximize the bandwidth, we require the server only to receive all the data and do not do any processing over them.

5.2.2 Prediction

Since TCP transmission is asynchronous, it does not require the client to wait for the ACK of the server, after it sends the data packet. From standard textbook, we know that "The positive acknowledgment with retransmission is used to guarantee reliability of packet transfers. The sender

also maintains a timer from when the packet was sent, and retransmits a packet if the timer expires before the message has been acknowledged. The timer is needed in case a packet gets lost or corrupted."

For remote transmission, the MTU of WLAN(WPA (TKIP) frame) is 2312 bytes, and the absolute limitation of TCP packet size is 65536 bytes, so the bottleneck is in WLAN. There is other headers on the WLAN frame, including header, IV, EIV, data, MIC, ICV, FCS, consisting of 54 bytes. The size of IPv4 header is 20 bytes. The size of TCP header is 20 bytes, plus 12 bytes optional TCP timestamps. So the fraction of data in on package is (2312 - 52) / (2312 + 54) = 95.5%.
On a loopback, we can expect very high throughput since our interface is no longer limited by the network card but rather it is a file on the local file system. To send 32MB of data, we would also need to receive 32MB of data in the loopback, so 64MB of total data would be transferred; this would fit inside our main memory comfortably. Given our previous experiment on RAM bandwidth, we found that this machine could achieve 2045 MB/s of throughput, so naturally we will assume that in this simulation of essentially moving data inside main memory, that the TCP local bandwidth would be the same.

5.2.3 Results

Type	Est (Mb/s)	Tested Mean	Tested Median(Mb/s)
TCP local	2045	2066.8	-
TCP remote	51.3	8.1855	-

5.2.4 Comments

Our prediction for TCP local matches up almost perfectly with our tested resulted. Intuitively this makes sense, since in a loopback with all data already on main memory, all reads and writes would be to and from memory, so there is no reason we wouldn't expect to see the same bandwidth as with our RAM bandwidth experiment.

5.2.5 Sources

http://en.wikipedia.org/wiki/Transmission_Control_Protocol
http://stackoverflow.com/questions/17481397/what-is-the-difference-between-synchronous-and-asynchronous-transmission-in-tcp
http://www.wildpackets.com/resources/compendium/wireless_lan/wlan_packets
http://nmap.org/book/tcpip-ref.html

5.3 Connection Overhead

In this experiment we measure the amount of time it takes to set up and tear down a network TCP connection to a remote and local host.

5.3.1 Methodology

We measure the time of executing in user process:

$$connect(cfd, (structsockaddr*)(\&s\_add), sizeof(structsockaddr));$$

as the setup time. On the other hand, the tear down time can be measured as the time measured in user process:

$$close(cfd);$$

5.3.2 Prediction

For the setup, on the server side, it is calling accept() function waiting for accepting a connect. On the client side, it is calling connect(), trying to connect to the server. There is a handshaking process going on: first the client sending a SYN, then the server SYN/ACK, then the client ACK.

For the teardown, on the server side, it is repeatedly reading, until the client has called close(), sending a FIN/ACK. The server's read() function returns 0, replying with a ACK. At this time the server also calls the close() function to the file descriptor indicating this connection. Thus the server sends a FIN/ACK again, and the client replies with a ACK.

As a result, we predict the hardware overhead for setting up a remote connection to be about 0.1ms and the software overhead to be around 1ms, with additional overhead due to presumably less than optimal management and sending of packets, another 1ms. For the remote teardown, we expect to see minimal hardware overhead for communication, and about 0.1ms of software overhead to invalidate the data.
For our local loopback prediction, we predict similar overheads as with establishing and destroying the remote server, however a factor less. Teardown for a local connection should once again be insignificant for the hardware, with just a few cycles of software overhead since we do not need to communicate through the network, estimate 0.05ms or half the time of remote server. To set up a local connection we estimate again insignificant hardware overhead, and software overhead that should be longer than the teardown for local, but shorter than it takes to communicate with network, so around 0.07ms.

5.3.3 Results

Type	Hardware Est	Software Est	Total Est	Tested Mean	Tested Median(ms)
Local Setup	0	0.07	0.07	0.0424	0.0398
Local Teardown	0	0.05	0.05	0.0114	0.0104
Remote Setup	0.1	2	2.1	2.992	2.992
Remote Teardown	0	0.1	0.1	0.0299	0.0297

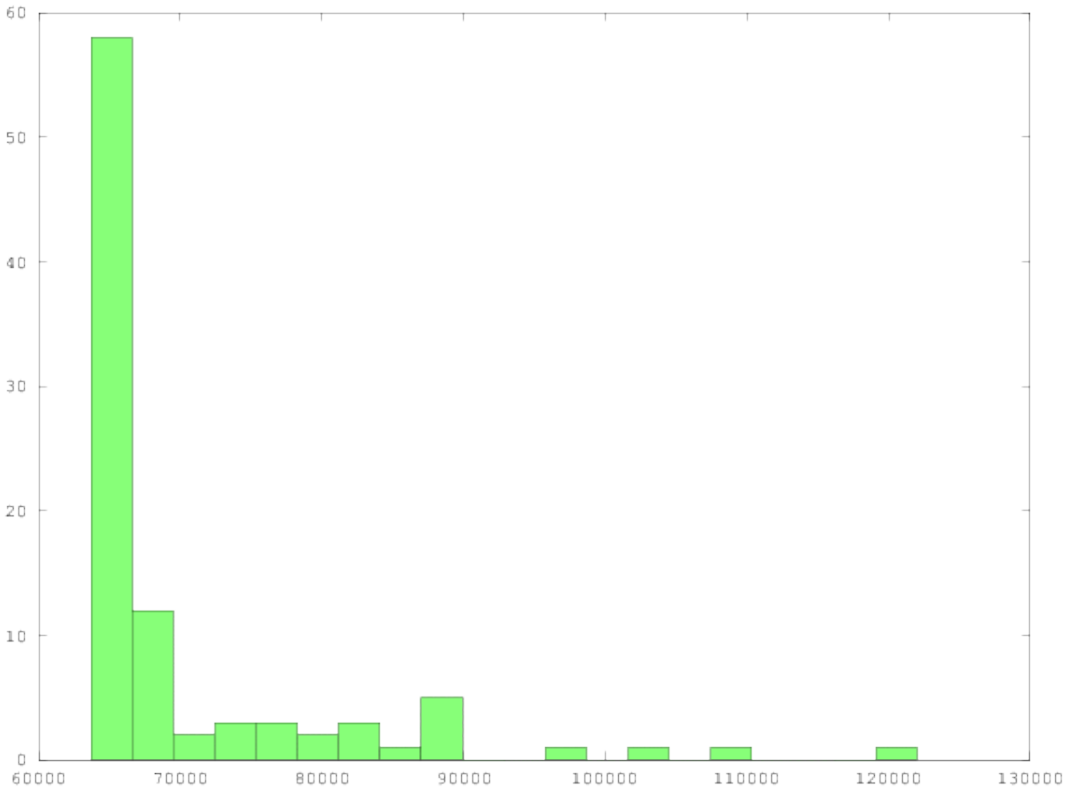

Figure 13: Histogram of local setup vs range of data

5.3.4 Comments

Our predictions for this experiment were not the most accurate, however we were on the right track with regards to what would take the longest versus what would take the shortest. Teardown of the connection clearly takes much shorter time due to the fact that the communication channel has already been opened and we do not need to re-establish it for the sake of closing it.

One explanation why the actual performance generally beat our prediction in the local experiment is due to the fact that the data structures are created in main memory, which may hit lower levels of cache and would receive some software optimization when storing and accessing these structures.

5.3.5 Sources

http://www.cs.rutgers.edu/ pxk/rutgers/notes/sockets/
http://pic.dhe.ibm.com/infocenter/iseries/v7r1m0/index.jsp?topic=%2Frzab6%2Fhowdosockets.htm
http://pic.dhe.ibm.com/infocenter/tpfhelp/current/index.jsp?topic=%2Fcom.ibm.ztpf-ztpfdf.doc_put.cur%2Fgtps

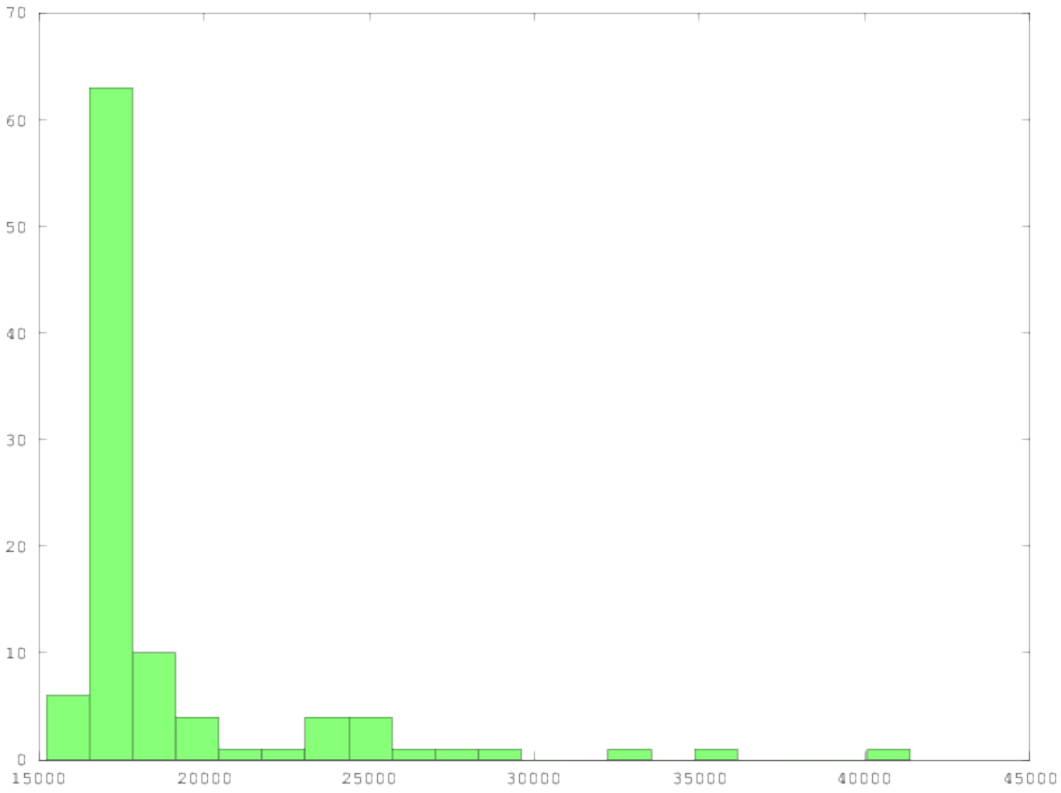

Figure 14: Histogram of local teardown vs range of data

6 File System

6.1 Size of File Cache

Our goal for this section is to find the size of the file cache, which is cache set aside for files for faster access.

6.1.1 Methodology

Our methodology for determining file cache size is relatively straight forward. We will create files of differing sizes, then try reading the files back and determining how long it takes. Then we will normalize the time based on the size of the file to get an overall idea throughout. When we plot the throughput of our reads on different file sizes, we should see a significant jump in access time/throughput when we miss the file cache, so that will determine what the size of our file cache is.

6.1.2 Prediction

The file cache should be close to the size of whatever free space there is in main memory. Given our main memory size of 2 GB and at the time of testing we have 0.97 GB of free physical memory, we can predict that the file cache size will take up the bulk of that remaining free space, so 0.97 GB.

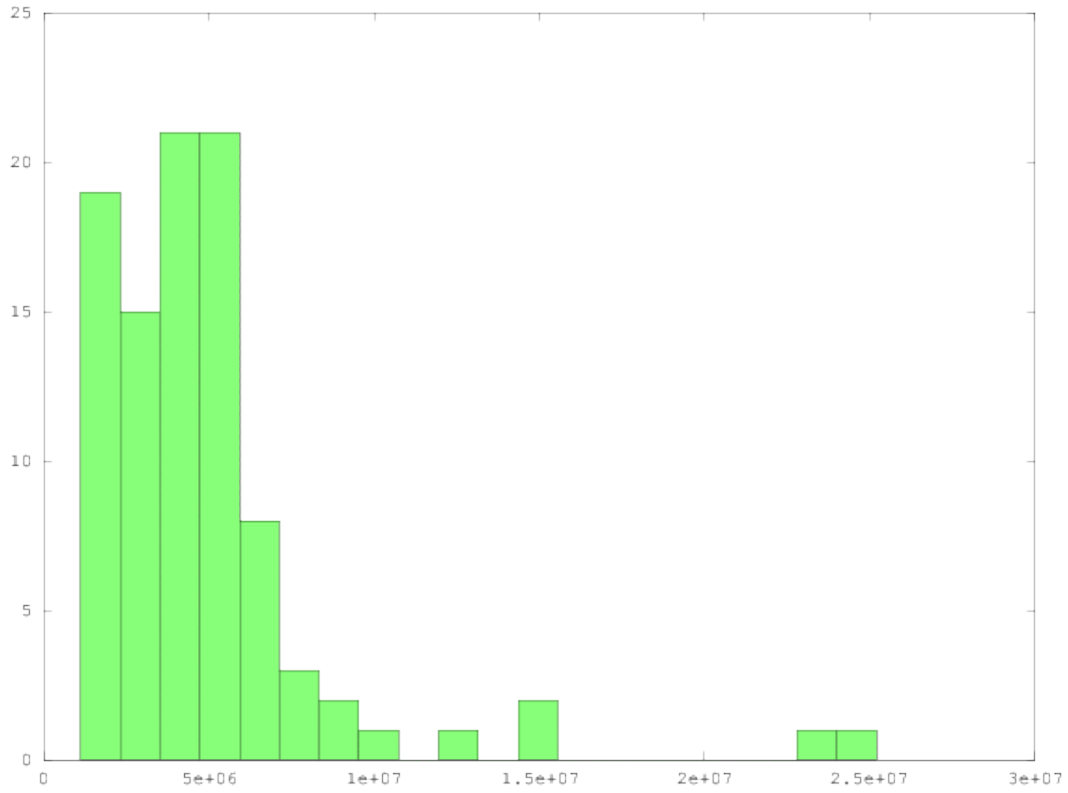

Figure 15: Histogram of remote setup vs range of data

6.1.3 Results

6.1.4 Comments

The graph is not the most exciting graph, but it does confirm our prediction to be rather accurate. You can see distinctly longer and longer access times per MB as the file size increases above 2^30 bytes or 1 GB, which was on par with our prediction.

6.1.5 Sources

http://www.westnet.com/ gsmith/content/linux-pdflush.htm

6.2 File Read Time

File read time is an extremely important statistic for real world application performance. We want to test the read times for sequential reads as with the case for media accesses, as well as small random read,s which would correspond with random disk accesses.

6.2.1 Methodology

We first want to disable file caching in Linux, because we strictly want to test the disk performance rather than memory performance, which we have already done. For this experiment, we create a file of size varying sizes from 4KB to 1 GB with block size of 4 KB using *dd* command. We then set $posix\_fadvise$ of the file to $POSIX\_FAD\_DONTNEED$ to denote that we will not be using

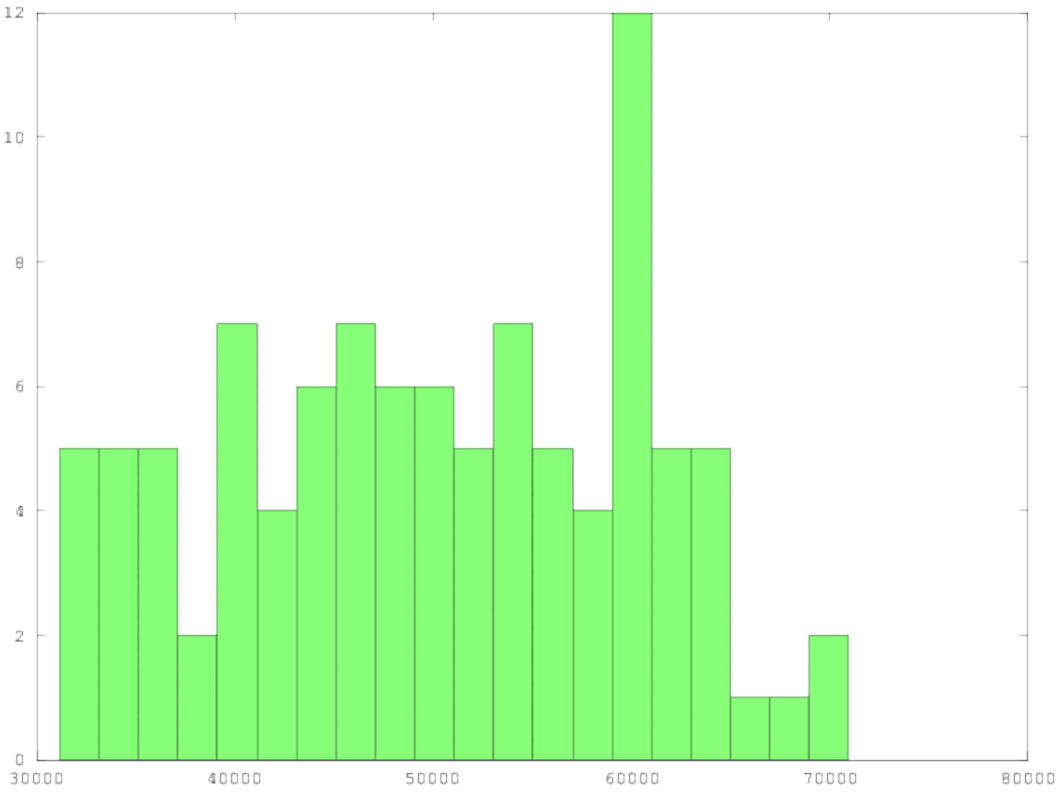

Figure 16: Histogram of remote teardown vs range of data

this file in the near future. This would ensure that the file gets placed on the disk rather than in the file cache/main memory. We then use *pread* on either random blocks of data to test random read speed, or we do it sequentially by reading larger

6.2.2 Prediction

From our hardware specifications, we know that this particular drive has sectors with 512 bytes and a buffer of 15151 bytes. That means theoretically with optimal prefetching it should be able to look ahead 15 KB.
Moreover, from earlier testing, we know that the average hardware seek time for the disk is around 12ms

6.2.3 Results

Result can be found in Figure 18 and 19.

6.2.4 Comments

There is always the chance (in fact a pretty good chance) that our test for "sequential" access is not truly 'sequential". As we have discussed in class, in most file systems where we do a lot of writes and deletes to the disk, we encounter fragmentation within the disk. Because of this, when we go and try to write some large file to the disk, the file system may not be able to find a large enough section in the disk to write the entire file to sequentially, so it may have to divide the file

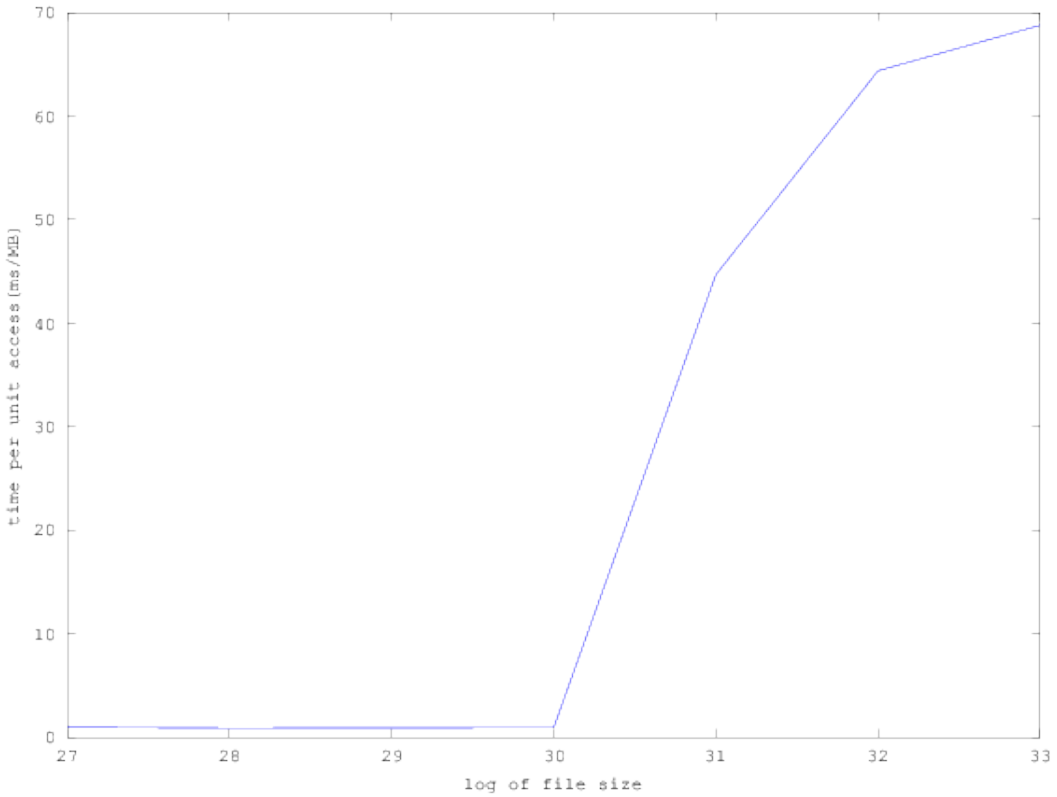

Figure 17: Access time vs File size

up into parts and scatter it throughout the disk. While logically our tests treat this as a sequential access, physically there may need to be seeks to different parts of the disk to retrieve parts of the file.

Our predictions were relatively accurate given our understanding that for small file sizes, the running time will primarily be dominated by seek times due to the nature of random reads. However, as the file size grows, we expect that because we are reading files sequentially, we will see the throughput approach the hardware specified value. However, as the file size grows even bigger yet, then we expect that due the nature of file systems not truly allowing "sequential reads" as argued aforementioned, we expect an additional form of latency in addition disk seeks.

6.2.5 Sources

http://www.gnu.org/software/libc/manual/html_node/Low_002dLevel-I_002fO.html#Low_002dLevel-I_002fO
http://en.wikipedia.org/wiki/Disk_buffer

6.3 Remote File Read Time

In this experiment we measured the time it took to read blocks of data from a network file system (SMB) both randomly and sequentially.

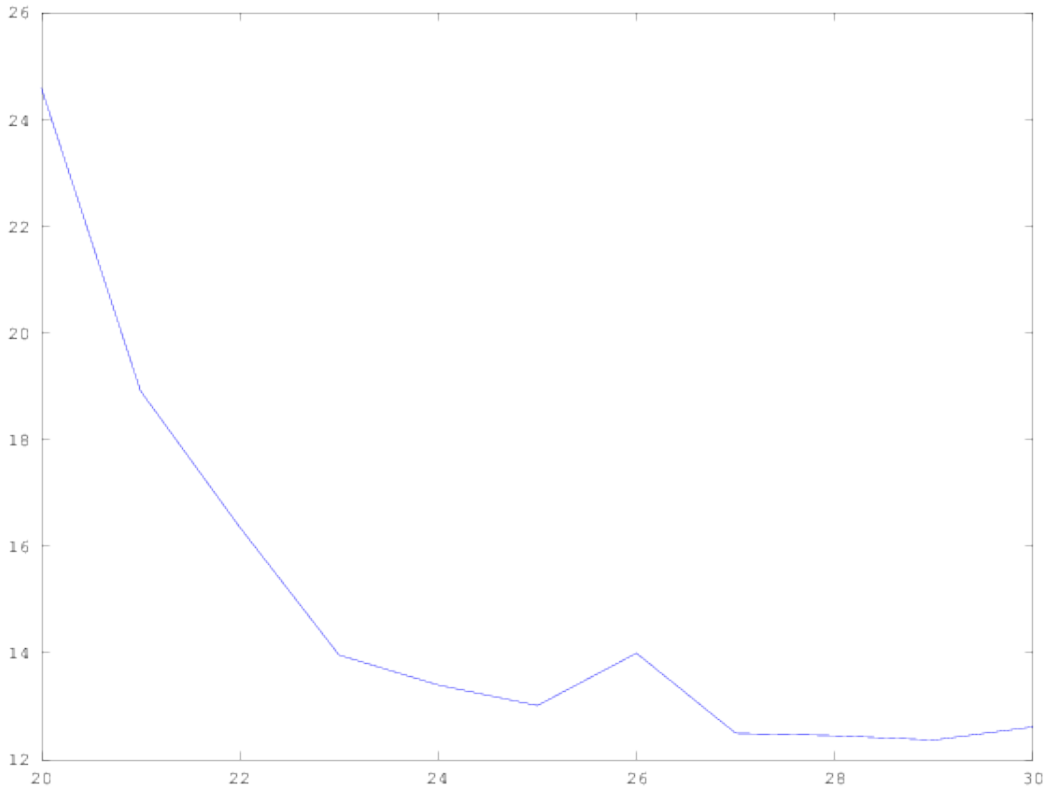

Figure 18: Sequential local file read time(s/MB)(y) to logarithm of file size(x)

6.3.1 Methodology

We first set up a network file system, and mount it on the same machine on /mnt. Then we use file read() system call to read the files just as it is on the local machine.

6.3.2 Prediction

When the client reading a file, it uses the system call read(). The read gets passed to the kernel VFS layer. VFS hands it off to the NFS module, which converts the read to an RPC. The RPC is sent to the NFS server. The server reads the file off its storage disk, packages, and sends it back across the network back to the client. The NFS module at the client receives the server's reply, and passes it to VFS, then back up to the program running in user space. So for NFS file read, there are two kernel traps: 1. for client process calling read(), 2. for NFS server calling read(). And there is a RPC request/reply over the network. Thus we expect the time overhead to be much higher than local file read.

6.3.3 Results

Result can be found in Figure 20 and 21.

6.3.4 Comments

We can see the effects of "network penalty" in this experiment when we compare it to conducting the same test on simply the local machine alone. We predict the network penalty to be somewhere

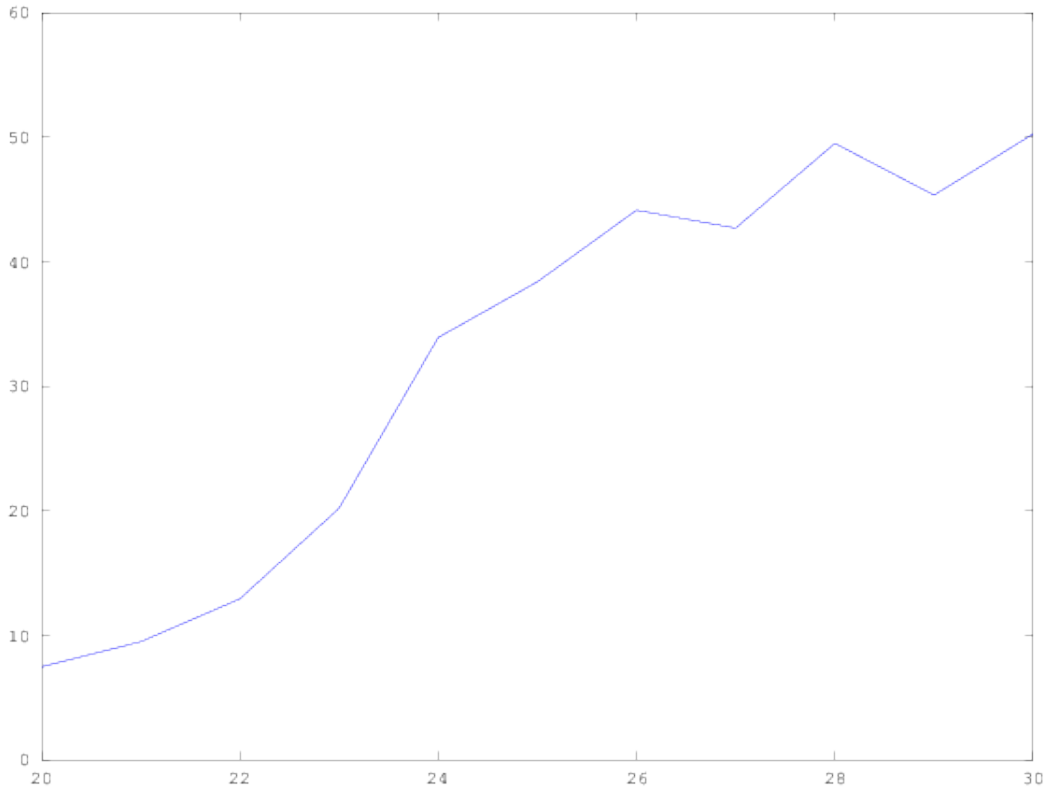

Figure 19: Random local file read time(s/MB)(y) to logarithm of file size(x)

in the neighborhood of our results for the round trip time experiment of 3ms.
One issue with this is the fact that the specs on the remote machine are not the same as on the local machine. Intuitively, we can reason out why there may definitely cases where we see better remote reads than we would local reads if say the remote machine used a solid-state drive instead of a traditional hard drive and we were not bottlenecked by our network interface.

6.3.5 Sources

http://www.read.cs.ucla.edu/111/2007spring/notes/lec17
http://www.tldp.org/LDP/nag/node140.html

6.4 File Contention

In this experiment, we want to know what happens when we have multiple processes trying to access data, e.g. contending for data.

6.4.1 Methodology

In order to conduct this experiment, we essentially had the parent thread fork a bunch of children threads, each of which would read their own 1MB block files. We ensured that the files were located on the disks themselves and not in the file buffer cache from our previous knowledge of file caches. We then calculated the amount of time it took to complete with varying n number of processes.

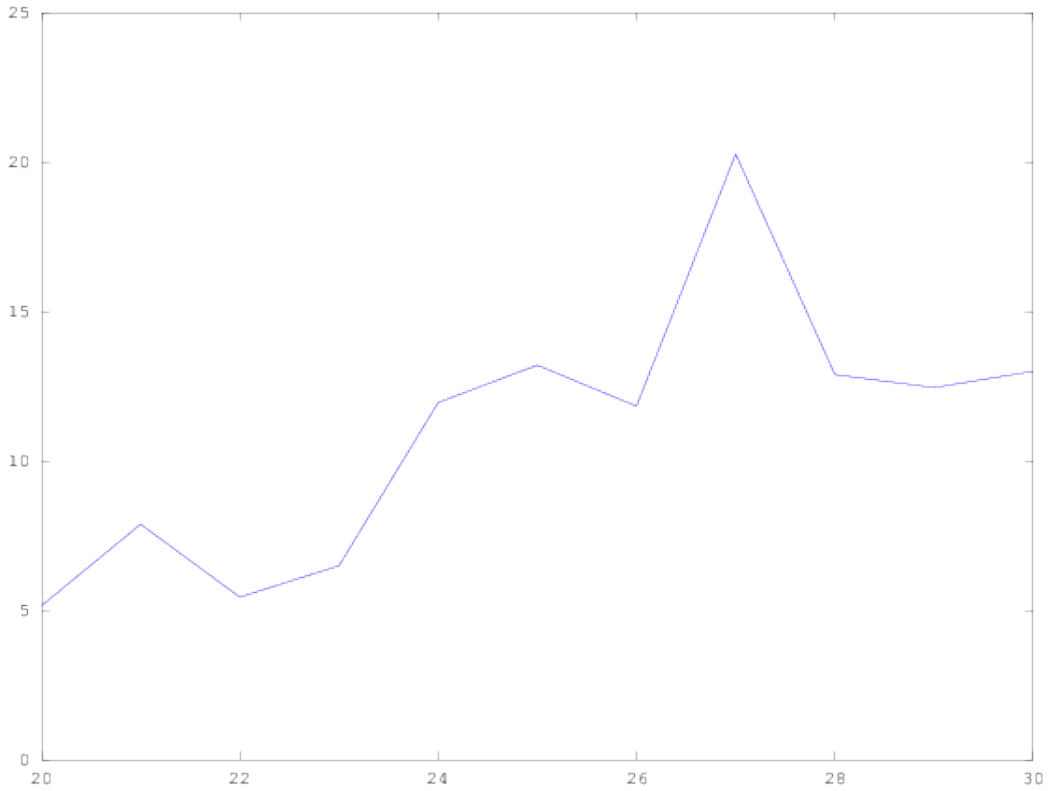

Figure 20: Sequential remote file read time(s/MB)(y) to logarithm of file size(x)

6.4.2 Prediction

Theoretically in this test, we should see higher and higher access times given more and more processes. This is because we have more and more blocks of data that are scattered throughout the disk, so because of this there will more likely be a page fault given more processes want to access different blocks of data on the disk. We know that the scheduler if it encounters a page fault, would invoke a context switch while the page fault is resolved. This context switch should let another processor run, which may then schedule another disk access somewhere else, resulting in another page fault and context switch. From our previous experiments, we know that the time for disk seek greatly outweighs the time it takes for a context switch, so our predicted time will largely be based on these seek times and how many seeks will occur given n processes. We estimate that the overall cost of software context switches will be around 0.1ms. For hardware, we need to read a block of 1MB data, combined with potential disk seeks. We estimate this overhead to be 0.8ms per additional process.

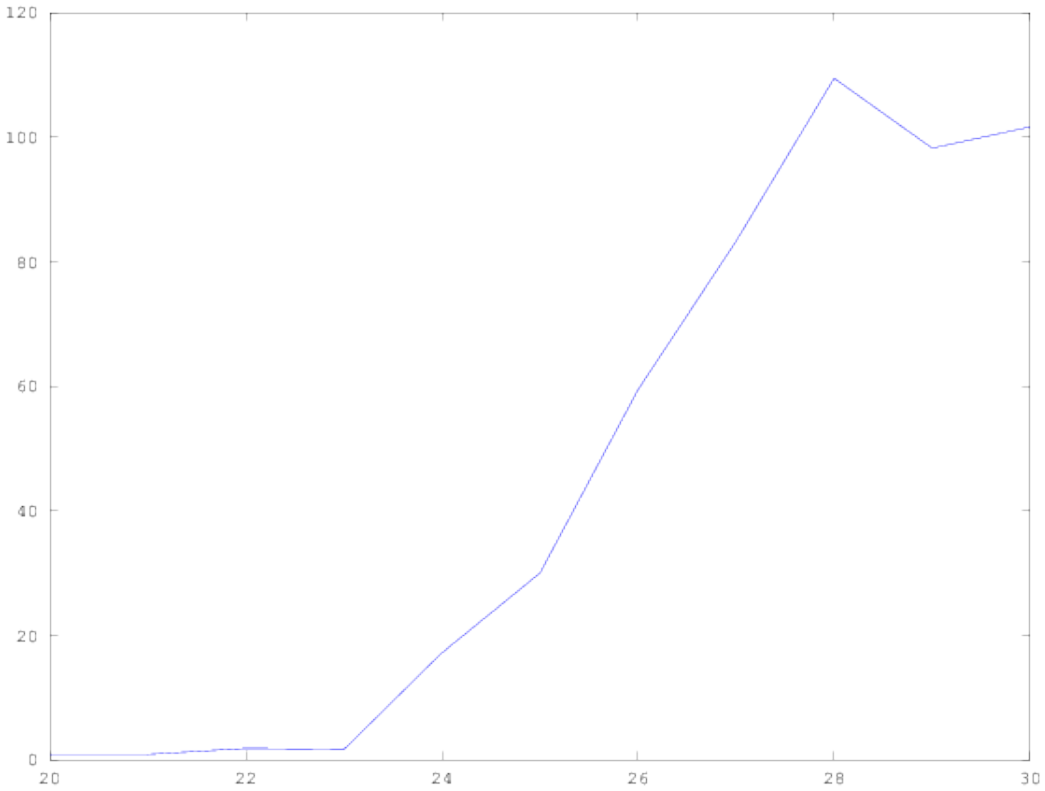

Figure 21: Random remote file read time(s/MB)(y) to logarithm of file size(x)

6.4.3 Results

# of proc	SW Est	HW Est	Total Est	Tested Mean (ms)
2	0.1	.8	0.9	.847
3	0.1	1.6	1.7	1.121
4	0.1	2.4	2.5	1.452
5	0.1	3.2	3.3	1.645
6	0.1	4	4.1	2.012
7	0.1	4.8	4.9	2.425
8	0.1	5.6	5.7	3.042
9	0.1	6.4	6.5	4.245

6.4.4 Comments

Our predictions were on the right track with regards to higher number of processes equals higher time until completion. With regards to the actual values themselves, we were still relatively accurate however not as precise as we could have been. This is largely due to the difficulty of predicting when a context switch will happen, along with inconsistencies in seek time between the varying blocks. We also notice a trend that with larger number of processes, we see increasingly longer completion times. The could be due to the fact that more processes are competing for resources and CPU time.

6.4.5 Sources

http://docs.oracle.com/cd/A87860_01/doc/server.817/a76992/ch21_res.htm